Andrew S. Tanenbaum

Számítógép-architektúrák

A kiadásért felel a Panem Könyvkiadó Kft. ügyvezetője, Budapest, 2006

Ez a könyv az Oktatási Minisztérium támogatásával,
a Felsőoktatási Tankönyv- és Szakkönyv-támogatási Pályázat
keretében jelent meg.

ISBN-10: 9-635454-57-0
ISBN-13: 978-9-635454-57-0

Lektorálta és szerkesztette: Dr. Máté Eörs

Fordította: Bohus Mihály, Alexin Zoltán, Gombás Éva, Nyúl László, Horváth Gyula, Erdőhelyi Balázs, Schrettner Lajos, Virágh János
Borítóterv: Tóth Attila
Tördelte: Pipaszó Bt.

panem@panem.hu
http://www.panem.hu

Tartalom

Előszó 13

1. Bevezetés 17
1.1. Strukturált számítógép-felépítés 18
1.1.1. Nyelvek, szintek és virtuális gépek 18
1.1.2. Korszerű többszintű számítógépek 20
1.1.3. A többszintű számítógépek fejlődése 23
1.2. Mérföldkövek a számítógépek felépítésében 28
1.2.1. Nulladik generáció: mechanikus számológépek (1642–1945) 30
1.2.2. Első generáció: vákuumcsövek (1945–1955) 31
1.2.3. Második generáció: tranzisztorok (1955–1965) 34
1.2.4. Harmadik generáció: integrált áramkörök (1965–1980) 36
1.2.5. Negyedik generáció: magas integráltságú áramkörök (1980–?) 38
1.2.6. Ötödik generáció: láthatatlan számítógépek 40
1.3. Számítógép-kiállítás 42
1.3.1. Technológiai és gazdasági mozgatórugók 42
1.3.2. A számítógépek termékskálája 44
1.3.3. Eldobható számítógépek 44
1.3.4. Mikrovezérlők 46
1.3.5. Játékgépek 48
1.3.6. Személyi számítógépek 49
1.3.7. Kiszolgálók 50
1.3.8. Munkaállomások gyűjteménye 50
1.3.9. Nagyszámítógépek 51
1.4. Néhány számítógépcsalád 52
1.4.1. A Pentium 4 áttekintése 52
1.4.2. UltraSPARC III áttekintése 57
1.4.3. A 8051 59
1.5. Mértékegységek 61
1.6. Könyvünk tartalmáról 62
1.7. Feladatok 63

2. Számítógéprendszerek felépítése 66
2.1. Processzorok 66
2.1.1. A CPU felépítése 67
2.1.2. Utasítás-végrehajtás 68
2.1.3. RISC és CISC 72
2.1.4. Korszerű számítógépek tervezési elvei 74
2.1.5. Utasításszintű párhuzamosság 75
2.1.6. Processzorszintű párhuzamosság 80
2.2. Központi memória 83
2.2.1. Bitek 84
2.2.2. Memóriacímek 84
2.2.3. Bájtsorrend 86
2.2.4. Hibajavító kódok 88
2.2.5. Gyorsítótár 92
2.2.6. Memóriatokozás és -típusok 95
2.3. Háttérmemória 96
2.3.1. Memóriahierarchia 96
2.3.2. Mágneslemezek 97
2.3.3. Hajlékonylemezek 100
2.3.4. IDE-lemezek 101
2.3.5. SCSI-lemezek 103
2.3.6. RAID 104
2.3.7. CD-ROM 108
2.3.8. Írható CD-k 112
2.3.9. Újraírható CD-k 114
2.3.10. DVD 115
2.3.11. Blu-Ray 117
2.4. Bemenet/Kimenet 117
2.4.1. Sínek 117
2.4.2. Terminálok 121
2.4.3. Egér 126
2.4.4. Nyomtatók 127
2.4.5. Telekommunikációs berendezések 133
2.4.6. Digitális kamerák 141
2.4.7. Karakterkódok 143
2.5. Összefoglalás 147
2.6. Feladatok 148

3. A digitális logika szintje 152
3.1. Kapuk és Boole-algebra 152
3.1.1. Kapuk 153
3.1.2. Boole-algebra 155
3.1.3. A Boole-függvények megvalósítása 157
3.1.4. Áramköri ekvivalencia 159
3.2. Alapvető digitális logikai áramkörök 163
3.2.1. Integrált áramkörök 163
3.2.2. Kombinációs áramkörök 165

3.2.3. Aritmetikai áramkörök 170
3.2.4. Órák 175
3.3. Memória 176
3.3.1. Tárolók 177
3.3.2. Flip-flopok 179
3.3.3. Regiszterek 181
3.3.4. Memóriaszervezés 183
3.3.5. Memórialapkák 186
3.3.6. RAM-ok és ROM-ok 188
3.4. CPU lapkák és sínek 191
3.4.1. CPU lapkák 192
3.4.2. Számítógépes sínek 194
3.4.3. Sínszélesség 196
3.4.4. Sínek időzítése 198
3.4.5. Sínütemezés 202
3.4.6. Sínműveletek 205
3.5. Példák CPU lapkákra 208
3.5.1. Pentium 4 208
3.5.2. UltraSPARC III 214
3.5.3. 8051 219
3.6. Példák sínekre 221
3.6.1. ISA sín 222
3.6.2. PCI sín 223
3.6.3. PCI Express 232
3.6.4. Univerzális soros sín 236
3.7. Kapcsolat a perifériákkal, interfészek 240
3.7.1. A B/K lapkák 240
3.7.2. Címdekódolás 242
3.8. Összefoglalás 245
3.9. Feladatok 246

4. A mikroarchitektúra szintje 251
4.1. Mikroarchitektúra-példa 251
4.1.1. Adatút 252
4.1.2. Mikroutasítások 258
4.1.3. Mikroutasítás-vezérlés: Mic–1 261
4.2. ISA-példa: az IJVM 266
4.2.1. Vermek 266
4.2.2. Az IJVM memóriamodellje 268
4.2.3. Az IJVM utasításkészlete 270
4.2.4. Java fordítása IJVM-re 273
4.3. Példa a megvalósításra 275
4.3.1. Mikroutasítások és jelölésrendszer 275
4.3.2. IJVM megvalósítása Mic–1 felhasználásával 280
4.4. A mikroarchitektúra szintjének tervezése 291
4.4.1. Sebesség vagy ár 291
4.4.2. A végrehajtási út hosszának csökkentése 294

4.4.3. Terv előre betöltéssel: a Mic–2 301
4.4.4. Csővonalas terv: a Mic–3 303
4.4.5. Hétszakaszú csővezeték: a Mic–4 309
4.5. A teljesítmény növelése 312
4.5.1. Gyorsítótár 313
4.5.2. Elágazásjövendölés 319
4.5.3. Sorrendtől eltérő végrehajtás és regiszterátnevezés 324
4.5.4. Feltételezett végrehajtás 329
4.6. Példák a mikroarchitektúra-szintre 332
4.6.1. A Pentium 4 CPU mikroarchitektúrája 332
4.6.2. Az UltraSPARC III Cu CPU-jának mikroarchitektúrája 338
4.6.3. A 8051 CPU mikroarchitektúrája 343
4.7. A Pentium, az UltraSPARC és a 8051 összehasonlítása 346
4.8. Összefoglalás 347
4.9. Feladatok 348

5. Az utasításrendszer-architektúra szintje 352
5.1. Az ISA-szint áttekintése 354
5.1.1. Az ISA-szint tulajdonságai 354
5.1.2. Memóriamodellek 356
5.1.3. Regiszterek 358
5.1.4. Utasítások 359
5.1.5. A Pentium 4 ISA-szintjének áttekintése 359
5.1.6. Az UltraSPARC III ISA-szintjének áttekintése 362
5.1.7. A 8051 ISA-szintjének áttekintése 365
5.2. Adattípusok 368
5.2.1. Numerikus adattípusok 368
5.2.2. Nem numerikus adattípusok 369
5.2.3. A Pentium 4 adattípusai 370
5.2.4. Az UltraSPARC III adattípusai 370
5.2.5. A 8051 processzor adattípusai 371
5.3. Utasításformátumok 371
5.3.1. Utasításformák tervezésének követelményei 372
5.3.2. A műveleti kód kiterjesztése 374
5.3.3. A Pentium 4 utasításformátumai 376
5.3.4. Az UltraSPARC III utasításformátumai 378
5.3.5. A 8051 utasításformátumai 379
5.4. Címzési módszerek 380
5.4.1. Címzési módok 380
5.4.2. Közvetlen címzés 380
5.4.3. Direkt címzés 381
5.4.4. Regisztercímzés 381
5.4.5. Regiszter-indirekt címzés 381
5.4.6. Indexelt címzés 382
5.4.7. Bázis-index címzési mód 384
5.4.8. Veremcímzés 384
5.4.9. Címzési módok elágazó utasításokban 388

5.4.10. A műveleti kód és a címzési mód ortogonalitása 388
5.4.11. A Pentium 4 címzési módjai 390
5.4.12. Az UltraSPARC III címzési módjai 392
5.4.13. A 8051 címzési módjai 392
5.4.14. A címzési módok összefoglalása 393
5.5. Utasítástípusok 394
5.5.1. Adatmozgató utasítások 394
5.5.2. Diadikus műveletek 395
5.5.3. Monadikus műveletek 396
5.5.4. Összehasonlító és feltételes elágazó utasítások 398
5.5.5. Eljáráshívó utasítások 400
5.5.6. Ismétléses vezérlés 400
5.5.7. Bemenet/kimenet 402
5.5.8. A Pentium 4 utasításai 405
5.5.9. Az UltraSPARC III utasításai 408
5.5.10. A 8051 utasításai 411
5.5.11. Az utasításrendszerek összehasonlítása 411
5.6. Vezérlési folyamat 414
5.6.1. Szekvenciális vezérlés és elágazás 414
5.6.2. Eljárások 415
5.6.3. Korutinok (társrutinok) 421
5.6.4. Csapdák 422
5.6.5. Megszakítások 423
5.7. Részletes példa: Hanoi tornyai 427
5.7.1. A Hanoi tornyai probléma megoldása Pentium 4 assembly nyelven 427
5.7.2. A Hanoi tornyai probléma megoldása UltraSPARC III assemblyben 429
5.8. Az Intel IA–64 architektúra és az Itanium 2 431
5.8.1. A Pentium 4 problémái 431
5.8.2. Az IA–64 modell: explicit utasításszintű párhuzamosság 433
5.8.3. A memóriahivatkozások csökkentése 433
5.8.4. Utasításütemezés 434
5.8.5. Feltételes elágazások csökkentése: predikáció 436
5.8.6. Spekulatív betöltés 438
5.9. Összefoglalás 439
5.10. Feladatok 439

6. Az operációs rendszer gép szintje 444
6.1. Virtuális memória 445
6.1.1. Lapozás 446
6.1.2. A lapozás megvalósítása 448
6.1.3. A kérésre lapozás és a munkahalmaz modell 452
6.1.4. Lapcserélő eljárások 453
6.1.5. Lapméret és elaprózódás 455
6.1.6. Szegmentálás 456
6.1.7. A szegmentálás megvalósítása 459
6.1.8. A Pentium 4 virtuális memóriája 462
6.1.9. Az UltraSPARC III virtuális memóriája 466

6.1.10. Virtuális memória és gyorsítótár 469
6.2. Virtuális B/K utasítások 469
6.2.1. Fájlok 470
6.2.2. A virtuális B/K utasítások megvalósítása 472
6.2.3. Könyvtárkezelő utasítások 475
6.3. A párhuzamos feldolgozás virtuális utasításai 476
6.3.1. Processzusok létrehozása 477
6.3.2. Versenyhelyzetek 478
6.3.3. Processzusok szinkronizációja szemaforokkal 482
6.4. Példák operációs rendszerekre 486
6.4.1. Bevezetés 486
6.4.2. Példák virtuális memória kezelésére 495
6.4.3. Példák virtuális B/K műveletekre 499
6.4.4. Példák processzusok kezelésére 510
6.5. Összefoglalás 516
6.6. Feladatok 517

7. Az assembly nyelv szintje 524
7.1. Bevezetés az assembly nyelvbe 525
7.1.1. Mi az assembly nyelv? 525
7.1.2. Miért használnak assembly nyelvet? 526
7.1.3. Az assembly utasítások alakja 528
7.1.4. Pszeudoutasítások 531
7.2. Makrók 534
7.2.1. A makrók definíciója, hívása, kifejtése 534
7.2.2. Paraméteres makrók 536
7.2.3. Előnyös tulajdonságok 537
7.2.4. A makróassembler működése 538
7.3. Az assembler menetei 539
7.3.1. Kétmenetes assemblerek 539
7.3.2. Első menet 540
7.3.3. Második menet 544
7.3.4. Szimbólumtábla 545
7.4. Szerkesztés és betöltés 547
7.4.1. A szerkesztő feladatai 548
7.4.2. A tárgymodul szerkezete 551
7.4.3. Hozzárendelési idő és dinamikus áthelyezés 552
7.4.4. Dinamikus szerkesztés 555
7.5. Összefoglalás 559
7.6. Feladatok 560

8. Párhuzamos számítógép-architektúra 563
8.1. Lapkaszintű párhuzamosság 564
8.1.1. Utasításszintű párhuzamosság 565
8.1.2. Lapkaszintű többszálúság 572
8.1.3. Egylapkás multiprocesszorok 578

8.2. Társprocesszorok 583
8.2.1. Hálózati processzorok 584
8.2.2. Médiaprocesszorok 592
8.2.3. Kriptoprocesszorok 597
8.3. Közös memóriás multiprocesszorok 598
8.3.1. Multiprocesszorok és multiszámítógépek 598
8.3.2. Memóriaszemantika 606
8.3.3. UMA sínrendszerű SMP-architektúrák 610
8.3.4. NUMA-multiprocesszorok 619
8.3.5. COMA-multiprocesszorok 628
8.4. Üzenetátadásos multiszámítógépek 629
8.4.1. Összekötő hálózatok 631
8.4.2. MPP – erősen párhuzamos processzor 635
8.4.3. Klaszterszámítógépek 644
8.4.4. A multiszámítógépek kommunikációs szoftvere 650
8.4.5. Ütemezés 652
8.4.6. Alkalmazásszintű közös memória 654
8.4.7. Teljesítmény 661
8.5. Grid számítások 667
8.6. Összefoglalás 670
8.7. Feladatok 671

9. Ajánlott olvasmányok és irodalomjegyzék 674
9.1. Javasolt további olvasmányok 674
9.1.1. Bevezető és általános művek 674
9.1.2. Számítógéprendszerek felépítése 676
9.1.3. Digitális logika szintje 677
9.1.4. A mikroarchitektúra szintje 678
9.1.5 Az utasításrendszer-architektúra szintje 679
9.1.6. Az operációs rendszer gép szintje 679
9.1.7. Assembly nyelv szintje 680
9.1.8. Párhuzamos számítógép-architektúrák 680
9.1.9. Bináris és lebegőpontos számok 682
9.1.10. Assembly nyelvű programozás 683
9.2. Bibliográfia 684

A) Bináris számok 696
A.1. Véges pontosságú számok 696
A.2. Számrendszerek alapszámai 698
A.3. Konverzió egyik alapról a másik alapra 700
A.4. Negatív bináris számok 702
A.5. Bináris aritmetika 705
A.6. Feladatok 706

B) Lebegőpontos számok 708
B.1. A lebegőpontos számábrázolás elvei 708
B.2. Az IEEE 754-es lebegőpontos szabvány 711
B.3. Feladatok 715

C) Assembly nyelvű programozás 717
C.1. Áttekintés 718
C.1.1. Az assembly nyelv 718
C.1.2. Egy rövid assembly nyelvű program 719
C.2. A 8088-as processzor 720
C.2.1. A processzorciklus 720
C.2.2. Az általános regiszterek 722
C.2.3. Mutatóregiszterek 723
C.3. Memória és címzés 725
C.3.1. Memóriaszervezés és szegmensek 725
C.3.2. Címzés 726
C.4. A 8088 utasításrendszere 730
C.4.1. Adatmozgatás, -másolás és aritmetika 730
C.4.2. Logikai, bit- és léptető műveletek 733
C.4.3. Ciklusszervezés és ismétlődő string műveletek 734
C.4.4. Ugró és eljáráshívó utasítások 735
C.4.5. Szubrutin hívások 737
C.4.6. Rendszerhívások és rendszerszubrutinok 738
C.4.7. Záró megjegyzések az utasításrendszerről 740
C.5. Az assembler 741
C.5.1. Bevezetés 741
C.5.2. Az ACK-alapú assembler, az *as88* 742
C.5.3. Eltérések más 8088-as assemblerektől 746
C.6. A nyomkövető 747
C.6.1. Nyomkövető parancsok 749
C.7. Alapismeretek 751
C.8. Példák 752
C.8.1. Helló Világ példa 752
C.8.2. Példa az általános regiszterekre 756
C.8.3. A call utasítás és a mutató regiszterek 758
C.8.4. Hibakeresés egy tömbkiíró programban 760
C.8.5. String-kezelés és string-utasítások 763
C.8.6. Ugrótáblák 766
C.8.7. Pufferelt és véletlen fájlelérés 768
C.9. Feladatok 771

Angol–magyar tárgymutató 773
Magyar–angol tárgymutató 795

Előszó

Könyvünk első négy kiadása azt az elvet követte, hogy a számítógép egymásra épülő szintek hierarchiájaként fogható fel, amelyek mindegyikének megvan a saját, jól meghatározott feladata. Ez az alapelv ma is ugyanúgy érvényben van, mint az első kiadás megjelenésekor volt, ezért az ötödik kiadás is ezen alapszik. Részletesen ismertetjük a digitális logika, a mikroarchitektúra, az utasításrendszer-architektúra, az operációs rendszer gép és az assembly nyelv szintjét egyaránt, amint azt az első négy kiadásban is tettük.

Jóllehet az alapfelépítés ugyanaz, az ötödik kiadás – követve a gyorsan változó számítógép-iparágat – számos vonatkozásában megváltozott. Néhol lényegi, máshol csak kisebb a változás. Például aktualizáltuk a szemléltetésre használt gépeket, amelyek így ebben a kiadásban az Intel Pentium 4, a Sun UltraSPARC III és az Intel 8051. A Pentium 4 a mai asztali számítógépek elterjedt mikroprocesszora. Az UltraSPARC III ugyancsak elterjedt, amelyet széles körben használnak a közepes és nagy többprocesszoros rendszerekben.

A 8051 választása azonban többeket meglephet. Ez a tiszteletre méltó lapka már évtizedek óta jelen van. A beágyazott rendszerek rendkívüli fejlődése azonban megszerezte számára a méltó helyet. Mivel a rádiós óráktól kezdve a mikrohullámú sütőkig szinte mindent számítógépek vezérelnek, a beágyazott rendszerek iránti érdeklődés egyre nő, és a 8051-es lapkát széles körben használják bennük rendkívül alacsony (filléres) ára, a hozzá már létező szoftverek és perifériák mennyisége, valamint a programozásához értő fejlesztők tömege miatt.

A könyvet az oktatásban jegyzetként használó számos egyetemi oktató ismételten kérte az évek során, hogy assembly nyelvű programozás is jelenjék meg benne. Az ötödik kiadás ezért kiegészült ezzel az anyaggal, amely a C függelékben és a könyvhöz tartozó CD-ROM-lemezen található. A választás az Intel 8088 nyelvére esett, mivel ez a határtalanul népszerű Pentiumnak is az alapkészlete. Választhattam volna az UltraSPARC vagy a MIPS nyelvét, esetleg egy olyan processzorét, amelyről szinte senki nem hallott, de mint ösztönző eszköz, a 8088 a jobb választás, hiszen a legtöbb hallgatónak van otthon Pentiumos gépe, amely képes a 8088-as programok futtatására. Mivel az assembly kódban rendkívül nehéz a hibakeresés, így a CD-melléklet a Windows-, Unix- és Linux-környezet mindegyikéhez tartalmaz assembly fejlesztést segítő eszközöket, például egy 8088 assemb-

lert, egy szimulátort és egy nyomkövetőt. A CD mellett ugyanezek az eszközök megtalálhatók a könyv webhelyén is (lásd 15. o.).

Az évek során a könyv egyre hosszabb lett. A terjedelem növekedése elkerülhetetlen egy folyamatosan fejlődés alatt álló terület ismertetése során, amelyben naponta új ismeretek bukkannak fel. Ennek eredményeképpen, ha ezt a könyvet tanfolyami segédletként vagy féléves jegyzetként kívánják használni, valószínű, hogy nem lehet vele egyetlen tanfolyam/félév alatt végezni. Egy lehetséges megközelítés szerint legalább az első három fejezet teljes egészében, a negyedik fejezet a 4.4. alfejezettel bezárólag és az ötödik fejezet szintén teljes egészében képezheti egy tanfolyam vagy félév anyagát. Az esetleges fennmaradó időben, az oktató választásától függően, a negyedik fejezet további részével, valamint a hatodik, hetedik és nyolcadik fejezet egyes részeivel foglalkozhatnak.

Az alábbiakban fejezetről fejezetre felsoroljuk a negyedik kiadáshoz képest történt változásokat. Az első fejezet továbbra is a számítógép-architektúrák történelmi áttekintését tartalmazza, bemutatva a jelenlegi helyzetet, valamint az ide vezető út jelentős mérföldköveit. E fejezet kiegészült még a jelenleg létező számítógéptípusok spektrumának ismertetésével és a három fő példaprocesszor (Pentium 4, UltraSPARC III és 8051) vázlatos bemutatásával.

A második fejezetben a bemeneti/kimeneti (B/K) eszközök témaköre kiegészült, kiemelve a modern eszközökben használt technológiákat, köztük a digitális kamerákat, valamint a DSL és kábeles internetet is.

A harmadik fejezetet alaposan átdolgoztuk, és a számítógépsínekkel, valamint a modern B/K lapkákkal foglalkozik. Itt a három új példaprocesszort lapkaszinten ismertetjük. E fejezet kiegészült a PCI Express sín leírásával, amely várhatóan hamarosan a PCI sín helyébe fog lépni.

A negyedik fejezet mindig is népszerű volt, hiszen a számítógép tényleges működését mutatja be, így nagy része változatlan a negyedik kiadás óta. Bár kiegészült új pontokkal is, amelyek a Pentium 4, az UltraSPARC III és a 8051 mikroarchitektúra szintjét ismertetik.

Az ötödik, hatodik és hetedik fejezetet – új példákat használva – szintén frissítettük, ettől eltekintve azonban viszonylag változatlan. A hatodik fejezet ugyan a Windows NT helyett a Windows XP rendszert használja példaként, de a tárgyalásnak ezen a szintjén ez alig jelent némi változást.

A nyolcadik fejezet ezzel szemben jelentős mértékben megváltozott, hogy a párhuzamos feldolgozás minden formájában fellelhető új tevékenységeket kövesse. A párhuzamos rendszerek öt különböző osztályát mutatja be, kezdve a lapkán belüli párhuzamossággal (utasításszintű párhuzamosság, lapkán belüli többszálúság és egylapkás multiprocesszorok), folytatva a társprocesszorokkal, megosztott memóriájú rendszerekkel és a számítógépklaszterekkel (cluster), majd a hálózat- rács- (grid) technológia rövid ismertetésével zárul. Számos új példa is található ebben a fejezetben, a TriMedia CPU-tól a GlueGene/L, Red Storm- és Google-klaszterekig.

A kilencedik fejezet hivatkozásai ugyancsak jelentősen frissültek. A számítógép-architektúra dinamikusan fejlődő terület. A jelen kiadás hivatkozásainak több mint a fele a negyedik kiadás óta megjelent írásokra mutat.

Az A és B függelék nem változott a legutóbbi kiadás óta, a gépi nyelvű programozást ismertető C függelék azonban teljesen új. Ez a CD-n és a weboldalon található eszközök segítségével történő assembly nyelvű programozás részletes útmutatása. A függeléket az amszterdami Vrije Egyetemen dolgozó Dr. Evert Wattel írta, aki többéves tapasztalattal rendelkezik a hallgatók ugyanezen eszközök segítségével zajló tanításában. Szeretném ezúton is kifejezni neki a köszönetemet.

A gépi nyelvű programozást segítő eszközökön kívül a weboldalon található egy – a negyedik fejezettel együtt használható – grafikus szimulátor is, amelyet az Oberlin Főiskolán dolgozó Richard Salter professzor készített, s amely a fejezetben tárgyalt alapelvek megértésében segíti a hallgatókat. Nagyon köszönöm, hogy rendelkezésemre bocsátotta ezt a szoftvert.

A könyvben található ábrák és az oktatók számára készített PowerPoint bemutatók is megtalálhatók a weboldalon. Az URL:

http://www.prenhall.com/tanenbaum

Ezen az oldalon kattintson a könyv képe alatti „Companion Website" hivatkozásra, majd a megjelenő menüből válassza ki a megfelelő oldalt.

Sokan olvasták a kéziratot vagy annak egyes részeit, és hasznos javaslatokkal vagy egyéb módon segítették munkámat. Különösen Nikitas Alexandridis, Shekar Borkar, Herbert Bos, Scott Cannon, Doug Carmean, Alan Charlesworth, Eric Cota-Robles, Michael Fetterman, Quinn Jacobson, Thilo Kielmann, Iffat Kazi, Saul Levy, Ahmed Louri, Abhijit Pandya, Krist Petersen, Mark Russinovich, Ronald Schroeder és Saim Ural segítségét köszönöm.

Köszönetet mondok Jim Goodmannek a könyv, különösen a 4. és az 5. fejezet megszületésében nyújtott segítségéért. Az övé volt az az ötlet, hogy a Java virtuális gépet használjuk, és ezáltal a könyv sokkal jobb lett.

Végül újfent köszönöm Suzanne soha el nem fogyó szeretetét és türelmét, amely még 15 könyv megírása után is kitart. Barbara és Marvin mindig örömet szereznek, és most már ők is tudják, miből él egy professzor. A Holland Királyi Művészeti és Tudományos Akadémia (Royal Netherlands Academy of Arts and Sciences) 2004-ben nekem adományozta a már régóta áhított akadémiai professzori kinevezést, ezzel megszabadított az akadémia néhány kevésbé vonzó aspektusától (mint például az unalmas, vég nélküli bizottsági ülések), amiért örökké hálás leszek.

Andrew S. Tanenbaum

1. Bevezetés

A digitális számítógép olyan gép, amely a neki szóló utasítások alapján az emberek számára problémákat old meg. Azt az utasítássorozatot, amely leírja, hogyan oldjunk meg egy feladatot, **program**nak nevezzük. Minden egyes számítógép elektronikus áramkörei egyszerű utasítások korlátozott halmazát képesek felismerni és közvetlenül végrehajtani. Végrehajtásuk előtt programjaikat ezen a halmazon kell leírni. Legtöbb utasításuk ritkán bonyolultabb az alábbiaknál:

Adj össze két számot!
Ellenőrizz egy számot, vajon nulla-e!
Egy adatot másolj a számítógép memóriájában egyik helyről a másikra!

Egy számítógép egyszerű utasításainak együttese egy olyan nyelvet alkot, amelyen az ember a számítógéppel képes kommunikálni. Az ilyen nyelvet **gépi nyelv**nek nevezzük. Egy új számítógép tervezőinek el kell dönteniük, hogy milyen egyszerű utasításokat vegyenek fel annak gépi nyelvébe. Általában az egyszerű utasításokat – a tervezett felhasználást és teljesítménykövetelményeket figyelembe véve – a lehető legegyszerűbbre választják, ezzel is csökkentve az elektronika bonyolultságát és árát. Éppen a gépi nyelvek ilyen egyszerű volta miatt az emberek számára használatuk nehézkes és fárasztó.

Az idők folyamán ez a felismerés vezetett a számítógépek absztrakciók sorozataként való strukturálására. Egy absztrakciós szint az alatta lévő absztrakciós szintre épül. Ezzel a komplexitás kezelhetővé válik, és a számítógépek tervezése szisztematikus, szervezett módon történhet. Ezt a szemléletet nevezzük **strukturált számítógép-felépítés**nek. A következő részben kifejtjük ennek a fogalomnak a jelentését, ezt követően pedig áttekintjük történeti fejlődését, a jelenlegi helyzetet, és bemutatunk néhány fontosabb példát is.

1.1. Strukturált számítógép-felépítés

Már említettük, hogy óriási különbség van aközött, hogy mi az, ami az embernek, és mi az, ami a számítógépnek a legmegfelelőbb. Az ember az X dolgot akarja, de a számítógép csak az Y dolgot tudja. Ez bizony probléma. Könyvünk célja, hogy elmagyarázza, hogyan oldható fel ez a probléma.

1.1.1. Nyelvek, szintek és virtuális gépek

A probléma két oldalról is megközelíthető. Mindkettő az ember számára a gépi nyelvnél kényelmesebben használható új utasításrendszer tervezésére épül. Az új utasítások együttese is egy nyelvet alkot – nevezzük ezt L1-nek –, ugyanúgy, ahogy a beépített utasítások nyelvet alkotnak – legyen ez utóbbi L0. A kétféle megközelítés abban különbözik egymástól, hogy a számítógép hogyan hajtja végre az L1 nyelven írott programokat, miközben csak a saját, L0 gépi nyelvén írt programokat képes végrehajtani.

Az L1 nyelvű program végrehajtásának egyik módja az, hogy először minden utasítását helyettesítjük az L0 nyelv utasításainak egy vele ekvivalens sorozatával. Az így nyert program teljes egészében az L0 utasításaiból áll. Ekkor az eredeti L1 nyelvű program helyett a számítógép ezt az L0 nyelvű programot hajtja végre. Ezt a módszert **fordítás**nak nevezzük.

A másik módszer szerint megírunk egy L0 nyelvű programot, amely az L1 nyelvű programokat bemenő adatokként kezeli, és úgy hajtja végre azokat, hogy minden utasításukat elemzi, és a vele ekvivalens L0 nyelvű utasássorozatot azonnal végrehajtja. Ez a módszer nem igényli, hogy először egy új, L0 nyelvű programot állítsunk elő. A módszert **értelmezés**nek, a végrehajtó programot pedig **értelmező**nek nevezzük.

A fordítás és az értelmezés módszere hasonló. Végül is mindkettő az L1 utasításait az L0 utasításaiból álló sorozatok végrehajtásával teljesíti. A különbség csak az, hogy a fordítás esetében először az egész L1 nyelvű programot átírjuk L0 nyelvűvé, majd az L1 programtól megszabadulunk, és az L0 programot töltjük a számítógép memóriájába, és azt hajtjuk végre. A végrehajtás során az újonnan előállított L0 program az, ami fut, és ami a számítógépet vezérli.

Az értelmezés esetében L1 minden utasításának elemzését és dekódolását az utasítás azonnali végrehajtása követi. Nem keletkezik lefordított program. A számítógépet az értelmező vezérli, számára az L1 nyelvű program csak adat. Mindkét módszert – és egyre gyakrabban a kettő kombinációját is – széles körben alkalmazzák.

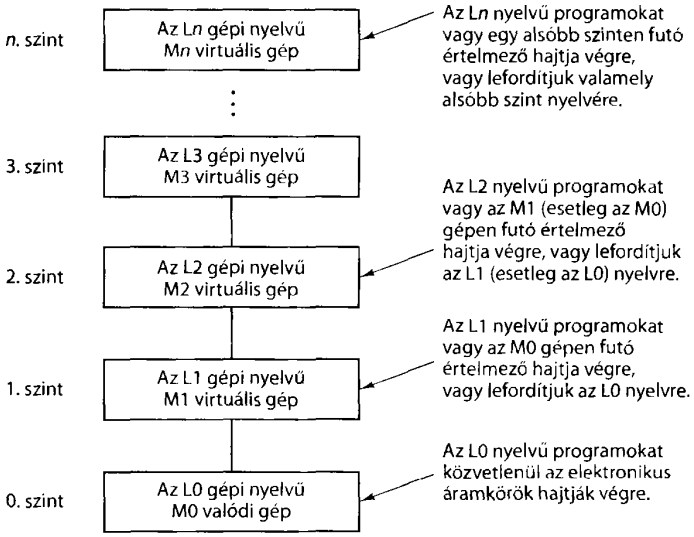
Ahelyett, hogy a fordítás, illetve értelmezés fogalomkörében maradnánk, egyszerűbb, ha egy hipotetikus, másként nevezve **virtuális gépet** képzelünk magunk elé, amelynek gépi nyelve az L1. Nevezzük ezt a virtuális gépet M1-nek (az L0-nak megfelelő valódi gépet pedig M0-nak). Ha az M1 gépet elég olcsón meg tudnánk építeni, nem lenne szükségünk semmiféle L0 nyelvre, sem L0 programokat végrehajtó gépre. Programjainkat egyszerűen L1 nyelven írnánk, amelyeket az M1 számítógép közvetlenül végrehajtana. Még ha annak a virtuális gépnek, amelynek

gépi nyelve az L1, elektronikus áramkörökből való megépítése túl drága, akkor is írhatunk rá programokat. Ezek a programok feldolgozhatók L0 nyelven írt értelmezőkkel vagy fordítókkal, amelyek közvetlenül végrehajthatók valóságosan létező számítógépeken. Más szóval úgy írhatunk programokat virtuális gépekre, mintha azok valóban léteznének.

Azért, hogy a fordítás vagy az értelmezés a gyakorlatban is alkalmazható legyen, az L0 és az L1 nyelv nem különbözhet „nagyon". Ez a korlát többnyire azt jelenti, hogy bár az L1 jobb az L0-nál, a legtöbb alkalmazáshoz még messze nem ideális. Ez talán riasztó annak fényében, hogy az L1-nek mi is volt az eredeti célja – megszabadítani a programozót attól a tehertől, hogy olyan nyelven fogalmazzon meg algoritmusokat, amely inkább gépre, mint emberre szabott. A helyzet azonban nem reménytelen.

A nyilvánvaló megoldás az, hogy képezzünk utasításokból egy újabb halmazt, amely az L1-hez képest már inkább emberre és kevésbé gépre szabott. Ez a harmadik halmaz is nyelvet alkot, nevezzük L2-nek (a hozzá tartozó M2 virtuális géppel). L2-ben úgy írhatunk programokat, mintha valóban létezne ez az L2 gépi nyelvű virtuális gép. Ezek a programok L1 nyelvre fordíthatók vagy értelmezhetők egy L1 nyelven írt fordítóval, illetve értelmezővel.

Nyelvek egy olyan sorozatát létrehozva, amelyben mindegyik a megelőzőnél már kényelmesebben használható, végül eljuthatunk egy számunkra már megfelelőhöz. Mindegyik nyelv az őt megelőzőre épül, ezért az ilyen módszert alkalmazó számítógépre úgy is tekinthetünk, mint egymás feletti **rétegek** vagy **szintek** sorozatára, ahogy az az 1.1. ábrán is látható. A legalsó nyelv, illetve szint a legegyszerűbb, míg a legfelső a legkifinomultabb.



1.1. ábra. *Egy többszintű gép*

A nyelvek és a virtuális gépek között szoros kapcsolat van. Minden gépnek van saját gépi nyelve, amely a gép által végrehajtható utasítások együtteséből áll, azaz egy gép definiál egy nyelvet. Hasonlóan egy nyelv is definiál egy gépet, nevezetesen azt a gépet, amely képes az ezen a nyelven írt bármely program végrehajtására. Természetesen egy nyelv által definiált gép elektronikus áramkörökből való közvetlen megépítése elképesztően bonyolult és drága is lehet, de nem elképzelhetetlen. Egy C, C++ vagy Java gépi nyelvű gép valóban bonyolult, de a mai technológiák mellett könnyen megépíthető lenne. Alapos oka van azonban annak, hogy ilyen gépek nem épülnek: nem lenne költséghatékony más módszerekkel összehasonlítva. Önmagában az még nem elég, hogy valamit meg lehet csinálni: egy gyakorlatias elgondolásnak költséghatékonynak is kell lennie.

Egy n szintű számítógépet n különböző virtuális gépnek is tekinthetünk, ahol mindegyikük rendelkezik a saját gépi nyelvével. A „szint" és a „virtuális gép" kifejezéseket egymással felcserélhető fogalmakként fogjuk használni. Csak az L0 nyelven írt programokat képesek az elektronikus áramkörök közvetlenül végrehajtani anélkül, hogy fordítást vagy értelmezést kellene közbeiktatnunk. Az L1, L2, ..., Ln nyelvű programokat vagy egy alsóbb szinten futó értelmezővel kell végrehajtatnunk, vagy le kell fordítanunk az alsóbb szint nyelvére.

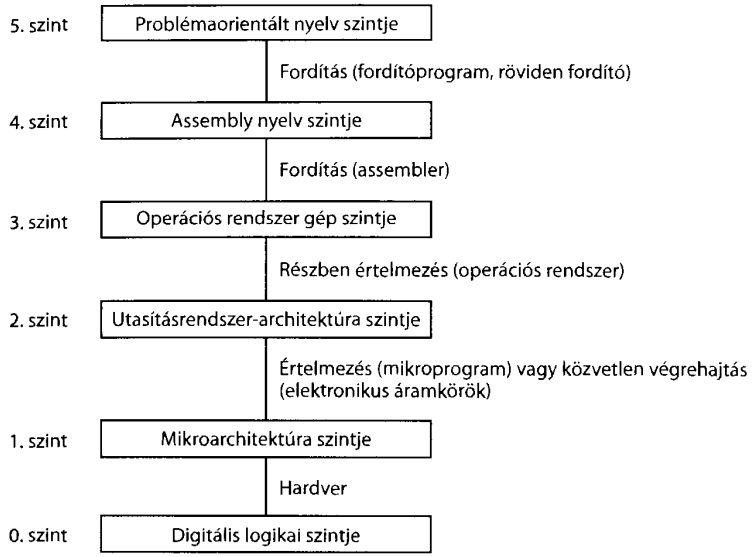
Amikor valaki az n. szintű virtuális gépre ír programokat, nem kell törődnie a szükséges értelmezőkkel és fordítókkal. A gép felépítése biztosítja, hogy programjai valahogyan végrehajthatók. Érdektelen, hogy a programot egy értelmező lépésenként fogja végrehajtani, az értelmezőt magát megint egy másik értelmező, vagy esetleg már az elektronika teljesíti a feladatot. Az eredmény mindkét esetben ugyanaz: a programok végrehajtása megtörténik.

Az n szintű gépeket használó programozók többségének csak a legfelső szintet kell ismernie, ami a legalsó szint gépi nyelvéhez a legkevésbé hasonlít. Akik viszont kíváncsiak, hogy hogyan is működik valójában egy számítógép, azoknak minden szintet tanulmányozniuk kell. Új számítógépek vagy új szintek (azaz új virtuális gépek) tervezőinek a legfelső mellett több más szintet is ismerniük kell. A szintek sorozatából felépülő gépek tervezésének elveiről és módszereiről, továbbá a szintek részleteiről szól könyvünk jelentős része.

1.1.2. Korszerű többszintű számítógépek

A legtöbb mai számítógép két- vagy többszintű. Még az 1.2. ábra szerinti hatszintű gépek is léteznek. Alul a 0. szint a gép valódi hardvere. Ennek az áramkörei hajtják végre az 1. szintű gépi nyelvű programokat. A teljesség kedvéért meg kell jegyeznünk, hogy a mi 0. szintünk alatt is van még egy szint. Ez az ún. **eszközszint** az elektronikai tervezés világához tartozik (és így könyvünk tárgykörén kívül esik), amelyet az ábra nem tartalmaz. Ezen a szinten találja a tervező az egyes tranzisztorokat, mint a számítógép-tervezés legalsó szintű építőköveit. Azzal a kérdéssel, hogy hogyan működnek a tranzisztorok, már a szilárdtestfizika területére vetődünk.

Az általunk vizsgált legalsó, **digitális logika szintjén** a **kapuk** a lényeges elemek. Bár a kapuk olyan analóg alkatrészekből épülnek fel, mint például a tranzisztorok, szerepük szerint digitális eszközöknek tekinthetők. Minden kapunak egy vagy több digitális bemenete van (a 0 vagy az 1 értéket reprezentáló jelek), kimenetként pedig ezekből egyszerű függvényértékeket számolnak ki, mint amilyen az AND (logikai „és") vagy az OR (logikai „vagy"). Egy kapu legfeljebb néhány tranzisztorból áll. Néhány kapuból összeállítható egy 1 bites memória, amely a 0 vagy az 1 értéket képes tárolni. Az 1 bites memóriákat 16-os, 32-es vagy 64-es csoportokba rendezve készíthetünk például regisztereket. Minden **regiszter**ben meghatározott értékhatárig egy bináris számot tárolhatunk. Kapukból építhetjük fel magát az aritmetikai egységet is. A kapukat és a digitális logika szintjét a 3. fejezetben tárgyaljuk részletesen.



1.2. ábra. *Egy hatszintű számítógép. Az egyes szinteket megvalósító módszert a szint alatt jelöltük (zárójelben a megvalósító program nevével)*

A következő felsőbb szint a **mikroarchitektúra szintje**. Ezen a szinten találjuk az (általában) 8–32 elemű, lokális memóriaként használt regiszterkészletet és az ún. **aritmetikai-logikai egység**et (**Arithmetic Logic Unit, ALU**), amely az egyszerű aritmetikai műveletek elvégzésére képes. A regiszterek az ALU-hoz kapcsolódnak, az adatok áramlásának útja az **adatút**. Az adatút alapfeladata az, hogy kiválasszon egy vagy két regisztert, az ALU-val műveletet végeztessen el rajtuk (például adja össze a tartalmukat), az eredményt pedig valamelyik regiszterben tárolja.

Egyes gépeken az adatút működését az ún. **mikroprogram** vezérli, míg más gépeken a vezérlés közvetlenül a hardver feladata. Könyvünk első három kiadásában ezt a szintet „mikroprogramszintnek" neveztük azért, mert ezen a szinten a múltban szinte kivétel nélkül szoftverértelmezőt használtak. Az adatutat manapság gyakran (legalább is részben) közvetlenül a hardver vezérli, emiatt változtattunk az elnevezésen.

Azokon a gépeken, amelyeken az adatút szoftvervezérlésű, a mikroprogram egy értelmező program, amely a 2. szintű utasításokat egyenként betölti, elemzi és az adatutat használva végrehajtja. Az ADD utasítás esetében például betölti az utasítást, megkeresi és regiszterekbe helyezi az operandusait, az ALU kiszámolja az összeget, végül az eredményt a megfelelő helyre elküldi. Ha a gépen az adatút hardvervezérlésű, akkor is hasonlók az egyes lépések, csak akkor nem tárolt program vezérli a 2. szintű utasítások értelmezését.

A 2. szintet **utasításrendszer-architektúra szintjének (Instruction Set Architecture, ISA-szint)** nevezzük. Minden számítógépgyártó vállalat az általa forgalmazott gépekhez ad egy kézikönyvet „A gépi nyelv referencia kézikönyve", „A Western Wombat Model 100X számítógép működésének elvei" vagy valami hasonló címmel. Ezek általában az ISA-szintről szólnak, az alacsonyabb szinteket nem tárgyalják. A gépi utasításrendszer leírása tulajdonképpen nem más, mint azoknak az utasításoknak a leírása, amelyeket a mikroprogram vagy a hardver végrehajtó áramkör értelmez. Ha a gyártó kétféle ISA-szintű értelmezőt biztosít ugyanahhoz a számítógépéhez, két „gépi nyelv" referencia kézikönyvet kell adnia, egyet-egyet mindegyik értelmezőhöz.

A következő szint általában egy kevert szint. A szint nyelvéhez tartozó utasítások többsége az ISA-szinten is megvan. (Semmi akadálya annak, hogy valamely szint utasításai más szintek utasításai között is szerepeljenek.) Ezen felül a szint új utasításokkal, eltérő memóriaszervezéssel, több program egyidejű futtatásának képességével és egyéb tulajdonságokkal rendelkezik. A 3. szinten sokkal változatosabbak a konstrukciók, mint az 1. vagy a 2. szinten.

A 3. szint új szolgáltatásait a 2. szinten futó értelmező biztosítja, amelyet hagyományosan operációs rendszernek szoktunk nevezni. A 2. szintről örökölt 3. szintű utasításokat nem az operációs rendszer, hanem közvetlenül a mikroprogram (vagy a hardver) hajtja végre. Másként fogalmazva, a 3. szintű utasítások egy részét az operációs rendszer, más részét közvetlenül a mikroprogram értelmezi. Ezt értjük a „kevert" szint elnevezés alatt. Könyvünkben ezt a szintet **operációs rendszer gép szintjének** nevezzük.

A 3. és a 4. szint között alapvető eltérés van. Az alsó három szintet nem egyszerű hétköznapi programozóknak találták ki, azok elsősorban a magasabb szinteken szükséges értelmezők és fordítók futtatására szolgálnak. Ezeket az értelmezőket és fordítókat **rendszerprogramozók** írják, akik új virtuális gépek tervezésére és megvalósítására szakosodtak. A 4. és az e fölötti szinteket az alkalmazási feladatokat megoldó programozóknak szánták.

A 4. szint és a magasabb szintek megvalósításának módjában is változás van. A 2. és a 3. szintet mindig értelmezővel, míg a 4. szintet és az e fölöttieket általában – de nem mindig – fordítóval valósítják meg.

Az 1., 2. és 3. szint, illetve a 4., 5. és magasabb szintek nyelveinek természetében találhatjuk a további különbséget. Az 1., 2. és 3. szint gépi nyelvei numerikusak, a rajtuk írt programok hosszú számsorozatok, ami kedvező a gépek, de kedvezőtlen az ember számára. A 4. szinttől kezdődően a nyelvek szavakból és az ember számára is jelentéssel bíró rövidítésekből állnak.

A 4., az assembly nyelv szintje valójában az alsóbb szintekhez tartozó nyelvek szimbolikus formája. Ezen a szinten lehet az 1., 2. és 3. szintekre programot írni azok virtuális gépeinek saját nyelveinél kényelmesebb nyelven. Az assembly nyelvű programokat először lefordítjuk az 1., 2. vagy 3. szint nyelvére, majd értelmeztetjük a megfelelő virtuális vagy valódi géppel. A fordítást végző programot **assemblernek** nevezzük.

A 5. szint nyelveit az alkalmazási feladatokat megoldó programozóknak tervezik. Az ilyen nyelveket szokták **magas szintű nyelveknek** nevezni, és több száz van belőlük. Néhány az ismertebbek közül: C, C++, Java, LISP, Prolog. Az ezeken írt programokat általában a 3. vagy a 4. szint nyelvére fordítják az ún. **fordítóprogramok**, de esetenként találkozunk értelmezőkkel is. A Java-programokat például először egy ISA-szerű nyelvre, Java bájtkódra szokták fordítani, amelyet azután egy értelmező hajt végre.

Néha az 5. szint egy speciális alkalmazási területre, például a szimbolikus matematikára kidolgozott értelmező. Ez a területen felmerülő problémák megoldásához az adatokat és műveleteket olyan formában biztosítja, amelyet a területen jártas szakemberek könnyen megértenek.

Összefoglalásként emlékeztetünk arra, hogy a számítógépeket egymásra épülő szintek sorozataként tervezik. Minden szint egy önálló absztrakciónak felel meg különböző elemekkel és műveletekkel. A számítógépek ilyen tervezése és tanulmányozása során időlegesen eltekinthetünk a lényegtelen részletektől, ezzel egy bonyolult tárgyat könnyebben érthetőre szűkíthetünk.

Egy-egy szint adattípusainak, műveleteinek és szolgáltatásainak összességét a szint **architektúrájának** nevezzük. Az architektúra a szint használója által látható tulajdonságokat foglalja egységbe. A programozó által látható tulajdonságok, mint például, hogy mennyi a rendelkezésre álló memória, az architektúrához tartoznak. A megvalósítás részletei – például, hogy milyen áramköri elemek valósítják meg a memóriát – nem része az architektúrának. A programozó által látható számítógépes rendszerelemek tervezésével a **számítógép-architektúra** foglalkozik. A hétköznapi gyakorlatban a számítógép-architektúra és a számítógépek felépítése lényegében ugyanazt jelentik.

1.1.3. A többszintű számítógépek fejlődése

Ahhoz, hogy a többszintű gépekről némi áttekintést nyújtsunk, röviden megvizsgáljuk történeti fejlődésüket, megmutatjuk, hogy az évek során hogyan nőtt a szintek száma, és hogyan bővültek a szintek szolgáltatásai. A számítógép valódi gépi nyelvén írt programokat (1. szint) a számítógép elektronikus áramkörei (0. szint) közvetlenül végre tudják hajtani, értelmező vagy fordító nem szükséges. Ezek az elektronikus áramkörök, valamint a memória és a bemeneti/kimeneti eszközök alkotják a számítógép **hardverét**. A hardver kézzelfogható dolgokból áll – integrált áramkörök, nyomtatott áramköri kártyák, kábelek, áramforrások, memóriák, nyomtatók –, nem pedig absztrakt fogalmakból, algoritmusokból vagy utasításokból.

A **szoftver** viszont **algoritmusok**ból (részletes előírás arra vonatkozóan, hogy hogyan kell valamit végrehajtani) és azok számítógépes reprezentációiból, azaz programokból áll. Programokat tárolhatunk merev- vagy hajlékonylemezeken, CD-ROM-on vagy más adathordozón, de a szoftver lényege a programot alkotó utasítássorozat, nem pedig a fizikai adathordozó, amelyen tároljuk.

A legelső számítógépek esetében kristálytiszta volt a hardver és a szoftver közötti határ. Időközben azonban ez egyre inkább elhomályosodott, nem kismértékben éppen azért, mert a számítógépek fejlődésével a szintek szaporodtak, eltűntek, összefonódtak. Ma már nehéz is a megkülönböztetés (Vahid, 2003). Könyvünk egy lényeges nézőpontja éppen az, hogy

a hardver és a szoftver logikailag ekvivalens.

Bármely szoftverrel végrehajtható művelet beépíthető közvetlenül a hardverbe is, de ez csak azután ajánlatos, ha megfelelően megismertük. Karen Panetta Lentz mondta: „A hardver megkövesedett szoftver." Természetesen fordítva is igaz: bármely hardverrel végrehajtható utasítás szoftverrel is szimulálható. Annak eldöntésében, hogy mely függvényeket valósítsuk meg hardverben és melyeket szoftverben, az ár, a sebesség, a megbízhatóság és a változtatás gyakoriságának szempontjai kapnak szerepet. Kevés szigorú szabályt találunk arra nézve, hogy X miért hardverben és Y miért szoftverben valósítandó meg. Döntéseinket a technológia és a számítógépek felhasználásának fejlődése befolyásolja.

A mikroprogramozás feltalálása

Az 1940-es években az első digitális számítógépek még csak kétszintűek voltak: az ISA-szint, amelyen a programokat írták, és a digitális logika szintje, amely e programokat végrehajtotta. A digitális logika szintjének áramkörei bonyolultak, nehezen áttekinthetők és megépíthetők, valamint megbízhatatlanok voltak.

Maurice Wilkes, a University of Cambridge kutatója, 1951-ben azt javasolta, hogy a hardver jelentős egyszerűsítésére háromszintű számítógépet tervezzenek (Wilkes, 1951). A gépnek beépített, megváltoztathatatlan értelmezője lenne (a mikroprogram), amelynek a feladata az ISA-szintű programok értelmezéssel történő végrehajtása. Mivel így a hardvernek csak az igen szűk utasításkészletű mikroprogramot kellene végrehajtania a jóval bővebb utasításkészletű ISA-szintű programok helyett, sokkal kevesebb elektronikus áramkörre lenne szükség. Ez az egyszerűsítés a vákuumcsöves elektronikus áramkörök korában a csövek számának csökkentését, ezzel a megbízhatóság növelését (azaz a napi összeomlások számának csökkenését) ígérte.

Néhány ilyen háromszintű gépet meg is építettek az 1950-es, és még többet az 1960-as években. 1970-re már az volt az uralkodó elv, hogy az ISA-szint értelmezése mikroprogrammal történjen, ne pedig közvetlenül elektronikus megvalósítással. Minden akkori komolyabb számítógép így működött.

Az operációs rendszer feltalálása

Ezekben a korai években a legtöbb számítógép „aki kapja, marja" elven üzemelt, ami azt jelentette, hogy a gépet a programozó személyesen kezelte. Minden gép mellett elhelyeztek egy időbeosztó lapot, és a programozó, ha programját futtatni akarta, lefoglalt egy intervallumot, mondjuk szerda reggel 3-tól 5-ig (számos programozó szeretett a gépterem csendes időszakaiban dolgozni). Amikor eljött az idő, a programozó – egyik kezében egy láda 80 oszlopos lyukkártyával (az akkori bemeneti adathordozóval), másik kezében egy hegyes ceruzával – belépett a gépterembe. Érkezését követően udvariasan az ajtó felé tessékelte az előtte ott dolgozó programozót, majd átvette a számítógépet.

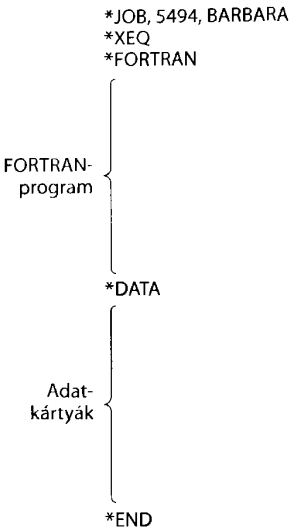
Ha FORTRAN-programot akart futtatni, akkor a következő lépések megtételére kényszerült:

1. Átszaladt a programkönyvtárakat tároló szekrényhez, kivette a FORTRAN-fordító feliratú nagy zöld ládát, tartalmát a kártyaolvasóba helyezte, és megnyomta az indítógombot.
2. FORTRAN-programját a kártyaolvasóba helyezte, és megnyomta a folytatás gombot, mire programját a gép beolvasta.
3. Amikor a számítógép megállt, még egyszer beolvastatta a FORTRAN-programot. Némelyik fordítónak ugyan csak egyszer kellett végigolvasnia a bemenetet, többségüknek azonban kétszer vagy többször. Minden olvasási menetben a vastag kártyaköteget újra kellett olvasni.
4. Végül a fordítás a végéhez közeledett. A vége felé a programozó gyakran idegeskedett, hiszen ha a fordító programhibát talált, ki kellett javítania, és az egész eljárást meg kellett ismételnie. Ha nem volt hiba, a fordító az elkészült gépi nyelvű programot lyukkártyára lyukasztotta.
5. A programozó ezután a gépi nyelvű programot és az eljáráskönyvtár kötegét a kártyaolvasóba helyezte, és mindkettőt beolvastatta.
6. A program végrehajtása megkezdődött. A legtöbb esetben a program nem működött, menetközben valahol váratlanul leállt. Általában a programozó babrált egy kicsit a vezérlőpult kapcsolóival és nézegette a jelzőlámpákat. Ha szerencséje volt, kitalálta mi a baj, kijavította a hibát, visszaszaladt a nagy zöld FORTRAN-fordítót tartalmazó szekrényhez, és kezdte az egészet elölről. Ha nem volt szerencséje, kinyomtatta a memória tartalmát, és ezt a **memóriamásolat**ot (**core dump**) hazavitte tanulmányozni.

Kis különbségekkel évekig ez volt az általános eljárás sok számítóközpontban. A programozónak meg kellett tanulnia, hogyan működik a számítógép, és mit kell tennie, ha leáll, ami elég gyakran előfordult. A gép gyakran csak várakozott, amíg az emberek a kártyákat a gépteremben ide-oda hordozgatták, vagy azon törték a fejüket, miért is nem működik a programjuk rendesen.

1960 körül a gépkezelő feladatának automatizálásával megkísérelték az elvesztegetett idő csökkentését. Az **operációs rendszer**nek nevezett programot állandóan a számítógépben tárolták. A programozó vezérlőkártyákat csatolt a programjához,

ezt olvasta be és hajtotta végre az operációs rendszer. Az FMS-nek (FORTRAN Monitor System, IBM 709), az egyik első, széles körben elterjedt operációs rendszernek szóló egyszerű kártyacsomag felépítését mutatja be az 1.3. ábra.



1.3. ábra. *Egy egyszerű feladat az FMS operációs rendszeren*

Az operációs rendszer beolvasta a *JOB kártyát, és a rajta lévő információt könyvelési célokra használta. (A csillagot a vezérlőkártyák megkülönböztetésére használták, így azok nem keveredtek össze a program- és az adatkártyákkal.) Később beolvasta a *FORTRAN kártyát, amely a FORTRAN-fordítónak mágnesszalagról való betöltését eredményezte. A fordító ezután beolvasta és lefordította a FORTRAN-programot. A fordítás befejeztével a vezérlés visszakerült az operációs rendszerhez, és megtörtént a *DATA kártya beolvasása. Ez a lefordított program végrehajtására utasított a *DATA kártyát követő kártyák tartalmának adatként való használatával.

Bár az operációs rendszert elsősorban a gépkezelő (operátor) munkájának automatizálására tervezték (ebből származik a neve is), egyben egy új virtuális gép fejlesztésének irányába tett első lépés is volt. A *FORTRAN kártyát egy virtuális „fordítsd le a programot" utasításnak, a *DATA kártyát pedig egy virtuális „hajtsd végre a programot" utasításnak is tekinthetjük. Ez a csupán két utasítást tartalmazó szint nem tekinthető igazán önálló szintnek, de nyitás volt afelé.

A következő években az operációs rendszerek egyre kifinomultabbakká váltak. Új utasítások, szolgáltatások és tulajdonságok kerültek az ISA-szint fölé, míg végül egy valódi új szint kezdett körvonalazódni. Az új szint utasításainak egy része megegyezett az ISA-szintű utasításokkal, mások viszont – különösen a bemeneti/kimeneti utasítások – ezektől alapjaiban különböztek. Az új utasításokat gyakran

operációs rendszeri makroutasításoknak vagy **felügyelő (supervisor) hívások**nak nevezték. Ma **rendszerhívás** az elfogadott nevük.

Az operációs rendszerek más tekintetben is fejlődtek. A korai változatok kártyakötegeket olvastak, és sornyomtatóra nyomtattak. Ezek voltak a **kötegelt rendszerek**. Általában több óra telt el a programok leadása és az eredmények kézhezvétele között. Ilyen körülmények között a programfejlesztés nehézkes volt.

Az 1960-as évek elején a Dartmouth College, az M. I. T. és más intézetek kutatói olyan operációs rendszereket fejlesztettek ki, amelyek biztosították a programozónak (akár többnek is egyidejűleg) a számítógéppel való közvetlen kommunikációt. A központi számítógépet ezeken a rendszereken telefonvonalak kötötték össze a távoli terminálokkal. A számítógépet a sok felhasználó megosztva használta. A hivatalokban, az otthoni garázsokban és bárhol, ahol terminált helyeztek el, a programozók begépelhették programjaikat, és majdnem azonnal visszakaphatták a futtatásuk eredményeit. Ezeket a rendszereket nevezzük **időosztásos rendszer**eknek.

Az operációs rendszerekből minket inkább a 3. szinten megvalósított, és az ISA-szinten nem létező utasítások és tulajdonságok érdekelnek, mint az időosztás kérdései. Bár külön nem hangsúlyozzuk, legyünk tudatában annak, hogy az operációs rendszer jóval több, mint az ISA-szinthez hozzátett tulajdonságok megvalósítása.

A szolgáltatások átterelése a mikroprogram szintjére

A mikroprogramozás széles körű elterjedésének idejére (1970-re) a tervezők észrevették, hogy új utasításokat a mikroprogram bővítésével is létre tudnak hozni. Más szóval a „hardvert" bővíteni lehet (új gépi utasításokkal) programozással. Ez a felismerés a gépi utasításrendszerek szinte robbanásszerű fejlődéséhez vezetett. A tervezők versenyeztek az egyre bővebb és használhatóbb utasításrendszerek előállításában. Az utasítások jó része nem volt nélkülözhetetlen abban az értelemben, hogy a meglévő utasításokkal is megvalósíthatók voltak, de sebességben többnyire meghaladták a létező utasítások sorozatával elérhető sebességet. Sok gépen például létezett az INC (INCrement) utasítás, amely eggyel növelt egy számot. Ezeken a gépeken megvolt az általános ADD utasítás is, ezért annak az 1 hozzáadására specializált változata nem volt nélkülözhetetlen (ahogy nyilvánvalóan a 720 hozzáadására specializált sem). Az INC azonban megmaradt, mert általában kicsit gyorsabb volt, mint az ADD.

Ugyanebből a megfontolásból nagyon sok más utasítást is felvettek a mikroprogramba. Ezek között gyakran a következők szerepeltek:

1. egészek szorzásának és osztásának műveletei;
2. lebegőpontos aritmetika;
3. eljárást hívó és eljárásból visszatérő utasítások;
4. ciklusokat gyorsító utasítások;
5. karaktersorozatokat kezelő függvények.

Amint a tervezők észrevették, hogy milyen egyszerű az új utasítások felvétele, kezdtek új tulajdonságokat is beépíteni mikroprogramjaikba. Néhány ezek közül:

1. Tömbökkel való számításokat felgyorsító tulajdonságok (indexelés, indirekt címzés).
2. A programok memóriában való áthelyezésének lehetősége futás közben (relokáció).
3. A megszakítások rendszere, amelyen keresztül a számítógép jelzést kaphat a bemeneti és kimeneti műveletek befejeződéséről.
4. Képesség arra, hogy néhány utasítással felfüggesszük az egyik, és folytassuk a másik programot (processzusátkapcsolás).
5. Speciális utasítások hang-, video- és multimédia-fájlok feldolgozására.

Számos egyéb tulajdonságot és szolgáltatást építettek be az évek során, többnyire bizonyos speciális tevékenységek felgyorsítására.

A mikroprogramozás száműzetése

A mikroprogramozás aranykorában (az 1960-as és 1970-es években) a mikroprogramok „meghíztak". Ahogy egyre többet és többet foglaltak magukba, úgy lettek egyre lassabbak és lassabbak. Végül néhány kutató felismerte, hogy mikroprogramozás nélkül, csupán az utasításkészlet radikális csökkentésével és a megmaradó utasítások közvetlen végrehajtásával (azaz az adatúthardver vezérlésével) a gépek felgyorsíthatók. Bizonyos értelemben a számítógép-tervezés egy teljes kört futott be, és visszatért arra az állapotra, amelyen még azelőtt volt, hogy Wilkes felfedezte a mikroprogramozást.

A kerék azonban tovább forog. A Java-programokat először általában egy köztes nyelvre (Java bájtkódra) fordítják, majd a Java bájtkódot interpreter hajtja végre.

Az eddigiekből kitűnhet, hogy a hardver és a szoftver közötti határ önkényes és állandóan változó. Ami ma még szoftver, holnapra hardverré válhat, és fordítva. Képlékeny a határ a különböző szintek között is. A programozó szempontjából közömbös, hogy az utasításokat (talán a sebességet kivéve) hogyan valósítják meg. Az ISA-szinten programozó személy a szorzó utasítást úgy használhatja, mintha hardverutasítás lenne, nem kell azzal foglalkoznia, sőt tudnia sem kell róla, hogy az tényleg hardverutasítás-e. Ami az egyiknek hardver, az a másiknak szoftver. Később még visszatérünk ezekre a kérdésekre.

1.2. Mérföldkövek a számítógépek felépítésében

A modern számítógépek fejlődése során több száz fajta számítógépet terveztek és építettek meg. Többségük már rég feledésbe merült, néhányuk azonban jelentős hatással volt a mai elgondolásokra. Ebben a fejezetben felvázolunk néhány jelentős régebbi fejlesztést, amiből jobban megismerhetjük, hogy hogyan jutottunk odáig, ahol most tartunk. Mondanunk sem kell, hogy a fejezet csak a fontosabb dolgokat érinti, és sok minden marad említés nélkül. Az 1.4. ábra a fejezetben tárgyalt mérföldkőnek tekinthető gépeket sorolja fel. A számítógépek korszakát megalapozó személyekről Slater könyvében (Slater, 1987) kitűnő kiegészítő történeti anyagot találunk. Ugyanezekről Louis Fabian Bachrach csodálatos színes fényképeivel illusztrált rövid életrajzokat közöl Morgan albuma (Morgan, 1997).

Év	Számítógép neve	Megépítő	Megjegyzés
1834	Analitikus gép	Babbage	Az első kísérlet digitális számítógép megépítésére.
1936	Z1	Zuse	Az első jelfogós számológép.
1943	COLOSSUS	Brit kormány	Az első elektronikus számítógép.
1944	Mark I	Aiken	Az első amerikai általános célú számítógép.
1946	ENIAC I	Eckert/Mauchley	A mai számítógépek történetének kezdete.
1949	EDSAC	Wilkes	Az első tárolt programú számítógép.
1951	Whirlwind I	M. I. T.	Az első valós idejű számítógép.
1952	IAS	Neumann János	A legtöbb mai gépnél is ezt a felépítést alkalmazzák.
1960	PDP-1	DEC	Az első miniszámítógép (50 eladott példány).
1961	1401	IBM	A rendkívül népszerű vállalati kisgép.
1962	7094	IBM	Az 1960-as évek elején a tudományos számítások uralkodó géptípusa.
1963	B5000	Burroughs	Az első magas szintű nyelvre tervezett gép.
1964	360	IBM	Az első számítógépcsaládként tervezett terméksorozat.
1964	6600	CDC	Az első tudományos szuperszámítógép.
1965	PDP-8	DEC	Az első tömegcikk miniszámítógépekből (50 000 eladott példány).
1970	PDP-11	DEC	Az 1970-es éveket uraló miniszámítógép.
1974	8080	Intel	Az első általános célú, egylapkás 8 bites számítógép.
1974	CRAY-1	Cray	Az első vektoros szuperszámítógép.
1978	VAX	DEC	Az első 32 bites szuper-miniszámítógép.
1981	IBM PC	IBM	A személyi számítógépek korszakának elindítója.
1981	Osborne-1	Osborne	Az első hordozható számítógép.
1983	Lisa	Apple	Az első grafikus felhasználói felülettel rendelkező személyi számítógép.
1985	386	Intel	A Pentium vonal első 32 bites előfutára.
1985	MIPS	MIPS	Az első piacra dobott RISC-munkaállomás.
1987	SPARC	Sun	Az első SPARC-alapú RISC-munkaállomás.
1990	RS6000	IBM	Az első szuperskaláris gép.
1992	Alpha	DEC	Az első 64 bites személyi számítógép.
1993	Newton	Apple	Az első kézi számítógép.

1.4. ábra. *Mérföldkövek a modern digitális számítógépek fejlődéstörténetéből*

1.2.1. Nulladik generáció: mechanikus számológépek (1642–1945)

Blaise Pascal (1623–1662) francia tudós volt az első, aki működő számológépet épített, és akinek tiszteletére a Pascal programozási nyelv a nevét kapta. Az 1642-ben épített géppel az akkor alig 19 éves Pascal apjának, a francia kormány adószedőjének akart segíteni. A teljesen mechanikus, fogaskerekes szerkezetet kézi forgatókarral kellett működtetni.

Pascal gépe még csak összeadni és kivonni tudott, de alig harminc évvel később a nagy német matematikus, Gottfried Wilhelm von Leibniz báró (1646–1716) megépítette ugyancsak mechanikus, szorozni és osztani is képes gépét. Leibniz három évszázaddal ezelőtt tulajdonképpen a mai négyműveletes zsebkalkulátoroknak megfelelő gépet szerkesztett meg.

Semmi említésre méltó nem történt 150 évig. Ekkor a University of Cambridge matematikaprofesszora, a sebességmérő feltalálója, Charles Babbage (1792–1871) megtervezte és megépítette **differenciagépét**. Babbage ezt a szintén mechanikus eszközt, mely Pascal gépéhez hasonlóan csak összeadni és kivonni tudott, hajózási navigációhoz használatos számtáblák összeállítására szerkesztette. Az egész szerkezetet egyetlen algoritmusra, a polinomokra alkalmazott véges differenciák módszerének végrehajtására tervezte. A differenciagép legérdekesebb tulajdonsága az volt, ahogyan az eredményeket kiadta: kimenő adatait rézbevonatú lemezbe lyukasztotta acél formanyomóval, előrevetítve a későbbi lyukkártyához, CD-ROM-hoz hasonló, egyszer írható adathordozók világát.

Bár a differenciagép igen jól működött, Babbage gyorsan ráunt a csak egyetlen algoritmust futtatni képes gépre. Egyre több idejét töltötte azzal – és családja vagyonát (nem szólva a 17000 fontos kormánytámogatásról) költötte arra –, hogy az újabb, ún. **analitikus gépet** megtervezze és megépítse. Analitikus gépének négy egysége volt: a tároló (memória), a malom (számolóegység), a bemeneti rész (lyukkártyaolvasó) és a kimeneti rész (lyukkártya és nyomtatott papír). A tároló 1000 darab, egyenként 50 decimális jegyű szóból állt, melyek változókat és eredményeket tartalmazhattak. A malom az összeadás, kivonás, szorzás és osztás műveletére volt képes, az operandusokat a tárolóból vette, az eredményt pedig a tárolóba helyezte. A differenciagéphez hasonlóan ez is teljesen mechanikus volt.

Az analitikus gép nagy előnye az volt, hogy általános célú volt. Lyukkártyáról utasításokat olvasott és ezeket hajtotta végre. Valamely utasítások hatására a gép kiolvasott két számot a tárolóból, hogy továbbítsa őket a malomba, majd elvégzett rajtuk egy műveletet (például az összeadást), és az eredményt visszaküldte a tárolóba. Más utasítások megvizsgáltak egy számot, és feltételes elágazást hajtottak végre attól függően, hogy a szám pozitív vagy negatív volt-e. A bemenő kártyákra más-más programot lyukasztva az analitikus géppel más-más számítást lehetett elvégeztetni, amire a differenciagép még nem volt képes.

Mivel az analitikus gép egy egyszerű assembly nyelven volt programozható, szoftvert kellett rá írni. A szoftver előállításával Babbage Ada Augusta Lovelace-t, Lord Byron, a híres brit költő leányát bízta meg. Így lett Ada Lovelace a világ első programozója. Az Ada® programozási nyelvet az ő tiszteletére nevezték el.

Sok mai tervezőhöz hasonlóan sajnos Babbage sem tudta hardverének tökéletes működőképességét ellenőrizni. Ezerszámra kellettek a megfelelő pontossággal

előállított fogak, kerekek és áttétek, ezt pedig a XIX. század technológiája nem tudta nyújtani. Elveivel viszont jóval megelőzte korát, hiszen a számítógépek még napjainkban is az analitikus géphez hasonló felépítésűek. Nyugodtan kijelenthetjük, hogy Babbage a modern digitális számítógépek (nagy)apja.

A következő jelentősebb fejlesztésre az 1930-as évek végén került sor, amikor Konrad Zuse német mérnökhallgató elektromágneses jelfogókból épített egy sorozat automata számológépet. Állami támogatást a háború kezdetétől nem kapott, mert a kormányzati hivatalnokok úgy gondolták, a háborút olyan gyorsan megnyerik, hogy addigra a gépek el sem készülhetnek. Zuse nem ismerte Babbage eredményeit, a szövetségesek berlini bombázásai alatt (1944-ben) gépei is elpusztultak, ezért munkája semmilyen befolyást nem gyakorolt a későbbi fejlődésre. Ennek ellenére a terület egyik úttörője volt.

Nem sokkal később az Egyesült Államokban John Atanasoff (Iowa State College) és George Stibbitz (Bell Labs) is számológépeket tervezett. Atanasoff gépe korához képest megdöbbentően fejlett volt. Bináris aritmetikát alkalmazott, memóriaként kondenzátorokat használt, a kisülés megakadályozására pedig ezeket periodikusan frissítette, ezt az eljárást „memóriafrissítésnek" nevezte. A mai dinamikus memória- (**DRAM, dinamikus véletlen elérésű memória**) lapkák pontosan ugyanilyen elven működnek. Sajnos a gépe soha nem lett működőképes. Atanasoff Babbage-hez némileg hasonlóan lángész volt, aki végül áldozatul esett a kor nem kielégítő hardvertechnológiájának.

Stibbitz számológépe egyszerűbb volt Atanasoffénál, de működött. 1940-ben nyilvánosan bemutatta a Dartmouth College konferenciáján. A hallgatóság soraiban ült John Mauchley, a University of Pennsylvania ismeretlen fizikaprofesszora. Később a számítástechnika világában Mauchley professzor nevét sokan megtanulták.

Miközben Zuse, Stibbitz és Atanasoff számológépeiket tervezték, egy fiatalember, Howard Aiken rettenetes mennyiségű numerikus számítást volt kénytelen kézzel elvégezni a Harvard Egyetemen folytatott PhD-kutatásai kapcsán. A fokozat megszerzését követően Aiken számára nem volt kétséges, hogy hasonló számításokat géppel kell végeztetni. Tanulmányozta az irodalmat, rátalált Babbage munkáira, és elhatározta, hogy jelfogókból felépíti azt az általános célú számítógépet, amelyet fogaskerekekből Babbage nem tudott megépíteni.

1944-ben a Harvard Egyetemen elkészült Aiken első gépe, a Mark I 72, egyenként 23 decimális jegyű szót tartalmazott, és egy utasítás végrehajtási ideje 6 másodperc volt. A bemeneti és kimeneti adathordozó lyukszalag volt. Amikorra Aiken elkészült a következő, a Mark II géppel, a jelfogós számítógépek kora lejárt. Beköszöntött az elektronika korszaka.

1.2.2. Első generáció: vákuumcsövek (1945–1955)

A II. világháborúban merült fel az igény elektronikus számítógépekre. A háború korai szakaszában a német tengeralattjárók nagy pusztítást végeztek a brit hajókon. A parancsokat a berlini német admiralitás rádión keresztül adta a tengeralattjáróknak, amelyeket a britek le tudtak hallgatni és le is hallgattak. A baj az

volt, hogy az üzeneteket egy **ENIGMA** nevű készülékkel kódolták, amelynek előfutárát éppen Thomas Jefferson, az amatőr felfedező és az Egyesült Államok volt elnöke tervezte.

A brit titkosszolgálatnak még a háború kezdetén sikerült szert tennie egy ENIGMA gépre, amelyet a lengyel titkosszolgálat lopott el a németektől. Egy kódolt üzenet megfejtéséhez azonban rengeteg számítást kellett elvégezni, ráadásul az üzenet lehallgatása után rövid időn belül, különben az eredmény haszontalan volt. Az üzenetek megfejtésére a brit kormány egy szupertitkos laboratóriumot állított fel, ahol megépítették a COLOSSUS nevű elektronikus számítógépet. Alan Turing, a neves brit matematikus is a gép tervezői között volt. 1943-ban a COLOSSUS működőképes lett, de mivel a brit kormány a projekt majd minden részletét 30 évre katonai titoknak minősítette, a COLOSSUS iránya zsákutcává vált. Mégis szóltunk róla, hiszen ez volt a világon az első elektronikus, digitális számítógép.

A háború azonfelül, hogy elpusztította Zuse gépeit és ösztönözte a COLOSSUS megépítését, az Egyesült Államok számítástechnikáját is befolyásolta. A nehéztüzérség számára a hadseregnek irányzéktáblázatokra volt szüksége. Ezeknek a táblázatoknak az előállítására nők százait alkalmazták, akik kézi számológépeken dolgoztak (a férfiaknál megbízhatóbbnak tartották a nőket). Az eljárás nagyon lassú volt, és sok hibával járt.

John Mauchley, aki ismerte Atanasoff és Stibbitz eredményeit, tisztában volt azzal, hogy a hadsereget érdeklik a mechanikus számológépek. Mint meganynyi számítástudós azóta is, összeállított egy pályázatot, amelyben egy elektronikus számítógép megépítésére kért támogatást a hadseregtől. Miután pályázatát 1943-ban elfogadták, Mauchley J. Presper Eckert nevű doktoranduszával együtt belefogott az **ENIAC (Electronic Numerical Integrator And Computer)** elnevezésű elektronikus számítógép megépítésébe. Az ENIAC 18 000 vákuumcsőből és 1500 jelfogóból állt, súlya 30 tonna, áramfogyasztása pedig 140 kilowatt volt. Felépítését tekintve 20, egyenként 10 jegyű decimális szám tárolására alkalmas regisztere volt. (Egy decimális regiszter, amely megadott számú számjeggyel felírható számok tárolására képes, valójában egy kis memória, akárcsak a gépkocsik kilométer-számlálója a lefutott távolság kijelzésére.) Az ENIAC programozása 6000 többállású kapcsoló beállításával és dugaszolóaljzatok tömegeinek – átkötő kábelek valóságos erdejével történő – összekötésével történt.

A gép építését 1946-ig nem sikerült befejezni, és addigra az eredeti célra való felhasználása már értelmét is vesztette. A háború befejeződött, így Mauchley és Eckert engedélyt kapott, hogy egy nyári egyetem keretében ismertessék munkájukat tudós kollégáikkal. Ez a nyári iskola jelentette a nagy digitális számítógépek megépítése iránti, robbanásszerű érdeklődés kezdetét.

A történelmi nevezetességű nyári egyetem után számos kutató kezdett elektronikus számítógép építésébe. Az EDSAC (1949) volt az első működő példány, amelyet Maurice Wilkes épített a Cambridge-i Egyetemen. Ezt követte többek közt a JOHNIAC (Rand Corporation), az ILLIAC (University of Illinois), a MANIAC (Los Alamos Laboratory) és a WEIZAC (Weizmann Institute, Izrael).

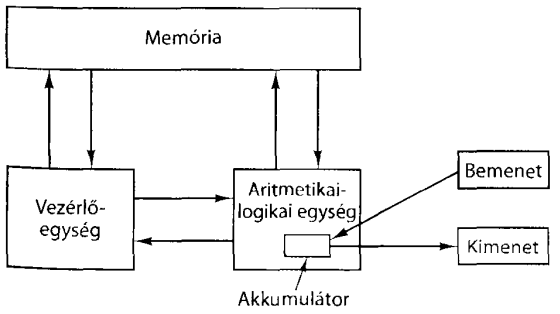
Eckert és Mauchley hamarosan egy újabb gép, az **EDVAC (Electronic Discrete Variabie Automatic Computer)** építésébe kezdett bele. A projekt azonban elvérzett, amikor elhagyták a Pennsylvaniai Egyetemet és megalapították az Eckert-Mauchley Computer Corporationt Philadelphiában (a Szilikon-völgyet még nem találták ki). Cégegyesítések sorozata után ebből a vállalkozásból alakult ki a jelenlegi Unisys Corporation.

Egy kis jogi kitérő. Eckert és Mauchley szabadalmi kérvényt nyújtottak be, miszerint ők a digitális számítógép feltalálói. Mai szemmel nézve, nem volna megvetendő dolog egy ilyen szabadalom birtokosának lenni. Évekig tartó pereskedés után a bíróság úgy ítélte, hogy az Eckert–Mauchley szabadalom érvénytelen, a digitális számítógép felfedezője pedig John Atanasoff, bár ő találmányát soha nem szabadalmaztatta.

Miközben Eckert és Mauchley az EDVAC-on dolgoztak, az ENIAC projekt egyik résztvevője, a magyar származású Neumann János a Princeton's Institute of Advanced Studies intézethez csatlakozott, hogy megépítse saját EDVAC-változatát, az **IAS gépet**. Neumann János éppolyan zseni volt, mint Leonardo da Vinci. Számos nyelvet beszélt, otthon volt a fizikai és a matematikai tudományokban, és bármikor tökéletesen fel tudta eleveníteni azt, amit egyszer hallott, látott vagy olvasott. Fejből idézett szó szerint szövegeket olyan könyvekből, amelyeket évekkel korábban olvasott. Amikor a számítógépek érdekelni kezdték, már a világ legkiemelkedőbb matematikusa volt.

Először is az tűnt fel neki, hogy a számítógépeknek a nagyszámú kapcsolóval és a kábelekkel történő programozása mennyire lassú, fáradságos és rugalmatlan. Felismerte, hogy a program digitális formában ugyanúgy tárolható a számítógép memóriájában, mint az adatok. Azt is látta, hogy az a szerencsétlen soros decimális aritmetika, amelyet az ENIAC is használt – minden egyes számjegy reprezentálására 10 vákuumcsövet pazarolva (1 bekapcsolt, 9 kikapcsolt állapotban) –, helyettesíthető párhuzamos bináris aritmetikával, hasonlóan Atanasoff évekkel korábbi felismeréséhez.

Az a tervezet, amelyet először leírt, ma **Neumann-gépként** ismert. Ezt valósította meg az EDSAC, az első tárolt programú számítógép, és ez még ma, több mint fél évszázaddal később is majdnem minden digitális számítógép alapelve. A terv és az IAS gép, amelyet Herman Goldstine-nel együtt épített, olyan mérhetetlen hatást gyakorolt, hogy érdemes röviden ismertetnünk. Bár ezt a tervezetet mindig is



1.5. ábra. *Az eredeti Neumann-gép*

Neumann János nevével fémjelzik, Goldstine és mások is közreműködtek. Az architektúra vázlatát az 1.5. ábrán láthatjuk.

A Neumann-gépnek öt építőköve volt: a memória, az aritmetikai-logikai egység, a vezérlőegység, valamint a bemeneti és kimeneti eszközök. A memória 4096 szóból, szavanként 40 0 vagy 1 értékű bitből állt. Minden szó vagy két 20 bites utasítást, vagy egy 40 bites előjeles számot tárolt. Az utasításokban 8 bit az utasítás típusát közölte, 12 bit pedig a 4096 memóriaszó egyikét azonosította. Az aritmetikai-logikai egység és a vezérlőegység együttesen alkották a számítógép „agyát". A modern számítógépekben ezeket egyetlen lapkára integrálják, melyet **CPU**-nak **(Central Processing Unit, központi feldolgozó egység)** nevezünk.

Az aritmetikai-logikai egységben volt egy speciális 40 bites belső regiszter, az **akkumulátor**. Egy tipikus utasítás az akkumulátor tartalmához hozzáadta egy memóriaszó tartalmát, vagy az akkumulátor tartalmát a memóriába tárolta. Lebegőpontos aritmetika nem volt a gépben, mert Neumann úgy gondolta, hogy minden valamirevaló matematikusnak fejben tudnia kell követni a tizedesvessző (valójában a kettedes vessző) helyét.

Nagyjából ugyanakkor, amikor Neumann az IAS gépet építette, az M. I. T. kutatói is építették a magukét. Míg az IAS, az ENIAC és a hasonló gépek szavai hosszúak voltak, hiszen komoly numerikus számolgatásokra szánták őket, addig az M. I. T. gépe, a Whirlwind I 16 bites szavakból épült, és folyamatirányításra tervezték. Ez a projekt vezetett Jay Forrester felfedezéséhez, a mágnesgyűrűs memóriához és tulajdonképpen az első kereskedelmi miniszámítógép megszületéséhez.

Ezen történések közben az IBM egy kis cég volt, amely kártyalyukasztók és mechanikus kártyarendező gépek gyártására szakosodott. Bár az IBM adott némi pénzügyi támogatást Aikennek, nem nagyon törődött a számítógépekkel egészen addig, amíg 1953-ban el nem készítette a saját 701-esét – jóval azután, hogy Eckert és Mauchley cége piacvezetővé vált UNIVAC gépével. A 701-esnek 2048 36 bites szava volt, szavanként két utasítással. Ez volt a tudományos számításokat végző gépek sorozatának első tagja, amely egy évtizeden belül az ipar meghatározó gépévé vált. Három évvel később jött ki a 704-es, eredetileg 4096 szavas mágnesgyűrűs memóriával, 36 bites utasításokkal és újításként lebegőpontos aritmetikával. Az IBM utolsó vákuumcsöves gépét, a 709-est – mely alapvetően egy megizmosított 704-es – 1958-ban kezdte gyártani.

1.2.3. Második generáció: tranzisztorok (1955–1965)

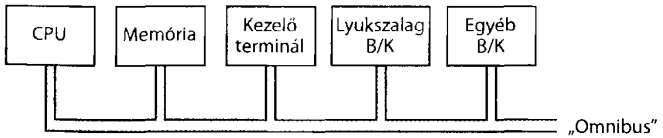
A tranzisztorokat 1948-ban John Bardeen, Walter Brattain és William Shockley, a Bell Labs munkatársai találták fel, amiért 1956-ban elnyerték a fizikai Nobel-díjat. Tíz éven belül a tranzisztorok forradalmasították a számítógépeket, és az 1950-es évek végére a vákuumcsöves számítógépek elavultak. Az első tranzisztoros számítógép az M. I. T. Lincoln Laboratóriumában épült meg, mely a Whirlwind I mintájára 16 bites volt. A **TX–0** (Transistored eXperimental computer 0) névre hallgató gépet alapvetően a TX–2, egy sokkal különlegesebb gép tesztelésére szánták.

Bár a TX–2-t sose becsülték sokra, Kenneth Olsen, a laboratórium egyik mérnöke a TX–0-hoz nagyon hasonló, kereskedelmi forgalmazásra szánt gép gyártására 1957-ben megalapította a Digital Equipment Corporation (DEC) céget. Ez a PDP–1 nevű gép azonban csak négy év múlva jelent meg, a késedelem legfőbb oka pedig az volt, hogy a DEC-et hivatalosan alapító befektetők úgy gondolták, hogy a számítógépekre nincs kereslet. Végül is T. J. Watson, az IBM korábbi elnöke egyszer azt mondta, hogy a számítógépek piaca a világon 4-5 darabra tehető. Ezért a DEC főleg kis áramköri kártyákat gyártott.

A végül is 1961-ben elkészült PDP–1 4096 18 bites szót tartalmazott, és másodpercenként 200 000 utasítást tudott végrehajtani. Ez a teljesítmény fele volt az IBM 7090-esének, amely a 709-es tranzisztoros változata, és akkor a világ leggyorsabb számítógépének számított. A PDP–1 120 000 dollárba került, míg a 7090-es több millióba. A DEC több tucat PDP–1-et adott el, és ezzel megszületett a miniszámítógép-ipar.

Az egyik első PDP–1-et az M. I. T. kapta, ahol azonnal felkeltette néhány bimbózó fiatal tehetség érdeklődését. A PDP–1 egyik újdonsága egy képernyős megjelenítő volt, amelyen az 512 × 512 felbontásban bárhová lehetett pontot kirajzolni. Hamarosan a hallgatók a PDP–1-re már űrháborús játékot programoztak, megajándékozva ezzel a világot az első videojátékkal.

A DEC néhány évvel később a PDP–1-nél sokkal olcsóbb (16 000 dolláros) 12 bites PDP–8-at kezdte gyártani. A PDP–8 nagy újdonsága az 1.6. ábra szerinti egyetlen sín („omnibus"). A **sín** egy többszörös vezeték, amellyel a számítógép különböző egységeit kapcsolják össze. Ez a megoldás jelentősen eltér az IAS gép memóriacentrikus felépítésétől, és azóta majdnem minden kisszámítógépben ezt alkalmazzák. A DEC mintegy 50 000 PDP–8-at adott el, és ezzel a miniszámítógép-kereskedelemben vezető pozíciót szerzett.



1.6. ábra. *A PDP–8 „omnibus"*

Már említettük, hogy eközben az IBM válasza a tranzisztorok megjelenésére a 709-es tranzisztoros változatának, a 7090-esnek (később a 7094-esnek) a megépítése volt. A 7094-es ciklusideje 2 mikroszekundum, 32 536 szavas, szavanként 36 bites mágnesgyűrűs memóriával. A 7090 és a 7094 zárták az ENIAC típusú gépek sorát, a tudományos számításokra való alkalmazásokat azonban még évekig ezek uralták az 1960-as években.

Miközben az IBM a tudományos számítások területén elsöprő erőre tett szert a 7094-essel, óriási összegeket zsebelt be egy kis üzleti alkalmazásokra való gép, az 1401-es eladásaiból. A gép a 7094-esét közelítő sebességgel, de árának töredékéért volt képes mágnesszalagokat olvasni, írni, lyukkártyát olvasni, lyukasztani és nyomtatni. Tudományos számításokra teljesen alkalmatlan, de üzleti adatok tárolására kiváló volt.

Az 1401-es szokatlan volt, mert nem voltak sem regiszterei, sem rögzített szóhossza. Memóriája 4000 8 bites bájtból állt, bár a későbbi modellek támogatták akár – az akkor meghökkentőnek számító – 16000 bájtot is. Minden bájt egy 6 bites karaktert, egy segéd- és egy szavak végét jelző bitet tárolt. A MOVE utasításnak például volt egy forrás- és egy célcím operandusa, és a forráscímtől kezdődően mindaddig, amíg a szóvéget jelző biten 1-et nem talált, a bájtok tartalmát a célcímtől kezdődő helyre másolta.

1964-ben egy apró új cég, a Control Data Corporation (CDC) piacra dobta a hatalmas 7094-esnél és minden egyéb akkor létező számítógépnél majdnem egy nagyságrenddel gyorsabb gépét, a 6600-ast. A numerikus számításokat végzők első látásra beleszerettek, és a CDC elindult a siker felé. Sebességének titka és a 7094-essel szembeni fölénye a CPU nagyfokú párhuzamos felépítésében rejlett. Egymással párhuzamos működésre képes funkcionális egységei voltak külön-külön az összeadásra, a szorzásra, az osztásra. Kis munkával egyidejűleg akár 10 utasítás is végrehajtható volt, bár ahhoz, hogy a legtöbbet kihozzák belőle, körültekintően kellett programozni.

Ez még nem minden, mert a Hófehérke és a hét törpe mintájára a 6600-ba több kisebb segédszámítógépet is beépítettek. Ennek köszönhetően a CPU minden idejét a számítások elvégzésére lehetett fordítani, a feladatszervezés, a bemenet/kimenet kezelése a kisszámítógépek feladata lett. Mai szemmel nézve a 6600-as évtizedekkel megelőzte a korát. A modern számítógépek számos lényeges eleme közvetlenül visszavezethető a 6600-asra.

Seymour Cray, a 6600-as tervezője Neumannhoz hasonló, legendás figura. Egész életét az egyre gyorsabb és gyorsabb gépek építésére fordította, olyanokra, amelyeket ma **szuperszámítógépek**nek nevezünk, és amilyen a 6600-as, a 7600-as vagy a Cray–1. Az autóvásárlásra is feltalált egy ma már közismert algoritmust: menj be a legközelebbi autószalonba, mutass rá az ajtóhoz legközelebbi kocsira, és mondd azt, hogy „ezt veszem meg". Ez az algoritmus a lehető legkevesebb időt fordítja lényegtelen dolgokra (mint például az autóvásárlás) azért, hogy a lehető legtöbb idő maradjon a fontos dolgokra (mint például szuperszámítógépek tervezése).

Sok más gép is épült ebben az időben, egyikük azonban kitűnik egy egészen más okból, a Burroughs B5000. A PDP-1, a 7094, a 6600 és a hasonló gépek tervezőit lekötötte a hardver, és annak olcsóvá (DEC) vagy gyorssá (IBM és CDC) tétele. A szoftver egyáltalán nem volt lényeges. A B5000 tervezői más taktikát választottak. Kifejezetten azzal a céllal építették gépüket, hogy azt Algol 60 (a C és Java elődje) nyelven lehessen programozni, és számos, a fordító feladatát segítő tulajdonságot építettek a hardverbe. Megszületett az a felismerés, hogy a szoftver is számít. Sajnos ezt nagyon rövid időn belül el is felejtették.

1.2.4. Harmadik generáció: integrált áramkörök (1965–1980)

Robert Noyce felfedezése (1958), a szilikon integrált áramkör lehetővé tette, hogy egyetlen lapkán több tucat tranzisztort helyezzünk el. Ennek a technikának köszönhetően a tranzisztoros elődöknél kisebb, gyorsabb és olcsóbb számítógépeket építhettek. Ennek a generációnak a jeles képviselőiről írunk ebben a fejezetben.

1964-re az IBM lett a vezető számítógépgyártó, de a két nagyon sikeres gépével, a 7094-essel és az 1401-essel kapcsolatban volt egy komoly probléma: azok a legkisebb mértékben sem voltak kompatibilisek. Az egyik nagy sebességű numerikus számológép volt párhuzamos bináris aritmetikával és 36 bites regiszterekkel, a másik csodálatos bemenet/kimenet szervező, de soros decimális aritmetikával és változó hosszúságú memóriaszavakkal. A vásárló vállalatok többségének mindkettőből volt, de egyáltalán nem örültek, hogy két teljesen különálló programozási részleget kell fenntartaniuk.

A két sorozat lecserélésének idején az IBM merész lépést tett. Egyetlen terméksort vezetett be, a System/360-at, melyet integrált áramköri alapokon terveztek olyanra, hogy mind a tudományos számítások, mind az üzleti alkalmazások céljainak megfeleljen. A sok egyéb újdonság mellett a System/360 sorozat egy családot alkotott mintegy fél tucat különböző méretű és teljesítményű géppel, amelyek mindegyike ugyanazon az assembly nyelven volt programozható. Egy vállalat az 1401-esét a 360 Model 30, a 7094-esét pedig a 360 Model 75 típusra cserélhette. A Model 75 nagyobb és gyorsabb volt (és drágább is), de a sorozat bármelyikére írt szoftver elvileg bármelyik másikon is futhatott. Gyakorlatilag a kis gépre írt szoftver a nagyobbakon gond nélkül futott, de egy kisebbre telepítve a program esetleg nem fért el a memóriában. Ennek ellenére óriási előrelépés volt a 7094-es és a 1401-es közötti állapotokhoz képest. A gépcsaládok gondolata azonnal elterjedt, és néhány éven belül a legtöbb számítógépgyártó már árban és teljesítményben különböző, egyébként azonos gépekből álló sorozatokat kínált. A 360-as család eredeti tagjainak néhány jellemzőjét mutatja az 1.7. ábra. Később újabb modelleket is bevezettek.

Tulajdonság	Model 30	Model 40	Model 50	Model 65
Relatív teljesítmény	1	3,5	10	21
Ciklusidő (ns, a másodperc milliárdod része)	1000	625	500	250
Maximális memória (bájt)	65536	262144	262144	524288
Ciklusonként betöltött bájtok száma	1	2	4	16
Adatcsatornák maximális száma	3	3	4	6

1.7. ábra. *Az IBM 360-as gyártmánycsalád bevezető modelljei*

A 360-as egy másik jelentős fejlesztése volt a **multiprogramozás**, amikor egyidejűleg több program van a számítógép memóriájában, és miközben egyik a bemenet/kimenet befejezésére vár, addig a másik számolhat. Így a CPU kihasználtsága megnőtt.

Ugyancsak a 360-as volt az első gép, amely más gépeket tudott emulálni (szimulálni). A kisebb modellek az 1401-est, a nagyobbak a 7094-est tudták emulálni, így a 360-asokra áttérő vásárlók régi bináris programjaikat minden változtatás nélkül futtathatták. Némelyik modell az 1401-esnél annyival gyorsabban futtatta az 1401-es programjait, hogy sokan sohasem írták át régi programjaikat.

A 360-ason könnyű volt az emuláció, mivel a bevezető és sok későbbi modell is mikroprogramozott volt. Az IBM-nek mindössze három mikroprogramot kellett megírnia, a 360-as saját, az 1401-es és a 7094-es utasításrendszereire. Ez a rugalmasság volt az egyik fő oka a mikroprogramozás elterjedésének.

A 360-as kompromisszumos megoldást hozott a párhuzamos bináris, illetve soros decimális aritmetika dilemmájára is: a bináris aritmetikára 16, egyenként 32 bites regisztere volt, de memóriája az 1401-eséhez hasonlóan bájtszervezésű maradt. Ezen felül, az 1401-eshez hasonlóan, soros utasításokkal is rendelkezett változó hosszúságú adatok memóriában való mozgatására.

Az (akkori mércével) óriási, 2^{24} (16 777 216) bájtos címtartomány volt a 360-as másik figyelemre méltó tulajdonsága. A memória akkori bájtonkénti néhány dolláros ára mellett ennyi memória végtelenül soknak tűnt. Sajnos a 360-as sorozatot követő 370-es, 4300-as, 3080-as, majd 3090-es sorozatok mind ugyanarra az architektúrára épültek. Az 1980-as évek közepére a memóriakorlát komoly problémát jelentett. Az IBM kénytelen volt a kompatibilitást részben feladni, amikor áttért a 2^{32} bájtos memóriaméret 32 bites címzési rendszerére.

Mai ésszel azt kérdezhetnénk, hogy ha a szavak és a regiszterek egyaránt 32 bitesek voltak, akkor a címek miért nem. Abban az időben azonban senki nem tudott elképzelni egy gépet 16 millió bájtnyi memóriával. Olyan lenne az IBM-et hibáztatni ezen előrelátás hiányáért, mint egy mai személyi számítógépeket gyártó céget azért okolni, hogy csak 32 bites címeket használ. Néhány éven belül a személyi számítógépeknek 4 milliárd bájtnál sokkal nagyobb memóriára lesz szükségük, és ekkor a 32 bites címek újra elégtelennek fognak bizonyulni.

A miniszámítógépes világ is nagyot lépett előre a harmadik generáció idejében, amikor a DEC a PDP–8 utódjaként bevezette a 16 bites PDP–11 sorozatát. Sok tekintetben a PDP–11 a 360-as kistestvére volt – amint azt a PDP–1-ről, mint a 7094-es kistestvéréről is elmondhatjuk. A 360-as és a PDP–11 is szavas regiszterekkel, bájtos memóriával rendelkezett, továbbá sorozataik jelentős ár/teljesítmény skálán mozogtak. A PDP–11 – különösen az egyetemeken – kimagaslóan sikeres volt, és hozzásegítette a DEC-et ahhoz, hogy megőrizze vezető helyét a miniszámítógép-gyártásban.

1.2.5. Negyedik generáció: magas integráltságú áramkörök (1980–?)

Az 1980-as évekre a **VLSI (Very Large Scale Integration, nagyon magas fokú integráltság)** technológia először több tízezer, majd százezer, végül pedig több millió tranzisztor elhelyezését tette lehetővé egyetlen lapkán. A fejlődés rövid időn belül kisebb és gyorsabb számítógépek előállításához vezetett. A PDP–1 előtt a vállalatoknak és az egyetemeknek speciális részlegekre, a **számítóközpontok**ra volt szükségük ezeknek a nagy és drága gépeknek az üzemeltetéséhez. A miniszámítógépek megjelenésével már egy részleg vagy tanszék is megvásárolhatta a saját számítógépét. 1980-ra az árak úgy lecsúsztak, hogy már magánszemélyek számára sem volt elérhetetlen a saját számítógép. Beköszöntött a személyi számítógépek kora.

A személyi számítógépeket egészen másként használták, mint a nagy számítógépeket. Szövegszerkesztésre, táblázatkezelésre és egyéb, kifejezetten interaktív alkalmazásokra (mint például játékok), amelyekkel a nagyszámítógépek nehezen boldogultak.

Az első személyi számítógépeket készletként árusították. A készlethez egy nyomtatott áramköri kártya, egy zacskónyi lapka – köztük általában az Intel 8080 –, kábelek, tápegység és esetleg egy 8 inches floppymeghajtó tartozott. A vásárlónak kellett az alkatrészekből a számítógépet összeszerelnie. Szoftver nem tartozott a készlethez. Akinek szüksége volt rá, meg kellett írnia a sajátját. Később Gary Kildall megírta a 8080-asokon népszerűvé vált CP/M operációs rendszert. Valódi, (hajlékony)lemezről futtatott operációs rendszer volt fájlrendszerrel, a felhasználói parancsokat pedig billentyűzetről lehetett bevinni a parancsértelmezőnek (shell, héj).

A korai személyi számítógépek másika az Apple, később az Apple II volt, melyet Steve Jobs és Steve Wozniak abban a nevezetes garázsban tervezett. A gép rendkívül népszerű volt az otthoni felhasználók és az iskolák körében egyaránt, és ezzel az Apple szinte egyik napról a másikra a piac komoly résztvevőjévé vált.

Hosszú megfontolás után – és látva, mit csinál a többi gyártó – végül az IBM, a számítógépipar akkori nagyhatalma is úgy döntött, hogy beszáll a személyiszámítógép-üzletbe. Egy, csak IBM-alkatrészekből összeállított, saját gép tervezése túlságosan sokáig tartott volna, ezért szokatlan döntés született. Philip Estridge, az IBM egyik vezetője kapott megbízást és jelentős pénzügyi támogatást arra, hogy a cég mindenbe beleavatkozó bürokráciájától és annak New York állambeli Armonk székhelyétől távol eső helyre távozzon, és vissza se jöjjön egy működő személyi számítógép nélkül. Estridge a floridai Boca Ratonban állította fel műhelyét, CPU-nak az Intel 8088-ast választotta, és az első IBM Personal Computert kereskedelmi forgalomban kapható alkatrészekből építette meg. 1981-ben kezdték meg a gyártását, és azonnal a történelem legnagyobb darabszámban eladott számítógépévé vált.

Az IBM még egy szokatlan lépést tett, amit azonban később megbánt. Ahelyett, hogy a gép terveit – ahogy általában szokta – titokban tartotta (vagy legalábbis szabadalmakkal védte) volna, egy mindössze 49 dollárért kapható könyvben közzétette a teljes tervet, beleértve az áramköri diagramokat is. Az volt a célja, hogy más cégek számára is lehetővé tegye kiegészítő kártyák gyártását az IBM PC-khez, ezzel is növelve az IBM PC alkalmazhatóságát és népszerűségét. Mivel azonban a tervek közismertek, az alkatrészek pedig a kereskedelemben könnyen beszerezhetők voltak, az IBM szerencsétlenségére számos gyártó ún. IBM PC-**klónok**at kezdett előállítani, többnyire az IBM árainál lényegesen olcsóbban. Ezzel egy teljesen új iparág született.

Más vállalatokat, mint például a Commodore, Apple, Amiga és Atari, akik nem Intel CPU-val gyártották személyi számítógépeiket, összeroppantotta az IBM PC-ipar súlya. Csak néhányuk élte túl, egyes szűk piaci szegmensekben.

Az Apple Macintosh is túlélte, bár csak éppen hogy. A Macintosh-t 1984-ben mutatták be a balsorsú Apple Lisa utódaként, amely az első **grafikus felhasználói felülettel (Graphical User Interface, GUI)** rendelkező számítógép volt. A GUI a manapság oly közkedvelt Windows-felülethez hasonlított. A Lisa elbukott, mert

túl sokba került, az egy évvel később bevezetett olcsóbb Macintosh azonban hatalmas sikert aratott, és sok csodálójából váltott ki túláradó érzelmeket.

A személyi számítógépek e korai piaca vezetett el a hordozható számítógépek addig elképzelhetetlen igényéhez is. Abban az időben a hordozható számítógépnek kb. annyi értelme volt, mint manapság egy hordozható hűtőszekrénynek. Az első valóban hordozható személyi számítógép, az Osborne-1, a maga 11 kilogrammjával inkább volt vonszolható, mint hordozható. Ennek ellenére bebizonyította, hogy lehetséges hordozható számítógépet készíteni. Az Osborne-1 csak mérsékelt kereskedelmi sikert hozott, azonban egy évvel később a Compaq kihozta az első hordozható IBM PC-klónt, és ezzel gyorsan megalapozta vezető helyét a hordozható gépek piacán.

Az első IBM PC az akkor még igen kicsi Microsoft Corporation által készített MS-DOS operációs rendszerrel működött. Ahogy az Intel egyre nagyobb teljesítményű CPU-kat gyártott, az IBM és a Microsoft kifejleszthette az MS-DOS utódját, az OS/2-t, amely az Apple Macintosh grafikus felületéhez hasonlót nyújtott a felhasználónak. Eközben a Microsoft kifejlesztette a saját, MS-DOS felett futó Windows operációs rendszerét is, arra az esetre, ha az OS/2 nem futna be. Rövidre fogva a történetet, az OS/2-nek nem volt sikere, az IBM és a Microsoft nyilvánosan jól összeveszték, a Microsoft pedig sikerre vitte a Windowst. Az, ahogy a kis Intel és a még kisebb Microsoft cégnek sikerült az IBM-et, a világtörténelem egyik legnagyobb, leggazdagabb és legbefolyásosabb vállalatát trónjától megfosztani, vitathatatlanul példaértékű, és világszerte üzleti főiskolák tananyaga.

A 8088-as sikerére építve az Intel egyre nagyobb és gyorsabb változatokat épített. Külön említésre méltó az 1985-ben forgalomba került 386-os, mely lényegében az első Pentiumnak tekinthető. Bár napjaink Pentiumjai sokkal gyorsabbak a 386-osnál, architektúráját tekintve egy mai Pentium is lényegében csak egy felpiszkált 386-os.

Az 1980-as évek közepére az új fejlesztésű, egyszerűbb (és gyorsabb) ún. RISC-architektúra kezdte felváltani a bonyolultabb (CISC-) architektúrákat. Az 1990-es években megjelentek a szuperskaláris CPU-k. Ezek a gépek egy időben több utasítást képesek végrehajtani, gyakran a programban szereplőtől eltérő sorrendben. A CISC, RISC és szuperskaláris fogalmakat a 2. fejezetben vezetjük be, és később részletesen is tárgyaljuk könyvünkben.

1992-ig a személyi számítógépek 8, 16 vagy 32 bitesek voltak. Akkor jelent meg a DEC a forradalmi 64 bites Alphával, egy valódi 64 bites RISC géppel, melynek a teljesítménye messze meghaladta az akkori számítógépekét. Mérsékelt sikere volt, de csaknem egy évtized múltán a 64 bites gépek igazán elterjedtek, többnyire nagy teljesítményű kiszolgálókként (szerverekként).

1.2.6. Ötödik generáció: láthatatlan számítógépek

1981-ben a japán kormány bejelentette, hogy 500 millió dollárral tervezi támogatni a japán vállalatoknak az ötödik generációs számítógépekre irányuló fejlesztéseit, melyek mesterséges intelligencián alapulnának és a „buta" negyedik generációs gépekhez képest hatalmas ugrást jelentenének. Látva, hogy a japán vállalatok átveszik a piacot az ipar több területén, a kameráktól a hangfelvevő és -lejátszó berendezéseken át a televíziókészülékekig, az amerikai és európai számítógépgyártók szinte azonnal kétségbeestek, és követelték az állami támogatást. A nagy beharangozás ellenére a japán ötödik generációs projekt lényegében kudarcot vallott, és csendben elhalt. Bizonyos értelemben olyan volt, mint Babbage analitikus gépe: egy látnoki ötlet, mely messze megelőzte korát, amikor is a megépítéséhez szükséges technológia még nem állt rendelkezésre.

Mindenesetre valami történt, amit akár ötödik generációnak is nevezhetünk, méghozzá váratlan módon: a számítógépek összementek. Az 1993-ban megjelent Apple Newton megmutatta, hogy egy hordozható kazettásmagnónál kisebb számítógép is építhető. A Newton kézírást használt felhasználói bevitelre, ami akadálynak bizonyult, azonban a kategória későbbi tagjai – melyeket ma **PDA**-nak **(Personal Digital Assistant, digitális személyi asszisztens)** nevezünk – javított felhasználói interfésszel rendelkeztek, és nagyon népszerűvé váltak. Sokuk ma majdnem olyan számítási kapacitással rendelkezik, mint a néhány évvel korábbi személyi számítógépek.

Azonban még a PDA-k sem számítanak igazán forradalminak. Ennél is fontosabbak a „láthatatlan" számítógépek, melyeket különféle készülékekbe, órákba, bankkártyákba és számos egyéb eszközbe építenek be (Bechini és társai, 2004). Ezek a processzorok alkalmazások széles választékában teszik lehetővé a funkcionalitás növelését és az árak csökkentését. Vitatható, hogy ezek a lapkák igazi generációt alkotnak-e (hiszen az 1970-es évek óta jelen vannak), viszont forradalmasítják a több ezer készülék és eszköz működését. Már most is nagy hatással vannak a világra, és befolyásuk a következő években gyorsan tovább fog növekedni. Ezek a beágyazott számítógépek abból a szempontból különösek, hogy a hardvert és a szoftvert gyakran együtt tervezik **(codesign)** (Henkel és társai, 2003). Ezekre könyvünkben később még visszatérünk.

Ha az első generációnak az elektroncsöves gépeket (például ENIAC) tekintjük, a második generációnak a tranzisztoros gépeket (például az IBM 7094), a harmadik generációnak a korai integrált áramkörös gépeket (például az IBM 360), a negyedik generációnak pedig a személyi számítógépeket (például az Intel CPU-k), akkor az igazi ötödik generáció inkább szemléletmódváltás, mintsem konkrét új architektúra. A jövőben a számítógépek mindenütt és mindenbe beágyazva, valóban láthatatlanok lesznek. Napi életünk részeként jelen lesznek ajtónyitáskor, lámpagyújtáskor, készpénzfelvételnél és több ezer egyéb dologban. Ezt a néhai Mark Weiser által kidolgozott modellt hívják **mindenütt jelenlévő számítástechnikának** (Weiser, 2002). Ez oly mértékben fogja megváltoztatni a világot, mint egykor az ipari forradalom. Könyvünkben nem tárgyaljuk tovább a témát; további részletekkel kapcsolatban lásd a szakirodalmat (Lyytinen és Yoo, 2002; Saha és Mukherjee, 2003; Sakamura, 2002).

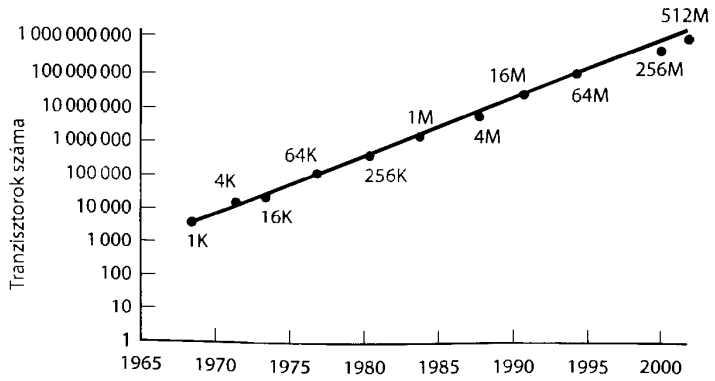
1.3. Számítógép-kiállítás

Az előző részben röviden áttekintettük a számítógépes rendszerek történetét. Most lássuk a jelent, és tekintsünk a jövőbe. Bár napjainkban a személyi számítógépek a legismertebbek, azért másféle gépek is léteznek, amelyekre szintén érdemes egy pillantást vetnünk.

1.3.1. Technológiai és gazdasági mozgatórugók

A számítógépipar olyan iramban fejlődik, mint ahogy egyetlen más iparág sem. Fő mozgatórugója, hogy a gyártók évről évre több és több tranzisztort képesek egyetlen lapkára elhelyezni. A több tranzisztor – amelyek tulajdonképpen apró elektronikus kapcsolók – nagyobb memóriát és hatékonyabb processzort eredményez. Gordon Moore, az Intel egyik alapítója és korábbi elnöke egyszer azzal viccelődött, hogy ha a repülőgép-technológia olyan gyorsan fejlődött volna, mint a számítógép-technológia, egy repülőgép 500 dollárba kerulne, 20 perc alatt kerülné meg a Földet 5 gallon (kb. 18,5 liter) üzemanyagot fogyasztva, és akkora lenne, mint egy cipősdoboz.

Moore egyik előadására készülve vette észre, hogy a memória-áramkörök esetében háromévente jelentek meg új generációk. Minden generáció megnégyszerezte az előző memóriakapacitását, tehát az egy lapkán elhelyezhető tranzisztorok száma állandó ütemben nőtt. Azt jósolta, hogy ez a növekedés még évtizedekig folytatódni fog. Ezt a megligyelést hívjuk **Moore-szabály**nak. Ma Moore szabályát inkább úgy fogalmazzák meg, hogy a tranzisztorok száma 18 havonta megkétszereződik, azaz a növekedés mintegy évi 60 százalék. A memória-áramkörök méreteit és bevezetésük időpontjait az 1.8. ábra mutatja; láthatjuk, hogy Moore szabálya több mint három évtizede teljesül.



1.8. ábra. *Moore szabálya szerint az egy lapkán elhelyezhető tranzisztorok száma évi 60 százalékkal nő. Az ábra bitekben adja meg a memóriaméreteket*

Természetesen Moore szabálya nem törvény, hanem a szilárdtestfizika és a gyártástechnológia előrehaladásának megfigyeléséből származó, és a jövőre nézve is érvényesnek tekintett fejlődési tendencia. Egyes ipari megfigyelők azt várják, hogy Moore szabálya még legalább egy évtizedig érvényben marad. Ekkorra a tranzisztorokat már olyan kevés atomból építik fel, hogy megbízhatatlanná válnak. Másrészről azonban elképzelhető, hogy a kvantumelektronika fejlődése ezt alapvetően megváltoztatja (Oskin és társai, 2002). Más megfigyelők szerint az energiakibocsátás, az áramszivárgás és egyéb hatások már korábban komoly problémákat állítanak elénk (Bose, 2004; Kim és társai, 2003).

A közgazdaság-tudomány Moore szabályát **bűvös kör**ként ismeri. A technológia fejlődése (tranzisztorok, lapkák) jobb és olcsóbb termékeket eredményez. Az alacsonyabb árak új alkalmazásokhoz vezetnek (10 millió dolláros gépekre senki sem készített videojátékokat). Az új alkalmazások új piacokat nyitnak, és ezek kiaknázására új vállalatok születnek. Kialakul a vállalatok közötti verseny, amely pedig még jobb technológiákat követel a győzelem érdekében. Ezzel a kör bezárul.

A technológiai fejlődés másik motorját Nathan első szoftvertörvénye fogalmazza meg (mely Nathan Myhrvoldtól, a Microsoft egyik volt vezetőjétől ered). Eszerint „a szoftver gáz, amely kitölti a rendelkezésére álló teret". Az 1980-as években a troff-hoz hasonló programokkal szerkesztettünk szövegeket (például e könyv angol eredetijét is). A troff csak néhány kilobájt memóriát foglalt. A mai szövegszerkesztőknek megabájtokra van szükségük. Nem kétséges, hogy a jövőbeliek gigabájtokat fognak használni. (A kilo, mega és giga előtagok rendre ezret, milliót és milliárdot jelentenek, de részletekért lásd az 1.5. fejezetet.) A szoftverekbe egyre újabb szolgáltatások épülnek be (mint ahogy a hajótestre rakódnak a kagylók), és ezért egyre gyorsabb processzorokra, nagyobb memóriára és B/K kapacitásra van igény.

Míg a lapkánkénti tranzisztorok száma évenként drasztikusan nőtt, a számítástechnika más ágainak növekedési üteme sem maradt le. Az IBM PC/XT például 1982-ben 10 megabájtos merevlemezzel került piacra. Húsz évvel később a PC/XT utódgépeiben megszokottak voltak a 100 gigabájtos merevlemezek. Ez a 20 év alatti négy nagyságrendnyi növekedés évi 58 százalékos kapacitásnövekedést jelent. A mágneslemezek fejlődésének mérése azonban ennél kicsit bonyolultabb, hiszen a kapacitáson kívül fontos paraméterek például az adatátviteli sebesség, a pozicionálási idő és az ár. Mindazonáltal, akárhogy is nézzük, azt látjuk, hogy az ár/teljesítmény arány 1982 óta évente legalább 50 százalékkal javult. Ez a mágneslemezek teljesítményében bekövetkezett óriási fejlődés és az a tény, hogy a Szilikon-völgyből (Silicon Valley) szállított lemezmennyiség értékben meghaladta a CPU lapkákét, Al Hoaglandet arra késztette, hogy megjegyezze, a hely elnevezése téves: Vas-oxid Völgynek (Iron Oxide Valley) kellene hívni (hiszen ez a mágneslemezek adathordozó rétegének alapja).

Látványos fejlődést mutatnak a telekommunikáció és a számítógép-hálózatok is. Két évtizeden belül a 300 bit/s modemektől eljutottunk az 56 kbit/s analóg modemekig és a 10^{12} bit/s üvegszáloptikás hálózatokig. Az Atlanti-óceánt átszelő száloptikás TAT-12/13 telefonkábel 700 millió dollárba került, és egyidejűleg 300 000 hívást teljesít. Ha ezt tíz évre vetítjük, egy 10 perces nemzetközi beszélgetés költ-

sége 1 cent alatt van. Kísérletek bizonyítják, hogy 100 km feletti távolságok külön erősítők nélkül áthidalhatók 10^{12} bit/s sebességgel működő optikai kommunikációs rendszerekkel. Az internet meredek felfutását már nem is kell csetelnünk.

1.3.2. A számítógépek termékskálája

A Bell Labs korábbi munkatársa, Richard Hamming megfigyelte, hogy egy nagyságrendnyi mennyiségi változás minőségi változáshoz vezet. A nevadai sivatagban 1000 km/óra sebességgel száguldó autó minőségileg is lényegesen különbözik attól, amelyik az autópályán 100 km/órával poroszkál. Hasonlóan a 100 emeletes felhőkarcoló sem csak egy megnyújtott 10 emeletes lakóház. A számítógépek világában ráadásul nem is 10-szeres különbségekről, hanem a három évtized alatt milliószoros növekedésről beszélhetünk.

A Moore-szabály által szolgáltatott fejlődést többféleképpen is alkalmazhatjuk. Egyrészről változatlan áron építhetünk egyre nagyobb teljesítményű számítógépeket. Másrészt pedig ugyanazt a gépet évről évre olcsóbban állíthatjuk elő. A számítógépipar mindkét megközelítést alkalmazta, és így ma számítógépek széles választékát szolgáltatja. A mai gépek egy durva osztályozását adja az 1.9. ábra.

Típus	Ár (dollár)	Példák felhasználási területekre
Eldobható számítógép	0,5	Üdvözlőlapok
Mikrovezérlő	5	Órák, autók, eszközök
Játékgép	50	Házi videojátékok
Személyi számítógép	500	Asztali és hordozható számítógép
Szerver	5000	Hálózati kiszolgáló
Munkaállomás-gyűjtemény (COW)	50 000-500 000	Tanszéki mini-szuperszámítógép
Nagyszámítógép	5 000 000	Banki kötegelt adatfeldolgozás

1.9. ábra. *A számítógépek mai típusválasztéka. Az árak (nagyon durva) becslések*

A következő részben röviden megvizsgáljuk ezeket a kategóriákat és tulajdonságaikat.

1.3.3. Eldobható számítógépek

A legkisebb számítógép egyetlen lapka, amely például képeslapra ragasztva lejátssza a „Boldog szülinapot" vagy egy ehhez hasonló, feledhető kis dalocskákat. A több millió dollárba kerülő nagyszámítógépek mellett felnőtt ember számára az eldobható számítógép úgy hangzik, mintha valaki eldobható repülőgépről beszélne.

Mindenesetre az eldobható számítógépek tartósan jelen vannak. A terület valószínűleg legfontosabb fejlesztése az **RFID (Radio Frequency IDentification, rádiófrekvenciás azonosító)** lapka. Képesek vagyunk néhány centért olyan 0,5 mm-nél vékonyabb, áramforrás nélküli RFID lapkákat gyártani, amelyekben egy apró

rádióadóvevő és egy beépített 128 bites szám van. Ha egy külső antennáról impulzust kapnak, a bejövő rádiójel elegendő energiát szolgáltat ahhoz, hogy a számukat az antennához visszasugározzák. Bár ezek a lapkák nagyon kicsik, jelentőségük egyáltalán nem az.

Kezdjük egy hétköznapi alkalmazással: a vonalkódok eltüntetése az árucikkekről. Kísérleteket végeztek, melyekben az üzletekben az árucikkeket (vonalkódok helyett) a gyártó által elhelyezett RFID lapkákkal látták el. A vásárló kiválasztja a terméket, beteszi a kosarába, és egyszerűen kitolja a boltból, elhaladva a pénztárpult mellett. Az üzlet kijáratánál egy antennával felszerelt leolvasó kiküld egy jelet, amellyel minden egyes terméket megkér, hogy azonosítsa magát, mindezt egy rövid, vezeték nélküli kommunikáción keresztül. A vásárlót szintén azonosítják a bankkártyáján tárolt lapkával. A hónap végén az üzlet tételes számlát küld az ügyfélnek a havi vásárlásairól. Ha a vásárlónak nincs érvényes RFID bankkártyája, megszólal a riasztó. A rendszer amellett, hogy kiküszöböli a pénztárosok szükségességét és a sorban állást, egyben lopásvédelmi rendszer is, hiszen a terméket teljesen haszontalan zsebbe vagy táskába rejteni.

A rendszer egyik érdekes jellemzője, hogy míg a vonalkódok csak az egyes terméktípusokat azonosítják, az egyedi darabokat nem, addig ezt az RFID lapkák a 128 bittel megteszik. Következésképpen, például az üzlet polcán található minden egyes doboz aszpirin különböző RFID kóddal rendelkezik. Eszerint, ha egy gyógyszergyártó azután fedezi fel, hogy egy köteg aszpirinban gyártási hiba történt, miután azt már elszállították, figyelmeztetheti az üzleteket a világ minden táján, hogy a riasztó jelezzen, ha valaki egy olyan csomagot vásárol, amelynek az RFID száma az érintett tartományba esik, még akkor is, ha a vásárlás egy távoli országban hónapokkal később történik. A nem a hibás kötegbe eső aszpirin nem fogja a riasztót megszólaltatni.

Az aszpirines, kekszes és kutyaeledeles csomagok címkézése azonban még csak a kezdet. Miért állnánk meg a kutyaeledel címkézésénél, amikor a kutyát is megjelölhetjük? Háziállat-tulajdonosok már most is arra kérik állatorvosukat, hogy ültessenek be RFID lapkákat kedvenceikbe, hogy az alapján megkereshetők legyenek, ha elvesznek vagy ellopják őket. Az állattenyésztők szintén meg akarják jelöltetni állataikat. A nyilvánvaló következő lépés az, hogy az aggódó szülők a gyermekorvossal RFID lapkákat ültettessenek be gyermekeikbe arra az esetre, ha elvesznének vagy elrabolnák őket. Ha már itt tartunk, miért nem építik be rögtön a kórházban az újszülöttekbe, és akkor elkerülhetők lennének a kórházi kavarodások is. A kormány és a rendőrség minden bizonnyal számos jó okot találna arra, hogy folyamatosan nyomon követhesse az állampolgárokat. Mostanra vélhetően sikerült rávilágítanunk az RFID lapkák „jelentőségére", amire korábban utaltunk.

Az RFID lapkák másik (kevésbé vitatható) alkalmazási területe a járművek nyomkövetése. Amikor RFID lapkákkal ellátott vasúti kocsik sora halad el a leolvasó előtt, a hozzácsatolt számítógép azonnal látja, mely vagonok haladtak át. Ezzel a rendszerrel könnyű az összes vagon helyzetét követni, ami segítség a szállítmányozóknak, az ügyfeleknek és a vasútnak egyaránt. Hasonló összeállítást alkalmazhatunk a kamionokra is. Gépkocsik esetén az ötlet az útdíjak elektronikus beszedésére használható.

A légiforgalmi csomagkezelő rendszereknek és számos egyéb csomagtovábbító rendszernek szintén hasznosak lehetnek az RFID lapkák. A londoni Heathrow repülőtéren tesztelt kísérleti rendszer lehetővé tette az érkező utasoknak, hogy megszabaduljanak csomagjaik cipelésétől. A szolgáltatást igénybe vevő utasok csomagjait RFID lapkákkal látták el, majd a repülőtéren külön útvonalra terelve, egyenesen az utasok szállodájába szállították. Az RFID lapkák további alkalmazási lehetőségei között találjuk az összeszerelő szalagon a festőállomásra érkező gépkocsi színének kiválasztását, az állatok vonulásának tanulmányozását, vagy azt, amikor a ruhák megmondják a villanyvasalónak, hogy milyen hőmérsékletet alkalmazzon. Egyes lapkákat érzékelőkkel is egybeépíthetnek, amikor is az alacsony helyértékű bitek a pillanatnyi hőmérsékletet, nyomást, páratartalmat vagy egyéb környezeti változót tartalmazhatnak.

A fejlett RFID lapkák állandó tárolóval is rendelkezhetnek. Ez a képességük vezetett az Európai Központi Bank (European Central Bank) azon döntéséhez, miszerint az elkövetkező években RFID lapkákat helyeznek az euró bankjegyekbe. A lapkák rögzítenék, hogy merre fordultak meg. Ez nemcsak az euró bankjegyek hamisítását tenné tulajdonképpen lehetetlenné, hanem az emberrablások váltságdíjai, a rablásokból származó zsákmány és a pénzmosáson átesett pénz is sokkal könnyebben nyomon követhetővé, illetve távolról érvényteleníthetővé válna. Ha a pénz már nem névtelen, az lehetne a szokványos rendőrségi eljárás a jövőben, hogy megvizsgálják, a gyanúsított pénze merre járt az utóbbi időben. Kinek lenne szüksége az emberekbe épített lapkákra, ha a tárcájuk tele lenne ilyenekkel? Hangsúlyozzuk, ha a nyilvánosság tudomására jut, hogy az RFID lapkák mire képesek, valószínűleg nyilvános viták tárgyát fogja képezni.

Az RFID lapkákban alkalmazott technológia gyorsan fejlődik. A legkisebbek passzívak (nincs belső áramforrásuk), és csak arra képesek, hogy a saját egyedi azonosítószámukat elküldjék, ha kérik tőlük. A nagyobbak viszont aktívak, tartalmazhatnak egy kis akkumulátort és egy egyszerű számítógépet, és képesek lehetnek bizonyos számítások elvégzésére is. Ebbe a kategóriába tartoznak a pénzügyi tranzakcióknál használatos intelligens kártyák.

Az RFID lapkák nemcsak aktív vagy passzív mivoltukban különböznek, hanem a rádiófrekvencia-tartományban is, amelyre reagálnak. Azok, amelyek alacsony frekvenciákon működnek, korlátozott adatsebességgel rendelkeznek, azonban az antennától nagy távolságban is érzékelhetők. A magas frekvenciákon működő típusok adatsebessége nagyobb, a hatótávolságuk viszont kisebb. A lapkák egyéb módon is eltérnek és folyamatosan fejlődnek. Az interneten rengeteg információ található az RFID lapkákról; a *www.rfid.org* jó kiindulópont lehet.

1.3.4. Mikrovezérlők

A létra következő fokán a különféle – nem számítógépként forgalmazott – eszközökbe beágyazott számítógépek állnak. A beágyazott számítógépek, melyeket néha **mikrovezérlők**nek neveznek, magát a berendezést és a felhasználói interfészt

is kezelik. A mikrovezérlők számos különféle eszközben megtalálhatók, köztük az alábbiakban is. Minden kategóriához adunk néhány példát is.

1. Háztartási berendezések (rádiós óra, mosógép, szárítógép, mikrohullámú sütő, riasztó).
2. Kommunikációs eszközök (vezeték nélküli telefon, mobiltelefon, fax, személyi hívó).
3. Számítógép-perifériák (nyomtató, lapolvasó, modem, CD-ROM-meghajtó).
4. Szórakoztatóelektronikai cikkek (videomagnó, DVD-lejátszó, hifiberendezés, MP3-lejátszó, beltéri vevőegység).
5. Képpel kapcsolatos berendezések (tv, digitális kamera, cameoder, objektívek, fénymásoló).
6. Orvosi berendezések (röntgenkészülék, MRI, szívmonitor, digitális hőmérő).
7. Katonai fegyverrendszerek (robotrepülőgép, interkontinentális rakéta, torpedó).
8. Vásárlással kapcsolatos eszközök (üdítő-, kávé-, jegy- és egyéb árusító automaták, ATM, pénztárgép).
9. Játékok (beszélő baba, játékkonzol, rádióvezérelt autó vagy hajó).

Egy felsőkategóriás autóban könnyen találhatunk akár 50 mikrovezérlőt is, melyek olyan részrendszereket működtetnek, mint a blokkolásgátló fékrendszer, üzemanyag-befecskendezés, rádió és a navigációs egység (GPS). Egy sugárhajtású repülőgép akár több mint kétszázat is tartalmazhat. Könnyen előfordulhat, hogy egy családban több száz számítógép is van anélkül, hogy tudnának róla. Néhány éven belül gyakorlatilag minden, ami elektromos hálózatról vagy akkumulátorról működik, mikrovezérlőket tartalmaz majd. Az évenként eladott mikrovezérlők száma mellett nagyságrendekkel eltörpülnek az egyéb típusú számítógépek eladásai, kivéve az eldobható számítógépeket.

Míg az RFID lapkák minimalista rendszerek, a mikrovezérlők kicsik ugyan, de teljes értékű számítógépek. Minden mikrovezérlőben van egy processzor, memória és B/K képességek. A B/K képesség gyakran jelenti a berendezés nyomógombjainak és kapcsolóinak érzékelését, valamint az eszköz lámpáinak, megjelenítőjének, hangjának és motorjainak vezérlését. Legtöbb esetben a szoftvert már gyártáskor beépítik a lapkába egy csak olvasható memóriába. A mikrovezérlőknek két nagy csoportja van: általános és speciális célú. Az előbbi típusba tartozók kisméretűek, de amúgy közönséges számítógépek. Az utóbbi csoportba tartozók mindig valamilyen konkrét alkalmazásra, mint például a multimédiára hangolt architektúrával és utasításrendszerrel rendelkeznek. A mikrovezérlőknek léteznek 4, 8, 16 és 32 bites fajtái.

Még az általános célú mikrovezérlők is több lényeges tekintetben különböznek a közönséges PC-ktől. Először is, rettenetesen árérzékenyek. Amikor egy vállalat több millió darabot vásárol, a választás akár a darabonkénti 1 cent árkülönbségen is múlhat. Ezért a mikrovezérlők gyártói gyakrabban hoznak architektúrára vonatkozó döntéseket gyártási költségek alapján, mint a több száz dolláros lapkák esetén. A mikrovezérlők ára nagyon változatos attól függően, hogy hány bitesek, mekkora és milyen típusú memóriát tartalmaznak stb. Viszonyításképpen, ha egy

8 bites mikrovezérlőt elég nagy tételben vásárolunk, valószínűleg megkaphatjuk akár darabonként 10 centért is. Ez az ár teszi lehetővé, hogy számítógépet építsünk egy 10 dolláros rádiós órába.

Másodsorban, jóformán minden mikrovezérlő valós időben működik. Amikor ingert kapnak, azonnali választ várunk tőlük. Például, amikor a felhasználó megnyom egy gombot, gyakran kigyullad egy lámpa, és ilyenkor a gomb megnyomása és a lámpa kigyulladása között nem lehet késleltetés. A valós idejű működés szükségessége gyakran hatással van az architektúrára is.

Harmadszor, a beágyazott rendszereknek gyakran megszorításoknak kell megfelelniük méret, súly, áramfelvétel és egyéb elektromos és mechanikai korlátokkal. A bennük használt mikrovezérlőket ezen megszorítások figyelembevételével kell tervezni.

1.3.5. Játékgépek

Egy fokkal magasabban állnak a videojátékgépek. Ezek közönséges számítógépek különleges grafikai és hangképességekkel, meghatározott szoftverrel és csak minimális bővítési lehetőséggel. A sort nyitó kisteljesítményű CPU-kkal egyszerű akciójátékokat, mint például pingpong, lehetett játszani a tévékészüléken. Az évek során azonban sokkal erőteljesebb rendszerekké fejlődtek, melyek a személyi számítógépekkel vetélkednek, sőt néhány tekintetben túl is szárnyalják azokat.

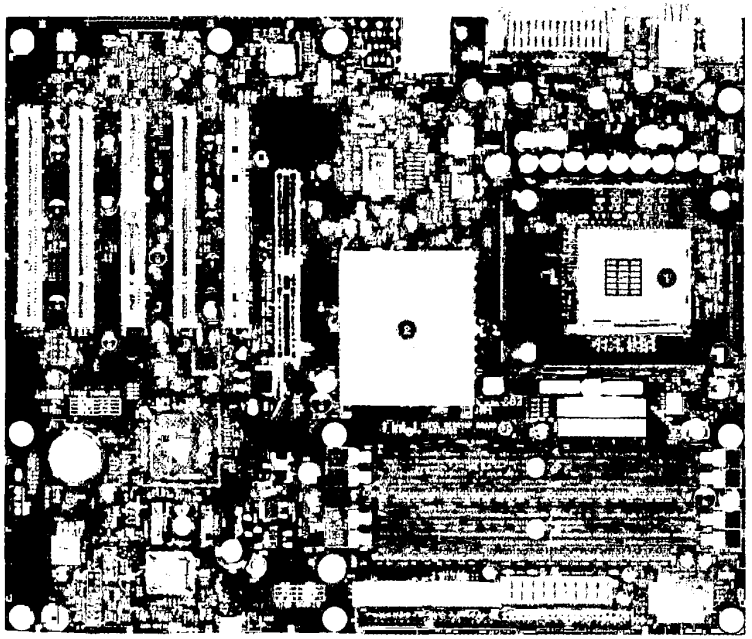
Ahhoz, hogy lássuk, miből is tevődik össze egy játékgép, nézzük három népszerű termék specifikációját. A Sony PlayStation 2-ben egy 295 MHz-es 128 bites saját fejlesztésű CPU található (a neve Emotion Engine), mely a MIPS IV RISC CPU mintájára épül. A PlayStation 2 tartalmaz továbbá 32 MB RAM-ot, egy 160 MHz-es speciális grafikai chipet, egy 48 csatornás speciális audiochipet és egy DVD-lejátszót. A Microsoft XBOX-ban található egy 733 MHz-es Intel Pentium III, 64 MB RAM, egy 300 MHz-es speciális grafikai chip, egy 256 csatornás speciális audiochip, egy DVD-lejátszó és egy 8 GB-os merevlemez. A Nintendo GameCube motorja egy 485 MHz-es 32 bites speciális CPU (a neve Gekko), melyet az IBM PowerPC RISC CPU-ból fejlesztettek; 24 MB RAM, egy 200 MHz-es speciális grafikus chip, egy 64 csatornás speciális audiochip és egy saját fejlesztésű 1,5 GB-os optikai lemez van benne.

Bár ezeknek a gépeknek a teljesítménye elmarad az ugyanakkor gyártott személyi számítógépekétől, de nem lényegesen, sőt, bizonyos tekintetben még előrébb is járnak (például a 128 bites CPU a PlayStation 2-ben szélesebb, mint bármely PC CPU-ja, jóllehet az órajelük sokkal alacsonyabb). A legfőbb eltérés a játékgépek és a PC-k között nem is annyira a CPU, mint inkább az a tény, hogy a játékgépek zárt rendszerek. A felhasználó nem bővítheti őket bővítőkártyákkal, bár néha azért biztosítanak USB vagy FireWire interfészeket. Továbbá, és ez talán a legfontosabb, a játékgépeket körültekintően egyetlen alkalmazási területre optimalizálják, például magas fokon interaktív 3D játékok kimagasló minőségű sztereóhanggal. Minden egyéb másodlagos kérdés. Ezek a hardver- és szoftvermegszorítások, az alacsony órajel, a kis memória, a nagyfelbontású monitor hiánya és (általában) a merevlemez hiánya teszi lehetővé, hogy ezek a gépek a személyi számítógépeknél olcsóbban előállíthatók és forgalmazhatók. A megszorítások ellenére már játékgépek millióit értékesítették.

A legnagyobb játékgépeket készítő vállalatok hordozható, akkumulátorról működő játékgépeket is gyártanak, melyek elférnek az ember kezében. Ezek azonban már közelebb állnak a beágyazott rendszerekhez, melyet korábban elemeztünk, mint a személyi számítógépekhez.

1.3.6. Személyi számítógépek

Eljutottunk a személyi számítógépekhez – a legtöbb embernek ez jut eszébe a „számítógép" szó hallatán. Az asztali és a hordozható (notebook, noteszgép) modellekre egyaránt gondoljunk. Több száz megabájt memóriával, 100 gigabájt körüli



1. Pentium 4 foglalat
2. 875P segédlapka
3. memóriafoglalatok
4. AGP-csatlakozó
5. lemezinterfész
6. gigabit ethernet
7. öt PCI-foglalat
8. USB 2.0 portok
9. hűtéstechnika
10. BIOS

1.10. ábra. *A nyomtatott áramköri kártya minden személyi számítógép lelke. Az ábrán az Intel D875PBZ kártya fényképe látható, melyet a szerzői jogok tulajdonosa, az Intel Corporation (2003) engedélyével tesszük közzé*

merevlemezzel, CD-ROM- és/vagy DVD-meghajtóval, modemmel, hangkártyával, hálózati csatolóval, nagyfelbontású monitorral és egyéb perifériákkal készülnek. Jól kidolgozott operációs rendszerrel, rengeteg kiegészítési lehetőséggel és bő szoftverkínálattal rendelkeznek. Egyesek csak az Intel CPU-s változatokat hívják „PC"-nek (személyi számítógépnek), míg a nagy teljesítményű RISC-processzorral (mint például a Sun UltraSPARC) felszerelteket „munkaállomásoknak" nevezik. Elvi eltérés azonban alig van közöttük.

Minden személyi számítógép lelke egy nyomtatott áramköri kártya, a doboza alján. Ez többnyire tartalmazza a CPU-t, memóriát, különféle B/K eszközöket (mint például a hangchip és esetleg egy modem), interfészeket a billentyűzet, egér, lemezek, hálózat stb. csatlakoztatásához, valamint néhány bővítő aljzatot. Egy ilyen áramköri kártya képe látható az 1.10. ábrán.

A hordozható számítógépek (noteszgépek) gyakorlatilag PC-k, csak kisebb kiszerelésben. Ugyanazokat a hardverelemeket használják, csak kisebb méretűre gyártják őket. És ugyanazok a szoftverek is futnak rajtuk, mint az asztali PC-ken.

Egy másik szorosan kapcsolódó géptípus a PDA (Personal Digital Assistant, digitális személyi asszisztens). Bár ezek még a noteszgépeknél is kisebbek, mindegyikben van CPU, memória, billentyűzet, megjelenítő és a személyi számítógépek egyéb jellemzői is, csak kicsiben. Mivel a legtöbb olvasó valószínűleg otthonosan mozog a személyi számítógépek világában, aligha van szükség további bevezetőre.

1.3.7. Kiszolgálók

A kissé megerősített személyi számítógépeket vagy munkaállomásokat gyakran alkalmazzák hálózati kiszolgálókként (szerverként) mind helyi hálózaton (többnyire egy vállalaton belül), mind pedig az interneten. Ezek lehetnek egy- vagy többprocesszorosak, és több gigabájt memória, több száz gigabájt merevlemez és nagy sebességű hálózati kapcsolat jellemző rájuk. Némelyik akár kérések ezreit is képes kezelni másodpercenként.

Architektúra tekintetében az egyprocesszoros kiszolgáló nem lényegesen különbözik egy egyprocesszoros személyi számítógéptől. Csak gyorsabb, több memóriával, nagyobb lemezterülettel és valószínűleg gyorsabb hálózati kapcsolattal rendelkezik. A kiszolgálók ugyanazt az operációs rendszert futtatják, mint a személyi számítógépek, többnyire a UNIX vagy a Windows valamelyik változatát.

1.3.8. Munkaállomások gyűjteménye

A munkaállomások és személyi számítógépek ár/teljesítmény arányának folyamatos javulásának köszönhető, hogy az utóbbi években a rendszertervezők ezekből nagyszámút összekapcsolva, **munkaállomások klasztereit (COW, Clusters of Workstations)** vagy csak egyszerűen **klasztereket** hoznak létre. Ezek közönséges személyi számítógépekből vagy munkaállomásokból állnak, amelyeket gigabites

hálózat köt össze, és amelyek egy speciális szoftvert futtatnak; ennek segítségével a gépek együtt dolgozhatnak egy konkrét, gyakran tudományos vagy mérnöki feladat megoldásán. Általában **COTS (Commodity Off The Shelf**, készen kapható termék) számítógépekről van szó, amelyeket bárki megvásárolhat egy közönséges PC-forgalmazótól. A legfőbb kiegészítés a nagy sebességű hálózat, de néha az is csak egy hétköznapi hálózati kártya. A klaszterek méretben könnyen skálázhatók, egy maroknyi géptől akár ezrekig. Többnyire a rendelkezésre álló pénz szabja meg a határokat.

Egy COW-t használhatunk internetes webszervernek is. Amikor a website másodpercenként kérések ezreire számíthat, a leggazdaságosabb megoldás gyakran egy több száz vagy akár több ezer szerverből álló klaszter. A beérkező kérések elosztásra kerülnek a szerverek között, és így tulajdonképpen párhuzamosan feldolgozhatók. Ilyen alkalmazás esetén a COW-t gyakran nevezik **kiszolgáló farm**nak.

1.3.9. Nagyszámítógépek

Megérkeztünk a nagygépekhez, az 1960-as éveket idéző, terem méretű számítógépekhez. Sok közülük az évtizedekkel korábbi beszerzésű IBM 360-asok közvetlen leszármazottja. Nem sokkal gyorsabbak a nagy teljesítményű szervereknél, de B/K átviteli képességük nagyobb, és gyakran több ezer gigabájt adatkapacitású hatalmas merevlemezfarmot kezelnek. Bár rendkívül drágák, a szoftverbe, adatba, ügymenetbe és személyzetbe befektetett óriási érték megóvása érdekében gyakran tovább üzemeltetik őket. Sok vállalat úgy gondolja, hogy kifizetődőbb időnként kiadni néhány millió dollárt egy új gépre, mint átgondolni az összes alkalmazás kisebb gépekre való újraprogramozásával járó erőfeszítéseket.

Ez a géposztály vezetett a hírhedt 2000. év problémához, mely az 1960-as és 1970-es évek COBOL-programozóinak memóriatakarékossági megfontolásából született, miszerint az évszámot csak két decimális számjegyen tárolják. Soha nem hitték volna, hogy szoftverük még három-négy évtized múlva is létezni fog. Bár a megjövendölt katasztrófa – a probléma megoldásába fektetett hatalmas mennyiségű munkának köszönhetően – sosem következett be, számos cég megismételte a régi hibát azzal, hogy az évszámhoz hozzácsaptak még két számjegyet. A szerző ezennel ünnepélyesen bejelenti az emberi civilizáció végóráját, 9999. december 31-én éjjeli 12 órát, amikor is a 8000 éves dicső COBOL-programok egyszerre lehelik ki lelküket.

Azon felül, hogy a megörökölt 30 éves szoftvereket futtatják, az utóbbi időben az internet új életet lehelt a nagygépekbe. Új piacot találtak mint nagy teljesítményű internetszerverek, például nagyszámú elektronikus üzleti tranzakció kezelését másodpercenként, kiváltképp olyan üzletekben, ahol hatalmas adatbázisokra van szükség. Bár könyvünk középpontjában a PC-k, szerverek és mikrovezérlők állnak, az 5. fejezetben tovább foglalkozunk kicsit a nagyszámítógépekkel is.

A számítógépeknek a közelmúltig létezett még a nagygépeknél is erősebb kategóriája, a **szuperszámítógépek**. Jellemzőjük a rendkívül gyors CPU, a sok gigabájt központi memória, a nagyon gyors lemezek és hálózat volt. Többnyire nagy meny-

nyiségű tudományos és mérnöki számításokra, köztük csillagrendszerek ütközésének szimulációjára, új gyógyszerek szintetizálására vagy a repülőgép szárnya körül kialakuló légmozgás modellezésére használták őket. Az utóbbi években azonban a COW-k ugyanazt a számítási kapacitást sokkal olcsóbban biztosítják, így a valódi szuperszámítógépek mára kihalófélben vannak.

1.4. Néhány számítógépcsalád

Könyvünkben három számítógéptípusra koncentrálunk: a személyi számítógépekre, a szerverekre és a beágyazott számítógépekre. A személyi számítógépekkel azért foglalkozunk, mert minden bizonnyal minden olvasónk kapcsolatba került már eggyel. A szerverekkel azért, mert ezek biztosítják az internet összes szolgáltatását. Végül, bár felhasználóik számára láthatatlanok, a beágyazott számítógépek vezérlik autóinkat, televízióinkat, mikrohullámú sütőinket, mosógépeinket és gyakorlatilag minden más, 50 dollárnál drágább elektromos berendezésünket.

Ebben a fejezetben röviden bemutatjuk azt a három számítógépet — a fenti kategóriákból egyet-egyet —, amelyeket könyvünkben állandó példaként fogunk használni. Ezek a Pentium 4, az UltraSPARC III és a 8051.

1.4.1. A Pentium 4 áttekintése

1968-ban Robert Noyce, a szilikon integrált áramkör feltalálója, a szabályáról híres Gordon Moore és egy San Franciscó-i befektető, Arthur Rock memórialapkák gyártására megalapította az Intel Corporationt. Az első évben az Intel csak 3000 dollár értékben adott el lapkákat, de az üzlet azóta fellendült.

Az 1960-as évek végén az elektromechanikus számológépek akkorák voltak, mint egy mai lézernyomtató, a súlyuk pedig a 20 kg-ot is elérhette. 1969 szeptemberében a japán Busicom cég egy 12 speciális lapkára szóló megrendeléssel kereste meg az Intelt; a lapkát egy tervezett elektronikus számológépbe kívánták beépíteni. Ted Hoff, az Intel mérnöke kapta meg a feladatot, és miután átnézte a terveket, felismerte, hogy a feladat egy egyetlen lapkára integrált 4 bites általános célú CPU-val is megoldható, ami ráadásul egyszerűbb és olcsóbb is. Így született meg 1970-ben az első, 2300 tranzisztoros egylapkás CPU, a 4004-es (Faggin és társai, 1996).

Érdemes megjegyeznünk, hogy sem az Intel, sem a Busicom nem tudta, mit is alkotott. Amikor az Intel elhatározta, hogy a 4004-est más célokra is kipróbálja, felajánlotta a Busicomnak, hogy visszavásárolja az új lapkára vonatkozó jogokat azért a 60 000 dollárért, amit a Busicom a fejlesztésért fizetett. Az ajánlatát azonnal elfogadták, az Intel pedig nekilátott a lapka 8 bites változatának, a 8008-asnak a fejlesztéséhez, amit 1972-ben be is fejezett. A 4004-essel és 8008-assal kezdődő Intel család az 1.11. ábrán látható.

Lapka	Dátum	MHz	Tranzisztorok száma	Memória	Megjegyzés
4004	1971/4	0,108	2300	640 B	Az első egylapkás mikroprocesszor
8008	1972/4	0,108	3500	16 KB	Az első 8 bites mikroprocesszor
8080	1974/4	2	6000	64 KB	Az első általános célú egylapkás CPU
8086	1978/6	5–10	29000	1 MB	Az első 16 bites egylapkás CPU
8088	1979/6	5–8	29000	1 MB	Az IBM PC processzora
80286	1982/2	8–12	134000	16 MB	Megjelent a memóriavédelem
80386	1985/10	16–33	275000	4 GB	Az első 32 bites CPU
80486	1989/4	25–100	1,2 millió	4 GB	8 KB beépített gyorsítótár
Pentium	1993/3	60–233	3,1 millió	4 GB	Két csővezeték, későbbi modellekben MMX
Pentium Pro	1995/3	150–200	5,5 millió	4 GB	Kétszintű beépített gyorsítótár
Pentium II	1997/5	233–450	7,5 millió	4 GB	Pentium Pro MMX utasításokkal
Pentium III	1999/2	650–1400	9,5 millió	4 GB	SSE utasítások 3D grafikához
Pentium 4	2000/11	1300–3800	42 millió	4 GB	Hyperthreading és még több SSE utasítás

1.11. ábra. *Az Intel processzorcsalád. Az órajeleket MHz-ben (megahertz) adtuk meg, ahol 1 MHz másodpercenkénti 1 millió ciklusnak felel meg*

Az Intel a 8008-asban nem sok esélyt látott, ezért kis teljesítményű gyártósort állított fel. Mindenki meglepetésére a kereslet hatalmas volt, ezért nekifogtak egy új, de a 8008-as 16 kilobájtos memóriakorlátját túllépő lapka tervezéséhez. (A korlát a lapka csatlakozópontjainak számából eredt.) Az eredmény az 1974-re elkészült 8080-as, egy kicsi, általános célú CPU lett. A PDP-8-hoz hasonlóan viharos gyorsasággal hódította meg az ipart, és azonnal tömegáru lett a piacon. Csakhogy a DEC ezres nagyságrendű eladásaival szemben az Intel milliószám adta el az új CPU-t.

1978-ban jött a 8086-os, egy vadonatúj egylapkás 16 bites CPU. Bár a 8080-hoz hasonlóra tervezték, nem volt vele teljesen kompatibilis. A 8086-ost a vele azonos felépítésű 8088-as követte, amely ugyanazokat a programokat tudta futtatni, de nem 16 bites, hanem 8 bites sínnel, emiatt lassabb, de olcsóbb volt a 8086-osnál. Amikor az IBM a 8088-ast választotta az eredeti IBM PC processzorának, ez a lapka lett a mértékadó a személyi számítógépek iparában.

Sem a 8088-as, sem a 8086-os nem tudott 1 megabájtnál nagyobb memóriát megcímezni. Az 1980-as évek elejére ez egyre súlyosabb probléma lett, így az Intel megtervezte a 8086-ossal felülről kompatibilis változatot, a 80286-ost. Alapvető utasításai a 8088-as, illetve 8086-oséval lényegében megegyeztek, de a memóriaszervezés merőben más, és a korábbi lapkákkal való kompatibilitás követelménye miatt elég ügyetlen volt. A 80286-ost az IBM PC/AT-ben és a PS/2-esek középtel-

jesítményű változataiban használták. A 8088-ashoz hasonlóan ez is igen sikeres volt, elsősorban azért, mert a 8088-as gyorsabb változatának tartották.

A következő lépés az 1985-ben bevezetett valódi 32 bites, egylapkás CPU, a 80386-os volt. A 80286-oshoz hasonlóan a 80386-os is a 8080-asig bezárólag minden régebbi változattal többé-kevésbé kompatibilis volt. A felülről kompatibilitás miatt áldás volt azoknak, akik régi szoftvereiket meg akarták tartani, ugyanakkor büntetés az egyszerű, átlátható és a divatjamúlt technológia hibáitól mentes új architektúrára vágyók számára.

Négy évvel később megjelent a 80486-os. Ez lényegében a 80386-os gyorsabb változata volt, lapkáján azonban lebegőpontos egység és 8 kilobájtnyi gyorsítótár is helyet kapott. A **gyorsítótár** a gyakran használt memóriaszavak tartalmát a processzorban, illetve ahhoz közel tárolja, ezzel küszöböli ki a lassabb memóriához fordulást. A többprocesszoros rendszereknek a 80486-osba beépített támogatása lehetővé tette, hogy a gyártók több CPU-t tartalmazó közös memóriát használó rendszereket építsenek.

Ekkorra az Intel is rájött (egy védjegysértési per elvesztése után), hogy számok (mint a 80486) nem védjegyezhetők, ezért a következő generációnak már nevet adott: **Pentium** (az ötös szám görög nevéből: πεντε). A 80486-ossal ellentétben, amely még csak egy belső csővezetékkel rendelkezett, a Pentiumnak már kettő volt, és így kétszer gyorsabb is lett (a csővezetékről a 2. fejezetben beszélünk részletesen).

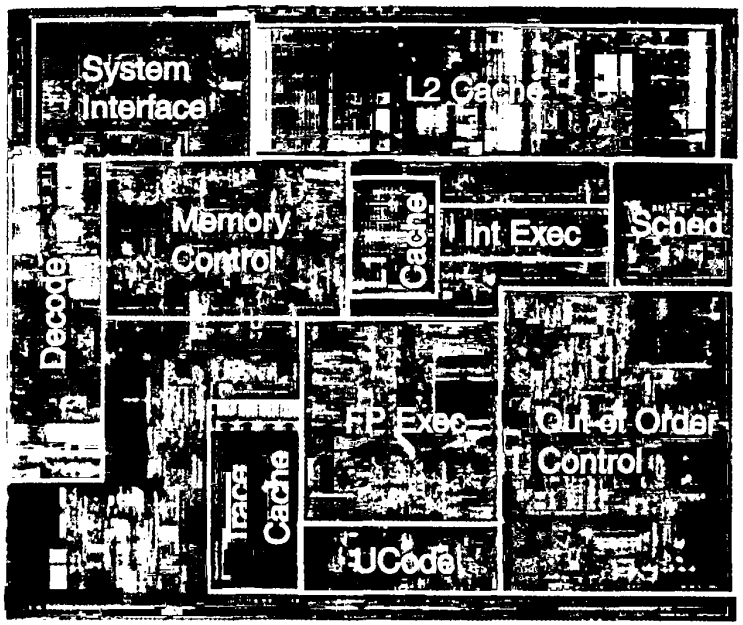
A termelés későbbi folyamán az Intel speciális **MMX (MultiMedia eXtension, multimédiás kiegészítések)** utasításokat épített be. Ezek célja a hang- és videoadatok feldolgozásához szükséges számítások felgyorsítása volt, amivel szükségtelenné tették a különleges multimédia-társprocesszorok alkalmazását.

Akik a megjelenő következő generációt Sexium (a hatos szám latin neve: *sex*) elnevezéssel várták, csalódtak. A Pentium név már olyan ismertté vált, hogy a reklámszakemberek meg akarták tartani, így az új lapka a Pentium Pro nevet kapta. Bár neve alig tér el elődjétől, ez a processzor komoly szakítás a múlttal. Nem a csővezetékek számát szaporították, hanem a Pentium Pro belső felépítését változtatták meg úgy, hogy akár öt utasítás egyidejű végrehajtására is képes volt.

A Pentium Pro másik újdonsága a kétszintű gyorsítótárban rejlett. A processzor lapkáján a gyakran használt utasítások és adatok tárolására külön-külön egy-egy 8 kilobájt méretű nagy sebességű memória szolgál. A Pentium Pro-csomaghoz – bár nem közvetlenül a lapkán – még egy 256 kilobájtos 2. szintű gyorsítótár is tartozott.

A nagy gyorsítótár ellenére a Pentium Pro nem rendelkezett az MMX-utasításkészlettel (mert az Intel nem volt képes ilyen nagyméretű megfelelő teljesítményű lapkát gyártani). Amikor a technológia fejlődése lehetővé tette az MMX és a gyorsítótár egy lapkára integrálását, az új termék Pentium II néven került forgalomba. A következő lépés még több multimédia-utasítás, **SSE (Streaming SIMD Extensions)** hozzáadása volt a tovább javított 3D grafika érdekében (Raman és társai, 2000). Az új lapkát ugyan Pentium III-nak hívták, de belül lényegében egy Pentium II volt.

A következő Pentium már alapvetően más belső architektúrára épült. Az esemény megünnepléseként az Intel a római számozásról áttért az arabra, és így Pentium 4-nek keresztelte. A Pentium 4, a szokásoknak megfelelően gyorsabb



System Interface – Rendszerinterfész
L2 Cache – L2 gyorsítótár
Decode – Dekódoló
Memory Control – Memóriavezérlő
L1 Cache – L1 gyorsítótár
Int Exec – Egészművelet-végrehajtó
Sched – Ütemező
FP Exec – Lebegőpontosművelet-végrehajtó
Out of Order Control – Sorrendenkívüliség-vezérlő
Trace Cache – Nyomkövető gyorsítótár
UCode

1.12. ábra. *A Pentium 4 lapka. A fényképet a szerzői jog tulajdonosa, az Intel Corporation (2003) engedélyével közöljük*

volt a korábbiaknál. A 3,06 GHz-es változatban egy érdekes új tulajdonságot is bevezettek, a hyperthreadinget. Ez a programok számára lehetővé tette, hogy a munkát két vezérlési szálra bontsák, amelyet a Pentium 4 párhuzamosan futtathat, ezzel felgyorsítva a végrehajtást. (A hyperthreadingről a 8. fejezetben részletesebben is szólunk majd.) Kiegészítésként további SSE utasítások kerültek a processzorba, ezzel tovább gyorsítva a hang- és videoadatok feldolgozását. Az 1.12. ábrán látható egy Pentium 4 lapka. A valóságban, a maga 16,0 mm × 13,5 mm-es méretével nagyon nagy lapkának számít.

A fent említett asztali CPU-k vonala mellett az Intel némely Pentium lapkájának különleges piacra szánt speciális változatait is elkészítette. 1998 elején beindította a **Celeron** termékvonalat, a Pentium II alacsonyabb árfekvésű és teljesítményű változatát a kisebb PC-k processzoraként. Felépítését tekintve a Celeron azonos a Pentium II-vel, ezért külön nem foglalkozunk vele. Ugyancsak nem tárgyaljuk külön a Pentium II egy másik változatát, az 1998 júniusában bejelentett **Xeon**

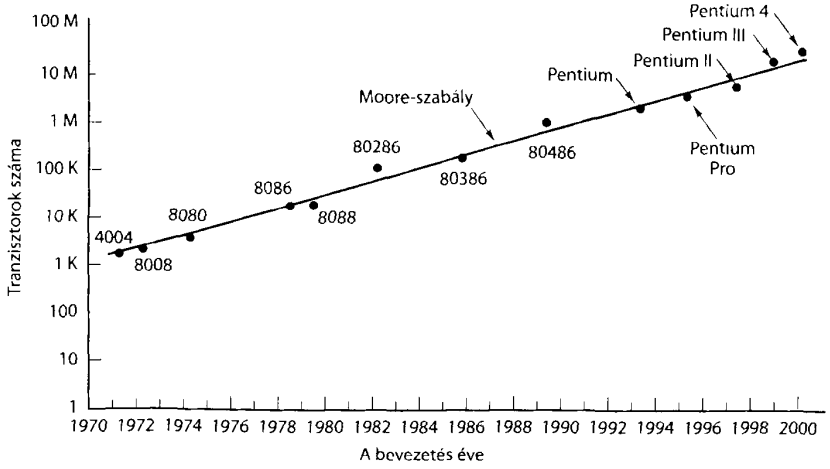
processzort sem. Ezt a nagy teljesítményű személyi számítógépek processzorának tervezték nagy gyorsítótárral, gyors sínnel és javított multiprocesszoros támogatással. A Pentium III-ból szintén készült Xeon-változat.

2000 novemberében az Intel tehát forgalomba hozta a Pentium 4-et, amely bár ugyanazokat a programokat futtatta, mint a Pentium III és a Xeon, belső felépítésében teljesen új volt.

2003-ban az Intel bemutatta a Pentium M (mint mobil) lapkát, melyet speciálisan hordozható számítógépekhez (noteszgépek) terveztek. Ez a lapka a Centrino architektúra része volt, melynek céljai az alacsonyabb energiafogyasztás, így hosszabb akkumulátor-élettartam, kisebb, könnyebb számítógépek és az IEEE 802.11 (WiFi) szabványra épülő beépített vezeték nélküli hálózat lehetősége. Az Intel tervei között szerepel újabb lapkák kifejlesztése különleges alkalmazások céljaira, mint például a házi szórakoztatóelektronikai berendezések és az IEEE 802.16 (WiMax) noteszgépek.

Az összes Intel-lapka felülről kompatibilis az elődjével, vissza egészen a 8086-osig. Más szóval a Pentium 4 a 8086-os programjait minden módosítás nélkül képes futtatni. Az Intel számára mindig is követelmény volt ez a kompatibilitás – azért, hogy felhasználói megőrizhessék a szoftverbe fektetett értékeiket. Természetesen a Pentium 4 három nagyságrenddel összetettebb, mint a 8086-os volt, tehát elég sok mindent tud, amit a 8086-os nem. Az apránkénti bővítgetés eredményeképpen messze nem olyan elegáns a felépítése, mintha a Pentium 4 tervezői 42 millió tranzisztorból az utasításkészlet ismeretében teljesen újraterveznék.

Érdekes, hogy bár Moore szabályát hosszú ideig a memória bitjeinek számára alkalmazták, az ugyanúgy érvényes a processzorlapkákra is. Ha az 1.11. ábrán megadott tranzisztorszámokat logaritmikus skálán ábrázoljuk a bevezetési időpontok függvényében, látható, hogy az megfelel Moore szabályának. Ez látszik az 1.13. ábrán.



1.13. ábra. *Moore szabálya (Intel-) processzorlapkákra alkalmazva*

Bár vélhetően a Moore-szabály még érvényben marad a következő években, egy másik probléma kezdi beárnyékolni: a hőkibocsátás. A kisebb tranzisztorok lehetővé teszik a magasabb órajelek használatát, amihez viszont magasabb feszültség szükséges. A felvett energia és a kibocsátott hő a feszültséggel négyzetesen arányosak, tehát minél gyorsabban fut egy processzor, annál több hőtől kell megszabadulni. 3,6 GHz-en a Pentium 4 fogyasztása 115 watt. Eszerint nagyjából annyi hőt termel, mint egy 100 wattos izzó. Még nagyobb órajelnél a helyzet tovább romlik.

2004 novemberében az Intel törölte a 4 GHz-es Pentium 4-es vonalat, mert problémák voltak a hőelvezetéssel. Hatalmas ventilátorok segítenének, de a zaj, amelyet keltenek, nem aratna osztatlan sikert a felhasználók körében. A nagyszámítógépek esetében használatos vízhűtés viszont az asztali gépek esetében nem jöhet szóba (noteszgépeknél pedig még kevésbé). Következésképpen, az órajelek korábbi könyörtelen menetelése megtorpanni látszik, legalábbis amíg az Intel mérnökei ki nem találják, hogyan szabaduljanak meg hatékonyan a termelt hőtől. Ehelyett az Intel jövőre vonatkozó tervei szerint két CPU kerülne egy lapkára egy nagy közös használatú gyorsítótárral egyetemben. Az energiafelvétel és a feszültség, illetve az órajel közötti kapcsolatból következően két CPU egy lapkán sokkal kevesebb energiát fogyasztana, mint egy CPU, amely kétszer olyan gyorsan működik. Ebből következik, hogy a Moore-szabály adta lehetőségeket a jövőben egyre nagyobb gyorsítótárakkal, nem pedig magasabb órajelekkel lehet kiaknázni (mivel a memória csak keveset fogyaszt).

1.4.2. UltraSPARC III áttekintése

Az 1970-es években a UNIX nagyon népszerű volt az egyetemeken, személyi számítógépeken azonban nem futott, így a UNIX szerelmesei kénytelenek voltak az olyan (gyakran túlterhelt) időosztásos minigépeken dolgozni, mint például a PDP–11 vagy a VAX. 1981-ben Andy Bechtolsheim, a Stanford Egyetem német hallgatója – megunva, hogy a számítóközpontba kell eljárnia, hogy UNIX-ot használhasson – elhatározta, hogy kész elemekből felépít magának egy saját UNIX-munkaállomást. A gépet SUN–1 névre keresztelte (Stanford University Network).

Bechtolsheim rögtön felkeltette Vinod Khosla, a 27 éves indiai érdeklődését, aki égett a vágytól, hogy 30 éves korára milliomosként nyugdíjba vonulhasson. Khosla meggyőzte Bechtolsheimet, hogy alapítsanak vállalatot Sun-munkaállomások építésére és forgalmazására. Khosla felvette Scott McNealyt, egy másik stanfordi hallgatót a gyártás irányítására. A szoftver készítésére Bill Joy-t, a Berkeley UNIX vezető tervezőjét alkalmazták. Ők négyen alapították meg 1982-ben a Sun Microsystems vállalatot.

Az első termék, a Motorola 68020 CPU-ra épülő Sun–1, majd az ezt követő, szintén Motorola CPU-t használó Sun–2 és Sun–3 is azonnal sikeres lett. A gépek teljesítménye az akkori személyi számítógépekéhez képest jóval nagyobb volt (innen a „munkaállomás” elnevezés), és eleve hálózatban működőknek tervezték őket. Minden Sun-munkaállomásban Ethernet-csatolókártya volt, és TCP/IP-szoftverrel kapcsolódott az internet elődjéhez, az ARPANET hálózathoz.

1987-re a Sun már évi félmilliárd dollár értékben adott el rendszereket, és ekkor elhatározták, hogy saját processzort terveznek a University of California at Berkeley forradalmian új modelljének (RISC II) a mintájára. Ez a CPU, a **SPARC (Scalable Processor ARChitecture, skálázható processzorarchitektúra)** lett a Sun–4 munkaállomás alapja. Rövid időn belül minden Sun-gépbe SPARC CPU került.

Sok más vállalattal szemben a Sun úgy döntött, hogy a SPARC processzort nem maga fogja előállítani. Ehelyett több cégnek is átadta a gyártási jogokat, és azt remélte, hogy a közöttük kialakuló verseny a teljesítmény növekedéséhez, és az árak csökkenéséhez vezet majd. Valóban, a gyártók számos, különböző technológiára épülő, különböző órajelsebességgel működő lapkát kezdtek előállítani változatos áron. A MicroSPARC, a HyperSPARC, a SuperSPARC, a TurboSPARC mind ezek közül való. Bár apró dolgokban eltérő processzorokról van szó, mindegyik binárisan kompatibilis és módosítás nélkül tudja ugyanazokat a felhasználói programokat futtatni.

A Sun a SPARC-ot mindig is nyílt architektúraként kezelte, számos alkatrész- és rendszerbeszállítóval, és olyan ipart akart felépíteni, amely versenyképes lehet az – akkor már Intel-alapú processzorok uralta – személyi számítógépek piacán. A Sun létrehozta a SPARC International ipari konzorciumot a SPARC architektúra folyamatos fejlesztésének koordinálására, ezzel megnyerve azon vállalatok bizalmát is, akik ugyan érdeklődtek a SPARC iránt, de egy versenytárs termékébe nem akartak pénzt fektetni. Emiatt meg kell különböztetnünk a SPARC architektúrát – amely az utasításrendszer és a programozó által látható tulajdonságok specifikációja – és annak tényleges megvalósításait. Könyvünk mindkettővel foglalkozik, az általános SPARC architektúrával és a Sun-munkaállomásokba épített egyik konkrét SPARC lapkával egyaránt; utóbbival a 3. és 4. fejezetben, a CPU lapkák tárgyalásánál.

Az első SPARC 36 MHz-es, valódi 32 bites gép volt. Processzora, az **IU (Integer Unit)** szegényes és egyszerű volt, mindössze három utasítástípussal és 55 utasítással rendelkezett. Ezt a lebegőpontos egység 14 utasítással bővítette ki. Láthatjuk az alapvető különbséget az Intel-irányzattal szemben, amely 8 és 16 bites lapkákkal indult (8088, 8086, 80286), és csak azután lett belőle a 32 bites 80386-os lapka.

A SPARC fejlődésében csak 1995-ben következett be komoly szakítás a múlttal, amikor is kifejlesztették a SPARC architektúra 9-es verzióját, amely egy valódi 64 bites architektúra, 64 címbittel és 64 bites regiszterkészlettel. Az **UltraSPARC I** volt 1995-ben az első Sun-munkaállomás, amely a V9 (Version 9) architektúrára épült (Tremblay és O'Connor, 1996). Annak ellenére, hogy 64 bites, binárisan kompatibilis volt az akkori 32 bites SPARC gépekkel.

Az UltraSPARC új területeket is megcélzott. Amíg a korábbi gépeket alfanumerikus adatok kezelésére tervezték, szövegszerkesztők és táblázatkezelők futtatására szánták, addig az UltraSPARC-ot tervezői kezdetektől fogva képek, hang, video és általában multimédia kezelésére szánták. A 64 bites architektúra és egyéb újdonságok mellett 23 új utasítás is megjelent, többek között a képpontok (pixelek) 64 bites szavakba történő be-, illetve kicsomagolására, képek skálázására és forgatására, adatblokkok mozgatására, valamint videók valós idejű be- és kitömörítésre. Az Intel MMX-utasításaihoz hasonlóan ez az ún. **VIS (Visual Instruction Set, vizuális utasításkészlet)** utasításkészlet biztosítja a gép általános multimédia-képességeit.

Az UltraSPARC komoly alkalmazásokat is megcélzott, mint például több tucat processzort tartalmazó webszerverek, akár 8 TB [1 TB (terabájt) = 10^{12} bájt] fizikai memóriával. Kisebb változatai azonban noteszgépekbe is megfelelők.

Az UltraSPARC I-et az UltraSPARC II, az UltraSPARC III, majd az UltraSPARC IV követte. Ezek a modellek elsősorban az órajelsebességben térnek el, de minden egyes fejlesztésbe néhány új tulajdonság is bekerült. Könyvünkben a SPARC architektúra tárgyalása során elsősorban a 64 bites V9 UltraSPARC III Cu lesz a példa. Az UltraSPARC IV lényegében egy kétprocesszoros változat, melyben két UltraSPARC III kapott helyet egyetlen CPU lapkán, és osztozik ugyanazon a memórián. Erről később még szólunk, amikor a multiprocesszoros rendszereket tárgyaljuk a 8. fejezetben.

1.4.3. A 8051

Harmadik példánk nagyban eltér az elsőtől (a Pentium 4-től, amelyet a személyi számítógépekben használnak) és a másodiktól (az UltraSPARC III-tól, amelyet szerverekben alkalmaznak). Ez a 8051-es, amely beágyazott rendszerekben használatos. A 8051-es története 1976-ban kezdődött, amikor a 8 bites 8080-as már két éve a piacon volt. A készülékgyártók elkezdték beépíteni a 8080-ast berendezéseikbe, de hogy egy teljes rendszert építhessenek, szükségük volt a 8080-as CPU lapkára, egy vagy több memórialapkára, valamint egy vagy több B/K lapkára. Ennek a legalább háromféle lapkának és azok összekapcsolásának költsége számottevő volt, és így a számítógépek beágyazott rendszerekben történő alkalmazását az alapvetően nagy és drága berendezésekre korlátozta. Sok gyártó arra kérte az Intelt, hogy a költségek csökkentése érdekében az egész számítógépet (CPU, memória és B/K) integrálja egyetlen lapkára.

Az Intel válaszlépésként elkészítette a 8748-as lapkát, a 17 000 tranzisztoros mikrovezérlőt, amely tartalmazott egy 8080-hoz hasonló processzort, 1 KB csak olvasható memóriát a program számára, egy 64 bájtos írható-olvasható memóriát a változóknak, egy 8 bites időzítőt és 27 B/K vonalat a kapcsolók, nyomógombok és lámpák vezérlésére. A lapka, bár kezdetleges volt, kereskedelmi sikert hozott, melynek hatására az Intel 1980-ban előrukkolt a 8051-essel. Az új lapkán 60 000 tranzisztor biztosította a sokkal gyorsabb CPU-t, továbbá 4 KB csak olvasható memóriát, 128 bájt írható-olvasható memóriát, 32 B/K vonalat, egy soros portot és két 16 bites időzítőt tartalmazott. Ezt hamarosan követték az Intel által **MCS–51 családnak** keresztelt sorozat tagjai, melyeket az 1.14. ábrán mutatunk be.

Lapka	Programmemória	Memóriatípus	RAM	Időzítők	Megszakítások
8031	0 KB		128	2	5
8051	4 KB	ROM	128	2	5
8751	8 KB	EPROM	128	2	5
8032	0 KB		256	3	6
8052	8 KB	ROM	256	3	6
8752	8 KB	EPROM	256	3	6

1.14. ábra. *Az MCS–51 család tagjai*

Ezen lapkák mindegyike csak olvasható memóriát használ a programok számára és egy kevés írható-olvasható memóriát, azaz **RAM**-ot (**Random Access Memory, véletlen elérésű memória**) az adatok tárolására. A 8031-esben és a 8032-esben a programtároló kívül helyezkedik el, így szükség esetén 8 KB-nál többet is lehet használni. A 3. fejezetben tanulmányozzuk a **ROM**-ot (**Read Only Memory, csak olvasható memória**) és az **EPROM**-ot (**Erasable Programmable ROM, törölhető, programozható ROM**). Pillanatnyilag elég annyit tudnunk, hogy a tényleges termékekben használt 8051-es és 8052-es egylapkás mikrovezérlők. Minden köteget a vásárló (például egy berendezéseket gyártó vállalat) igényei szerint, egyedileg gyártanak le, és az már tartalmazza a vásárló által biztosított programot is.

Ahhoz, hogy kifejleszthesse a szoftvert, a megrendelőnek szüksége van egy fejlesztőrendszerre. Itt jelentek meg a 8751-es és a 8752-es. Ezek ugyan sokkal drágábbak, mint a 8051-es és a 8052-es, de a vásárló programozhatja őket szoftverellenőrzési céljaira. Ha hibát találnak a programkódban, törölhetik a 8751-es vagy a 8752-es lapkát ultraibolya sugárzásnak kitéve. Ezután pedig beégethetik az új programot. Amikor elkészül a szoftver, eljuttatják a lapka gyártójához, aki azután legyártja a speciális kódot tartalmazó 8051-eseket vagy 8052-eseket.

Architektúra, interfész és programozás tekintetében az MCS–51 család összes tagja nagyon hasonló. Az egyszerűség kedvéért többnyire a 8051-esre hivatkozunk, de ahol szükséges, rámutatunk a többi lapka eltéréseire.

Egyesek számára elég furcsának tűnhet, hogy egy több mint 20 esztendős 8 bites lapkát még használnak, de nagyon komoly oka van ennek. Évente mintegy 8 milliárd mikrovezérlőt adnak el, és ez a szám gyorsan növekszik. Ezek a számok nagyságrendekkel nagyobbak, mint a Pentiumok eladási számai. 2001 előtt a 8 bites mikrovezérlők eladása éves szinten elmaradt a 4 bites mikrovezérlőkétől. Manapság a 8 bites mikrovezérlők túltesznek az összes többin együttvéve, és közülük az MCS–51-es a legnépszerűbb család. A beágyazott rendszerek egyre növekvő fontosságát figyelembe véve, minden számítógép-architektúrát tanulmányozó ember számára elengedhetetlen, hogy az ezekben alkalmazott lapkákat, köztük az egyik legnépszerűbbet, a 8051-est megismerje.

A 8051-es sikerének számos oka van. Először is az ára. A megrendelt mennyiségtől függően már darabonként 10-15 centért is megkaphatjuk, nagyobb tételben pedig akár még kevesebbért. Ezzel szemben a 32 bites mikrovezérlő gyakran 30-szor annyiba kerül, a 16 bites pedig árát tekintve valahol a kettő között foglal helyet. Egy versengő piacon egy 50 dollár alatti termék esetén néhány dollár lefaragása a gyártási költségekből meghatározó hatással lehet az árakra és az eladásokra. A 8051-es nagy népszerűségének legfőbb oka pontosan az, hogy nagyon olcsó.

Másrészt, több mint féltucat vállalat készít 8051-eseket az Intel licencei alapján. Termékeik széles sebességtartományt fednek le, az eredeti 12 MHz-estől a 100 MHz-es változatokig, és számos eltérő gyártási és tokozási technológiát alkalmaznak. Ez a verseny nemcsak az árakat tartja alacsonyan, hanem a nagy megrendelők is sokkal jobban érzik magukat, ha nem kell egyetlen beszállítóra hagyatkozniuk.

Harmadsorban, mivel a 8051-es már olyan régóta jelen van, hatalmas mennyiségű szoftver létezik rá, beleértve assemblereket, fordítókat C és egyéb nyelvekhez, különféle könyvtárakat, hibakeresőket, szimulátorokat, tesztelő szoftvereket és sok mást. Számos teljes fejlesztőrendszer található a piacon, amelyekkel felgyorsítható a beágyazott hardver és szoftver fejlesztése. Végül, nagyszámú, a 8051-est jól ismerő programozó és hardvermérnök között könnyű jól képzett munkaerőt találni.

Ez a népszerűség önmagát élteti. A beágyazott rendszerek iránt érdeklődő kutatók gyakran éppen elterjedtsége miatt választják munkájukhoz, például egy új energiatakarékos technológia tesztelésére (Martin és társai, 2003) vagy hibatűrési vizsgálatokra (Lima és társai, 2002) a 8051-est.

A 8051-ről is rengeteg információ található az interneten. Jó kiindulópont lehet a *www.8051.com*. Ezen kívül még napjainkban is jelennek meg könyvek a témában (Ayala, 2005; Calcutt és társai, 2004; MacKenzie és társai, 2005; Mazidi és társai, 2005).

1.5. Mértékegységek

A félreértések elkerülése végett érdemes külön megjegyeznünk, hogy könyvünkben, mint a számítástechnikában általában, metrikus egységeket használunk a hagyományos angolszász egységek (a furlong-stone-fornight rendszer) helyett. Az alapvető metrikus előtagokat az 1.15. ábrán soroltuk fel. Az előtagokat többnyire kezdőbetűjükkel rövidítjük, az egynél nagyobb egységeket pedig nagybetűkkel (KB, MB stb.). Egyetlen kivétel (történelmi okokból) a kbps a kilobit/másodperc (kilobit/s) esetében. Így az 1 Mbps kommunikációs vonal 10^6 bitet továbbít másodpercenként, és a 100 psec (vagy 100 ps) óra 10^{-10} másodpercenként ketyeg. Mivel a milli és a micro is m betűvel kezdődik, választani kellett. Rendszerint m jelenti a millit és μ (a görög mű betű) jelöli a mikrót.

Hatvány	Explicit	Előtag	Hatvány	Explicit	Előtag
10^{-3}	0,001	milli	10^{3}	1000	Kilo
10^{-6}	0,000001	micro	10^{6}	1000000	Mega
10^{-9}	0,000000001	nano	10^{9}	1000000000	Giga
10^{-12}	0,000000000001	pico	10^{12}	1000000000000	Tera
10^{-15}	0,000000000000001	femto	10^{15}	1000000000000000	Peta
10^{-18}	0,000000000000000001	atto	10^{18}	1000000000000000000	Exa
10^{-21}	0,000000000000000000001	zepto	10^{21}	1000000000000000000000	Zetta
10^{-24}	0,000000000000000000000001	yocto	10^{24}	1000000000000000000000000	Yotta

1.15. ábra. *Az alapvető metrikus előtagok*

Megjegyezzük, hogy a memória, lemez, fájl és adatbázisok mérete esetén a mindennapi ipari gyakorlatban az egységek kicsit mást jelentenek. Itt a kilo jelentése 2^{10} (1024), nem pedig 10^3 (1000), mivel a memóriák mérete mindig a kettő hatványa. Így 1 KB memória 1024 bájtból, nem pedig 1000 bájtból áll. Hasonlóan, 1 MB memória 2^{20} (1048576) bájtból, 1 GB memória 2^{30} (1073741824) bájtból, 1 TB

memória pedig 2^{40} (1 099 511 627 776) bájtból áll. Az 1 kbps kommunikációs vonal azonban 1000 bitet továbbít másodpercenként és a 10 Mbps LAN másodpercenkénti 10 000 000 bites sebességgel működik, mert ezek a sebességek nem a kettő hatványai. Sajnálatos módon sokan hajlamosak ezt a két rendszert összekeverni, főleg lemezek méreténél. A félreértések elkerülése végett könyvünkben a KB, MB, GM és TB jelölést rendre a 2^{10}, 2^{20}, 2^{30} és 2^{40} bájtra, míg a kbps, Mbps, Gbps és Tbps jelölést rendre a másodpercenkénti 10^3, 10^6, 10^9 és 10^{12} bps-re használjuk.

1.6. Könyvünk tartalmáról

Könyvünk a többszintű számítógépekről (a mai számítógépek elsöprő többségéről) és ezek felépítéséről szól. Négy szintet fogunk részletesen tárgyalni: a digitális logika szintjét, a mikroarchitektúra szintjét, az ISA-szintet és az operációs rendszer gép szintjét. Mindegyiknél elsősorban azt vizsgáljuk, hogy mi a szint meghatározó tervezési elve (és miért pont ez), milyen utasítás- és adattípusok használhatók, hogyan szerveződik a memória- és a címzési rendszer, valamint mi a szint megvalósítási módszere. Az ezeket és a hasonló kérdéseket tárgyaló tudományágat a számítógépek felépítése vagy a számítógépek architektúrája néven ismerjük.

Elsősorban az elveket fogjuk hangsúlyozni, és kevésbé foglalkozunk a részletekkel és a matematikai háttérrel. Néhány példánk ezért rendkívül leegyszerűsített, hogy a hangsúly az elveken legyen, ne pedig a részleteken.

A könyvünkben bemutatott elvek gyakorlati alkalmazásának lehetőségére és tényleges alkalmazására a Pentium 4, az UltraSPARC III és a 8051 szolgál majd állandó példaként. Több okból választottuk pont ezeket. Először is mindegyik széles körben használt, és az olvasó is valószínűleg hozzáfér legalább egyikükhöz. Másodszor, architektúrájuk egyedi, így alkalmasak az összehasonlításra, és bátorítják a „milyen alternatívák vannak?" hozzáállást. A csak egy géppel foglalkozó könyvek gyakran olyan érzést kelthetnek az olvasóban, hogy „lám, így kell megtervezni egy gépet", holott a tervezők számos kompromisszumra és sok kérdésben önkényes döntésre kényszerülnek. Az olvasót arra bátorítjuk, hogy ezeket és minden egyéb gépet kritikus szemmel tanulmányozzon, próbálja megérteni, miért éppen úgy vannak a dolgok ahogy, és hogyan lehetett volna azokat másként megcsinálni, ne pedig csak elfogadja a dolgokat olyanoknak, amilyenek.

Mindjárt az elején tisztáznunk kell, hogy ez a könyv nem a Pentium 4, az UltraSPARC III vagy a 8051 programozásáról szól. Ezeket a gépeket a megfelelő helyeken szemléltetési célokra használjuk, a teljesség igénye nélkül. Mélyreható ismeretekért az olvasó forduljon a gyártók kiadványaihoz.

A 2. fejezet a számítógépek fő komponenseivel – processzor, memória és bemeneti/kimeneti egységek – foglalkozik. A rendszerek felépítéséről ad áttekintést, és bevezetőül szolgál a további fejezetekhez.

A 3., 4., 5. és a 6. fejezet egy-egy, az 1.2. ábra szerinti szinttel foglalkozik. Alulról felfelé haladunk, ahogyan hagyományosan a számítógépeket is tervezték. A k. szint tervezését nagymértékben meghatározzák a $k-1$. szint tulajdonságai, ezért egyik szint sem érthető igazán az alsóbb szint – amely annak kialakítását ösztönözte – alapos megértése nélkül. Didaktikailag is logikusabb az egyszerűbb alsó szintek felől a bonyolultabb magasabb szintek felé haladni, mint fordítva.

A 3. fejezet a digitális logika szintjéről, a gép valódi hardveréről szól. Tárgyaljuk a kapukat és ezekből az áramkörök felépítésének módját. A digitális áramkörök elemzésének egyik eszközét, a Boole-algebrát is megismerjük. A sínek, különösen az elterjedt PCI sín is a fejezet tárgya. Számos ipari példát mutatunk be, többek között a már említett három állandó példánkat.

A 4. fejezet bemutatja a mikroarchitektúra szintjét és ennek vezérlését. A szint funkcióinak fő feladata a fölötte lévő 2. szint utasításainak értelmezése, ezért erre a témára és az ezt illusztráló példákra koncentrálunk. A fejezetben néhány konkrét gép mikroarchitektúra szintjét is megismerjük.

Az 5. fejezet a számítógépgyártók által a gép nyelveként hirdetett ISA-szint ismertetője. Itt ismertetjük részletesen példaként használt gépeinket.

A 6. fejezetben az operációs rendszer gép szintjén szokásos utasításokat, memóriaszervezést és vezérlési szerkezeteket ismertetjük. Az itt használt példák a nagyobb Pentium 4-es kiszolgáló rendszereken népszerű Windows XP és az UltraSPARC III-on használt UNIX.

A 7. fejezet az assembly nyelv szintjéről szól. Magát a nyelvet és az assembler meneteit is ismerteti. Itt érintjük a programszerkesztés kérdéskörét is.

A 8. fejezet napjaink egyre fontosabb témakörét, a párhuzamos számítógépeket tárgyalja. Egyes párhuzamos gépekben több CPU van, és ezek közös memóriát használnak. Másokban több CPU van, de közös memória nélkül. Némelyek szuperszámítógépek vagy lapkára integrált rendszerek, míg mások COW-k (munkaállomás-klaszterek).

A 9. fejezetben témakörök szerint csoportosítva, megjegyzésekkel ellátott ajánlott irodalmat és az irodalmi hivatkozások betűrendes listáját találjuk. Ez a könyv legfontosabb fejezete.

1.7. Feladatok

1. Saját szavaival magyarázza meg a következő fogalmakat.
   a) Fordító
   b) Értelmező
   c) Virtuális gép
2. Mi a különbség az értelmezés és a fordítás között?
3. Elképzelhető-e, hogy egy fordító a mikroarchitektúra szintjére fordítson az ISA-szint helyett? Soroljon fel érveket mellette és ellene.
4. El tud-e képzelni olyan többszintű számítógépet, amelyben a digitális logika szintje és az eszközszintek nem a legalsó szintek? Indokolja véleményét.
5. Legyen a többszintű gépünk minden szintje különböző. Minden egyes szint utasításai legyenek m-szer hatékonyabbak, mint az alsóbb szint utasításai, azaz egy r szintű utasítás m darab $r-1$ szintű utasítás munkáját végzi. Továbbá téte-

lezzük fel, hogy n darab r szintű utasítás szükséges egyetlen $r + 1$ szintű utasítás végrehajtásához. Ha egy 1. szintű program futási ideje k másodpere, mennyi ideig futna az ezzel ekvivalens program a 2., 3. és 4. szinten?

6. Az operációs rendszer gép szintjének néhány utasítása azonos az ISA nyelv utasításaival. Ezeket az utasításokat közvetlenül a mikroprogram hajtja végre, és nem az operációs rendszer. Az előző feladatra adott válaszának tükrében mit gondol, vajon miért ez a helyzet?
7. Legyen a gépünknek az 1., 2. és a 3. szintje egyaránt értelmező. Mindegyik értelmezőnek n utasításra van szüksége, hogy beolvasson, megvizsgáljon és végrehajtson egy utasítást. Egy 1. szintű utasítás végrehajtási ideje k nanoszekundum. Mennyi a 2., 3. és 4. szintű utasítások végrehajtási ideje?
8. Milyen értelemben azonos a hardver és a szoftver, és milyen értelemben nem?
9. Babbage differenciagépének volt egy rögzített programja, amelyet nem lehetett módosítani. Vajon ez lényegében ugyanaz, mint ahogy nem módosítható a CD-ROM? Indokolja válaszát.
10. Neumann János elveinek egyik következménye, hogy a memóriában tárolt program ugyanúgy módosítható, mint az adatok. Tud mondani példát, amelyben ez a lehetőség hasznos lehetett? (*Tipp:* gondoljon tömbökön elvégzett műveletekre.)
11. A 360-as sorozat 75-ös modellje 50-szer gyorsabb a 30-as modellnél, a ciklusideje mégis csak ötször gyorsabb. Hogyan magyarázza az eltérést?
12. Az 1.5. és az 1.6. ábra két alapvető architektúrát mutat be. Részletezze, hogy hogyan folyhat a bemenet/kimenet ezeken a rendszereken. Melyiket tartja jobbnak az egész rendszer teljesítményére nézve?
13. Tegyük fel, hogy az Egyesült Államokban élő 300 millió ember naponta teljesen elfogyaszt két csomag, RFID címkével ellátott terméket. Mennyi RFID címkét kell évente készíteni az igény kielégítésére? Címkénként egy penny költséggel számolva mennyi az összes címke költsége? A GDP nagyságát ismerve vajon ez a pénzmennyiség akadályozza-e a címkék használatát minden egyes eladásra kínált csomagon?
14. Nevezzen meg három olyan készüléket, amelyek beépített CPU-val működhetnek.
15. Valamikor 0,1 mikron átmérőjű volt a mikroprocesszor egy tranzisztora. Moore szabálya szerint mennyi volt az átmérője egy egy évvel későbbi modellben?
16. Azt a jogi kérdést, hogy ki fedezte fel a számítógépet, 1973 áprilisában döntötte el Earl Larson bíró, aki az ENIAC szabadalmaira szert tevő Sperry Rand Corporation benyújtotta szabadalombitorlási pert vizsgálta. Sperry Rand álláspontja szerint mindenki, aki számítógépet épített, jogdíjjal tartozik neki, mivel ő volt a kulcsfontosságú szabadalmak tulajdonosa. Az eset 1971 júniusában került bíróság elé, ahol több mint 30 000 bizonyítékot mutattak be, és a bírósági jegyzőkönyv több mint 20 000 oldalas volt. Tanulmányozza az esetet még körültekintőbben az interneten elérhető terjedelmes mennyiségű információra építve, és írjon beszámolót, melyben az esetet technikai szempontból tárgyalja. Pontosan mit is szabadalmaztatott Eckert és Mauchley, és miért érezte úgy a bíró, hogy rendszerük Atanasoff korábbi munkáján alapult?

17. Válasszon három személyt, akikről úgy gondolja, napjaink számítógép-hardverének megalkotásában a legmeghatározóbbak voltak. Írjon rövid beszámolót közreműködésükről, mutassa be azt is, hogy miért őket választotta.
18. Ismételje meg az előző feladatot a számítógép-szoftverre vonatkozóan.

2. Számítógéprendszerek felépítése

Egy digitális számítógép egymással összekapcsolt processzorok, memóriák és bemeneti/kimeneti egységek rendszere. Ez a fejezet alapvető ismereteket nyújt e három komponens és kapcsolataik megismeréséhez, biztosítja a hátteret az egyes szintek részletes tanulmányozásához a következő öt fejezetben. A processzorok, memóriák és a bemeneti/kimeneti egységek olyan kulcsfogalmak, amelyek minden szinten megjelennek majd, így a számítógépek felépítésének tanulmányozását ezzel a háromval kezdjük.

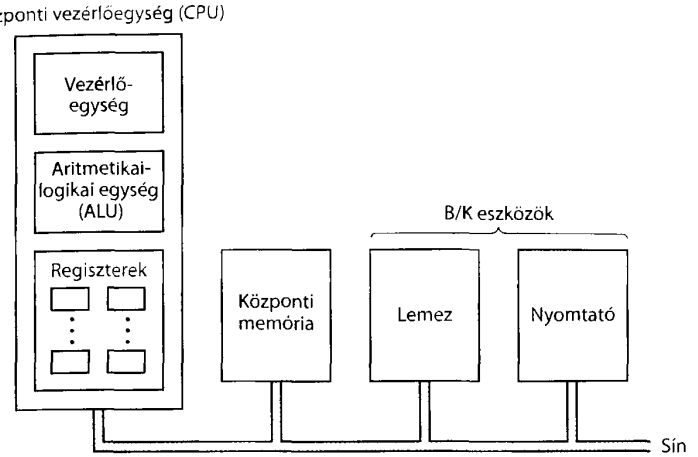
2.1. Processzorok

Egy egyszerű sínalapú számítógép felépítése látható a 2.1. ábrán. A **CPU (Central Processing Unit, központi feldolgozóegység)** a számítógép „agya". Feladata az, hogy a központi memóriában tárolt programokat végrehajtsa úgy, hogy a programok utasításait egymás után beolvassa, értelmezi és végrehajtja. Az egyes részegységeket egy **sín (bus)** köti össze, amely címek, adatok és vezérlőjelek továbbítására szolgáló párhuzamos vezetékköteg. A sín lehet a CPU-t tekintve külső, amely összekapcsolja azt a memóriával és a B/K egységekkel, illetve lehet belső, ahogy azt hamarosan látni fogjuk.

A CPU több különálló részegységből áll. A vezérlőegység feladata az utasítások beolvasása a központi memóriából és az utasítások típusának megállapítása. Az aritmetikai-logikai egység a program utasításainak végrehajtásához szükséges műveleteket végez, mint például az összeadás vagy a logikai ÉS.

A CPU egy kisméretű, gyors memóriát is tartalmaz, amelyben részeredményeket és bizonyos vezérlőinformációkat tárol. Ez a memória több regiszterből áll, mindegyiknek meghatározott mérete és funkciója van. Legtöbbször az összes regiszter azonos méretű. Minden regiszter képes tárolni egy számot, amelynek az értéke kisebb a regiszter mérete által meghatározott maximumnál. A regisztereket nagy sebességgel lehet olvasni és írni, mivel a CPU-n belül vannak.

A legfontosabb regiszter az **utasítás- vagy programszámláló (Program Counter, PC)**, amely a következő végrehajtandó utasítás címét tartalmazza. Az „utasításszámláló" név kissé félrevezető, mert semmit sem *számlálunk meg* vele, ennek ellenére az elnevezés általánosan elterjedt. Fontos még az **utasításregiszter (Instruction Register, IR)**, amely az éppen végrehajtás alatt levő utasítást tartalmazza. A legtöbb számítógép még számos egyéb regisztert is tartalmaz, ezek némelyike általános célú, míg mások speciális célúak.



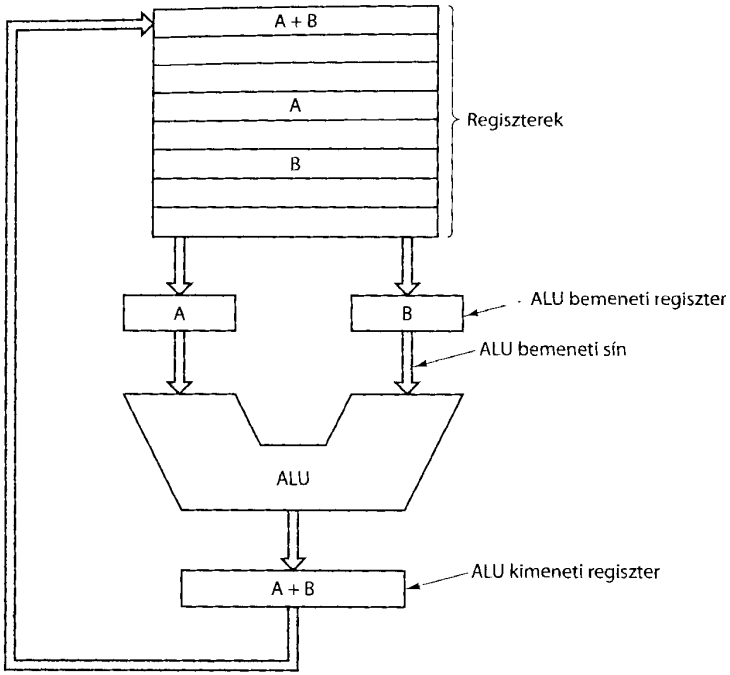
2.1. ábra. *Egy egyszerű, egy processzorból és két B/K egységből álló számítógép felépítése*

2.1.1. A CPU felépítése

Egy tipikus Neumann-elvű számítógép egy részének belső felépítése a 2.2. ábrán látható részletesebben. Ez a rész az ún. **adatút (data path)**, amelynek részei a regiszterek (tipikusan 1-től 32-ig), az **aritmetikai-logikai egység (ALU, Arithmetic Logic Unit)** és az ezeket összekötő néhány sín. A regiszterek két ALU bemeneti regiszterbe csatlakoznak, ezeket az ábrán *A*-val és *B*-vel jelöltük. Ezek a regiszterek tárolják a bementi adatokat, amíg az ALU más számításokon dolgozik. Az adatút minden számítógépben nagyon fontos, ezért hosszasan fogjuk tárgyalni a könyv fejezeteiben.

Maga az ALU a bemenő adatokon összeadást, kivonást és egyéb egyszerű műveleteket végez, és az eredményt a kimeneti regiszterbe teszi. Ennek a kimeneti regiszternek a tartalma visszaírható egy regiszterbe. Később, ha szükséges, a regiszter tartalma beírható (azaz eltárolható) a memóriába. Nem minden számítógép terve tartalmazza az *A*, *B* és a kimeneti regisztert. Az ábrán az összeadást mutattuk be.

A legtöbb utasítás a következő két kategória egyikébe sorolható: regiszter-memória vagy regiszter-regiszter. A regiszter-memória utasítások segítségével tölthetünk át szavakat a memóriából regiszterekbe, ahol a soron következő utasítások például az ALU bemeneteként használhatják. (A „szavak" a memória és a regisz-



2.2. ábra. *Egy tipikus Neumann-elvű számítógép adatútja*

terek közötti adatforgalom egységei. Egy szó lehet egy egész szám. A memória felépítését e fejezetben később tárgyaljuk.) Más regiszter-memória utasítások segítségével a regiszterek tartalmát írhatjuk vissza a memóriába.

A másik csoportba tartoznak a regiszter-regiszter utasítások. Egy tipikus regiszter-regiszter utasítás vesz két operandust a regiszterekből, elhelyezi őket az ALU bemeneti regisztereibe, az ALU elvégez rajtuk valamilyen műveletet – például összeadást vagy logikai ÉS-t –, majd az eredményt tárolja az egyik regiszterbe. A két operandusnak az ALU-n történő átfuttatásából és az eredmény regiszterbe tárolásából álló folyamatot **adatútciklus**nak nevezzük, ez a legtöbb CPU lelke. Jelentős mértékben ez határozza meg, hogy a gép mire képes. Minél gyorsabb az adatútciklus, annál gyorsabban dolgozik a gép.

2.1.2. Utasítás-végrehajtás

A CPU minden utasítást apró lépések sorozataként hajt végre. Ezek a lépések durván a következők:

1. A soron következő utasítás beolvasása a memóriából az utasításregiszterbe.

2. Az utasításszámláló beállítása a következő utasítás címére.
3. A beolvasott utasítás típusának meghatározása.
4. Ha az utasítás memóriabeli szót használ, a szó helyének megállapítása.
5. Ha szükséges, a szó beolvasása a CPU egy regiszterébe.
6. Az utasítás végrehajtása.
7. Vissza az 1. pontra, a következő utasítás végrehajtásának megkezdése.

A fenti lépéssorozatot gyakran nevezik **betöltő-dekódoló-végrehajtó** ciklusnak, és központi szerepet tölt be minden számítógép működésében.

A központi egység működésének ilyetén leírása nagyon hasonlít egy magyarul írt programra. A 2.3. ábra ezt az informális programot mutatja be egy *interpret*

```
public class Interp {
   static int PC;                        // a PC a következő utasítás címét tartalmazza
   static int AC;                        // az akkumulátor, aritmetikai műveletek elvégzésére
   static int instr;                     // tárolóregiszter az aktuális utasítás tárolására
   static int instr_type;                // az utasítás típusa (opcode)
   static int data_loc;                  // az adat címe, vagy –1, ha nincs adat
   static int data;                      // az aktuális operandust tárolja
   static boolean run_bit = true;        // e bit kikapcsolásával megállítható a gép

public static void interpret(int memory[ ], int starting_address) {
   // Ez az eljárás egy egyszerű, egyetlen memóriaoperandust tartalmazó utasításokkal
   // ellátott gép programjait értelmezi. A gépnek van egy AC regisztere (akkumulátora),
   // ami aritmetikai műveletekhez használható. Az ADD művelet például egy
   // memóriabeli egész számot ad az akkumulátorhoz. Az értelmező addig működik,
   // amíg a HALT utasítás hamis értékűre nem állítja a run_bit értékét. A gépen futó
   // folyamat állapota a memóriából, az utasításszámlálóból, a run_bit-ből és az
   // akkumulátorból áll. A bemenő paraméter a feltöltött memória képe és a kezdőcím.

   PC = starting_address;
   while (run_bit) {
      instr = memory[PC];                          // instr feltöltése a következő utasítással
      PC = PC + 1;                                 // utasításszámláló növelése
      instr_type = get_instr_type(instr);          // utasítás típusának meghatározása
      data_loc = find_data(instr, instr_type);     // adat megkeresése (–1, ha nincs)
      if (data_loc >= 0)                           // ha data_loc = –1, nincs operandus
         data = memory[data_loc];                  // adat beolvasása
      execute(instr_type, data);                   // utasítás végrehajtása
   }
}

private static int get_instr_type(int addr) { ... }
private static int find_data(int instr, int type) { ... }
private static void execute(int type, int data) { ... }
}
```

2.3. ábra. *Egy egyszerű számítógép (Javában írt) értelmezője*

nevű Java metódusként (az Interp osztály eljárásaként) megírva. Az interpretált gépnek két olyan regisztere van, amelyek a felhasználói programok számára láthatók: az utasításszámláló (PC) a következő betöltésre váró utasítás címének követésére, illetve az akkumulátor (AC) a számítások részeredményeinek tárolására. Ezeken kívül vannak belső regiszterei az aktuális utasítás végrehajtás alatti tárolására (instr), az aktuális utasítás típusa (instr_type), az utasítás operandusának címe (data_loc) és maga az operandus (data) számára. Feltételezzük, hogy az utasítások egyetlen memóriacímet tartalmaznak. A megcímzett memóriarekesz tartalmazza az operandust, például az akkumulátorhoz hozzáadandó számot.

Maga a tény, hogy lehetséges olyan programot írni, amely a CPU funkcióit tudja imitálni, azt bizonyítja, hogy a programokat nem szükséges egy áramkörökkel teli dobozzal, vagyis egy „hardver" CPU-val végrehajtatni. Ehelyett elég, ha van egy másik olyan programunk, amelyik az utasításait egyenként beolvassa, értelmezi és végrehajtja. Az olyan programot (lásd 2.3. ábra), amely beolvassa, értelmezi és végrehajtja egy másik program utasításait **értelmező**nek **(interpreter)** nevezzük, ahogyan azt az 1. fejezetben említettük.

A processzoráramkörök és az értelmezők közötti ekvivalenciának fontos következményei vannak a számítógépek felépítésére és a számítógéprendszerek tervezésére nézve. Miután egy új számítógép L nyelvét specifikálták, a tervezők eldönthetik, hogy az L nyelvű programok végrehajtására processzoráramkört építenek, vagy az L nyelvű programokat értelmezni tudó értelmezőt írnak inkább. Ha az értelmező megírása mellett döntenek, annak futtatásához is kell valamilyen számítógép. Bizonyos hibrid megoldások is elképzelhetők, részben hardveres végrehajtással, részben szoftveres értelmezéssel.

Az értelmező a célgép utasításait kis lépésekre bontja. Ennek következtében az értelmezőt futtató számítógép a célgép hardverprocesszoránál sokkal egyszerűbb és olcsóbb lehet. Ez a megtakarítás különösen jelentős, ha a célgépnek nagyszámú összetett utasítása van, az utasítások meglehetősen komplikáltak, sok opcióval rendelkeznek. A megtakarítás lényegében abból ered, hogy a hardvert szoftverrel (az értelmezővel) helyettesítjük, és a hardver megvalósítása többe kerül, mint a szoftveré.

Az első számítógépeknek kicsi, egyszerű utasításkészlete volt. Azonban az egyre nagyobb teljesítményű gépekért folytatott verseny többek között nagyobb teljesítményű egyedi utasításokhoz vezetett. Már nagyon korán felfedezték, hogy összetettebb utasítások alkalmazása esetén a programok végrehajtási ideje sok esetben csökken, annak ellenére, hogy az egyes utasítások végrehajtása több időt vehet igénybe. Jó példák összetettebb utasításokra a lebegőpontos utasítások vagy a tömbelemek közvetlen elérését lehetővé tevő gépi utasítások. Néha csak annyit kellett észrevenni, hogy két utasítás gyakran fordult elő egymás után, így egyetlen új utasítás elvégezheti mindkettő feladatát.

Az összetettebb utasítások előnyösebbek voltak amiatt is, hogy különböző hardvermegoldásokkal néha több utasítást átlapolva vagy más módon párhuzamosítva lehetett végrehajtani. A drága, nagy teljesítményű gépeknél ennek a kiegészítő hardvernek a többletköltsége könnyen indokolható. Így olyan helyzet alakult ki, hogy a drága, nagy teljesítményű gépeknek sokkal több utasítása lett, mint az olcsóbbaknak. Azonban a szoftverfejlesztési költségek emelkedése és az utasítások kompatibilitása iránti követelmények szükségessé tették az összetett utasítások megvalósítását az olcsó gépeken is, ahol az ár fontosabb volt a sebességnél.

Az 1950-es évek végére az IBM (az akkor meghatározó számítógépes cég) felismerte, hogy egyetlen számítógépcsalád kifejlesztése, amelyben a számítógépek mindegyike ugyanazokat az utasításokat hajtja végre, mind az IBM, mind vásárlói számára számos előnnyel jár. Az IBM vezette be az **architektúra** elnevezést az ilyen szintű kompatibilitás jellemzésére. Az új számítógépcsalád tagjainak ugyanaz lenne az architektúrája, de különböző megvalósításaik lennének, amelyek ugyanazokat a programokat tudnák futtatni, árban és sebességben különböznének. De hogyan lehet olyan olcsó számítógépet építeni, amely képes egy nagy teljesítményű, drága gép minden összetett utasítását végrehajtani?

A válasz az interpretálásban rejlik. Ez az először Wilkes javasolta technika lehetővé teszi egyszerű, olcsó számítógépek tervezését, amelyek azonban képesek sokféle utasítás végrehajtására (Wilkes, 1951). Az eredmény az IBM System/360 architektúra lett, kompatibilis gépek árban és teljesítményben közel két nagyságrendet felölelő családja. Közvetlen hardver- (azaz nem interpretált) megvalósítást csak a legdrágább modellekben használtak.

Az interpretált utasításokkal ellátott egyszerű gépeknek egyéb előnye is volt. A legfontosabbak ezek közül:

1. Hibásan implementált utasítások helyszíni javításának vagy akár az alaphardverben előforduló tervezési hibák áthidalásának lehetősége.
2. Lehetőség új utasítások hozzáadására minimális költséggel, akár a számítógép leszállítása után is.
3. Strukturált felépítés, amely lehetővé tette az összetett utasítások hatékony fejlesztését, tesztelését és dokumentálását.

Ahogy a számítógépek piaca robbanásszerű növekedésnek indult az 1970-es években, a számítási képességek gyors növekedésnek indultak, az olcsó gépek iránti kereslet az értelmezőket használó számítógépek tervezését részesítette előnyben. A hardver és az értelmező egy bizonyos utasításkészlethez szabása rendkívül költséghatékony processzortervezési megoldásnak bizonyult. Mivel az alapul szolgáló félvezető-technológia gyorsan fejlődött, az alacsony költségek fontosabbnak bizonyultak a nagyobb teljesítménnyel szemben, így az értelmezőalapú architektúrák tervezése vált megszokottá. Az 1970-es években tervezett új számítógépek majdnem mindegyike, a minigépektől a nagyszámítógépekig, értelmezőalapú volt.

Az 1970-es évek végére az értelmezőt használó egyszerű processzorok nagyon elterjedtté váltak, kivéve a legdrágább, legnagyobb teljesítményű modelleket, mint a Cray–1 és a Control Data Cyber sorozat. Az értelmező alkalmazása kiküszöbölte az összetett utasítások használatából fakadó költségkorlátokat, így az architektúrák elkezdtek sokkal összetettebb utasításokat felfedezni, különösen a felhasznált operandusok megadásának módját illetően.

Ez a fejlődési irány a Digital Equipment Corporation VAX gépével érte el tetőfokát. Ennek több száz utasítása volt, mindegyikben több mint 200 módon lehe-

tett meghatározni az utasítás által használt operandust. Sajnos a VAX-architektúra kezdetektől fogva interpretálást tételezett fel, és nem fordítottak figyelmet egy nagy teljesítményű modell megvalósíthatóságára. Ez a beállítottság nagyon sok, csekély jelentőségű utasítás bevételéhez vezetett, amelyeket aztán nehéz volt közvetlenül végrehajtani. Ez a hiányosság végzetesnek bizonyult a VAX, de végső soron a DEC számára is (a Compaq megvásárolta a DEC-et 1998-ban, a HP pedig a Compaqot 2001-ben).

Habár az első 8 bites mikroprocesszorok nagyon egyszerű utasításkészlettel rendelkező, nagyon egyszerű gépek voltak, az 1970-es évek végére még a mikroprocesszorok is áttértek az értelmezőalapú felépítésre. Ebben az időszakban a mikroprocesszor-tervezők előtt álló egyik legnagyobb kihívás az integrált áramkörökkel elérhető egyre nagyobb bonyolultság kihasználása volt. Az értelmezőalapú megközelítés legnagyobb előnye az volt, hogy egyszerű processzort lehetett tervezni, a bonyolultság nagy része az értelmezőt tároló memóriára korlátozódott. Vagyis egy bonyolult hardver tervét bonyolult szoftveres tervvé lehetett változtatni.

A nagy interpretált utasításkészlettel rendelkező Motorola 68000 sikere és a Zilog Z8000 (hasonlóan nagy utasításkészletű, de interpretálás nélküli) kudarca megmutatta az interpretálás előnyeit egy új mikroprocesszor gyors piacra dobásakor. Ez a siker annál is inkább meglepő, mert a Zilog sokkal jobban kezdett (a Z8000 elődje, a Z80 sokkal népszerűbb volt, mint a 68000-es elődje, a 6800). Természetesen más tényezők is fontosak voltak ennél az esetnél, nem utolsósorban a Motorola lapkagyártó múltja, illetve az, hogy az Exxon (a Zilog tulajdonosa) hosszú ideig olajkitermelő cég volt, és nem lapkagyártó.

Abban az időszakban az a körülmény is kedvező volt az értelmezőalapú számítógépek számára, hogy léteztek gyors, csak olvasható tárak, ún. **vezérlőtár**ak (**control store**) az értelmező tárolására. Tegyük fel, hogy a 68000-es egy tipikus interpretált utasítását az értelmező 10 darab, egyenként 100 ns (nanoszekundum) idejű ún. **mikroutasítás** végrehajtására és 2 darab, egyenként 500 ns idejű memória-hozzáférésre bontotta le. A teljes végrehajtási idő tehát 2000 ns volt, mindösszesze 2-szer annyi, mint amit közvetlen végrehajtással el lehetett érni. Ha a vezérlőtár nem létezett volna, a végrehajtási idő 6000 ns lett volna. Hatszoros lassulást sokkal nehezebb lenyelni, mint egy kétszeres lassulást.

2.1.3. RISC és CISC

Az 1970-es évek során sokat kísérleteztek rendkívül komplex utasításokkal, amit az értelmezők tettek lehetővé. A tervezők megpróbálták bezárni azt a „szemantikai rést", amely a számítógépek képességei és a magas szintű programozási nyelvek követelményei között húzódott. Szinte senki sem gondolt arra, hogy egyszerűbb gépeket tervezzen, éppen úgy, ahogy manapság sem végeznek kutatásokat arra, hogy egyszerűbb operációs rendszereket, hálózatokat, szövegszerkesztőket stb. tervezzenek (talán kár, hogy így van).

Az IBM John Cocke vezette csoportja mégis ebben az irányban indult el, és megpróbálta Seymour Cray néhány ötletét alkalmazni egy nagy teljesítményű miniszámítógépben. Ez a munka vezetett el egy kísérleti miniszámítógéphez, a **801**-hez. Habár az IBM soha nem dobta piacra ezt a gépet, és az eredményeket csak évekkel később publikálták (Radin, 1982), mégis híre ment, és mások is elkezdtek hasonló architektúrákkal kísérletezni.

1980-ban David Patterson és Carlo Séquin vezetésével egy csoport a Berkeley-n olyan VLSI-processzorokat kezdett el tervezni, amelyek nem használtak interpretálást (Patterson, 1985; Patterson és Séquin, 1982). A koncepciójuk a **RISC** nevet kapta, és az első CPU lapkát RISC I-nek, a rövidesen elkészülő másodikat pedig RISC II-nek nevezték el. Kicsit később, 1981-ben a San Franciscó-i öböl másik partján, Stanfordban John Hennessy tervezett és el is készített egy ettől kicsit eltérő lapkát, amelyet **MIPS**-nek nevezett (Hennessy, 1984). Mindkettő továbbfejlesztett változata kereskedelmi forgalomba is került SPARC és MIPS néven.

Ezek az új mikroprocesszorok alapvetően különböztek a kereskedelemben kapható kortársaiktól. Mivel nem kellett visszafelé kompatibilisnek lenniük egyetlen létező termékkel sem, tervezőik úgy választhatták meg az utasításkészletet, hogy a rendszer teljesítménye maximális legyen. Míg a kezdeti hangsúly az egyszerű, gyorsan végrehajtható utasításokon volt, hamar felismerték, hogy a teljesítmény szempontjából a kulcskérdés az, hogy milyen gyorsan lehet az utasításokat egymás **után kiadni** (elindítani). Az utasítás időtartamánál többet számított az, hogy hányat lehetett elindítani egy másodperc alatt.

Amikor először terveztek ilyen egyszerű processzorokat, mindenki figyelmét felkeltette, hogy viszonylag milyen kevés utasításuk van, tipikusan 50 körül. Ez a szám sokkal kisebb volt, mint a megszokott gépek 200 és 300 közötti utasítása, mint például a DEC VAX vagy az IBM-nagygépek esetében. Tulajdonképpen a RISC rövidítés a **Reduced Instruction Set Computer (csökkentett utasításkészletű számítógép)** kifejezésből ered, szemben a CISC, vagyis **Complex Instruction Set Computer (összetett utasításkészletű számítógép)** elnevezéssel, ami egy alig leplezett utalás a VAX-ra, amely abban az időben uralta az egyetemi számítóközpontokat. Manapság már csak kevesen tulajdonítanak nagy jelentőséget az utasításkészlet méretének, de az elnevezések megmaradtak.

Hogy rövidre fogjuk, egy nagy vallási háború kerekedett, amelyben a RISC-hívők támadták a fennálló rendszert (VAX, Intel, IBM-nagygépek). Azt állították, hogy a számítógépek tervezésének legjobb módja, ha kevés egyszerű utasításunk van, amelyek a 2.2. ábra adatútjának egyszeri bejárásával végrehajthatók. Ez tehát azt jelenti, hogy vesszük két regiszter tartalmát, valahogyan kombináljuk ezeket (például összeadjuk vagy ÉS-eljük), végül az eredményt elhelyezzük egy regiszterben. Úgy érveltek, hogy még ha egy CISC-utasítás helyettesítéséhez 4-5 RISC-utasítás kell is, és ha a RISC-utasítások 10-szer gyorsabbak (mivel nem interpretáltak), még mindig a RISC a nyerő. Érdemes még megjegyezni, hogy eddigre a központi memória sebessége csaknem beérte a csak olvasható vezérlőtárak sebességét, így az interpretálás miatti relatív időveszteség nagymértékben nőtt, ezzel is erősen kedvezve a RISC gépeknek.

Azt gondolhatnánk, hogy a RISC technológia, a teljesítményben nyújtott előnyét kihasználva, a RISC gépek (mint a Sun UltraSPARC) kiszoríthatták volna a CISC gépeket (mint az Intel Pentium) a piacról. Semmi ilyesmi nem történt. Vajon miért?

Mindenekelőtt, itt van a visszafelé kompatibilitás kérdése, no meg az a sok milliárd dollár, amit a cégek az Intel-processzoros gépeken futó programokra költöttek. Másodszor, meglepő módon az Intel képes volt alkalmazni ugyanezeket az ötleteket még egy CISC-architektúra esetén is. A 486-ossal kezdődően az Intel-processzorok tartalmaznak egy RISC-magot, amely a legegyszerűbb (és egyben leggyakoribb) utasításokat egyetlen adatútciklus alatt hajtja végre, míg a komplikáltabb utasításokat interpretálja a CISC-elvnek megfelelően. Ennek az az eredménye, hogy a gyakori utasítások gyorsak, míg a kevésbé gyakoriak lassúk. Bár ez a hibrid megközelítés nem olyan gyors, mint egy tiszta RISC-processzor, mégis versenyképes teljesítményre képes, és lehetővé teszi a régi programok módosítás nélküli futtatását.

2.1.4. Korszerű számítógépek tervezési elvei

Most, hogy már több mint húsz év eltelt az első RISC gépek megjelenése óta, bizonyos tervezési elvek alkalmazása, figyelembe véve a mai hardvertechnológia lehetőségeit, a számítógépek tervezésének elfogadott módszerévé vált. Ha alapvető változás állna be a hardvertechnológiában (például egy új gyártási eljárás a memóriák ciklusidejét a CPU ciklusidejénél 10-szer kisebbé tenné), teljesen új helyzet állna elő. Ezért aztán a gépek tervezőinek egyik szemüket mindig az olyan technológiai változásokon kell tartaniuk, amelyek befolyásolhatják a komponensek közötti egyensúlyt.

Elmondhatjuk, hogy létezik a tervezési elveknek egy gyűjteménye, amelyeket időnként **RISC tervezési elveknek** hívnak, és amelyeket ma minden általános célú processzor tervezője igyekszik legjobb képességei szerint követni. Egyéb külső követelmények, mint amilyen egy korábbi architektúrával való visszafelé kompatibilitás, időről időre kompromisszumokat igényelhetnek, de ezeket a tervezési elveket a legtöbb tervező igyekszik betartani. A következőkben a legfontosabbakat tárgyaljuk.

Minden utasítást közvetlenül a hardver hajtson végre

Az összes gyakori utasítást a hardver hajtja végre, ezek nem bonthatók fel interpretált mikroutasításokra. Az interpretációs szint kiküszöbölésével a legtöbb utasítás gyors lesz. A CISC-utasításkészletű számítógépek az összetettebb utasításokat részekre bonthatják, és mikroutasítások sorozataként hajthatják végre. Ez a többletmunka lassítja a gépet, de ritkán előforduló utasítások esetén elfogadható.

Maximalizálni kell az utasítások kiadásának ütemét

A mai számítógépek sok trükköt alkalmaznak teljesítményük növelésére, ezek közül a legfontosabb, hogy megpróbálják egy másodperc alatt a lehető legtöbb utasítás végrehajtását elkezdeni. Végül is, ha másodpercenként el tudunk indítani 500 millió utasítást, van egy 500 MIPS-es processzorunk, függetlenül attól, hogy

mennyi ideig tart még az utasítások befejezése. (A **MIPS** a Millions of Instructions Per Second rövidítése; a MIPS processzor neve erre utaló szójáték.) Ez az elv azt sugallja, hogy a párhuzamosság fontos szerepet játszhat a teljesítmény növelésében, mivel nagyszámú lassú utasítás végrehajtását csak akkor tudjuk rövid időn belül megkezdeni, ha több utasítást tudunk egyszerre végrehajtani.

Az utasítások könnyen dekódolhatók legyenek

Az utasítások indítási ütemének egyik felső korlátja a dekódolásához szükséges idő, amely során megállapításra kerül, hogy az utasításoknak milyen erőforrásokra van szükségük. Minden jól jöhet, ami segít ebben a folyamatban. Például az, ha az utasítások szabályosak, egyforma hosszúak és kevés mezőből állnak. Minél kevesebb utasításformátum van, annál jobb.

Csak a betöltő és tároló utasítások hivatkozzanak a memóriára

Az utasítások részekre bontásának legegyszerűbb módja, ha megköveteljük, hogy az utasítások operandusai regiszterek legyenek, és az eredmények is regiszterekbe kerüljenek.

Az operandusok mozgatása a regiszterek és a memória között külön utasításokkal történhet. Mivel a memóriaműveletek sok időt vehetnek igénybe, ráadásul a késleltetés hossza előre nem ismert, ezeket legjobb más utasításokkal átfedve végrehajtani, amennyiben semmi mást nem tesznek, csak adatokat mozgatnak a regiszterek és a memória között. Ez a megfigyelés azt jelenti, hogy csak a LOAD és STORE utasításoknak szabad a memóriára hivatkozni. Minden más utasítás csak regisztereket használhat.

Sok regiszter kell

Mivel a memóriaműveletek lassúak, sok regiszterre (legalább 32-re) van szükség, hogy egy beolvasott szó mindaddig regiszterben maradhasson, amíg szükség van rá. Nem kívánatos, és el kell kerülni, amennyire csak lehet, hogy kifogyjunk a regiszterekből, és csak azért kelljen a memóriába menteni az adatokat, hogy később visszaolvassuk. Ennek legjobb módja, ha elég regiszterünk van.

2.1.5. Utasításszintű párhuzamosság

A számítógép-tervezők állandóan igyekeznek javítani gépeik teljesítményét. Ennek egyik módja a lapkák gyorsítása az órajel frekvenciájának növelésével, de a nyers erő alkalmazásában rejlő lehetőségek az adott történelmi pillanatban mindig korlátozottak. Emiatt a legtöbb tervező a párhuzamosság (több dolgot végez-

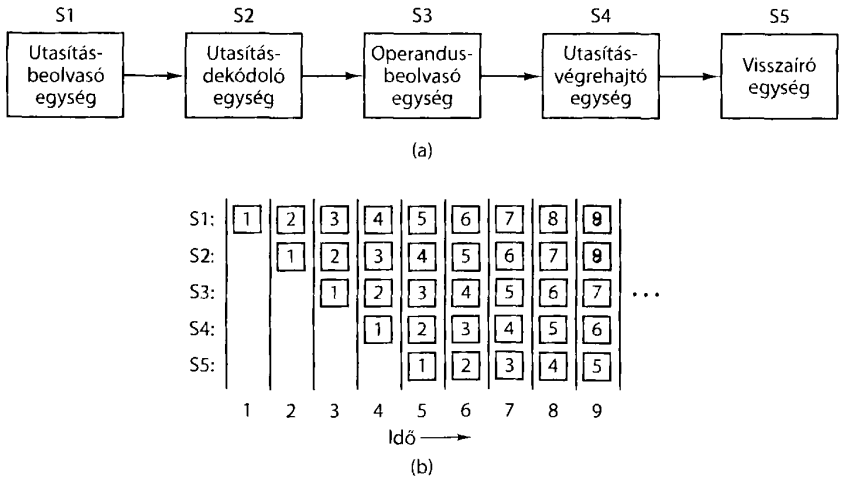
ni egyszerre) kiaknázásában lát lehetőséget adott órajel-frekvencia mellett a még nagyobb teljesítmény elérésére.

A párhuzamosság kétféleképpen lehet jelen: utasításszintű párhuzamosság vagy processzorszintű párhuzamosság formájában. Az előbbiben, az egyes utasításokban rejlő párhuzamosságot használjuk ki, hogy több utasítást tudjunk másodpercenként kiadni. A másik esetben több processzor dolgozik egyszerre ugyanazon a feladaton. Mindkét megközelítésnek megvannak a maga előnyei. Ebben az alfejezetben az utasításszintű párhuzamosságot tekintjük át, a következőben pedig a processzorszintű párhuzamosságot.

Csővezeték

Évek óta ismert, hogy az utasítások végrehajtásának egyik legszűkebb keresztmetszete az utasítások kiolvasása a memóriából. E probléma enyhítésére már az olyan régi számítógépek is, mint az IBM Stretch (1959) képesek voltak előre beolvasni utasításokat, hogy azok rendelkezésre álljanak, amikor szükség van rájuk. Ezeket az utasításokat egy **előolvasási puffer (prefetch buffer)** elnevezésű regiszterkészletben tárolták. Ilyen módon a soron következő utasítást általában az előolvasási pufferből lehetett venni ahelyett, hogy egy memóriaolvasás befejeződésére kellett volna várni.

Lényegében az előolvasás az utasítás végrehajtását két részre osztja: beolvasás és tulajdonképpeni végrehajtás. A **csővezeték** ezt a stratégiát viszi sokkal tovább. Az utasítás végrehajtását kettő helyett több részre osztja (gyakran egy tucatra vagy még többre), minden részt külön hardverelem kezel, amelyek mind egyszerre működhetnek.



2.4. ábra. *(a) Ötfázisú csővezeték. (b) A fázisok állapota az idő függvényében. Az ábrán kilenc órajelciklus látható*

A 2.4. (a) ábra egy öt egységből, más néven **fázis**ból álló csővezetéket szemléltet. Az első fázis beolvassa az utasítást a memóriából, és elhelyezi egy pufferben, amíg szükség nem lesz rá. A második fázis dekódolja az utasítást, meghatározza a típusát és a szükséges operandusokat. A harmadik fázis megkeresi és beolvassa az operandusokat akár regiszterből, akár a memóriából. A negyedik fázis hajtja végre valójában az utasítást, ez tipikusan azt jelenti, hogy az operandusokat átviszi a 2.2. ábra adatútján. Végül az ötödik fázis visszaírja az eredményt a megfelelő regiszterbe.

A 2.4. (b) ábrán láthatjuk a csővezeték működését az idő függvényében. Az első órajel alatt az S1 fázis az első utasításon dolgozik, beolvassa a memóriából. A második órajel alatt az S2 fázis dekódolja az első utasítást, ez idő alatt az S1 fázis már a második utasítást olvassa be. A harmadik órajel alatt az S3 fázis előkészíti az első utasítás operandusait, az S2 fázis dekódolja a második utasítást, míg az S1 fázis beolvassa a harmadik utasítást. A negyedik órajel alatt az S4 fázis végrehajtja az első utasítást, S3 előkészíti a második utasítás operandusait, S2 dekódolja a harmadik utasítást, S1 pedig beolvassa a negyedik utasítást. Végül az ötödik órajel alatt S5 visszaírja az első utasítás eredményét, míg a többi fázis a soron következő utasításokkal van elfoglalva.

A következő hasonlat még világosabbá teszi a csővezeték elvet. Képzeljünk el egy édességgyárat, ahol a torták gyártása és csomagolása szét van választva. Tegyük fel, hogy a csomagolórészlegben van egy hosszú csővezeték, amely mellett öt munkás (feldolgozóegység) sorakozik fel. 10 másodpercenként (órajel) az első munkás egy üres dobozt tesz a szalagra. A doboz a második munkáshoz kerül, aki beletesz egy tortát. Kicsit később a doboz a harmadik munkáshoz érkezik, aki lezárja és leragasztja. A negyedik egy címkét ragaszt rá, ezután az ötödik munkás leveszi a dobozt a szalagról és egy nagy konténerbe teszi, amelyet később majd egy áruházba szállítanak. Alapjában véve ugyanígy működik a számítógépes csővezeték is: minden utasítás (torta) több feldolgozási lépésen megy keresztül, mielőtt elkészülve előbukkan a túlsó végén.

Visszatérve a 2.4. ábra csővezetékéhez, tegyük fel, hogy ennek a gépnek az órajele 2 ns. Ekkor egy utasítás 10 ns alatt jut át az öt fázison. 10 ns utasításokkal első ránézésre úgy tűnhet, hogy a gép 100 MIPS sebességre képes, igazából azonban annál sokkal gyorsabb. Minden órajel (2 ns) alatt befejeződik egy utasítás, ezért a valódi sebesség 500 MIPS, nem 100 MIPS.

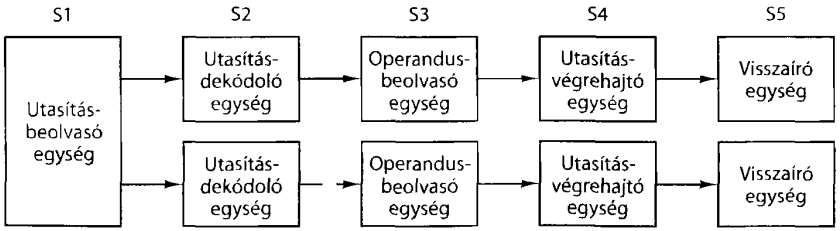
A csővezeték lehetővé teszi, hogy kompromisszumot kössünk **késleltetés** (mennyi ideig tart egy utasítás végrehajtása) és **áteresztőképesség** (hány MIPS a processzor sebessége) között. Ha az órajel T nanoszekundum, és a csővezeték n fázisú, a késleltetés nT nanoszekundum, mivel minden utasítás n állapoton halad keresztül és mindegyikben T ideig tartózkodik.

Mivel minden órajelben egy utasítás befejeződik, és mivel $10^9/T$ az órajelek száma másodpercenként, így a másodpercenként végrehajtott utasítások száma is $10^9/T$. Például, ha $T = 2$ ns, másodpercenként 500 millió utasítás hajtódik végre. Ha az utasítások számát MIPS-ben szeretnénk megkapni, a végrehajtási sebességet el kell osztanunk 1 millióval, ezt kapjuk $(10^9/T)/(10^6) = 1000/T$ MIPS. Elméletileg mérhetnénk a végrehajtási sebességet BIPS-ben (Billion Instruction Per Second, milliárd utasítás másodpercenként) MIPS helyett, de mivel senki sem tesz így, mi sem fogunk.

Szuperskaláris architektúrák

Ha egy csővezeték jó, kettő biztos még jobb. Egy két csővezetékes CPU-terv látható a 2.4. ábra alapján készített 2.5. ábrán. Itt az egyetlen utasítást előolvasó egység két utasítást olvas be egyszerre, majd ezeket az egyik, illetve a másik csővezetékre teszi. A csővezetékeknek saját ALU-juk van, így párhuzamosan tudnak működni, feltéve, hogy a két utasítás nem használja ugyanazt az erőforrást (például regisztert), és egyik sem használja fel a másik eredményét. Ugyanúgy, mint egyetlen csővezeték esetén, a feltételek betartását vagy a fordítóprogramnak kell garantálnia (vagyis a hardver nem ellenőriz és hibás eredményeket ad, ha az utasítások nem kompatibilisek), vagy a konfliktusokat egy kiegészítő hardvernek kell a végrehajtás során felismernie és kiküszöbölnie.

Habár az egyszeres és kétszeres csővezetékeket főként a RISC gépekben használják (a 386-osban és elődeiben ilyen nem volt), a 486-ostól kezdődően az Intel elkezdett csővezetékeket alkalmazni a processzoraiban. A 486-osban egy csővezeték van, a Pentiumban két 5 fázisú, nagyjából a 2.5. ábrának megfelelően, habár a második és harmadik fázis (decode-1 és decode-2) közötti munkamegosztás egy kicsit más, mint a mi példánkban. A fő csővezeték az **u pipeline** tetszőleges Pentium-utasítást végre tud hajtani. A második csővezeték a **v pipeline** csak egyszerű egész műveleteket tud elvégezni (és még egy igen egyszerű lebegőpontos műveletet, például FXCH).



2.5. ábra. *Kettős csővezeték közös utasítás-beolvasó egységgel*

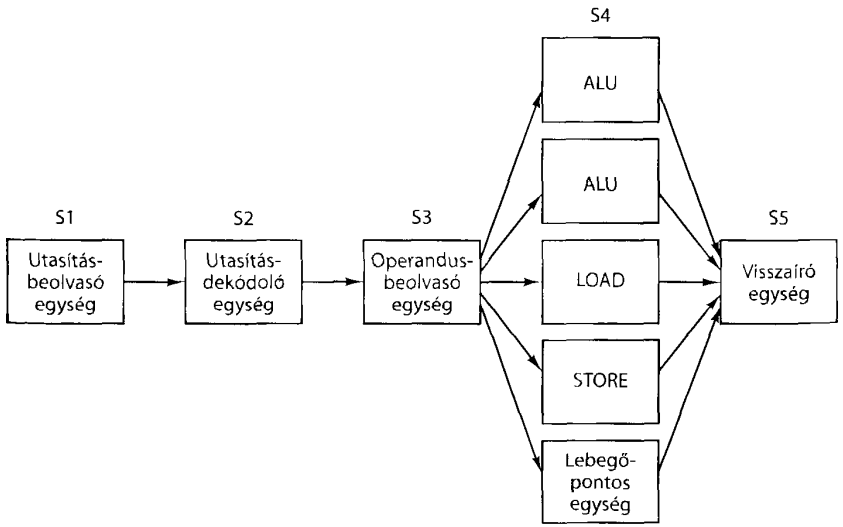
Bonyolult szabályok határozzák meg, hogy két utasítás kompatibilis-e, vagyis végrehajtható-e párhuzamosan. Ha a két utasítás nem elég egyszerű vagy inkompatibilis, csak az egyik kerül végrehajtásra (a fő csővezetéken). A másik utasításhoz egy újabbat olvas be a processzor, majd a folyamat folytatódik. Az utasítások mindig az eredeti sorrendben hajtódnak végre. Kompatibilis utasításpárokat képző speciális Pentium-fordítóprogramok gyorsabb programokat tudnak előállítani, mint a régebbi fordítóprogramok. Mérések azt mutatták, hogy egy Pentium a hozzá optimalizált, egész aritmetikát használó kódot éppen kétszer olyan gyorsan tudta futtatni, mint egy ugyanolyan órajelű 486-os (Pountain, 1993). Ez a gyorsulás teljes egészében a második csővezetéknek tulajdonítható.

A csővezetékek számának négyre emelése még elképzelhető (a számítógépes szakemberek a néprajzkutatókkal ellentétben nem hisznek a hármas számban),

de ekkor már túl sok hardverelemet kell megduplázni. Ehelyett a nagy teljesítményű processzorokban más megoldást alkalmaznak. Az alapötlet az, hogy csak egy csővezetéket használnak, de több **funkcionális egységgel**, ahogyan ez a 2.6. ábrán látható. Például a Pentium 4 az ábrán láthatóhoz hasonló felépítésű. A 4. fejezetben majd részletesebben is tárgyaljuk. A **szuperskaláris architektúra** kifejezés ennek az elrendezésnek a jelölésére 1987-ben született (Agerwala és Cocke, 1987), a gyökerei azonban 30 évvel korábbra a CDC 6600-as számítógépig nyúlnak vissza. A 6600-as 100 ns-ként olvasott be egy utasítást és adta tovább a 10 funkcionális egység valamelyikének párhuzamos végrehajtásra, mialatt a CPU újabb utasítás beolvasásába kezdett.

A „szuperskaláris" definíciója az évek során fejlődött. Ma az olyan processzorok jellemzésére használják, amelyek több – gyakran négy vagy hat – utasítás végrehajtását kezdik el egyetlen órajel alatt. Természetesen egy szuperskaláris CPU-nak több funkcionális egységének kell lennie, amelyek kezelik mindezeket az utasításokat. Mivel a szuperskaláris processzoroknak általában egy csővezetéke van, ezért felépítésük hasonlít a 2.6. ábrán látható modellhez.

Ezt a definíciót használva a CDC 6600 technikailag nem volt szuperskaláris, mivel egy órajel alatt csak egy utasítás végrehajtását kezdte meg. Azonban a hatás lényegében ugyanez volt: az utasítások megkezdését sokkal nagyobb ütemben végzik, mint amilyen ütemben azokat végre lehet hajtani. Nagyon kicsi a különbség két CPU között, ha az egyik 100 ns órajelenként ad ki egy utasítást a funkcionális egységek egy csoportja számára, a másik pedig 400 ns órajelenként négy utasítást ad ki ugyanennek a csoportnak. Mindkét esetben az az alapötlet, hogy az utasítások kiadásának sebessége nagyobb, mint a végrehajtás sebessége, így a terhelés megoszlik a funkcionális egységek között.



2.6. ábra. *Szuperskaláris processzor 5 funkcionális egységgel*

A szuperskaláris processzor elvében implicit módon benne van az a feltételezés, hogy az S3 fázis lényegesen gyorsabban tudja előkészíteni az utasításokat, mint ahogy az S4 fázis képes azokat végrehajtani. Ha az S3 fázis 10 ns-ként produkál egy utasítást, és az összes funkcionális egység végezni tud 10 ns alatt, mindig csak legfeljebb egy fog dolgozni, így az egésznek nincs semmi haszna. Valójában a negyedik fázis funkcionális egységeinek többsége egy órajelnél jóval több időt igényel feladata elvégzéséhez – a memóriához fordulók vagy a lebegőpontos műveleteket végzők biztosan. Ahogy az ábrán is látható, lehet több ALU is az S4 fázisban.

2.1.6. Processzorszintű párhuzamosság

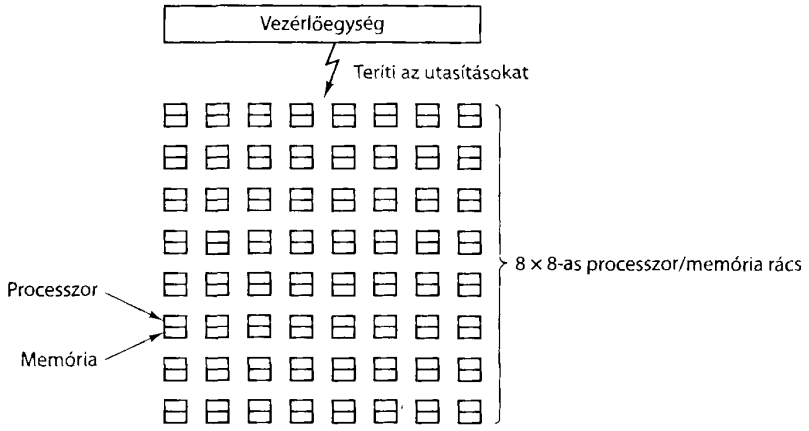
Az egyre gyorsabb számítógépek iránti igény kielégíthetetlennek tűnik. A csillagászok szimulálni akarják a Nagy Bumm első ezredmásodpercében történteket, a közgazdászok modellezni akarják a világpiacot, a tinédzserek pedig virtuális barátaikkal 3 dimenziós interaktív multimédiás játékokat akarnak játszani az interneten. Habár a processzorok egyre gyorsabbak lesznek, előbb vagy utóbb a fénysebesség végességéből adódó korlátokba fognak ütközni, ugyanis ez valószínűleg 20 cm/ns fog maradni a rézdrótban és a fénykábelben függetlenül attól, hogy az Intel mérnökei milyen okosak. A gyorsabb lapkák több hőt is termelnek, aminek az elvezetése szintén problémás.

Az utasításszintű párhuzamosság segít egy kicsit, de a csővezeték és a szuperskaláris működési mód ritkán növeli a sebességet 5–10-szeresnél jobban. Ha 50-szeres, 100-szoros vagy ennél is nagyobb gyorsulást szeretnénk, ennek egyedüli módja, hogy több CPU-t tartalmazó számítógépet tervezünk; a következőkben azt fogjuk áttekinteni, hogyan lehet ilyeneket építeni.

Tömbszámítógépek

Sok fizikai és mérnöki probléma tömbökkel vagy más nagyon szabályos szerkezetű objektumokkal fogalmazható meg. Gyakran ugyanazokat a műveleteket kell egyszerre elvégezni különböző adathalmazokon. A feladatoknak a szabályossága és a szerkezete különösen megfelelővé teszi ezeket párhuzamos feldolgozásra. Két olyan módszer van, amelyeket nagyméretű tudományos problémák gyors megoldására használnak. Habár ez a két séma meglepően hasonló, érdekes módon az egyiket az egyprocesszoros rendszer kiterjesztésének, míg a másikat párhuzamos számítógépnek tekintik.

Egy **tömbprocesszor** nagyszámú egyforma processzorból áll, ezek ugyanazt a műveletsorozatot végzik el különböző adathalmazokon. A világ első tömbprocesszora a University of Illinois 2.7. ábrán látható ILLIAC IV számítógépe volt (Bouknight és társai, 1972). Az eredeti terv szerint egy 4 negyedből álló gépet építettek volna, minden negyedben egy 8 × 8-as négyzethálóban processzor/memória párokkal. Negyedenként egy vezérlőegység adta ki az utasításokat, melyeket a hozzá tartozó processzorok szinkronizálva hajtottak végre, az adatokat mindegyik a saját memóriájából vette (amit egy inicializálási fázisban töltöttek fel). Ez a kialakítás, nyilvánvalóan nagyon eltér a szokásos Von Neumann géptől, és időnként **SIMD (Single Instruction-stream Multiple Data stream)** processzorként hivatkoznak rá. Mivel a valódi költségek a tervezett négyszeresére rúgtak, csak egyetlen negyedet építettek meg, de ez is elérte az 50 megaflop (million floating-point operations per second, millió lebegőpontos utasítás másodpercenként) sebességet. Azt mondják, hogy ha az egész gépet megépítették volna, és az valóban elérte volna a kitűzött 1 gigaflop sebességet, egymaga megduplázta volna a világ akkori számítási kapacitását.



2.7. ábra. *Egy ILLIAC IV típusú tömbszámítógép*

A **vektorprocesszor** a programozó szemszögéből nagyon hasonlít a tömbprocesszorra. Ahhoz hasonlóan nagyon hatékonyan tud egy utasítássorozatot végrehajtani adatelempárokon. A tömbprocesszortól eltérően azonban, minden összeadás egyetlen csővezeték elven működő összeadóegységben zajlik. A Seymour Cray által alapított Cray Research cég napjainkig sok vektorprocesszort gyártott, kezdve az 1974-ben megjelent Cray–1 számítógéppel és folytatva a jelenlegi modellekkel (a Cray Research ma az SGI része).

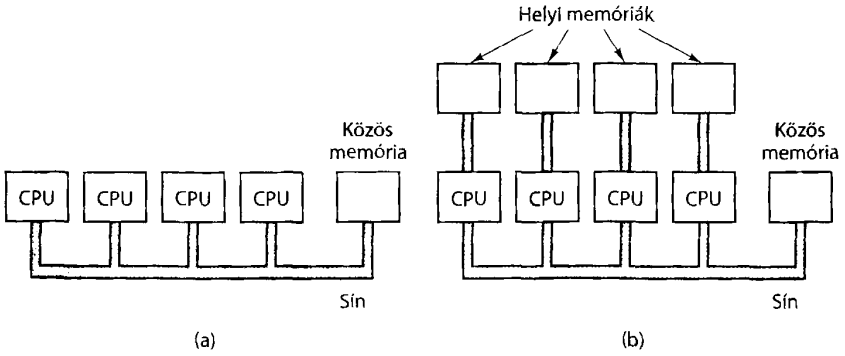
A tömb- és a vektorprocesszorok is adattömbökkel dolgoznak. Mindkettő olyan egyedi utasításokat hajt végre, mint amilyen például két vektor elemeinek páronkénti összeadása. De míg a tömbprocesszorok ezt úgy végzik, hogy a vektor elemszámával megegyező számú összeadóegységet tartalmaznak, a vektorprocesszorok **vektorregisztereket** használnak. Egy vektorregiszter több hagyományos regiszterből áll, ezeket a betöltő utasítás egymás után, sorosan tölti fel a memóriából. Ezután a vektorösszeadó utasítás végrehajtja két ilyen vektor elemeinek páronkénti összeadását úgy, hogy egy csővezetékes összeadóba irányítja a párokat a két vektorregiszterből. A vektorösszeadás eredménye egy újabb vektor, amelyet egy vektorregiszterbe lehet tárolni, vagy közvetlenül fel lehet használni egy újabb vektorművelet operandusaként.

Tömbprocesszorokat jelenleg nem gyártanak, azonban az ötlet egyáltalán nem halt meg. Az MMX és az SSE utasítások, amelyek a Pentium 4 utasításkészletében találhatók, ezt a végrehajtási modellt használják a multimédia-szoftver felgyorsítására. Ebben a tekintetben az ILLIAC IV a Pentium 4 egyik elődjének tekinthető.

Multiprocesszorok

Egy tömbprocesszor feldolgozóegységei nem függetlenek egymástól, mert mindegyiküket egy közös vezérlőegység felügyeli. Az első olyan párhuzamos rendszerünk, amelyben több teljes CPU van, a **multiprocesszor**, egy olyan rendszer, amelyben közös memóriát használó egynél több CPU található, hasonlóan ahhoz, mint amikor egy szobában több ember használ egy közös táblát. Mivel mindegyik CPU írhatja és olvashatja a memória bármely részét, együtt kell működniük (szoftveresen), hogy ne legyenek egymás útjában. Amikor két vagy több CPU rendelkezik azzal a képességgel, hogy szorosan együttműködjenek, mint ahogyan a multiprocesszorok esetében, akkor azokat szorosan kapcsoltaknak nevezik.

Több implementációs séma lehetséges. A legegyszerűbb, ha egyetlen sín van, amelyhez csatlakoztatjuk a memóriát és az összes processzort. Egy ilyen sínalapú rendszert mutat a 2.8. (a) ábra. Sok cég gyárt ilyen rendszereket.



2.8. ábra. *(a) Egysínes multiprocesszor. (b) Multiprocesszor lokális memóriákkal*

Nem kell nagy képzelőerő annak belátására, hogy ha sok gyors processzor próbálja állandóan elérni a memóriát a közös sínen keresztül, az konfliktusokhoz vezet. A multiprocesszorok tervezői sokféle módon próbálták meg csökkenteni a versenyhelyzeteket, és ezáltal növelni a teljesítményt. A 2.8. (b) ábrán látható megoldás minden processzornak biztosít valamekkora saját lokális memóriát, amelyet a többiek nem érhetnek el. Ez a memória felhasználható a programkód és az olyan adatok számára, amelyeket nem kell megosztani a többi processzorral. A lokális memória elérése nem a közös sínen történik, így annak forgalmát jelentősen csökkenti. Más megoldások is elképzelhetők (például gyorsítótár használata).

A multiprocesszoroknak a többi párhuzamos számítógéppel szemben megvan az az előnyük, hogy a közös memória programozási modelljét könnyű használni. Képzeljünk el például egy programot, amely valamilyen szövet mikroszkópos fényképén rákos sejteket keres. A digitalizált fényképet a közös memóriában tárolva minden processzor a fénykép egy számára kijelölt területét fésülheti át. Mivel minden processzor elérheti az egész memóriát, nem okoz problémát, ha valamelyik sejtet vizsgálva – amely a kijelölt régióban kezdődik, de azután a kijelölt határt átlépi, át kell menni az egyik szomszédos területre.

Multiszámítógépek

Habár kevés (≤ 256) processzorból álló multiprocesszorok aránylag könnyen építhetők, nagyokat meglepően nehéz konstruálni. A nehézséget az összes processzor és a memória összekötése jelenti. Ezeknek a nehézségeknek az elkerülésére sok tervező felhagyott a közös memória alkalmazásával, és sok összekapcsolt számítógépből álló rendszereket építenek, amelyeknek csak saját memóriájuk van, közös memóriájuk nincs. Ezeket a rendszereket nevezik **multiszámítógépek**nek. A multiszámítógépek CPU-it időnként **lazán kapcsoltak**nak nevezik, megkülönböztetve őket a multiprocesszorokban található szorosan kapcsolt CPU-któl.

A multiszámítógép processzorai üzenetek küldésével kommunikálnak egymással, ami olyasmi, mint az e-mail, csak sokkal gyorsabb. Nagy rendszerekben nem célszerű minden számítógépet minden másikkal összekötni, ezért 2 és 3 dimenziós rácsot, fákat és gyűrűket használnak. Ennek következtében egy gép valamelyik másikhoz küldött üzeneteinek gyakran egy vagy több közbenső gépen vagy csomóponton kell áthaladniuk ahhoz, hogy a kiindulási helyükről elérjenek a céljukhoz. Mindazonáltal néhány mikroszekundumos nagyságrendű üzenetküldési idők nagyobb nehézség nélkül elérhetők. Közel 10000 processzort tartalmazó multiszámítógépeket is építettek már és vettek használatba.

Mivel a multiprocesszorokat könnyebb programozni, de multiszámítógépeket könnyebb építeni, sok kutató foglalkozik azzal, hogy hibrid rendszerekben a kettő előnyös tulajdonságait ötvözze. Ezek a gépek a közös memória illúzióját próbálják kelteni anélkül, hogy pénzt kellene kiadni a tényleges megépítésükre. A 8. fejezetben részletesen foglalkozunk a multiprocesszorokkal és a multiszámítógépekkel.

2.2. Központi memória

A **memória** a számítógépnek az a része, ahol a programokat és az adatokat tároljuk. Sok informatikus (különösen a britek) a memória helyett szívesebben használja a **tár (store)** vagy **tároló (storage)** megnevezést, bár ez utóbbi egyre inkább a diszk egységet jelenti. Memória nélkül, ahonnan a processzorok az adatokat ki tudják olvasni, és ahova be tudják írni, nem létezhetne tárolt programú digitális számítógép.

2.2.1. Bitek

A memória alapegysége a bináris számjegy, a **bit**. Egy bit egy 0-t vagy egy 1-est tartalmazhat. Ez a lehető legegyszerűbb egység. (Egy olyan eszköz, amely csak nullákat tudna tárolni, aligha képezhetné a memória alapját; legalább két érték kell.)

Az emberek gyakran mondják, hogy a számítógépek azért használnak bináris aritmetikát, mert az „hatékony". Ezen azt értik (ritkán tudatosan), hogy digitális információ valamilyen folytonos fizikai mennyiség, mint például feszültség vagy áramerősség különböző értékeinek szétválasztásával tárolható. Minél több értéket kell megkülönböztetni, annál nehezebb a szomszédos értékeket szétválasztani, és annál kevésbé megbízható a memória. A bináris számrendszer csak két érték megkülönböztetését igényli. Emiatt ez a digitális információ legmegbízhatóbb kódolási formája. A kettes számrendszert nem ismerő olvasóink figyelmébe ajánljuk az A) függeléket.

Némely számítógépet, mint például az IBM-nagygépeket, úgy reklámozzák, hogy a bináris aritmetika mellett decimálist (tízes számrendszerbelit) is tudnak. Ezt a trükköt úgy csinálják, hogy egy decimális számjegyet 4 biten tárolnak az ún. **BCD (Binary Coded Decimal, binárisan kódolt decimális)** kódolással. A négy bit 16 kombinációt ad, a számjegyekhez 0-tól 9-ig elég 10 kombináció, a maradék 6 kihasználatlan. Alább látható az 1944 decimálisan, majd 16 biten binárisan kódolva:

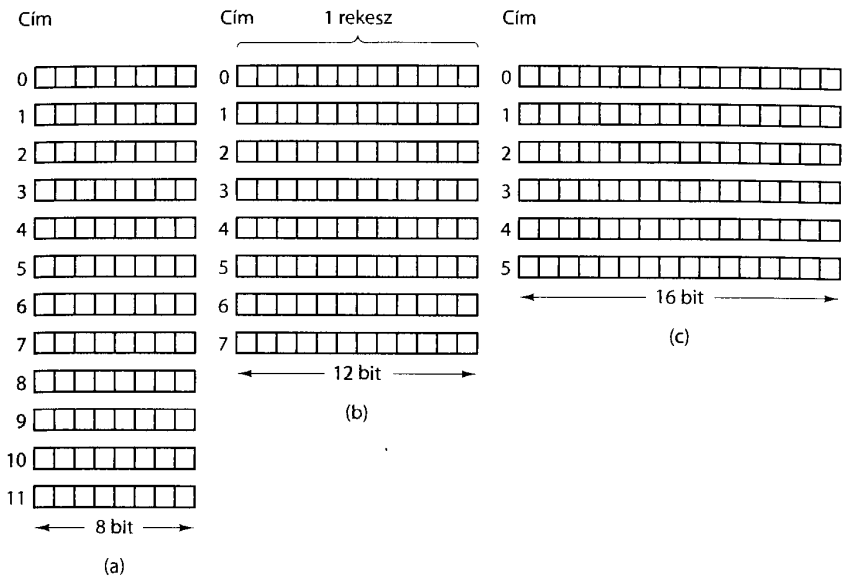
decimális: 0001 1001 0100 0100 bináris: 0000011110011000

Decimális formátumban (BCD) 16 biten 0-tól 9999-ig tudjuk tárolni a számokat, ez összesen csak 10 000 kombináció, míg egy tiszta 16 bites bináris szám 65 536 különböző kombinációt tárolhat. Emiatt mondják azt az emberek, hogy a bináris hatékonyabb.

Képzeljük el azonban, hogy egy nagyon okos fiatal elektronikai mérnök felfedezne egy olyan eszközt, amely nagyon megbízhatóan tudná közvetlenül tárolni a decimális számjegyeket 0-tól 9-ig azáltal, hogy a 0-tól 10 voltig terjedő tartományt tíz intervallumra osztaná. Négy ilyen eszköz 10 000 kombinációt adna. Bináris számok tárolására is lehetne ezeket használni úgy, hogy csak 0-t és 1-est használnánk, de ekkor négy pozíción csak 16 kombinációt tárolhatnánk. Ilyen eszközökkel a decimális rendszer nyilván hatékonyabb lenne.

2.2.2. Memóriacímek

A memóriák **rekeszek**ből (**cellák**ból) állnak, amelyek mindegyike valamilyen információt tárolhat. Minden rekeszhez hozzá van rendelve egy szám, a **rekesz címe**, a programok ezzel hivatkoznak rájuk. Ha egy memóriában n rekesz van, a címek 0-tól n – 1-ig terjednek. A memóriában minden rekeszben ugyanannyi bit van. Ha egy rekesz k bites, a 2^k különböző bitkombináció bármelyikét tárolhatja. A 2.9. ábra egy 96 bites memória három lehetséges beosztását mutatja. Figyeljük meg, hogy (definíció szerint) a szomszédos rekeszek címei eggyel különböznek egymástól.



2.9. ábra. *Egy 96 bites memória háromféle szervezési módja*

A kettes számrendszert (akár nyolcas vagy tizenhatos számrendszerbeli jelöléssel) használó számítógépek a memóriacímeket bináris számokkal fejezik ki. Ha egy cím m bites, a megcímezhető rekeszek száma 2^m. Például a 2.9. (a) ábra egy rekeszére hivatkozó címnek legalább 4 bitesnek kell lennie, hogy a 0 és 11 közötti számokat elő tudja állítani. Elegendő azonban 3 bit a 2.9. (b) és (c) ábrához. A cím bitjeinek száma határozza meg a memória közvetlenül megcímezhető rekeszeinek számát; ez független attól, hogy a rekeszek hány bitesek. Egy 2^{12} darab 8 bites rekeszből és egy 2^{12} darab 64 bites rekeszből álló memória egyaránt 12 címbitet igényel.

Számítógép	Bit/rekesz
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
DCD 3600	48
CCD Cyber	60

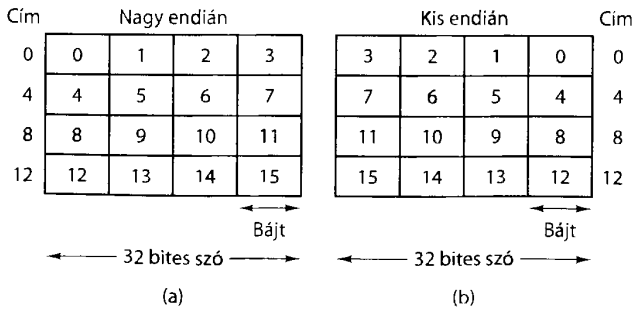
2.10. ábra. *Bitek száma rekeszenként néhány számítógép-történetileg érdekes, kereskedelmi forgalomba került gépben*

Néhány kereskedelmi forgalomba került gép rekeszhosszát tartalmazza a 2.10. ábra.

A rekesz jelentősége, hogy ez a legkisebb címezhető egység. Az utóbbi években majdnem minden számítógépgyártó szabványosította a 8 bites rekeszt, amelyet **bájt**nak nevezünk. Bájtokból épülnek fel a **szavak**. Egy 32 bit szóhosszúságú számítógép szavai 4 bájtosak, míg egy 64 bit szóhosszúságúé 8 bájtosak. A szó jelentősége abban áll, hogy a legtöbb utasítás teljes szavakkal dolgozik, például összead két szót. Vagyis egy 32 bites gépnek 32 bites regiszterei vannak, hogy 32 bites szavakat dolgozhasson fel, míg egy 64 bites gépnek 64 bites regiszterei vannak, és olyan utasításai, amelyek 64 bites szavakat mozgatnak, adnak össze, vonnak ki és egyéb módon manipulálnak.

2.2.3. Bájtsorrend

Egy szó bájtjai sorszámozhatók balról jobbra vagy jobbról balra. Első ránézésre ez lényegtelennek tűnhet, de rövidesen látni fogjuk, hogy ennek fontos következményei vannak. A 2.11. (a) ábra egy olyan 32 bites számítógép memóriájának részletét mutatja, amelynek bájtjai balról jobbra vannak számozva, ilyenek például a SPARC vagy az IBM-nagygépek. A 2.11. (b) ábra egy 32 bites, jobbról balra számozást használó gép megfelelő részét ábrázolja, ilyenek az Intel-processzorok. Az első rendszer neve **nagy endián (big endian)**, mert a számozás a legnagyobb helyértékű bájtnál kezdődik, ezzel ellentétben a 2.11. (b) ábrán látható neve **kis endián (little endian)**. Az angol elnevezések Jonathan Swifttől származnak, aki *Gulliver utazásai* című művében kigúnyolta azokat a politikusokat, akik háborút szítottak, mert nem tudtak megegyezni, hogy a tojást a vastagabbik vagy a keskenyebbik felén kell-e feltörni. A számítógép-architektúrákkal kapcsolatban a kifejezést Cohen (1981) használta először egy élvezetes cikkében.

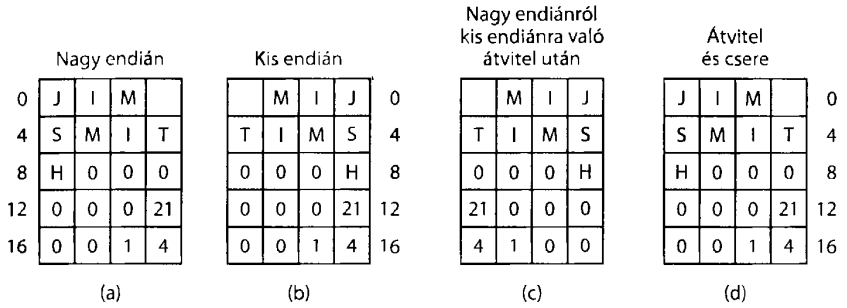


2.11. ábra. *(a) Nagy endián memória. (b) Kis endián memória*

Fontos megérteni azt, hogy mind a nagy endián, mind a kis endián rendszerben egy 32 bites szám, például a 6, ugyanúgy a 3 legkisebb helyértékű biten elhelyezett 110 bitkombinációval van reprezentálva, a többi 29 bit pedig mind 0. A nagy

endián séma szerint az 110 bitek a 3. (vagy 7., vagy 11. stb.) bájtban vannak, míg a kis endián séma szerint a 0. (vagy 4., vagy 8. stb.) bájtban. A számot tartalmazó szó címe mindkét esetben 0.

Ha a számítógépek csak egész számokat tárolnának, nem lenne semmi probléma. De sok alkalmazás számok, karakterláncok és egyéb adattípusok keverékével dolgozik. Tekintsük például a következő egyszerű személyi adatokat tároló struktúrát, amely egy karakterláncból (alkalmazott neve) és két egész számból (életkor, osztálykód) áll. A karakterlánc egy vagy több 0 bájttal ki van egészítve, hogy egész számú szót foglaljon el. A nagy endián reprezentáció a 2.12. (a) ábrán látható, a kis endián reprezentáció pedig a 2.12. (b) ábrán, mindkettő Jim Smith, 21 éves 260-as osztályon ($1 \times 256 + 4 = 260$) dolgozó alkalmazott adataival.



2.12. ábra. *(a) Személyi adatstruktúra nagy endián gépben. (b) Ugyanaz kis endián gépben. (c) Nagy endián gépről kis endián gépre történt átvitel eredménye. (d) Előző átvitel a bájtsorrend megfordításával*

Mindkét reprezentáció teljesen jó, és önmagában konzisztens. A problémák akkor kezdődnek, amikor az egyik gép adatokat akar küldeni a másiknak hálózaton keresztül. Tegyük fel, hogy a nagy endián elküldi a struktúrát bájtonként a kis endiánnak, a 0. bájtnál kezdi és a 19. bájtnál fejezi be. (Optimista módon feltételezzük, hogy a bájtokon belül a bitek sorrendje nem fordul meg, enélkül is van elég bajunk.) Tehát a nagy endián 0. bájtja a kis endián 0. bájtjába kerül, majd így tovább a 2.12. (c) ábrán látható módon.

Amikor a kis endián megpróbálja kinyomtatni a nevet, akkor még minden rendben van, de az életkor 21×2^{24} lesz és az osztály száma is hasonlóképpen eltorzul. Ez a helyzet azért áll elő, mert az átvitel során a szöveg bájtjai helyesen, szavanként fordított sorrendbe kerültek, de így az egész számok bájtjai is fordítva vannak, ami viszont már baj.

Kézenfekvő megoldás lehet, hogy átvitel után egy programmal visszafordítsuk a sorrendet minden szóban. Ha ezt megtesszük, a 2.12. (d) ábrán látható helyzetbe kerülünk, amikor a két egész szám stimmel, de a név „MIJTIMS" lesz, ráadásul a „H" valahol a semmi közepén lóg. A karakterlánc azért kavarodik össze, mert a gép először a 0. bájtot (egy szóköz) olvassa, majd az 1. bájtot (M) és így tovább.

Nincs egyszerű megoldás. Egy lehetséges, de nem hatékony út lehet az, ha minden adatelem elé egy mezőt helyezünk, amely megmondja, hogy milyen típusú adat követi (karakterlánc, egész szám stb.), és az milyen hosszú. Ennek segítségével a vevőnek csak a szükséges konverziókat kell elvégeznie. Bárhogy is legyen, annyi látszik, hogy a bájtsorrend egyértelmű rögzítésének hiánya nagy kellemetlenségeket okoz különböző gépek közötti adatátvitel során.

2.2.4. Hibajavító kódok

Az elektromos hálózatban keletkező áramlökések és egyéb okok miatt a számítógépek memóriái néha hibáznak. Az ilyen hibák ellen védekezésül bizonyos memóriák hibafelismerő vagy hibajavító kódot alkalmaznak. Ezek használata esetén minden memóriabeli szót kiegészítenek speciális bitekkel. Egy szó kiolvasása után a kiegészítő biteket ellenőrzik, hogy lássák, történt-e hiba.

Ahhoz, hogy megértsük, hogyan kezelhetők a hibák, először azt kell megnéznünk közelebbről, hogy valójában mi is a hiba. Tegyük fel, hogy egy memóriabeli szó m adatbitből áll, ehhez adunk még r redundáns, más néven ellenőrző bitet. A teljes hossz legyen n (vagyis $n = m + r$). Egy n bites, m adatbitet és r ellenőrző bitet tartalmazó egységet gyakran n bites **kódszó**nak neveznek.

Ha adott két kódszó, mondjuk 10001001 és 10110001, megállapíthatjuk, hogy hány bitpozíción térnek el. Jelen esetben 3 bit különbözik. Az eltérő bitpozíciók számának megállapításához egy egyszerű logikai KIZÁRÓ-VAGY műveletet kell végezni a két kódszón, majd megszámolni az eredményben az 1-es biteket. Az eltérő bitpozíciók számát a két kódszó **Hamming-távolság**ának nevezzük (Hamming, 1950). Ennek az a jelentősége, hogy ha két kódszó távolsága d, d darab egyszeres bithibának kell előfordulnia ahhoz, hogy az egyik kódszó a másikba alakulhasson. Például az 11110001 és a 00110000 Hamming-távolsága 3, mert 3 egyszeres bithiba szükséges ahhoz, hogy az egyik a másikba alakulhasson.

m bites memóriaszavak esetén mind a 2^m bitminta előfordulhat, de a redundáns bitek kiszámításának szabálya miatt a 2^n kódszó közül csak 2^m érvényes. Ha egy memóriaolvasás érvénytelen kódszót ad, a számítógép azonnal tudja, hogy memóriahiba lépett fel. Az ellenőrző biteket kiszámító algoritmus ismeretében meghatározható az összes érvényes kódszó listája, majd ebben a listában meg lehet keresni azt a két kódszót, amelyek Hamming-távolsága minimális. Ez az érték lesz az összes kód Hamming-távolsága.

Egy kódolás hibafelismerő és hibajavító képessége Hamming-távolságától függ. d egyszeres bithiba felismeréséhez $d + 1$ távolságú kódolás kell, mert ebben az esetben d egyszeres bithiba semmiféleképpen nem alakíthat át egy érvényes kódszót egy másik érvényes kódszóvá. Hasonlóképpen, d egyszeres bithiba javításához $2d + 1$ távolságú kódolás kell, mert így az érvényes kódszavak olyan távol vannak egymástól, hogy még d egyszeres bithiba esetén is az eredeti kódszó minden más kódszónál közelebb van, így egyértelműen meghatározható.

Egyszerű példaként a hibafelismerő kódolásra tekintsük azt a kódolást, amikor az adatbitekhez egyetlen **paritásbit**et adunk hozzá. A paritásbit értékét úgy választjuk

meg, hogy a kódszóban páros (vagy páratlan) számú 1-es legyen. Ennek a kódolásnak a Hamming-távolsága 2, mert minden egyszeres bithiba hibás paritású kódszót állít elő. Más szóval, két bithiba kell ahhoz, hogy egy érvényes kódszóból egy másik érvényes kódszóhoz jussunk. Ezt egyetlen bithiba felismerésére lehet használni. Ha hibás paritású szó kerül kiolvasásra a memóriából, hibaüzenetet kapunk. A program nem tud tovább folytatódni, de legalább nem keletkeznek hibás eredmények.

Egyszerű hibajavító kódolásra példaként tekintsük azt a kódolást, ahol csak a következő négy kódszó érvényes:

0000000000, 0000011111, 1111100000 és 1111111111

Ennek a kódolásnak a Hamming-távolsága 5, ami azt jelenti, hogy ki tud javítani dupla bithibákat. Ha a 0000000111 kódszó érkezik, a vevő tudja, hogy az eredeti csak a 0000011111 lehetett (ha nem volt 2 bithibánál több). Ha azonban a 0000000000 kódszó 3 bithiba miatt lett 0000000111, ez a hiba már nem javítható.

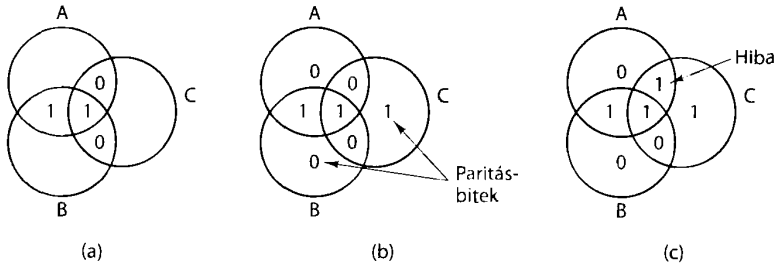
Képzeljük el, hogy m adatbit és r ellenőrző bit felhasználásával olyan kódolást akarunk tervezni, amely minden egyszeres hibát ki tud javítani. A 2^m érvényes memóriaszó bármelyikének van n érvénytelen kódszó szomszédja tőle 1 Hamming-távolságra. Ezeket úgy kaphatjuk meg, hogy egyenként invertáljuk a hozzá tartozó n bites kódszó minden egyes bitjét. Így mind a 2^m érvényes memóriaszóhoz tartoznia kell különböző $n + 1$ bitmintának (a helyes kódszó és n darab hibás). Az összes bitminta száma 2^n, ezért $(n + 1)2^m \leq 2^n$. Felhasználva, hogy $n = m + r$, azt kapjuk, hogy $(m + r + 1) \leq 2^r$. Az m adott, ezért a kapott eredmény megad egy alsó határt az egyszeres hiba javításához szükséges ellenőrző bitek számára. A 2.13. ábrán látható, hogy különböző memóriaszó-hosszúság esetén hány ellenőrző bit kell.

Szó hossza	Ellenőrző bitek	Teljes hossz	Hozzáadott bitek százaléka
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

2.13. ábra. *Egyetlen bithibát javítani képes kódoláshoz szükséges ellenőrző bitek száma*

Ez az elméleti alsó korlát elérhető egy Richard Hamming által megalkotott módszerrel (Hamming, 1950). Mielőtt a Hamming-algoritmusba belefognánk, nézzünk meg egy egyszerű rajzot, amely világosan mutatja egy 4 bites szavakra tervezett hibajavító kódolás alapötletét. A 2.14. (a) ábra Venn-diagramján három kör látszik, A, B és C, amelyek hét részre osztják a síkot. Példaként kódoljuk az 1100 négybites memóriaszót az AB, ABC, AC és BC régiókkal, régiónként 1 bittel (szótári rendezésben). Ez a kódolás látható a 2.14. (a) ábrán.

Ezután a 2.14. (b) ábrán látható módon mindhárom üres régióhoz egy paritásbitet rendelünk úgy, hogy páros paritást kapjunk. Definíció szerint mindhárom körben a bitek összegének párosnak kell lennie. Az *A* körben a 0, 0, 1, 1 bitek vannak, az összegük 2, ami páros. A *B* körben az 1, 1, 0, 0 bitek vannak, ismét páros összeget adva. Végül a *C* körben ugyanez a helyzet. Ebben a példában éppen minden körben ugyanazok a bitek szerepelnek, de más példákban a 0 és 4 összeg előfordulhat. Az ábra egy 4 adatbitből és 3 paritásbitből álló kódszónak felel meg.



2.14. ábra. *(a) Az 1100 kódolása. (b) Páros paritásbitek hozzáadása. (c) Hiba az AC-ben*

Most tegyük fel, hogy az *AC* régióban található bit elromlik, 0-ról 1-re változik, ahogy a 2.14. (c) ábrán látható. A számítógép észleli, hogy az *A* és a *C* köröknek rossz (páratlan) a paritása. Csak egyféleképpen lehet 1 bit javításával helyreállítani a rendet, az *AC* régió bitjét kell visszaállítani 0-ra, megszüntetve ezzel a hibát. Ilyen módon a számítógép automatikusan kijavíthatja az egyszeres bithibákat.

Most nézzük a tetszőleges hosszúságú memóriaszavak hibajavító kódolására használható Hamming-algoritmust. A kódolás során *r* redundáns bitet adunk egy *m* bites szóhoz, így a kódszó teljes hossza $n = m + r$ bit lesz. A biteket nem 0-val, hanem 1-gyel kezdődően sorszámozzuk, a legnagyobb helyértékű az 1-es sorszámú lesz. Minden olyan bit paritásbit lesz, amelynek sorszáma 2 hatványa, a többi pedig adatbit. Például egy 16 bites kódszóhoz 5 paritásbitet adunk. Az 1., 2., 4., 8. és 16. pozíción vannak a paritásbitek, a többi pedig mind adatbit. A memóriaszó összesen 21 bites (16 adat, 5 paritás). Ebben a példában páros paritást fogunk használni (használhatnánk páratlant is).

Minden paritásbit meghatározott bitpozíciókat ellenőriz; a paritásbit értékét úgy állítjuk be, hogy az ellenőrzött pozíciókon lévő 1-esek száma páros legyen. Az egyes paritásbitek által ellenőrzött bitpozíciók a következők:

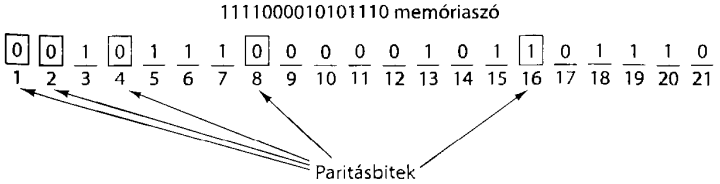
1. bit: 1., 3., 5., 7., 9., 11., 13., 15., 17., 19., 21.
2. bit: 2., 3., 6., 7., 10., 11., 14., 15., 18., 19.
4. bit: 4., 5., 6., 7., 12., 13., 14., 15., 20., 21.
8. bit: 8., 9., 10., 11., 12., 13., 14., 15.
16. bit: 16., 17., 18., 19., 20., 21.

Általánosan a *b*. bitet azok a $b_1, b_2, ..., b_j$ paritásbitek ellenőrzik, amelyekre $b_1 + b_2 + ... + b_j = b$. Például az 5. bitet az 1. és a 4. bit ellenőrzi, mert 1 + 4 = 5. A 6. bitet a 2. és a 4. bit ellenőrzi, mert 2 + 4 = 6 és így tovább.

A 2.15. ábra a 16 bites 1111000010101110 memóriaszóhoz tartozó Hamming-kód kiszámítását mutatja. Az eredmény a 21 bites 001011100000101101110 kódszó. A hibajavítás működésének bemutatásához nézzük meg, hogy mi történik, ha például egy áramlökés következtében az 5. bit átfordul. Az új kódszó 001001100000101101110 lesz 001011100000101101110 helyett. Az öt paritásbitet ellenőrizve a következő eredményre jutunk:

1. paritásbit hibás (1., 3., 5., 7., 9., 11., 13., 15., 17., 19., 21. bitek között öt 1-es van)
2. paritásbit helyes (2., 3., 6., 7., 10., 11., 14., 15., 18., 19. bitek között hat 1-es van)
4. paritásbit hibás (4., 5., 6., 7., 12., 13., 14., 15., 20., 21. bitek között öt 1-es van)
8. paritásbit helyes (8., 9., 10., 11., 12., 13., 14., 15. bitek között két 1-es van)
16. paritásbit helyes (16., 17., 18., 19., 20., 21. bitek között négy 1-es van)

Az 1., 3., 5., 7., 9., 11., 13., 15., 17., 19. és 21. bitek között páros számú 1-esnek kell lennie, mivel páros paritást használunk. A hibás bitnek azok között kell lennie, amelyeket az 1. paritásbit ellenőriz, vagyis az 1., 3., 5., 7., 9., 11., 13., 15., 17., 19. vagy 21. valamelyike hibás. A 4. paritásbit is hibás, ami azt jelenti, hogy a 4., 5., 6., 7., 12., 13., 14., 15., 20. vagy 21. bitek egyike hibás. A hibás bitnek mindkét listában benne kell lennie, vagyis az 5., 7., 13., 15. vagy 21. bit a bűnös. De a 2. paritásbit helyes, ez kizárja a 7. és a 15. bitet. Hasonlóan a 8. paritásbit helyessége kizárja a 13. bitet. Végül, a 16. paritásbit is helyes, ez kizárja a 21. bitet. Az egyetlen megmaradt bit az 5., ennek kell hibásnak lennie. Mivel 1 értékűnek olvastuk, ezért a helyes értékének 0-nak kell lennie. Ezzel a módszerrel a hibákat ki lehet javítani.



2.15. ábra. *A 16 bites 1111000010101110 memóriaszó Hamming-kódjának kiszámítása 5 ellenőrző bit hozzáadásával*

A hibás bit megtalálásának egyszerű módja, hogy először kiszámítjuk az összes paritásbitet. Ha mindegyik helyes, nem volt hiba (vagy 1-nél több hiba fordult elő). Ezután összeadjuk a hibás paritásbitek sorszámát, az 1. paritásbit esetén 1-et,

a 2. paritásbit esetén 2-t, a 4. paritásbit esetén a 4-et és így tovább. Az eredményül kapott összeg a hibás bit pozíciója. Például, ha a paritásbit 1 és 4 hibás, de a paritásbit 2, 8, 16 helyes, az 5. (1 + 4) bit fordult át.

2.2.5. Gyorsítótár

A számítógépek története során a processzorok mindig gyorsabbak voltak a memóriáknál. A memóriákkal együtt a processzorok is mindig gyorsabbak lettek, így a különbség mindig fennmaradt. Valójában azt a lehetőséget, hogy egyre több áramkört zsúfoljanak rá egy lapkára a CPU-tervezők arra használták, hogy csővezetékeket és szuperskaláris funkciókat valósítsanak meg, amellyel meggyorsíthatják a processzorok működését. A memóriák tervezői ezzel ellentétben az új technológiákat nem a sebesség, hanem a kapacitás növelésére használták fel, így a helyzet idővel egyre rosszabb lett. A gyakorlatban ez azt jelenti, hogy miután a processzor kér egy szót a memóriából, a kért szót még jó néhány CPU-ciklus alatt nem kapja meg. Minél lassabb a memória, annál több ciklust kell várnia a processzornak.

Ahogy arra korábban rámutattunk, két megoldás képzelhető el. A legegyszerűbb módszer, hogy kiadjuk a memóriaolvasási parancsot, amikor az jelentkezik a végrehajtás során, majd folytatjuk a végrehajtást, és csak akkor várakoztatjuk a processzort, ha egy gépi utasítás használni próbálná a memória szót, mielőtt az megérkezne. Minél lassabb a memória, ez annál gyakrabban előfordulhat, és annál nagyobb az időveszteség is, amikor előfordul. Például, ha a memóriakésés 10 ciklus, nagyon valószínű, hogy a következő 10 utasítás valamelyike használni akarja a kért szót.

A másik megoldás a számítógépek várakozásának megelőzésére, hogy a fordítóprogramoktól megköveteljük, hogy olyan kódot állítsanak elő, amelyekben a program nem használ fel egy memória szót, mielőtt az megérkezne. A baj az, hogy ezt sokkal egyszerűbb mondani, mint megvalósítani. Gyakran egy LOAD után semmi mást nem lehet tenni, mint várni, így a fordítóprogram kénytelen NOP (no operation, nincs művelet) utasításokat elhelyezni a kódban, amelyek nem csinálnak semmit, csak a helyet foglalják, és az időt fecsérlik. Valójában ez a megoldás szoftveres várakozás a hardveres várakozás helyett, de a teljesítmény csökkenése ugyanakkora.

Tulajdonképpen a probléma nem technológiai, hanem gazdaságossági. A mérnökök tudják, hogy kell olyan gyors memóriát építeni, mint a CPU, de ehhez a CPU lapkára kellene tenni (mert a sínen keresztül nagyon lassú eljutni a memóriához). Nagy memória elhelyezése a CPU lapkán megnöveli a lapka méretét, emiatt még drágább lenne, de még ha az ár nem is számítana, egy CPU lapka méretének még akkor is lennének gyakorlati korlátai. Így aztán két dolog közül választhatunk: kicsi gyors memória vagy nagy lassú memória. Nagy, gyors és olcsó memóriát szeretnénk.

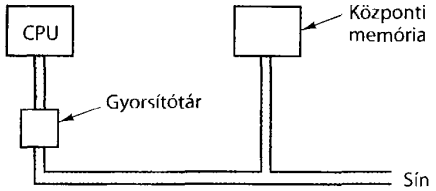
Érdekes módon vannak olyan módszerek, amelyek a kicsi gyors és nagy lassú memória kombinálásával egyszerre nyújtják mérsékelt áron a gyors memória sebességét (megközelítőleg) és a lassú memória méretét. A kis, gyors memória neve

gyorsítótár (**cache**, a francia *cacher* szóból származik, jelentése: elrejteni). A következőkben a gyorsítótárak működését és használatát tárgyaljuk. Részletesebben a 4. fejezetben foglalkozunk majd velük.

A gyorsítótár alapötlete egyszerű: a leggyakrabban használt memóriaszavakat a gyorsítótárban tartjuk. Amikor a processzornak szüksége van egy szóra, először belenéz a gyorsítótárba. Ha a szó nincs benne, csak akkor fordul a központi memóriához. Ha a szavak jelentős része a gyorsítótárban van, az átlagos elérési idő nagymértékben csökken.

A siker tehát azon múlik, hogy a keresett szavak mekkora hányada van a gyorsítótárban. Évek óta ismert, hogy a programok nem teljesen véletlenszerűen kezelik a memóriát. Ha egy adott hivatkozás az A memóriacímre történik, a következő memóriahivatkozás valahol A egy tágabb környezetében lesz. Erre egy egyszerű példa maga a program. A vezérlésátadó utasítások és az eljáráshívások kivételével az utasítások egymás utáni címekről kerülnek beolvasásra. Ezen túlmenően a programok futási idejük nagy részét ciklusokban töltik, amikor is ugyanazt a néhány utasítást hajtják végre újra és újra. Egy mátrixokat kezelő program is valószínűleg sokszor hivatkozik a mátrixra, mielőtt valami más tevékenységbe fogna.

Az a megfigyelés, hogy egy rövid időintervallumban a memóriahivatkozások a teljes memóriának csak egy kis részét érintik, a **lokalitási elv**. Ez az alapja a gyorsítótáras rendszereknek, ahol nagy vonalakban az történik, hogy ha egy memóriaszóra hivatkozik a program, a szó és még néhány szomszédja a nagy lassú tárból beolvasásra kerül a gyorsítótárba, így ha legközelebb használják, már gyorsan elérhető. A CPU, a gyorsítótár és a központi memória egy szokványos elrendezését a 2.16. ábrán láthatjuk. Ha a CPU rövid időn belül egy szóra k-szor hivatkozik, ebből 1-szer kell a lassú memóriához fordulni, $k - 1$-szer pedig a gyorshoz. Minél nagyobb a k, annál jobb a teljesítmény.



2.16. ábra. *A gyorsítótár logikailag a CPU és a központi memória között helyezkedik el. Fizikailag számos olyan lehetséges hely van, ahová elhelyezhető.*

Néhány paraméter bevezetésével formálisan is elvégezhetjük a számítást, jelöljük c-vel a gyorsítótár elérési idejét, m-mel a központi memória elérési idejét és h-val a **találati arányt**, ami azt mutatja, hogy az összes hivatkozás mekkora hányadát lehetett a gyorsítótárból kielégíteni. Az előző bekezdés példájában $h = (k - 1)/k$. Bizonyos szerzők a **hibaarányt** is definiálni szokták, ennek értéke $1 - h$.

Ezekkel a definíciókkal kiszámíthatjuk az átlagos elérési időt a következőképpen:

átlagos elérési idő = $c + (1 - h)\,m$

Ha $h \to 1$, minden hivatkozás kiszolgálható a gyorsítótárból, az elérési idő pedig közelíti c-t. Másrészt, ha $h \to 0$, minden esetben memóriaműveletet kell végezni, az elérési idő pedig közelíti $c + m$-et, mert először c időegység alatt ellenőrizni kell a gyorsítótárat (sikertelenül), majd m időegység kell a memóriaművelet elvégzéséhez. Bizonyos rendszerekben a memóriaműveletet el lehet kezdeni a gyorsítótárban való kereséssel párhuzamosan, így ha nincs találat a gyorsítótárban, a memóriaciklus már elkezdődött. Ennél a stratégiánál azonban szükség van arra, hogy gyorsítótár-találat esetén a memóriát munka közben megszakíthassuk, ami a megvalósítást bonyolultabbá teszi.

A lokalitási elvtől vezéreltetve a központi memória és a gyorsítótár kötött méretű blokkokra van osztva. Amikor a gyorsítótáron belüli blokkokról esik szó, akkor ezeket általában **gyorsítósor**nak (**cache line**) nevezik. Ha egy keresett szó nincs a gyorsítótárban, egy egész sort betöltenek a központi memóriából a gyorsítótárba, nemcsak a szükséges szót. Például 64 bájtos sormérettel, a 260-as memóriacímre történő hivatkozás behúzza a 256. bájttól a 319. bájtig terjedő sort egy gyorsítósorba. Egy kis szerencsével a gyorsítósor többi bájtja közül is szükség lesz néhányra hamarosan. Ez a módszer hatékonyabb, mint egyesével olvasni a szavakat, mert egyszerre k szót olvasni hatékonyabb, mint egy-egy szót olvasni k-szor. Ezenkívül a több szóból álló sorok azt is jelentik, hogy kevesebb van belőlük, ez pedig kevesebb többletmunkát jelent.

A nagy teljesítményű processzorok esetében a gyorsítótárak tervezése egyre fontosabb feladat. Az egyik szempont a gyorsítótár mérete. Minél nagyobb, annál jobban működik, de egyben drágább is. Másik szempont a gyorsítósor mérete. Egy 16 KB-os gyorsítótár felosztható 1024 darab 16 bájtos sorra, 2048 darab 8 bájtos sorra, és még más kombinációk is elképzelhetők. A harmadik szempont a gyorsítótár felépítése, vagyis az, hogy milyen módon tartja nyilván a gyorsítótár, hogy mely memóriaszavak vannak benne. A gyorsítótárakat a 4. fejezetben tárgyaljuk részletesen.

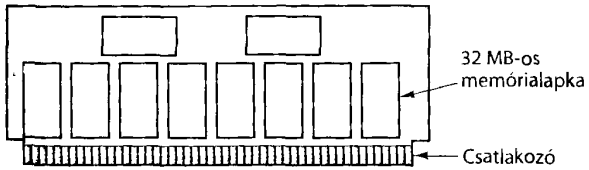
A negyedik tervezési szempont az, hogy az utasításokat és az adatokat közös vagy külön tárban tartjuk. Az **egyesített gyorsítótár** (az utasítások és adatok ugyanazt a gyorsítótárat használják) egyszerűbb szerkezetű, és automatikusan egyensúlyban tartja az utasítások és az adatok mozgatását. Mindazonáltal mára az **osztott gyorsítótár** felé történt elmozdulás, amikor is az utasítások és az adatok külön gyorsítótárban vannak. Ez utóbbit **Harvard-architektúrá**nak is nevezik, ugyanis gyökerei egészen Howard Aiken Mark III gépéig nyúlnak vissza, amelynek külön memóriája volt az utasítások és az adatok számára. A csővezetékes központi egységek elterjedt használata készteti erre a tervezőket. Az utasítást előre beolvasó egységnek ugyanakkor kell hozzáférnie az utasításokhoz, mint az operandusbeolvasó egységnek az adatokhoz. A szétválasztott gyorsítótár lehetővé teszi a párhuzamos működést, az egyesített gyorsítótár nem. Ezenkívül, mivel az utasításokat általában nem módosítják végrehajtás közben, az utasításokat tároló gyorsítótár tartalmát soha nem kell visszaírni a memóriába.

Végül, az ötödik szempont a gyorsítótárak száma. Nem ritka manapság, hogy van egy elsődleges gyorsítótár a processzorlapkán, egy másodlagos a lapkán kívül, de a processzorral egy tokban, végül egy harmadik még messzebb.

2.2.6. Memóriatokozás és -típusok

A félvezető-memóriák első napjaitól kezdve az 1990-es évek elejéig a memóriát külön áramköri lapkákon gyártották, adták el és szerelték be. A lapkák sűrűsége 1 kbitről 1 megabitre vagy e fölé emelkedett, de minden lapka külön egység volt. Az első PC-kben gyakran voltak fenntartott aljzatok, amelyekbe kiegészítő memórialapkákat lehetett betenni, ha a vásárlónak szüksége volt rá.

Jelenleg gyakran más elrendezést használnak. Több, tipikusan 8 vagy 16 lapkát egy kicsi nyomtatott áramköri lapkára rögzítenek, és egységként árusítják. Ez az egység a **SIMM (Single Inline Memory Module, egy érintkezősoros memóriamodul)** vagy **DIMM (Dual Inline Memory Module, két érintkezősoros memóriamodul)**, attól függően, hogy a lapkának csak az egyik vagy mindkét oldalán vannak-e érintkezői. A SIMM moduloknak 72 érintkezője van, és 32 bitet tudnak továbbítani egy órajelciklus alatt. A DIMM moduloknak mind a két oldalon 84 érintkezője van, összesen 168, és 64 bitet tudnak továbbítani egyetlen órajelciklus alatt. Egy SIMM látható a 2.17. ábrán.



2.17. ábra. *Egy 256 MB-os SIMM (Single Inline Memory Module). A két felső lapka vezérli a SIMM működését*

Egy tipikus SIMM vagy DIMM állhat nyolc, egyenként 256 megabites (32 MB) lapkából. A teljes modul mérete 256 MB. Sok számítógépbe négy modult lehet betenni, így ez összesen 1 GB, ha 256 MB-os modulokat használunk, és több, ha nagyobbak a modulok.

A fizikailag kisebb DIMM-eket **SO-DIMM**-eknek (**Small Outline DIMM**) nevezik, és a noteszgépekben használják. A SIMM és DIMM moduloknak lehet paritásbites hibajavító kódja, mivel azonban az átlagos hibaarány nagyon kicsi, 1 hiba minden 10 évben, ezért a legtöbb mezei számítógépben a hibafelismerést és -javítást el szokták hagyni.

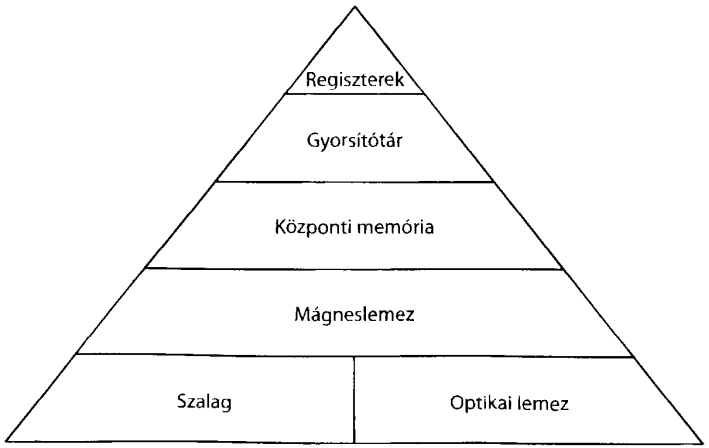
2.3. Háttérmemória

Nem számít mekkora a központi memória, mindig túl kicsi. Az emberek mindig több adatot akarnak tárolni, mint amennyi belefér, elsősorban azért, mert ahogy a technológia fejlődik, az emberek mindjárt olyan dolgokat akarnak tárolni, ami korábban csak a tudományos-fantasztikus művekben volt lehetséges. Például, mivel az Egyesült Államok költségvetési irányelvei szerint a kormányzati szerveknek saját bevétellel kell rendelkezniük, elképzelhető, hogy a Kongresszusi Könyvtár úgy dönt, digitalizálja teljes anyagát, és fogyasztási cikként árulni kezdi („Az emberiség összes tudása csak 99,95 dollár!"). A durva becsléssel 50 millió könyv, mindegyikben 1 MB szöveg és 1 MB tömörített képi anyag, tárolásához 10^{14} bájt, vagyis 100 terabájt kell. Az összes eddig készített mozifilm tárolása is ebben a tartományban van. Ennyi információ nem fér a központi memóriába, legalábbis még néhány évtizedig nem.

2.3.1. Memóriahierarchia

Nagy mennyiségű adat tárolására hagyományosan memóriahierarchia szolgál, ahogyan ez a 2.18. ábrán látható. Legfelül a CPU teljes sebességével elérhető regiszterek helyezkednek el. A következő szinten a gyorsítótár van, ez jelenleg 32 KB és néhány megabájt közötti nagyságrendű. A központi memória a következő, 16 MB kezdő mérettől néhány tíz gigabájtig terjed a legjobb gépekben. Ezután a mágneslemezek, a huzamos tárolás jelenlegi „igáslovai" jönnek, végül pedig a mágnesszalag és az optikai lemez, az archiválás eszközei.

Ahogy lefelé haladunk a hierarchia szintjein, három kulcsfontosságú paraméter értéke növekszik. Először is az elérési idő. A CPU-regiszterek néhány nanoszekundum alatt elérhetők. A gyorsítótár néhányszor lassabb a CPU-regisztereknél.



2.18. ábra. *Ötszintű memóriahierarchia*

A központi memória elérése tipikusan néhány tíz nanoszekundumot vesz igénybe. Ezen a ponton nagy rés van, a mágneslemez elérési ideje legalább 10 ms, a szalag és az optikai lemez elérési ideje pedig másodpercekben mérhető, ha az adathordozót elő kell venni és be kell tenni a meghajtóba.

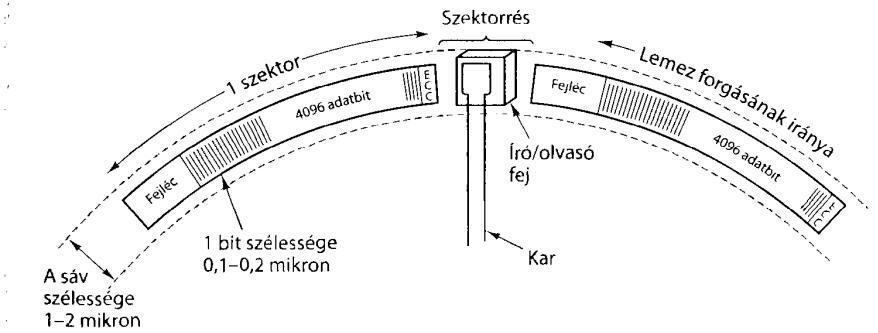
Másodszor, a tárolási kapacitás lefelé haladva növekszik. A CPU regiszterei körülbelül 128 bájt tárolására jók, a gyorsítótár néhány megabájtos, a központi memória néhányszor tíz megabájttól néhányszor ezer megabájtig, a mágneslemezek pedig néhány gigabájttól néhányszor tíz gigabájtig terjednek. A szalagok és az optikai lemezek jelenleg cserélhetők, ezért kapacitásuknak csak a tulajdonos költségvetése szab határt.

Harmadszor, az 1 dollárért megvásárolható bitek száma a hierarchiában lefelé növekszik. Jóllehet az aktuális árak gyorsan változnak, a központi memóriát dollár/megabájtban, a mágneslemezt penny/megabájtban, a mágnesszalagot pedig dollár/gigabájtban mérik.

A regisztereket, a gyorsítótárat és a központi memóriát már megismertük. A következő fejezetekben a mágneslemezekkel, azután az optikai lemezekkel ismerkedünk. Nem foglalkozunk a szalagokkal, mert a biztonsági mentésen kívül ritkán használják, és amúgy sincs róluk túl sok mondanivaló.

2.3.2. Mágneslemezek

Egy mágneslemez (egység) egy vagy több mágnesezhető bevonattal ellátott alumíniumkorongból áll. Eredetileg a korongok átmérője mintegy 50 cm volt, de manapság már csak 3 és 12 cm közöttiek, a noteszgépekhez használatos lemezek átmérője pedig már 3 cm alatt van, és egyre kisebb lesz. Egy indukciós tekercset tartalmazó fej lebeg a lemez felszíne felett egy vékony légpárnán (kivéve a hajlékonylemezeknél, ahol a fej hozzáér a felülethez). Ha pozitív vagy negatív áram folyik az indukciós tekercsben, a fej alatt a lemez magnetizálódik, és az áram polaritásától függően a mágneses részecskék balra vagy jobbra állnak be. Amikor a fej egy mágnesezett terület felett halad át, akkor pozitív vagy negatív áram indu-



2.19. ábra. *Egy sáv részlete. Két szektor látható a képen*

kálódik benne, így a korábban tárolt biteket vissza lehet olvasni. Ahogy a korong forog a fej alatt, bitsorozatokat lehet felírni és később visszaolvasni. A lemez egy sávjának felépítését a 2.19. ábra mutatja.

Egy teljes körülfordulás alatt felírt bitsorozat a **sáv**. Minden sáv rögzített méretű, tipikusan 512 adatbájtot tartalmazó **szektorok**ra van osztva, melyeket egy **fejléc** előz meg, lehetővé téve a fej szinkronizálását írás és olvasás előtt. Az adatok után hibajavító kód (ECC, Error-Correcting Code) található, ez vagy a Hamming-kód, vagy egyre gyakrabban a többszörös hibákat is javítani képes **Reed–Solomon-kód**. Az egymást követő szektorok között keskeny **szektorrés** van. Egyes gyártók a lemezek formázatlan kapacitását adják meg (mintha a sávok csak adatokat tartalmaznának), de becsületesebb mérőszám a formázott kapacitás, amely nem számítja adatnak a fejléceket, az ECC-ket és a szektorréseket. A formázott kapacitás általában 15 százalékkal kisebb, mint a formázatlan.

Minden lemeznek vannak mozgatható karjai, melyek a forgástengelytől sugárirányban ki-be tudnak mozogni. Minden sugárirányú pozíción egy-egy sáv írható fel. A sávok tehát forgástengely középpontú koncentrikus körök. A sáv szélessége attól függ, hogy milyen széles a fej, illetve hogy milyen pontosan lehet sugárirányban pozicionálni. A jelenlegi technológia mellett a lemezek centiméterenként 5000–10000 sávot tartalmaznak, ez 1–2 mikron szélességű sávoknak felel meg (1 mikron = 1/1000 mm). Meg kell jegyeznünk, hogy egy sáv nem egy fizikailag létező barázda a lemezen, hanem csak egy mágnesezett gyűrű, amelyet kis mágnesezetlen hézagok választanak el a külső és belső szomszédos sávoktól.

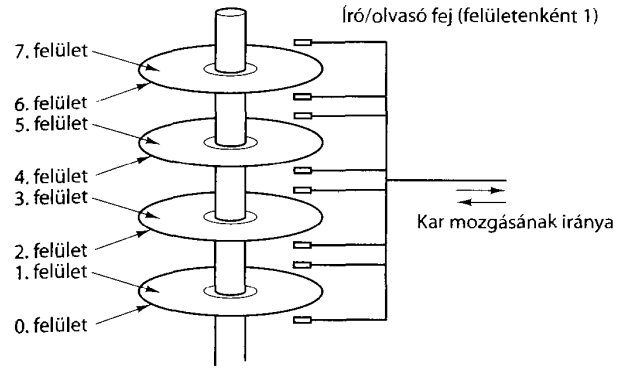
A bitek lineáris sűrűsége egy sáv kerülete mentén különbözik a sugárirányú sűrűségtől. Ezt nagyrészt a felület tisztasága és a levegő minősége határozza meg. A mai lemezek 50 000 bit/cm és 100 000 bit/cm közötti kerületi sűrűséget érnek el. Tehát egy bit körülbelül 50-szer nagyobb sugárirányban, mint a kerület mentén.

A még nagyobb írássűrűség érdekében a gyártók olyan eljárásokat fejlesztenek ki, amelyeknél a biteket nem a diszk kerületén hosszirányban, hanem függőlegesen a vas-oxid belseje felé rögzítik. Ezt **merőleges rögzítés**nek nevezik, és hamarosan piacra is kerül.

A felület tisztasága és a levegő minősége érdekében a legtöbb lemezt gyárilag légmentesen lezárják, hogy por ne kerülhessen bele. Az ilyen lemezeket **winchester**nek hívják. Az első ilyen (az IBM által gyártott) lemezegységekben 30 MB lezárt, fix tárolóhely és 30 MB cserélhető tárolóhely volt. Feltehetőleg ezek a 30-30-as lemezek az amerikai vadnyugat 30-30-as Winchester puskáira emlékeztették az embereket, és a „winchester" név megragadt.

A legtöbb lemezegység a 2.20. ábrán látható módon több, egymás felett elhelyezett korongból áll. Minden felülethez tartozik egy fej és egy mozgatókar. A karok rögzítve vannak egymáshoz, így a fejek mindig ugyanarra a sugárirányú pozícióra állnak be. Egy adott sugárirányú pozícióhoz tartozó sávok összességét **cilinder**nek nevezzük. A jelenlegi PC diszkekben 6–12 korong található egymás felett, ami 12–24 rögzítésre használható felületet jelent.

A lemezegység teljesítménye sok tényezőtől függ. Egy szektor beolvasásához vagy kiírásához először a fejet a megfelelő sugárirányú pozícióba kell állítani. Ezt a műveletet **keresés**nek **(seek)** hívják. Az átlagos (véletlenszerűen kiválasztott sávok közötti) keresési idők 5 és 10 ms között vannak, míg egymás melletti sávok esetén ez az idő már 1 ms alatti. A fej kívánt sugárirányú pozícióba való beállása után van egy kis szünet, az ún. **forgási késleltetés**, amíg a keresett szektor a fej alá fordul. A legtöbb lemez 5400, 7200 vagy 10800 fordulatot tesz meg percenként, ezért az átlagos késleltetés (egy félfordulat ideje) 3 és 6 ms között van. Az adatátviteli idő a kerületi bitsűrűségtől és a fordulatszámtól függ. A tipikus 20 és 40 MB/s adatátviteli sebesség mellett egy 512 bájtos szektor 13 és 26 μs közötti időt igényel. Ebből következik, hogy a keresési idő és a forgási késleltetés teszi ki az adatátviteli idő nagy részét. A lemezen szétszórt, véletlenszerűen kiválasztott szektorokat olvasni nyilvánvalóan nem hatékony adatfeldolgozási módszer.

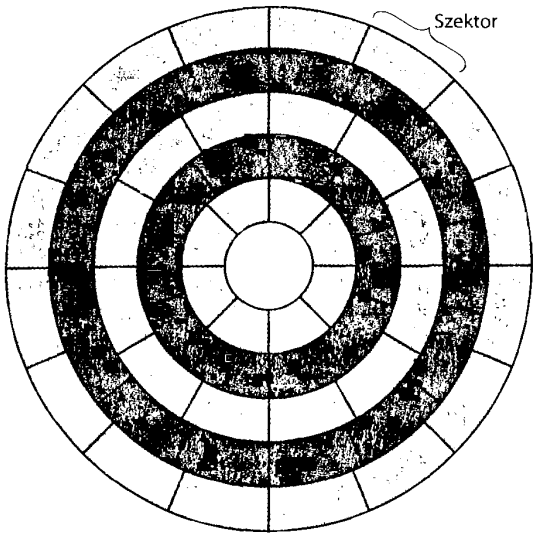


2.20. ábra. *Lemezegység négy koronggal*

Érdemes megjegyezni, hogy a fejléceket, az ECC-ket, a szektorok közötti réseket, a keresési időt és a forgási késleltetést figyelembe véve nagy különbség mutatkozik a meghajtó maximális adatátviteli képessége és a folyamatos adatátviteli képessége között. A maximális adatátviteli sebesség az a sebesség, amelyet az első adatbit beolvasását követően mérhetünk. A számítógépnek képesnek kell lennie az ezzel a sebességgel beérkező adatok fogadására. A meghajtó azonban ezt az iramot csak egy szektornyi ideig képes tartani. Bizonyos (például multimédia-) alkalmazások számára a másodperceken keresztül folyamatosan tartható átlagos sebesség a fontos, ehhez figyelembe kell venni a keresési időt és a forgási késleltetést is.

Egy kis gondolkodás után, a kör kerületét megadó, a középiskolából jól ismert $c = 2\pi r$ képlet alkalmazásával rájöhetünk, hogy a külső sávok hosszabbak, mint a belsők. Mivel a lemezek a fejek pozíciójától függetlenül állandó szögsebességgel forognak, ez problémát vet fel. Régebbi meghajtókban az elérhető legnagyobb bitsűrűséget a legbelső sávon használták, majd egyre csökkentették a sávokon kifelé haladva. Ha egy lemezen például 18 szektor volt sávonként, mindegyik 20 foknyi ívet foglalt el, függetlenül attól, hogy melyik cilinderen volt.

Manapság más stratégiát alkalmaznak. A cilindereket (tipikusan 10 és 30 közötti) zónákba osztják, és a külső zónákban több szektort tesznek egy sávba. Ez bonyolítja



2.21. ábra. *Egy lemezegység öt zónával. Minden zónában sok sáv található*

az információ követését, de növeli a kapacitást, és ez a legfontosabb. Minden szektor mérete egyforma. Egy öt zónával rendelkező diszk ábrája látható a 2.21. ábrán.

Minden lemezhez tartozik egy **lemezvezérlő**, egy lapka, amely vezérli a meghajtót. Egyes vezérlőkben egy teljes CPU van. A vezérlő feladatai közé tartozik a szoftverből érkező parancsok fogadása, mint a READ, WRITE és FORMAT (az összes fejléc felírása), a kar mozgatása, hibák felismerése és javítása, valamint a 8 bites memóriabájtok oda- és visszaalakítása bitek sorozatává. Egyes vezérlők még több szektorra kiterjedő pufferceléssel is foglalkoznak, a beolvasott szektorokat gyorsítótárba helyezik a jövőbeni használat gyorsítása érdekében, és áthelyezik a fizikailag hibás szektorokat. Ez utóbbira akkor van szükség, amikor valamelyik szektorban egy hibás (véglegesen mágnesezett) hely keletkezik. Ha a vezérlő felfedez egy ilyen hibás helyet, kiváltja egy olyan tartalék szektorral, amelyet kifejezetten erre a célra tartanak fenn minden cilinderen vagy zónában.

2.3.3. Hajlékonylemezek

A személyi számítógépek megjelenésével szükséggé vált valamilyen módszer kidolgozása a programok terjesztésére. A megoldást a **floppy** vagy **hajlékonylemez** adta, egy kisméretű, cserélhető adathordozó, amelyet azért hívnak így, mert a legelsőket még fizikailag meg lehetett hajlítani. A hajlékonylemezt tulajdonképpen az IBM találta fel, és arra használták, hogy a nagygépek karbantartásához szükséges információt tárolják a kiszolgáló személyzet számára, de a személyi számítógépek gyártói hamar átvették mint az eladásra szánt programok terjesztésének kényelmes eszközét.

Általános jellemzőik ugyanazok, mint az előzőkben leírt lemezeké, de míg a merevlemezeknél a fejek a lemezek felszíne felett egy vékony, gyorsan áramló levegőrétegen lebegnek, a hajlékonylemezek esetén a fejek hozzáérnek a lemezhez. Ennek következtében mind az adathordozó, mind a fejek aránylag hamar elkopnak. Hogy a kopást és az ebből eredő bosszúságot elkerülhessük, a személyi számítógépek visszahúzzák a fejeket és leállítják a forgást, amikor a meghajtó éppen nem olvas vagy ír. Emiatt a soron következő olvasás vagy írás parancs kiadásakor körülbelül fél másodpercre van szüksége a motornak a megfelelő sebesség eléréséhez. A floppydiszkeket mintegy 20 évig használták, de a modern számítógépek már nélkülük kerülnek forgalomba.

2.3.4. IDE-lemezek

A mai személyi számítógépekben használt lemezek az IBM PC XT 10 MB-os Seagate lemezéből fejlődtek ki, amelyhez egy bővítőkártyán található Xebec vezérlő tartozott. A Seagate lemezegységben 4 fej, 306 cilinder és sávonként 17 szektor volt. A vezérlő két meghajtó kezelésére volt képes. Az operációs rendszer a lemezre íráshoz, és a lemezről olvasáshoz a paramétereket CPU-regiszterekbe töltötte, majd meghívta a PC beépített, csak olvasható memóriájában tárolt **BIOS**-t **(Basic Input Output System)**. A BIOS adta ki azokat a gépi utasításokat, amelyek feltöltötték a lemezvezérlő regisztereit és elindították az adatátvitelt.

A technológia gyorsan fejlődött a külön kártyán elhelyezett vezérlőtől a meghajtóba integrált vezérlőig, ezek közül az elsők az **IDE- (Integrated Drive Electronics, beépített eszközelektronika)** meghajtók voltak az 1980-as évek közepén. A BIOS-hívási konvenciókat a visszafelé kompatibilitás érdekében nem változtatták meg. Ezek szerint a szektorok címzése a fej, a cilinder és a szektor sorszámának megadásával történt, a fejek és a cilinderek számozása 0-val kezdődött, a szektoroké 1-gyel. Utóbbi választás valószínűleg az eredeti BIOS-programozó hibája, aki a remekművét 8088 assemblyben írta. 4 bit jut a fejre, 6 bit a szektorra és 10 bit a cilinderre, tehát maximálisan 16 fej, 63 szektor és 1024 cilinder lehet, ez 1032192 szektor összesen. Így a maximális diszkkapacitás 504 MB, ez valószínűleg végtelenül nagynak tűnt annak idején, de semmi esetre sem tűnik annak napjainkban. (Panaszkodna ma valaki azért, mert egy új gép nem tud 1 terabájtnál nagyobb meghajtókat kezelni?)

Nem sok idő telt el, és 504 MB-nél ugyan kisebb lemezegységek kezdtek megjelenni, de rossz felosztással (például 4 fej, 32 szektor, 2000 cilinder). Az operációs rendszer nem tudta kezelni ezeket a régóta megkövesedett BIOS-hívásokkal. Ennek következtében a lemezvezérlők elkezdtek hazudni, úgy tettek mintha az értékek a BIOS-határok között lettek volna, de ténylegesen átszámították a látszólagos értékeket a valós értékekre. Ez a módszer működött, csupán akkor voltak zűrök, ha az operációs rendszer a keresési idők minimalizálása érdekében gondosan ügyelt az adatok elhelyezésére.

Végül az IDE-meghajtókat felváltották az **EIDE- (Extended IDE, kiterjesztett IDE)** meghajtók, amelyek támogatnak egy másik címzési módot is; ez az **LBA**

(**Logical Block Addressing, logikai blokk címzés**), amely a szektorokat egyszerűen 0-tól a maximális 2^{28} – 1-ig számozza. Ez a mód megkívánja, hogy a vezérlő az LBA-címeket fej, szektor és cilinder sorszámokká konvertálja, de így képes az 504 MB-os határ fölé kerülni. Sajnos, egyúttal egy újabb szűk keresztmetszetet hozott létre, a $2^{28} \times 2^9$ bájtot (128 GB). A szabványokról döntő bizottságoknak, mint a politikusoknak is, szokása a problémák továbbgörgetése, hogy a következő bizottságnak kelljen megoldania őket.

Az EIDE-meghajtók és -vezérlők más előnyökkel is rendelkeztek. Például az EIDE-vezérlőknek két csatornájuk lehetett, mindegyik egy elsődleges és egy másodlagos diszkegységgel. Ez az elrendezés legfeljebb 4 meghajtót tett lehetővé vezérlőegységenként. Támogatta a CD-ROM- és a DVD-meghajtókat is, az átviteli sebességük pedig 4 MB/s-ről 16,67 MB/s-re nőtt.

Ahogyan a mágneslemez-technológia folyamatosan javult, az EIDE-szabvány is folytatta a fejlődést, de valamilyen okból kifolyólag az EIDE utódja az **ATA–3 (AT Attachment, AT kiegészítő)** lett, az elnevezés az IBM PC/AT-re utal (az AT jelentése Advanced Technology, fejlett technológia, ami a 8 MHz-es 16 bites CPU-t jelentette). A szabvány következő változatában, az **ATAPI–4**-ben **(ATA Packet Interface, ATA-csomaginterfész)** a sebességet 33 MB/s-re növelték. Az ATAPI–5 esetében pedig 66 MB/s-re.

Erre az időre a 128 GB-os határ, amelyet a 28 bites LBA-címek alkalmazása jelentett, elkezdett egyre fenyegetőbb alakot ölteni, így az ATAPI–6 megváltoztatta az LBA-címek méretét 48 bitre. Az új szabvány akkor kerül majd bajba, ha a diszkek mérete eléri a $2^{48} \times 2^9$ bájtot (128 PB). Ha minden évben 50%-kal nő a kapacitás, a 48 bites határ kitart 2035-ig. Ha kíváncsi arra, hogyan oldották meg ezt a problémát, lapozza fel könyvünk 11. kiadását. Le merem fogadni, hogy fel fogják emelni az LBA méretét 64 bitre. Az ATAPI–6 szabvány ugyancsak megemelte az átviteli sebességet 100 MB/s-ra, és egyúttal első ízben foglalkozott a diszkek zajának csökkentésével.

Az ATAPI–7 szabvány radikálisan szakít a múlttal. Ahelyett, hogy a lemezvezérlő csatlakozójának a méretét növelnék meg (az átviteli sebesség növelése érdekében), ez a szabvány **soros ATA** átviteli módot használ arra, hogy egyszerre 1 bitet továbbítson egy 7 érintkezős csatlakozón keresztül kezdetben 150 MB/s sebességgel, ami várhatóan 1,5 BG/s fölé emelkedik. A jelenlegi 80 vezetékes szalagkábel lecserélése egy néhány milliméter vastag kerek kábellel, javítja majd a légáramlást a számítógépházban. Továbbá, a soros ATA 0,5 V feszültséget használ (összehasonlítva az ATAPI–6 meghajtóinak 5 V-os jeleivel), amely csökkenti az energiafogyasztást. Valószínűleg néhány éven belül minden számítógép soros ATA-t fog használni. A diszkek által elfogyasztott energia egyre fontosabb probléma, mind a legjobb minőségű gépekben, ahol az adatfeldolgozó központokban óriási diszkfarmok vannak, mind pedig az olcsó gépeknél, ahol a noteszgépek telepeinek energiája korlátozott.

2.3.5. SCSI-lemezek

Az SCSI-lemezek nem különböznek az IDE-lemezektől abban a tekintetben, hogy ezek is cilinderekre, sávokra és szektorokra vannak osztva, de más az interfészük, és sokkal nagyobb az adatátviteli sebességük. Az SCSI története Howard Shugart, a hajlékonylemez feltalálójának nevéhez kapcsolódik, akinek a cége bevezette a SASI- (Shugart Associates System Interface) lemezeket 1979-ben. Ezt kisebb módosítások és sok vita után az ANSI 1986-ban szabványosította, és nevét **SCSI**-re **(Small Computer System Interface, kis számítógéprendszerek interfésze)** módosította. (A SCSI kiejtése „szkazi".) Azóta az egyre gyorsabb változatokat az alábbi neveken szabványosították: Fast SCSI (10 MHz), Ultra SCSI (20 MHz), Ultra2 SCSI (40 MHz), Ultra3 SCSI (80 MHz), és Ultra4 SCSI (160 MHz). Mindegyikből van széles (16 bites) változat is. A fontosabb kombinációkat a 2.22. ábra mutatja be.

Név	Adatbitek	Sín MHz	MB/s
SCSI–1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80
Ultra3 SCSI	8	80	80
Wide Ultra3 SCSI	16	80	160
Ultra4 SCSI	8	160	160
Wide Ultra4 SCSI	16	160	320

2.22. ábra. *Néhány lehetséges SCSI-paraméter*

A SCSI-lemezek a nagyobb adatátviteli sebességük miatt a Sun, HP, SGI és más cégek legtöbb UNIX-munkaállomásának szabványfelszereléséhez tartoznak. Hasonlóan az alapfelszereléshez tartoznak a Macintosh-gépekben, és gyakoriak az igényesebb Intel PC-kben, különösen a hálózati szerverekben.

A SCSI több egy merevlemez-interfésznél. Ez egy sín, amelyre egy SCSI-vezérlő és legfeljebb hét eszköz csatlakoztatható. Ezek között lehet egy vagy több SCSI-merevlemez, CD-ROM, CD-író, szkenner, szalagegység és más SCSI-periféria. Minden SCSI-egységnek van egy 0 és 7 (széles SCSI esetén 15) közötti egyértelmű azonosítója. Minden egységnek két csatlakozója van: egy bemeneti és egy kimeneti. Az egyik egység kimenetét kábelek kötik össze a következő bemenetével sorban, mint egy olcsó karácsonyfaégő-füzérnél. A sorban utolsó eszközt le kell zárni, nehogy a SCSI sín végéről induló visszaverődések zavart okozzanak az adatforgalomban. Tipikus esetben a vezérlő egy bővítőkártyán van, amely az eszközsor első eleme, habár ezt az elrendezést nem követeli meg szigorúan a szabvány.

A leggyakoribb 8 bites SCSI-kábel 50 eres, ebből 25 földpotenciálon van, ezek mindegyikéhez párosítva van egy a többi 25 érből, így lehet elérni a nagy sebességű adatátvitelhez nélkülözhetetlen zajtűrő képességet. A 25 érből 8 adat, 1 paritás és 9 vezérlővonal, a többi pedig tápfeszültség vagy jövőbeli fejlesztésekre fenntartott. A 16 bites (és 32 bites) egységekhez szükség van egy második kábelre is. A kábelek hosszúsága több méter is lehet, így külső meghajtók, szkennerek stb. is csatlakoztathatók.

A SCSI-vezérlők és -perifériák kezdeményező és fogadó üzemmódban működhetnek. Általában a kezdeményezőként működő vezérlő adja ki a parancsokat a fogadóként viselkedő lemezegységeknek és egyéb perifériáknak. Ezek a parancsok legfeljebb 16 bájtot tartalmazó blokkok, amelyek megmondják a fogadóknak, hogy mit kell tenniük. A parancsok és a válaszok fázisonként követik egymást, különféle vezérlőjeleket használnak a fázisok kialakításához és ahhoz, hogy a sín használatát szabályozzák, ha több eszköz is egyszerre próbál hozzáférni. Ez az ütemezés fontos, mert a SCSI-szabvány megengedi, hogy az összes eszköz egyszerre működjön, így nagyban növelhető a hatékonyság több folyamatot futtató környezetben (mint például a UNIX vagy Windows NT). Az IDE- és EIDE-vezérlők esetében egyszerre csak egy eszköz lehet aktív.

2.3.6. RAID

Az elmúlt évtizedben a CPU-k teljesítménye exponenciálisan növekedett, másfél évenként nagyjából megkétszereződött. Nem így a lemezek teljesítménye. Az 1970-es években a miniszámítógépek lemezeinek átlagos keresési ideje 50 és 100 ms közé esett. Jelenleg a keresési idő 10 ms. A legtöbb iparágban (például a gépkocsigyártásban vagy a repülőgépiparban) két évtized alatt 5-10-szeres teljesítménynövekedés jelentős eredmény volna, de a számítógépiparban ez szégyen. A CPU és a lemezek teljesítménye közötti hézag tehát sokkal nagyobb lett.

Ahogy láttuk, a CPU teljesítményének növelésére a párhuzamosságot egyre jobban kihasználják. Többekben felvetődött már az évek során, hogy a párhuzamos B/K szintén jó ötlet lehetne. 1988-as cikkükben, Patterson és társai hat olyan lemez-összekapcsolási módot javasoltak, amelyek vagy a teljesítmény, vagy a megbízhatóság, vagy mindkettő növelésére használhatók (Patterson és társai, 1988). Ezeket az ötleteket hamarosan ipari körülmények között is alkalmazták, és ez elvezetett egy új típusú B/K eszköz megszületéséhez, amelynek a neve: **RAID**. Pattersonék RAID-definíciója eredetileg **olcsó lemezek redundáns tömbje (Redundant Array of Inexpensive Disks)** volt, de az ipar az I-t Independentre (független) változtatta meg az Inexpensive-ről (talán azért, hogy drága lemezeket is használhassanak?). Mivel egy negatív szereplőre is szükség volt (mint a RISC és CISC esetében, Patterson szerint), a rosszfiú neve **SLED (Single Large Expensive Disk, egyetlen nagy, drága lemez)** lett.

A RAID alapötlete az, hogy a számítógép (tipikusan egy nagy szerver) mellé telepítünk egy dobozt tele lemezegységekkel, a lemezvezérlőt kicseréljük egy RAID-vezérlőre, átmásoljuk az adatokat a RAID-re, aztán folytatjuk a munkát.

Más szóval, a RAID-nek az operációs rendszer felé úgy kell viselkednie, mint egy SLED-nek, amelynek azonban nagyobb a teljesítménye és a megbízhatósága. Mivel a SCSI-lemezeknek jó a teljesítménye, alacsony az ára és egyetlen vezérlő akár 7 (széles SCSI esetén 15) meghajtót is képes kezelni, elég természetes, hogy a legtöbb RAID egy RAID SCSI-vezérlőből és sok SCSI-lemezből áll, ami az operációs rendszer felé egyetlen nagy lemeznek látszik. Ezzel a módszerrel a RAID használatához semmilyen szoftverváltoztatás nem szükséges, és ez bizony fontos értékesítési szempont sok rendszeradminisztrátor számára.

Azon túl, hogy a RAID szoftverszempontból egyetlen lemeznek látszik, az adatok szét vannak osztva a meghajtók között, lehetővé téve a párhuzamos működést. Pattersonék ennek több lehetséges módját is definiálták, ezek ma a RAID 0-tól RAID 5-ig terjedő szintek. Ezeken túl van még jó néhány kisebb szint is, amelyet nem tárgyalunk. A „szint" elnevezés sem igazán jogos, mert valójában nincs hierarchia, csupán hat különböző szervezési mód.

A RAID-0 a 2.23. (a) ábrán látható. Ennél a RAID által szimulált virtuális lemez k darab szektorból álló csíkokra van felosztva, a 0-tól a $k - 1$-ig terjedő szektorok a 0. csoporthoz tartoznak, k-tól a $2k - 1$-ig a 1. csoporthoz és így tovább. $k = 1$ esetén minden csík egyetlen szektorból áll, $k = 2$ esetén két szektorból stb. A RAID-0 az egymás után következő csíkokat körben forgó módszerrel írja fel a lemezekre, ahogy azt a 2.23. (a) ábrán láthatjuk 4 RAID-meghajtó esetén. Ezt a fajta adatelosztást **csíkozás**nak **(striping)** hívják. Például, ha a program egy olyan adatblokkot akar beolvasni, amelyik csoporthatáron kezdődik, és négycsoportnyi helyet foglal el, a RAID-vezérlő az olvasási parancsot felbontja négy különálló parancsra az egyes diszkek számára, és párhuzamosan működteti a meghajtókat. Ezen a módon párhuzamos B/K műveleteket lehet végezni a program tudta nélkül.

A RAID-0 nagyméretű blokkokkal működik a legjobban, minél nagyobbak, annál jobb. Ha egy kérés nagyobb, mint a lemezek száma szorozva a csík méretével, egy meghajtó több kérést is fog kapni, amikor az elsőt befejezte, kezdi a másodikat. A vezérlő feladata a kérések feldarabolása, a megfelelő részkérések eljuttatása a megfelelő lemezekhez a megfelelő sorrendben, végül a részeredmények helyes összeállítása a memóriában. A teljesítmény kiváló, a megvalósítás pedig magától értetődik.

A RAID-0 olyan operációs rendszerekkel teljesít a leggyengébben, amelyeknek az a szokása, hogy szektoronként kezelik az adatokat. Az eredmények helyesek lesznek, de nincs párhuzamosság, és így nincs teljesítménynövekedés sem. Másik hátrány, hogy a megbízhatóság még rosszabb is lehet, mint SLED esetén. Ha egy RAID négy lemezből áll, és egy meghajtó átlagosan 20 000 óránként hibásodik meg, kb. 5000 óránként egy meghibásodik a négyből, és az összes adat elvész. Egyetlen SLED 20 000 órás átlagos meghibásodási idővel négyszer megbízhatóbb lenne. Mivel nincs igazi redundancia, ez nem is igazi RAID.

A 2.23. (b) ábrán látható a következő lehetőség, a RAID-1, amely már igazi RAID. Minden lemezt megdupláz, tehát négy elsődleges és négy tartalék lemez van. Írás esetén minden csoport kétszer kerül kiírásra. Olvasás esetén a kettő közül bármelyik másolat használható, így a terhelés több meghajtóra oszlik el. Tehát az írás nem hatékonyabb, mint egyetlen meghajtó esetén, de az olvasás kétszer

olyan gyors lehet. A meghibásodások elleni védelem kiváló: ha egy meghajtó elromlik, a párját használják helyette. A helyreállítás mindössze abból áll, hogy beállítanak egy új meghajtót, és rámásolják az épen maradt diszk tartalmát.

Eltérően a 0-s és az 1-es szintektől, amelyek szektorcsoportokkal dolgoznak, a RAID-2 szóalapú, esetleg bájtalapú is lehet. Képzeljük el, hogy a virtuális lemez minden bájtját két 4 bites részre vágunk, majd Hamming-kódolással olyan 7 bites szavakat képezünk, amelyekben az 1., a 2. és a 4. bitek paritásbitek. Ezen túl képzeljük még el, hogy a 2.23. (c) ábra hét meghajtójának karjai sugár irányban és forgási pozíció tekintetében szinkronizálva vannak. Ekkor lehetséges a Hamming-kódolással előállított 7 bites szót a hét meghajtóra úgy felírni, hogy mindegyikre 1 bit kerüljön.

A Thinking Machines CM–2 számítógép ezt a sémát használta, egy 32 bites adatszóhoz 6 paritásbitet adva 38 bites Hamming-kódszó jött létre, ehhez még egy kiegészítő paritásbitet adtak, és mindezt 39 meghajtón tárolták bitenként. Az adatátviteli sebesség rendkívüli volt, mert egy szektornyi idő alatt 32 szektornyi adatot lehetett felírni. Egy meghajtó elvesztése nem okozott gondot, mert az 1 bit elvesztését jelentette minden egyes 39 bites szóból, amit a Hamming-kóddal menet közben lehetett kezelni.

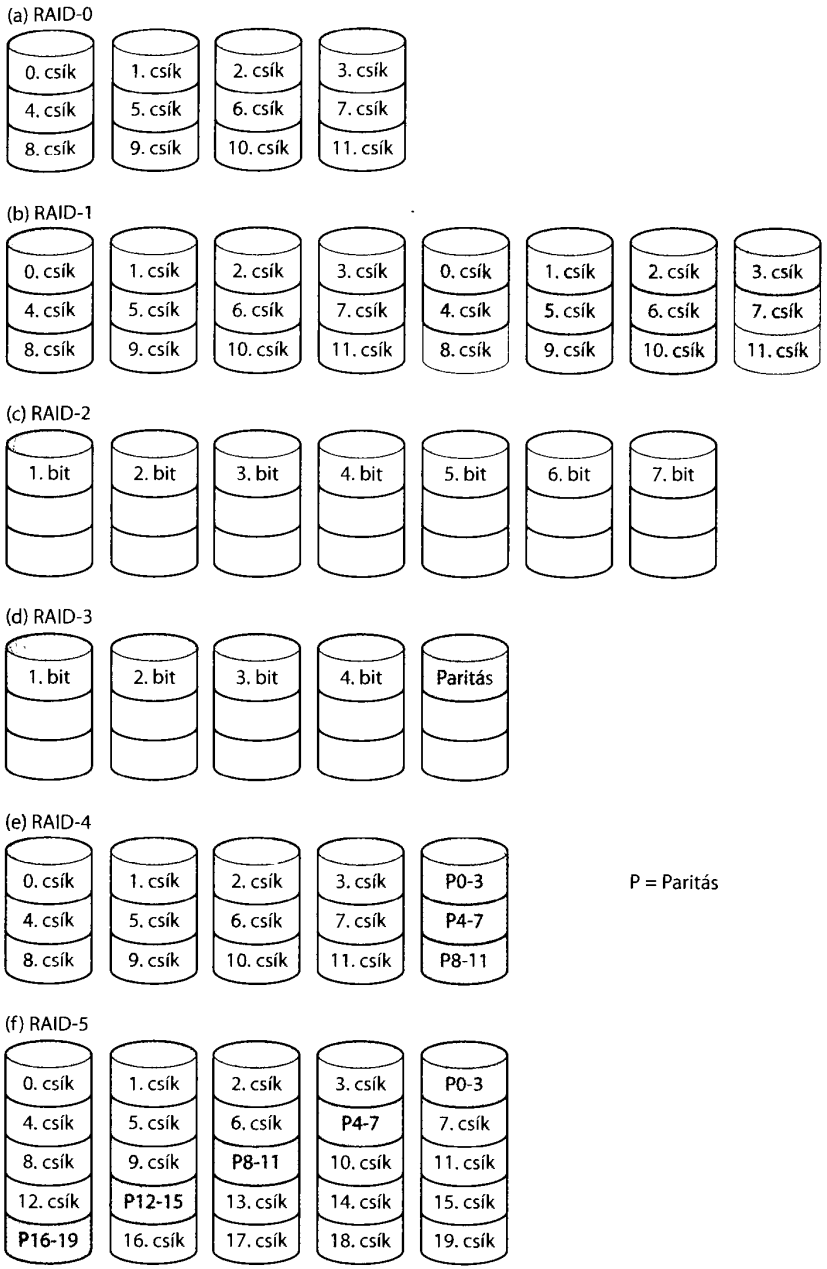
Hátránya ennek a sémának, hogy az összes meghajtó forgásának szinkronban kell lennie, és csak elég sok meghajtó használata esetén van értelme (még 32 adat- és 6 paritásmeghajtó esetén is a többletráfordítás 19%). A vezérlőnek is sok a dolga, mert minden bitmozgatási idő alatt egy Hamming-kódot is ki kell számítania.

A RAID-3 a RAID-2 egy egyszerűsített változata, amelyet a 2.23. (d) ábra illusztrál. Itt minden adatszóhoz egyetlen paritásbit van rendelve, amelyet egy külön paritásmeghajtón tárolunk. Mint a RAID-2 esetében is, a meghajtókat szinkronizálni kell, mert az adatszavak szét vannak terítve több meghajtóra.

Első ránézésre úgy tűnhet, hogy egyetlen paritásbit csak hibafelismerésre jó, hibajavításra nem. Véletlenül felmerülő hibák esetén ez igaz is, csakhogy egy meghajtó meghibásodása esetén ez egy teljes 1 bites hibajavítást tesz lehetővé, hiszen pontosan tudjuk, melyik diszk romlott el, ismerjük a hibás bit pozícióját, tehát 1 bithibát mindig ki tudunk javítani. Ha valamelyik meghajtó meghibásodik, a vezérlő úgy tesz, mintha a hibás diszkről érkező minden bit 0 lenne. Ha egy szóban paritáshiba van, a rossz meghajtóról származó bitnek 1-esnek kellett volna lennie, és ennek megfelelően javításra kerül. Jóllehet mind a RAID-2, mind a RAID-3 nagyon magas adatátviteli sebességet nyújt, a másodpercenként feldolgozható B/K kérések száma nem nagyobb, mint egyetlen meghajtó esetén.

A RAID-4 és RAID-5 megint csak csíkozással dolgozik, nem paritásbitekkel ellátott egyedi szavakkal, ezért ezek nem igénylik a meghajtók szinkronizálását. A RAID-4 [lásd 2.23. (e) ábra] hasonlít a RAID-0-ra, azonban a csíkonkénti paritást felírja egy külön meghajtóra. Például, ha egy csoport k bájtból áll, az összetartozó csíkokat KIZÁRÓ VAGY művelettel összekapcsolva egy k bájtos paritáscsík jön létre. Ha egy meghajtó tönkremegy, az elvesztett bájtokat visszakaphatjuk a paritásmeghajtó segítségével.

Ez a megoldás védelmet nyújt egy meghajtó elvesztése ellen, de nem hatékony, ha gyakran kell kis mennyiségű adatot újraírnunk. Ha egyetlen szektor változik



2.23. ábra. *RAID-szintek 0-tól 5-ig. A tartalék és paritásmeghajtók szürke tónusúak*

meg, minden meghajtóról kell olvasni a paritásbit kiszámításához, amit ezután fel kell írni a paritásmeghajtóra. Alternatív megoldásként be lehet olvasni a régi felhasználói adatot és a régi paritást, majd az új paritást ezekből kiszámítani. Még ezzel az optimalizálással is egyetlen kicsi változtatás két olvasást és két írást igényel, így ez a megoldás nyilvánvalóan rossz.

A paritásmeghajtóra nehezedő nagy terhelés miatt az válhat a szűk keresztmetszetté. Ezt a szűk keresztmetszetet a RAID-5 azzal szünteti meg, hogy a paritásbiteket egyenletesen, körbejárásos módszerrel szétosztja a meghajtók között, ahogyan az a 2.23. (f) ábrán látható. Ha egy meghajtó tönkremegy, annak tartalmát ugyanúgy lehet rekonstruálni, mint RAID-4 esetén.

2.3.7. CD-ROM

Az optikai lemezeket eredetileg televízióadások rögzítésére fejlesztették ki, de számítógépes adathordozóként sokkal esztétikusabban használhatók. Nagy kapacitásuk és alacsony áruk miatt széles körben használják szoftverek, könyvek, mozifilmek és mindenféle adat terjesztésére, valamint mágneslemezek tartalmának biztonsági mentésére.

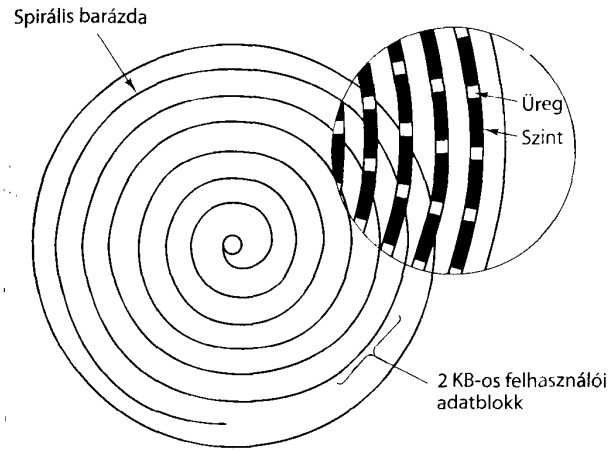
Az első generációs optikai lemezeket a holland elektronikai cég, a Philips fejlesztette ki mozifilmek tárolására. Átmérőjük 30 cm volt, és LaserVision név alatt kerültek piacra, de Japán kivételével nem volt nagy sikerük.

1980-ban a Philips és a Sony megtervezte a CD-t (Compact Disk, kompaktlemez), amely gyorsan felváltotta a $33^1/_3$ percenkénti fordulatszámmal lejátszott bakelit hanglemezeket. A CD pontos technikai részleteit egy hivatalos nemzetközi szabványban (IS 10149) rögzítették, amelynek elterjedt elnevezése **Red Book (Vörös Könyv)**, borítójának színe után. (A nemzetközi szabványokat a Nemzetközi Szabványügyi Hivatal adja ki, amely a nemzeti szabványügyi hivatalok, mint például az ANSI, American National Standard Institute vagy a DIN nemzetközi megfelelője. Minden szabványnak van egy IS száma.) A CD-lemez- és a meghajtóspecifikáció nemzetközi szabványként való publikálásának a célja, hogy ezáltal a zenekiadók és az elektronikai berendezéseket gyártó cégek együtt tudjanak működni. Minden CD 120 mm átmérőjű és 1,2 mm vastag, közepén egy 15 mm-es lyukkal. Az audio-CD volt az első sikeres tömeggyártású digitális adathordozó. Élettartamukat legalább 100 évre becsülik. Kérjük az olvasót, 2080-ban nézzen utána, milyen állapotban vannak az első példányok.

Egy CD úgy készül, hogy nagy energiájú infravörös lézerrel 0,8 mikron átmérőjű lyukakat égetnek egy bevonattal ellátott, üveg mesterlemezbe. Erről a lemezről negatív öntőforma készül, amelyen kiugrások vannak az eredeti lyukak helyén. Az öntőformába olvadt polikarbonát gyantát töltenek, így egy olyan CD-t kapnak, amelyen a lyukak mintázata azonos a mesterlemezével. Ezután egy nagyon vékony fényvisszaverő alumíniumréteg, majd egy lakk védőréteg és egy címke kerül a lemezre. A polikarbonát rétegben elhelyezkedő mélyedéseket **üreg**nek **(pit)**, az üregek közötti érintetlen területeket pedig **szint**nek **(land)** hívják.

Visszajátszáskor egy kis energiájú lézerdióda 0,78 mikron hullámhosszúságú infravörös fénnyel világítja meg az üregeket és a szinteket, ahogy elhaladnak alatta. A lézer a polikarbonát oldalon van, ezért az üregek a lézer felé kidudorodnak az amúgy sima felületből. Mivel az üregek magassága negyede a lézerfény hullámhosszának, az üregekről visszaverődő fény fél hullámhossznyi fáziseltolódásban van a környező területről visszaverődő fényhez képest. Ennek eredménye, hogy a két rész interferenciája gyengíti egymást, emiatt a lejátszó fényérzékelőjébe kevesebb fény jut annál, mint amikor a fény a szintről verődik vissza. Így különbözteti meg a lejátszó az üreget a szinttől. Jóllehet az tűnik a legegyszerűbbnek, hogy üreget használjunk a 0, szintet az 1 tárolásához, ennél azonban megbízhatóbb, ha az üreg/szint vagy a szint/üreg átmenetet használjuk az 1-hez, az átmenet hiányát pedig a 0-hoz, ezért ez utóbbi módszert alkalmazzák.

Az üregek és a szintek egyetlen folytonos spirálban kerülnek felírásra, amely a lyuk közelében kezdődik és a lemez széle felé tartó 35 mm széles sávot foglal el. A spirál 22 188 fordulatot tesz meg a középpont körül (kb. 600-at milliméterenként). Ha letekernénk, 5,6 km hosszú lenne. A spirál a 2.24. ábrán látható.



2.24. ábra. *Adattárolás a kompaktlemezen vagy CD-ROM-on*

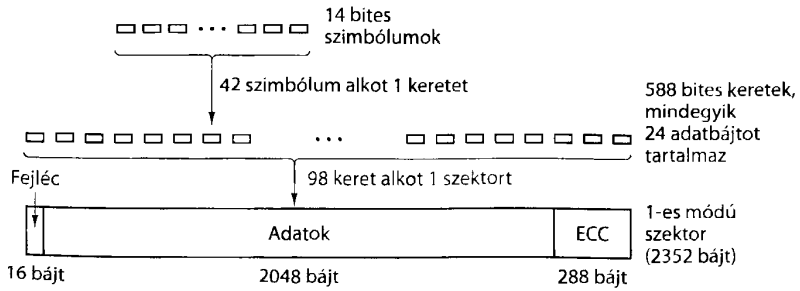
A zene egyenletes lejátszásához szükséges, hogy az üregek és a szintek állandó sebességgel mozogjanak. Emiatt a CD forgási sebességét fokozatosan csökkenteni kell, ahogy az olvasófej a belső területekről kifelé tart. A belső részen 530 RPM (Rotation Per Minute, percenkénti fordulatszám) szükséges a kívánt 120 cm/s olvasási sebességhez; a külső részen 200 RPM-re kell lassítani, hogy a fejnél ugyanazt a kerületi sebességet kapjuk. Egy állandó kerületi sebességgel működő meghajtó nagyban különbözik egy mágneslemez-meghajtótól, amely a fej pozíciójától függetlenül állandó szögsebességgel forog. Ezenkívül az 530 RPM nagyon messze van a 3600–7200 RPM-től, amivel a legtöbb mágneslemez forog.

1984-ben a Philips és a Sony felismerte, hogy lehetséges számítógépes adatokat is tárolni a CD-n, így aztán kiadták a **Yellow Book**ot (**Sárga Könyv**), amely a mai CD-ROM (**Compact Disk – Read Only Memory**) pontos szabványát írja le. Az akkor már jelentős audio CD-piacot meglovagolva, a CD-ROM-oknak az audio-CD-kkel megegyező méretűnek kellett lenniük, mechanikusan és optikailag kompatibiliseknek egymással, és ugyanazokkal a polikarbonát fröccsöntő gépekkel lehetett gyártani őket. Ennek a döntésnek a következménye, hogy lassú, változtatható sebességű motorokra volt szükség, de egyúttal egy CD-ROM előállítási költsége nagyobb széria esetén jóval 1 dollár alatt lehet.

A Yellow Book a számítógépes adatok tárolási formátumát definiálta. A rendszer hibajavító képességét is növelték, ami elengedhetetlen is volt, mert a zenekedvelőknek ugyan nem fáj egy-két bit elvesztése itt vagy ott, a számítógép-felhasználók viszont nagyon nyűgösek tudnak lenni miatta. A CD-ROM alapformátuma szerint minden bájtot egy 14 bites szimbólum tárol. Ahogy fentebb láttuk, 14 bit elegendő ahhoz, hogy egy 8 bites bájtot Hamming-kódolással kódoljunk, még marad is 2 bit. Tulajdonképpen egy ennél még erősebb kódolási rendszert használnak. Olvasáskor a 14 bit 8 bitre történő leképezése hardveres úton történik egy beépített konverziós tábla segítségével.

A következő szinten 42 egymás után következő szimbólum képez egy 588 bites **keret**et. Minden keret 192 adatbitet tartalmaz (24 bájt). A maradék 396 bit hibajavításra és vezérlő információk tárolására szolgál. Eddig ez a séma ugyanaz az audio-CD-k és a CD-ROM-ok esetén.

A Yellow Book ehhez azt adja még hozzá, hogy 98 keretet egy CD-ROM-szektorba csoportosít, ahogyan ez a 2.25. ábrán látható. Minden **CD-ROM-szektor** egy 16 bájtos bevezetővel indul, ennek első 12 bájtja (hexadecimális) 00FFFFFFFFFF FFFFFFFFFF00, a lejátszó ebből ismeri fel a szektor elejét. A következő 3 bájt a szektor számát tartalmazza – erre azért van szükség, mert az egyetlen hosszú spirálra felírt adatok esetén a keresés sokkal nehezebb, mint a koncentrikus sávokkal rendelkező mágneslemeznél. Kereséskor a meghajtó szoftvere meghatározza, hogy a célterület nagyjából hol helyezkedik el, odaviszi a fejet, majd elkezd kutatni egy bevezető után. A bevezető utolsó bájtja a mód.



2.25. ábra. *Az adatok logikai elhelyezkedése egy CD-ROM-on*

A Yellow Book két módot definiál. Az 1-es mód a 2.25. ábra szerinti elrendezést használja 16 bájtos bevezetővel, 2048 adatbájttal és 288 bájt hibajavító kóddal (cross-interleaved Reed–Solomon-kód). A 2-es mód az adat és az ECC mezőket egyetlen 2336 bájtos adatmezőbe foglalja azoknak az alkalmazásoknak a kedvéért, amelyek nem igénylik a hibajavítást (vagy nem tudnak időt fordítani rá), mint például audio- és videoalkalmazások. Figyeljük meg, hogy a kiváló megbízhatóság elérése érdekében három különböző hibajavító sémát alkalmaznak: egyet szimbólumokra, egyet keretekre és egyet a CD-ROM-szektorokra. Egyszeres bithibákat a legalsó szinten, rövid szakaszra kiterjedő többszörös hibákat a keretek szintjén, míg minden más hibát a szektorok szintjén javítanak ki. Ennek a megbízhatóságnak az az ára, hogy 98 darab 588 bites keret (7203 bájt) kell 2048 bájt tárolásához, ami csak 28 százalékos kihasználtságot jelent.

Az egyszeres sebességű CD-ROM-meghajtók 75 szektor/s sebességgel működnek, ez 153 600 bájt/s adatátviteli sebességet jelent 1-es módban, 175 200 bájt/s sebességet 2-es módban. A kétszeres sebességű meghajtók kétszer ilyen gyorsak, és így tovább a legnagyobb sebességűekig. Egy szokványos audio-CD 74 percnyi zene tárolására alkalmas, ha ezt adatok tárolására használjuk 1-es módban, 681 984 000 bájtos kapacitást jelent. Ezt általában 650 MB-nak írják, mert 1 MB az 2^{20} bájt (1 048 576 bájt), nem 1 000 000 bájt.

Még egy 32-szeres CD-ROM-meghajtó (4 915 200 bájt/s) sem versenyezhet egy gyors SCSI-2 mágneslemez-meghajtó 10 MB/s sebességével, habár sok CD-ROM-meghajtó a SCSI-csatolót használja (IDE CD-ROM-meghajtók is léteznek). Ha még hozzávesszük, hogy a keresési idő gyakran több száz milliszekundum, világossá kell válnia, hogy a CD-ROM-meghajtók nagy kapacitásuk ellenére egyáltalán nincsenek a mágneslemez-meghajtókkal egy kategóriában.

1986-ban a Philips újra hallatott magáról a **Green Book**kal (**Zöld Könyv**), amely a grafikus anyagok tárolását határozta meg, valamint lehetővé tette egy szektoron belül audio-, video- és egyéb adatok egyidejű elhelyezését, ami a multimédiás CD-ROM-okhoz elengedhetetlen tulajdonság.

A CD-ROM-történet utolsó darabja a fájlrendszer. Ahhoz, hogy különböző számítógépekben lehessen használni a CD-ROM-okat, meg kellett egyezni a CD-ROM-fájlrendszer formátumában. Az egyezség megszületése érdekében számos számítógépgyártó cég képviselője találkozott a Kalifornia és Nevada határán lévő High Sierra hegységben, a Tahoe-tónál, és javaslatot tettek egy fájlrendszerre, amelyet **High Sierrá**nak neveztek el. Ez később nemzetközi szabvánnyá fejlődött (IS 9660). Három szintje van. Az 1-es szint maximum 8 karakteres fájlneveket használ, ezt követheti egy kiterjesztés legfeljebb 3 karakteren (ez az MS-DOS fájlnévkonvenció). A fájlnevek csak nagybetűket, aláhúzás jeleket és számjegyeket tartalmazhatnak. Az alkönyvtárak legfeljebb 8 mélységig lehetnek egymásba ágyazva, és az alkönyvtárneveknek nem lehet kiterjesztése. Az 1-es szint megköveteli, hogy minden állomány folytonos legyen, de ez nem is probléma egy csak egyszer írható adathordozó esetén. Bármely IS 9660 szerinti 1-es szintű CD-ROM olvasható MS-DOS-, Apple- és UNIX-gépen, vagy ami azt illeti, szinte bármelyik gépen. A CD-ROM-ok kiadói ezt nagy előnynek tekintik.

Az IS 9660 2-es szint megenged legfeljebb 32 karakter hosszúságú fájlneveket, a 3-as szint pedig nem folytonos állományokat. A Rock Ridge kiterjesztések [furcsa módon a *Fényes Nyergek* (Blazing Saddles) című Mel Brooks-filmben szereplő városról kapta a nevét] megengednek nagyon hosszú fájlneveket (a UNIX számára), UID-ket, GID-ket és szimbolikus linkeket, de az 1-es szint követelményeit nem teljesítő CD-ROM-ok nem olvashatók minden számítógépen.

2.3.8. Írható CD-k

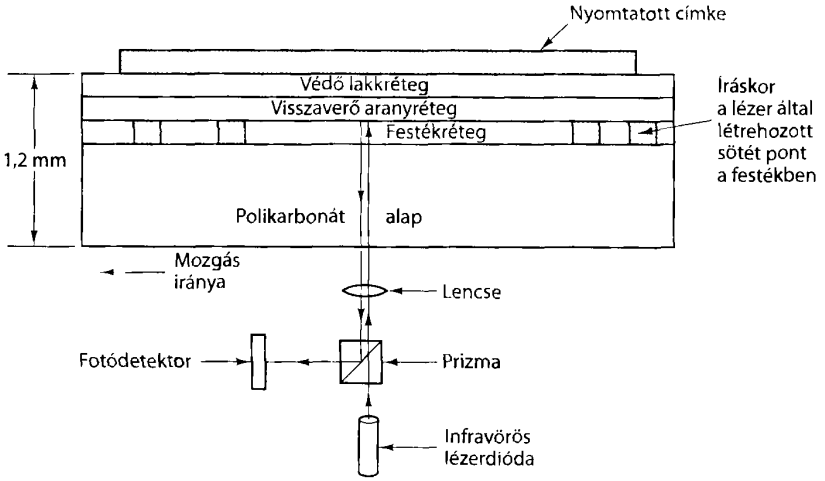
Kezdetben a mester CD-ROM (vagy éppenséggel audio-CD) elkészítéséhez szükséges berendezés rendkívül drága volt. De ahogy az a számítógépiparban megszokott, semmi sem marad sokáig drága. Az 1990-es évek közepére a CD-lejátszóknál nem nagyobb CD-írók a legtöbb számítógépboltban kaphatók voltak. Ezek az eszközök még mindig különböztek a mágneslemezektől, mert ha egyszer megírták, a CD-ROM-okat nem lehetett törölni. Ennek ellenére hamar felhasználták őket nagy mágneslemezek biztonsági másolataihoz, ezenkívül egyének és kisebb cégek elkészíthették saját, kis sorozatú CD-ROM-jaikat vagy egy mesterlemezt a nagy szériát előállító kereskedelmi CD-sokszorosító üzemek számára. Ezeknek a meghajtóknak **CD-R (CD-Recordable, írható CD)** volt a nevük.

Fizikailag az írható CD-khez ugyanolyan üres 120 mm-es polikarbonát lemezeket használnak, amelyek hasonlítanak a CD-ROM-okra, kivéve, hogy egy 0,6 μm szélességű barázda van rajtuk az író lézernyaláb irányítására. A barázdának van egy 0,03 μm-es szinusz hullámú kitérése pontosan 22,05 kHz frekvenciával, ami folyamatos visszacsatolást biztosít, így a forgási sebesség pontosan ellenőrizhető, és szükség esetén korrigálható. A CD-R-ek úgy néznek ki, mint a szokványos CD-ROM-ok, csak annyi a különbség, hogy a felső oldaluk aranyszínű, nem ezüst. Az aranyszín azért van, mert valódi aranyat használnak visszaverő rétegnek alumínium helyett. Az ezüstös CD-kkel ellentétben, amelyekben valódi bemélyedések vannak, a CD-R-ek esetében az üregek és szintek különböző visszaverő képességét utánozni kell valahogy. Ezt úgy érik el, hogy egy festékréteget helyeznek el a polikarbonát és a visszaverő aranyréteg közé, ahogy az a 2.26. ábrán látható. Kétféle festék használatos: cianin, amely zöld, és ftalocianin, amely sárgás narancsszínű. A kémikusok vég nélkül tudnak vitatkozni azon, hogy melyik jobb. Ezek a festékek hasonlók a fotózásban használtakhoz, ami megmagyarázza, hogy miért a Kodak és a Fuji a legnagyobb CD-R-gyártó. Néha alumínium visszaverő réteg helyettesíti az aranyat.

Kezdeti állapotában a festékréteg átlátszó, a lézerfényt átengedi, és visszaverődik az aranybevonatról. Az íráshoz a CD-R-lézert nagy energiára (8–16 mW) kapcsolják. Amikor a sugár egy festékfoltot talál el, az felmelegszik, és egy kémiai kötés felbomlik. A molekuláris szerkezet megváltozása sötét foltot hoz létre. Visszaolvasáskor (0,5 mW) a fényérzékelő egység különbséget tud tenni a lézerrel elroncsolt festék sötét foltjai és az épen hagyott átlátszó területek között. A színkülönbségeket úgy kezelik, mintha üregek és szintek közötti különbségek lennének, még akkor is, ha közönséges CD-ROM-olvasón vagy egy audio-CD-lejátszón olvassák le.

Egyetlen újfajta CD sem állhatna a világ elé emelt fővel egy színes könyv nélkül, a CD-R-nek az **Orange Book (Narancssárga Könyv)** jutott, amelyet 1989-ben adtak ki. Ebben a dokumentumban van a CD-R, valamint egy új formátum, a CD-ROM XA definíciója; ez utóbbi az adatok inkrementális felírását teszi lehetővé, néhány szektort ma, néhányat holnap, néhányat a jövő héten. Az egyszerre felírt egymás után következő szektorokat **CD-ROM sáv**nak **(CD-ROM track)** nevezzük.

A CD-R egyik legelső felhasználása a Kodak PhotoCD volt. Ennél a vásárló behoz egy tekercs filmet és a PhotoCD-jét az üzletbe, ahol az új fényképeit a régiek mellé felírják. A negatívok digitalizálásával kapott új sorozatot egy új CD-ROM-sávként hozzák létre a PhotoCD-n. Az inkrementális írásra azért van szükség, mert az üres CD-R-lemezek túl drágák ahhoz, hogy minden tekercs filmhez elhasználjanak egyet.



2.26. ábra. *Egy CD-R keresztmetszete a lézerrel (nem méretarányos). A CD-ROM-oknak hasonló a szerkezete, kivéve, hogy hiányzik a festékréteg, és üreges alumíniumréteg van a fényvisszaverő réteg helyett*

Az inkrementális írás azonban új problémát is felvet. Az Orange Book előtt minden CD-ROM-nak egyetlen **tartalomjegyzéke (VTOC, Volume Table of Contents)** volt, az elején. Ez a módszer nem felel meg inkrementális (többsávos) írás esetén. Az Orange Book azt a megoldást adta, hogy minden sávhoz külön tartalomjegyzéket rendelt. A tartalomjegyzékben szereplő fájlok között szerepelhetnek az előző sáv fájljai is. Miután egy új CD-R kerül a meghajtóba, az operációs rendszer az összes CD-R-sávot végignézi, hogy a legutoljára felírt tartalomjegyzéket megtalálja, ami egyúttal a lemez aktuális állapotát is megadja. Ha a korábbi sávok fájljai közül nem mindegyiket adjuk hozzá a következő sáv tartalomjegyzékéhez, a fájltörlés illúzióját kelthetjük. Több sáv összefogható egy **szekció**ba **(session)**, így **többszekciós (multisession)** CD-ROM-okat kapunk. A hagyományos audio-CD-

lejátszók nem képesek kezelni a többszekciós CD-ROM-okat, mert feltételezik, hogy egyetlen tartalomjegyzék van a lemez elején.

Minden sávot egyetlen folytonos művelettel, megszakítás nélkül kell felírni. Következésképpen az adatok forrásául szolgáló mágneslemeznek elég gyorsnak kell lennie, hogy a felírandó bájtok időben rendelkezésre álljanak. Ha a felírandó fájlok a lemezen szerteszét vannak, a keresési idő túl hosszúra nyúlhat, és a CD-R puffere kiürülhet; ez okozza a rettegett pufferkiürülést. A puffer kiürülésének eredménye egy szépen fénylő (habár egy kicsit drága) söralátét, avagy egy 120 mm-es arany- vagy ezüstszínű frizbi. A CD-R-szoftverek általában felajánlják azt a lehetőséget, hogy a felírandó fájlokat egyetlen folytonos 650 MB méretű CD-ROM-fájlba gyűjtsük össze, de ez rendszerint megkétszerezi a felírási időt, 650 MB szabad diszkhelyet igényel, és még mindig nem segít az olyan merevlemezek esetén, amelyek pánikba esnek, és újrakalibrálják magukat, ha nagyon felmelegednek.

A CD-R lehetővé teszi, hogy egyes személyek vagy cégek könnyen lemásoljanak CD-ROM-okat (vagy audio-CD-ket), rendszerint megsértve ezzel a kiadók szerzői jogait. Sok olyan módszert fejlesztettek ki, amely megnehezíti az ilyen kalózkodást, valamint azt, hogy a CD-ROM-ot a kiadóén kívül más szoftverrel is el lehessen olvasni. Egyik módszer az, hogy a CD-ROM-on található összes fájl hosszát több gigabájtosra állítják be, megakadályozva ezzel, hogy a szokványos másolóprogramokkal a fájlokat merevlemezre másolják. Az igazi hosszak a kiadó szoftverében vannak eltárolva vagy a CD-ROM-on elrejtve (esetleg titkosítva) olyan helyre, ahol nem számítanak rá. Egy másik módszer szándékosan hibás ECC-ket használ bizonyos kiválasztott szektorokban, arra számítva, hogy a CD-t másoló program „kijavítja" ezeket a hibákat. Az alkalmazói program aztán megvizsgálja az ECC-ket a kiválasztott szektorokban, és nem hajlandó működni, ha helyesek. A sávok közötti nem szabványos méretű hézagok használata, és más fizikai „hiba" is a lehetőségek közé tartozik.

2.3.9. Újraírható CD-k

Habár az emberek más, csak írható adathordozókhoz is hozzászokhattak – mint például a papír vagy a fényképezéshez használt film –, igény van újraírható CD-ROM-ra. Egy rendelkezésre álló technológia ma a **CD-RW (CD-ReWritable)**, amely CD-R méretű lemezeket használ. Utóbbitól eltérően azonban az adattároló réteg cianin és ftalocianin festék helyett ezüst, indium, antimon és tellúr egy ötvözetét tartalmazza. Ennek az ötvözetnek két stabil állapota van: kristályos és amorf, különböző fényvisszaverő tulajdonságokkal.

A CD-RW-meghajtók három eltérő energiájú lézert alkalmaznak. A legmagasabb energián az ötvözet megolvad, és a nagy visszaverő képességű kristályos állapotból a kis visszaverő képességű amorf állapotba kerül, ezzel egy üreget reprezentálva. Közepes energián az ötvözet megolvad, és visszatér természetes kristályos állapotába, ezzel újból szintállapotba kerül. Kis energián az anyag állapotát lehet érzékelni (olvasáshoz), de nem történik átalakulás.

A CD-RW azért nem szorította ki a CD-R-t teljesen, mert az üres CD-RW-lemezek drágábbak, mint az üres CD-R-lemezek. Ezenkívül az olyan alkalmazási területeken, mint a merevlemezek biztonsági mentése, nagy előny, hogy az egyszer felírt CD-R-lemezt nem lehet véletlenül törölni.

2.3.10. DVD

Az alap CD/CD-ROM formátum már 1980 óta létezik. Azóta a technológia sokat fejlődött, ezért már nagyobb kapacitású optikai lemezek készíthetők gazdaságosan, és nagy is a kereslet iránt uk. Hollywood rendkívül szeretné az analóg videoszalagokat lecserélni digitális lemezekre, mert a lemezeknek jobb a minősége, olcsóbb őket előállítani, tovább tartanak, kevesebb helyet foglalnak a videoüzletekben, és nem kell őket visszatekercselni. A szórakoztatóelektronikai cégek egy új, elsöprő sikerű terméket keresnek, sok számítógépes cég pedig multimédia-tulajdonságokkal akarja szoftvereit ellátni.

A technológia és a három hihetetlenül gazdag és hatalmas iparágban mutatkozó kereslet kombinációja vezetett a **DVD**-hez, amely eredetileg a **Digital Video Disk**, ma hivatalosan a **Digital Versatile Disk (Sokoldalú Digitális Lemez)** rövidítése. A DVD alapfelépítése ugyanaz, mint a CD-é, egy 120 mm-es, öntőformában készített polikarbonát lemez, amely üregeket és szinteket tartalmaz; ezeket egy lézerdióda világítja meg, és egy fotódetektor olvassa le a jeleket. Amiben más:

1. kisebb üregek (0,4 mikron a CD 0,8 mikronos méretével szemben);
2. szorosabb spirál (a sávok között 0,74 mikron a rés, a CD-nél 1,6 mikron);
3. vörös lézer (0,65 mikron, míg a CD-nél 0,78 mikron).

Ezek a javítások együtt hétszeresre, 4,7 GB-ra növelték a kapacitást. Egy 1 x DVD-meghajtónak 1,4 MB/s az adatátviteli sebessége (szemben a CD 150 KB/s sebességével). Sajnos a bevásárlóközpontokban is használt vörös lézerre történt váltás azt is jelenti, hogy a DVD-meghajtóknak egy második lézerre vagy körmönfont átalakító optikára lesz szükségük ahhoz, hogy a CD-ket és CD-ROM-okat olvasni tudják. Lehet, hogy nem mindegyik fogja ezeket tartalmazni, és hogy a CD-R- és a CD-RW-lemezeket nem lehet majd DVD-meghajtóban olvasni.

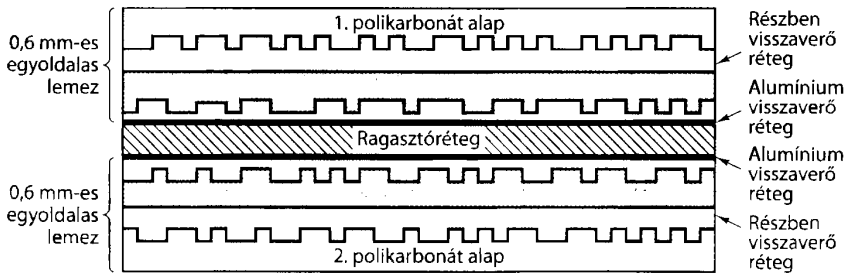
Elég-e a 4,7 GB? Elég lehet. MPEG–2 (IS 13346-os szabvány) tömörítést alkalmazva egy 4,7 GB-os DVD-lemez 133 percnyi teljes képernyős, folyamatos videofilmet képes tárolni nagy (720 x 480) felbontással, ezen kívül 8 nyelven szinkronhangot és további 32 nyelven feliratokat. A Hollywood által valaha is készített filmek 92 százaléka 133 percnél rövidebb. Ennek ellenére bizonyos alkalmazások, mint például játékok vagy enciklopédiák többet is igénylehetnek, és Hollywood is akarhat hosszabb filmet tenni egy lemezre, ezért aztán négy formátumot definiáltak:

1. Egyoldalas, egyrétegű (4,7 GB);
2. Egyoldalas, kétrétegű (8,5 GB);

3. Kétoldalas, egyrétegű (9,4 GB);
4. Kétoldalas, kétrétegű (17 GB).

Miért van ennyi formátum? Egy szóban összefoglalva: politika. A Philips és a Sony egyoldalas, kétrétegű lemezt akart a nagy kapacitású verzióhoz, míg a Toshiba és a Time Warner kétoldalas, egyrétegűt. A Philips és a Sony nem hitte, hogy az emberek szeretik, ha meg kell fordítaniuk a lemezt, a Time Warner nem hitte, hogy meg lehet oldani két réteg felvitelét egy oldalra. A kompromisszumos megoldás az lett, hogy minden kombinációt bevettek, és a piacra bízták annak eldöntését, melyik az életképes.

A kétréteges technológia úgy működik, hogy legalul egy visszaverő réteget helyeznek el, fölötte pedig egy részben visszaverő réteget. Attól függően, hogy a lézert hova fókuszálják, az egyik vagy a másik rétegről verődik vissza. Az alsó réteg egy kicsit nagyobb üregeket és szinteket igényel a biztonságos visszaolvasáshoz, ezért a kapacitása egy kicsit kisebb, mint a felső rétegé. A kétoldalas lemezeket úgy készítik, hogy két 0,6 mm vastagságú egyoldalas lemezt háttal egymásnak összeragasztanak. Azért, hogy minden verzió egyforma vastagságú legyen, az egyoldalas lemezeket egy 0,6 mm vastagságú üres alapra ragasztják (talán a jövőben 133 perc reklám lesz rajta, remélve, hogy az emberek olyan kíváncsiak lesznek, hogy megfordítják). A kétoldalas, kétrétegű lemez szerkezete a 2.27. ábrán látható.



2.27. ábra. *Kétoldalas, kétrétegű DVD-lemez*

A DVD-t egy tíz tagból álló konzorcium alkotta meg, mindegyik szórakoztatóelektronikai cikkeket gyártó cég, közülük hét japán. Szorosan együttműködtek a hollywoodi stúdiókkal (melyek közül néhány szintén a konzorciumban szereplő japán elektronikai cégek birtokában van). A számítógépes és telekommunikációs iparágak képviselőit nem hívták meg a piknikre, a jelenlévők pedig a mozifilmek kölcsönzésére és a termékbemutatókra fektették a hangsúlyt. Például beépített funkció a durva jelenetek valós idejű átlépésére (így a szülők egy felnőtteknek szánt filmet kisgyerekek számára is levetíthetővé tehetnek), hatcsatornás hang, valamint a Pan-and-Scan támogatás. Ez utóbbi segítségével a DVD-lejátszó dinamikusan eldöntheti, hogy a mozifilmek bal és jobb széléből mennyit vágjon le (mivel azok szélesség/magasság aránya 3:2) a képernyőn való megjelenítéshez (amelynél ez az arány 4:3).

Egy másik dolog, amire a számítógépipar valószínűleg nem is gondolt volna, az a szándékos inkompatibilitás az Egyesült Államokba, Európába és végül a többi kontinensre szánt lemezek szabványai között. Ezt Hollywood követelte, mert az új filmeket először mindig az Egyesült Államokban mutatják be, és csak azután kerülnek Európába, hogy az Egyesült Államokban a videofilm-változatot kiadják. Azt szerették volna elérni, hogy az európai videofilmüzletek ne vásárolhassák meg a filmeket az Egyesült Államokból idő előtt, mert így csökkennének az új filmek mozibevételei. Ha Hollywood irányítaná a számítógépipart, 3,5 inches hajlékonylemezt használnának az Egyesült Államokban és 9 cm-est Európában.

2.3.11. Blu-Ray

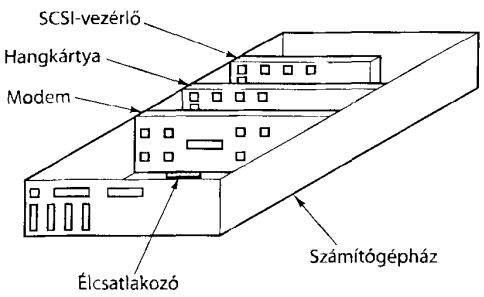
A számítógépiparban semmi sem állandó, így van ez természetesen az adattárolás technológiájában is. A DVD-t épphogy csak bevezették, amikor megjelent következő versenytársa, hogy elavulttá tegye. A DVD utódja az ún. Blu-Ray; így nevezik, mivel kék lézert használ a DVD-ben használt piros helyett. A kék fénynek rövidebb a hullámhossza, mint a pirosnak, ezért pontosabban fókuszálható, és így kisebb mélyedéseket tesz lehetővé. Az egyoldalas Blu-Ray-lemez 25 GB, a kétoldalas 50 GB adatot tárol. Az átviteli sebesség 4,5 MB/s, ami jó egy optikai lemezegység számára, de még mindig jelentéktelen, ha egy mágneslemezegységgel hasonlítjuk össze (vö. az ATAPI-6 100 MB/s-os, a Wide Ultra4 SCSI 320 MB/s-os sebességével). Arra számítanak, hogy a Blu-Ray végül le fogja váltani a CD-ROM-ot és a DVD-t, de ez néhány évet biztosan igénybe fog venni.

2.4. Bemenet/Kimenet

Ahogy a fejezet elején említettük, egy számítógépes rendszernek három fő komponense van: a CPU, a memóriák (központi és a háttér) és a **B/K (Bemeneti/Kimeneti)** berendezések (más néven perifériák), mint például a nyomtatók, szkennerek és a modemek. Eddig a CPU és a memória volt terítéken, most itt az ideje, hogy a B/K berendezéseket vizsgáljuk meg, valamint azt, hogy hogyan kapcsolódnak a rendszer többi részéhez.

2.4.1. Sínek

Fizikailag a személyi számítógépek és a munkaállomások szerkezete hasonló a 2.28. ábrán láthatóhoz. A szokásos felépítés egy fémdoboz, az alján egy nagy, nyomtatott áramköri lap, amelyet **alaplap**nak nevezünk. Az alaplap tartalmazza a CPU lapkát, néhány csatlakozót, ahova DIMM modulokat lehet betenni, és még mindenféle kiegészítő lapkát. Hosszában található rámaratva egy sín (PCI sín) és csatlakozók, amelyekbe a B/K kártyák élcsatlakozóit dughatók be. A régebbi

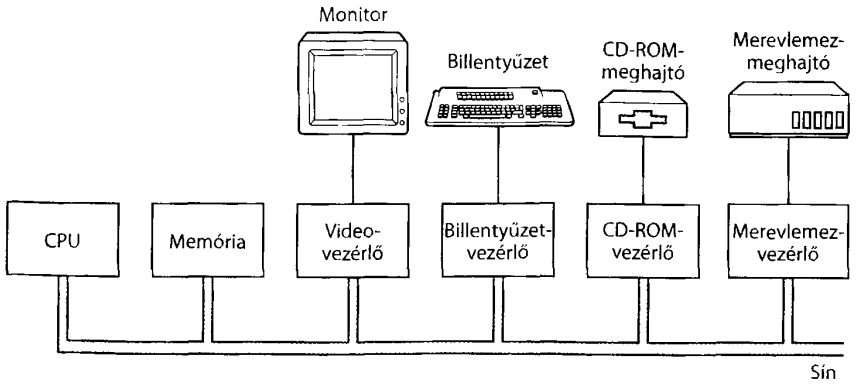


2.28. ábra. *Személyi számítógép fizikai felépítése*

PC-kben van egy második sín is (ISA sín) a hagyományos B/K kártyák számára, de az újabbak ezt már nélkülözik, és hamarosan el is fog tűnni.

Egy egyszerű, olcsóbb személyi számítógép logikai felépítése látható a 2.29. ábrán. Ebben van egy sín, amely a CPU-t, a memóriát és a B/K eszközöket kapcsolja össze; a legtöbb rendszerben kettő vagy több sín van. Minden B/K eszköz két részből áll: az egyik rész tartalmazza az elektronika nagy részét; ez a **vezérlő** vagy **kontroller**, a másik pedig az eszköz maga, mint például egy lemezegység. A vezérlő általában egy kártyán van, amely be van dugva valamelyik szabad csatlakozóba, kivéve a nem opcionális vezérlőket (mint például a billentyűzet), amelyeket néha az alaplapon helyeznek el. Bár a monitor nem tartozik az opcionális eszközök közé, a videovezérlő gyakran mégis külön kártyán van, hogy a felhasználók szabadon választhassanak a különböző kártyák közül, legyen-e benne grafikus gyorsító, vagy sem, legyen-e több memória stb. A vezérlő a vezérelt eszközhöz egy kábellel csatlakozik, amelyet a számítógépház hátoldalán elhelyezett csatlakozóba kell bedugni.

A vezérlőnek az a feladata, hogy a hozzá tartozó eszközt vezérelje, és a sínhez való hozzáférését kezelje. Ha például egy program adatokat akar olvasni a lemez-



2.29. ábra. *Egy egyszerű személyi számítógép logikai felépítése*

egységről, a vezérlőnek ad ki parancsot, amely keresési és egyéb parancsokat ad a lemezegységnek. Amikor a megfelelő sáv és szektor megvan, akkor a meghajtó az adatokat bitsorozatként elkezdi továbbítani a vezérlő felé. A vezérlő feladata, hogy a bitsorozatot nagyobb egységekké állítsa össze, és amint összeálltak, beírja a memóriába. Egy egység tipikusan egy vagy több szóból áll. A CPU közreműködése nélkül a memóriát olvasó vagy oda író vezérlő ún. **közvetlen memóriaelérést (Direct Memory Access, DMA)** hajt végre; ez DMA rövidítésként jobban ismert. Az adatátvitel befejeződése után a vezérlő egy **megszakítást** vált ki, ezzel kényszeríti a CPU-t, hogy az éppen futó programot azonnal felfüggessze, és egy speciális eljárást, a **megszakításkezelőt** végrehajtsa, amely a hibaellenőrzést és egyéb speciális teendőket elvégezve értesíti az operációs rendszert, hogy a B/K művelet befejeződött. A megszakításkezelő befejeződése után a CPU folytatja a megszakítás bekövetkezésekor felfüggesztett program futtatását.

A sínt nem csak a B/K vezérlők használják, a CPU is azon keresztül éri el az utasításokat és az adatokat. Vajon mi történik akkor, ha a CPU és egy B/K vezérlő egyszerre akarja használni a sínt? Ekkor egy **sínütemező** lapka dönti el, hogy ki lesz az első. Általában a B/K eszközök kapnak elsőbbséget, mert a lemezeket és egyéb mozgó alkatrészeket nem lehet megállítani, és várakoztatás esetén adatvesztés lépne fel. Amikor nincs végrehajtás alatt álló B/K művelet, a CPU kaphatja meg magának az összes sínciklust a memória elérésére. Amikor azonban egy B/K eszköz is működik, az eszköz kérni fogja, és meg is kapja a sínhasználat jogát, ha szüksége van rá. Ez a jelenség a **ciklustopás**, és lelassítja a számítógép működését.

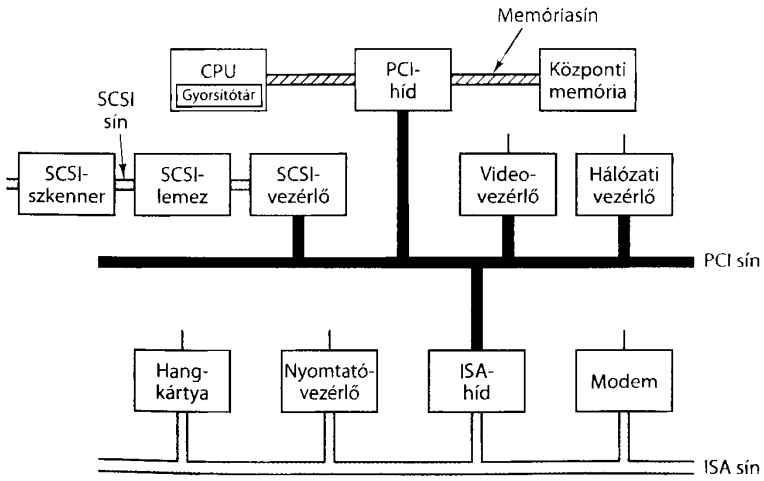
A fenti megoldás remekül működött az első személyi számítógépekben, mert minden komponens nagyjából egyensúlyban volt. Ahogy azonban a CPU-k, memóriák és B/K eszközök gyorsabbak lettek, probléma adódott: a sín nem bírta a terhelést. Egy zárt rendszerben, mint például egy mérnöki munkaállomáson, megoldás lehetett egy új, gyorsabb sín tervezése a következő modell számára. Mivel senki sem vitt át a régi modellből B/K eszközöket az újba, ez a megközelítés megfelelő volt.

Azonban a PC-k világában az emberek gyakran úgy javítják fel a gépeiket, hogy közben szeretnék a régi nyomtatójukat, szkennerüket és modemjüket megtartani. Közben egy hatalmas ipari ágazat is felnőtt, amely óriási mennyiségű B/K egységet állít elő az IBM PC sínhez, és ennek az ágazatnak nagyon kis érdeke fűződik ahhoz, hogy eddigi befektetéseit kidobja, és mindent elölről kezdjen. Az IBM a saját kárán tanulta meg ezt, amikor az IBM PC utódját, a PS/2 sorozatot kihozta. A PS/2-ben új, gyorsabb sín volt, de a legtöbb PC-gyártó továbbra is a régi PC sínt használta, amelyet ma **ISA (Industry Standard Architecture, ipari szabványos felépítés)** sínnek hívnak. A legtöbb lemezegységet és B/K egységet gyártó cég is folytatta az ISA-vezérlők gyártását, így az IBM abban az érdekes helyzetben találta magát, hogy ő az egyetlen, nem IBM-kompatibilis PC-gyártó. Végül is rákényszerült, hogy visszatérjen az ISA sínhez. Mellesleg megjegyezzük, hogy a gépi szintek tárgyalása során az ISA az utasításrendszer-architektúra (Instruction Set Architecture), míg a sínek esetén az ipari szabványos felépítés/architektúra (Industry Standard Architecture) rövidítése.

Mindezek ellenére – habár a piaci érdek az volt, hogy semmi se változzon – a régi sín tényleg nagyon lassú volt, ezért valamit tenni kellett. Ez a helyzet oda ve-

zetett, hogy több cég olyan gépeket tervezett, amelyek több sínt használtak. Ezek egyike a régi ISA vagy a vele visszafelé kompatibilis utódja, az **EISA (Extended ISA, bővített szabványos ipari felépítés)** sín volt. Ma a legnépszerűbb közülük a **PCI (Peripheral Component Interconnect, perifériális komponensek összekapcsolása)** sín. Ezt az Intel tervezte, de úgy döntött, hogy a szabadalmakat mindenki számára elérhetővé teszi, hogy bátorítsa felhasználásukat az egész iparágban (beleértve a versenytársakat is).

A PCI sín sokféle konfigurációban használható, a 2.30. ábrán egy tipikus konfigurációt mutatunk be. Itt a CPU egy külön erre a célra fenntartott, nagy sebességű vonalon keresztül kommunikál a memóriavezérlővel. A vezérlő közvetlen összeköttetésben van a memóriával és a PCI sínnel, így a CPU és a memória közötti forgalom nem halad át a PCI sínen. A nagy adatátviteli sebességű perifériák azonban, mint például SCSI-lemezegységek közvetlenül kapcsolódhatnak a PCI sínhez. Ezen kívül a PCI sín tartalmaz egy hidat az ISA sínhez, így az ISA-vezérlők és a hozzájuk tartozó eszközök is tovább használhatók – bár ahogyan korábban említettük, az ISA sín megszűnőben van. Egy ilyen gép általában 3-4 üres PCI-csatlakozót és egy-két ISA-csatlakozót is tartalmaz, hogy a felhasználók a régi (általában lassú eszközökhöz tartozó) ISA B/K kártyákat is és az új (általában gyors eszközökhöz tartozó) PCI-kártyákat is használhassák.



2.30. ábra. *Egy tipikus modern PC egy PCI és egy ISA sínnel. A modem és a hangkártya ISA-eszköz; a SCSI-vezérlő PCI-eszköz*

Manapság sokfajta B/K eszköz van forgalomban. Az ismertebbek közül néhányat a következőkben tárgyalunk.

2.4.2. Terminálok

A számítógépes terminálok két részből állnak: egy billentyűzetből és egy monitorból. A nagyszámítógépes világban ez a két rész gyakran egybeépített, és soros vonalon vagy a telefonhálózaton keresztül van a központi géphez kapcsolva. Légitársaságok helyfoglalási rendszereiben, bankokban és egyéb nagyszámítógépes ágazatokban ezeket még ma is széles körben használják. A személyi számítógépek világában a billentyűzet és a monitor független eszközök. A két részhez alkalmazott technológia mindkét esetben ugyanaz.

Billentyűzet

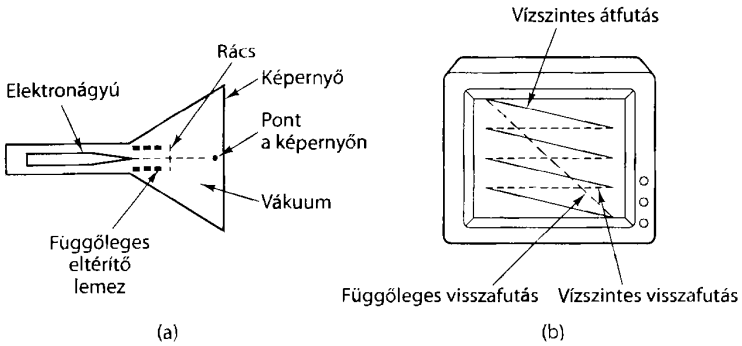
A billentyűzeteknek számos fajtája létezik. Az eredeti IBM PC billentyűzetének minden billentyűje alatt egy olyan kapcsoló volt, amely jól érzékelhető visszacsatolást és kattanó hangot is adott, ha eléggé lenyomták. Manapság az olcsóbb billentyűzetek egyszerűen csak mechanikus kontaktust hoznak létre leütéskor. A jobbakban rugalmas anyag (egyfajta gumi) van a billentyűk és az alattuk levő nyomtatott áramköri lap között. Minden billentyű alatt egy rugalmas kiemelkedés található, amely behorpad, ha elég erősen nyomjuk. A kiemelkedés alján lévő vezető anyag zárja az áramkört. Némelyik billentyűzet billentyűi alatt kis mágnesek találhatók, ezek leütéskor egy kis tekercsben elmozdulnak; az így indukált áramot detektálni lehet. Sok más, mechanikus és elektromágneses módszer is használatos.

A személyi számítógépeken egy billentyű leütésekor megszakítás generálódik, és a billentyűzet-megszakításkezelő (egy program, amely az operációs rendszer része) elindul. A megszakításkezelő kiolvassa a billentyűzet vezérlő regisztereiből a leütött billentyű kódját (1-től 102-ig). Amikor egy billentyűt felengedünk, egy második megszakítás keletkezik. Így ha a felhasználó lenyomja a SHIFT billentyűt, aztán leüti és felengedi az M billentyűt, végül felengedi a SHIFT-et, akkor az operációs rendszer láthatja, hogy a felhasználó egy nagy M betűt akar, és nem pedig egy kis m-et. A több billentyű leütéséből álló, a SHIFT, CTRL és ALT billentyűket is tartalmazó sorozatok kezelése teljesen szoftveres úton történik (beleértve a rossz hírű CTRL-ALT-DEL sorozatot is, amely az IBM PC-kompatibilis gépek újraindításához használatos).

Katódsugárcsöves monitorok

A monitor egy **katódsugárcsövet (Cathode Ray Tube, CRT)** és a hozzá tartozó energiaellátó berendezéseket tartalmazó doboz. A katódsugárcsőben van egy olyan ágyú, amely elektronsugarat tud lőni a cső elülső részéhez közel elhelyezkedő foszforeszkáló ernyőre, ahogy az a 2.31. (a) ábrán látható. (A színes monitorokban három elektronágyú van, egy-egy a vörös, a zöld és a kék színekhez.) A vízszintes átfutás alatt a sugár körülbelül 50 μs alatt keresztben átfut a képernyőn, és egy majdnem vízszintes vonalat rajzol ki az ernyőre. Ezután a vízszintes vissza-

futás következik, amely során a sugár visszatér az ernyő bal szélére, hogy egy újabb pásztázást kezdhessen. Az ilyen eszközöket, amelyek a képeket soronként állítják össze, **raszteres** eszközöknek nevezzük.



2.31. ábra. *(a) Katódsugárcső keresztmetszete. (b) Az elektronsugár útja*

A vízszintes átfutás úgy jön létre, hogy lineárisan növekvő feszültséget kapcsolnak az elektronágyú bal és jobb oldalán elhelyezkedő vízszintes eltérítő lemezekre. A függőleges mozgás egy sokkal lassabban lineárisan növekvő feszültség hatására következik be, amelyet az ágyú alatt és felett elhelyezett eltérítő lemezekre kapcsolnak. Minden sor végén a vízszintes, és valahol 400 és 1000 közötti átfutás után a függőleges eltérítő lemezeken a feszültségeket hirtelen felcserélik, ezzel a sugarat visszairányítják az ernyő bal felső sarkába. A teljes képernyőt általában másodpercenként 30–60-szor újrarajzolják. A sugár mozgását a 2.31. (b) ábrán követhetjük nyomon. Habár a CRT működését úgy írtuk le, hogy a sugár eltérítésére elektromos tereket használ, főleg a nagyobb teljesítményű modellekben az elektromos helyett mágneses teret alkalmaznak.

Azért, hogy képpontokat lehessen megjeleníteni a képernyőn, a CRT-ben egy rács helyezkedik el. Ha pozitív feszültséget kapcsolnak a rácsra, az elektronok felgyorsulnak, és az ernyőbe csapódva rövid felvillanást okoznak. Ha negatív feszültséget kapcsolunk a rácsra, az elektronok visszapattannak, ezért nem tudnak keresztülhaladni a rácson, és a képernyő nem fog világítani. Így megfelelő feszültséget kapcsolva a rácsra elérhető, hogy a megfelelő bitmintázat jelenjen meg a képernyőn. Ezzel a módszerrel egy bináris elektromos jel világos és sötét pontokból álló képpé alakítható.

Lapos megjelenítők

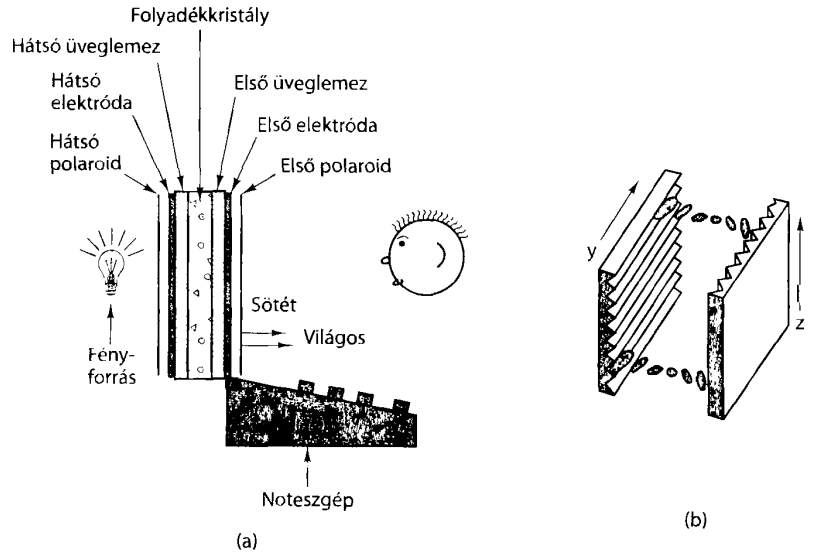
A katódsugárcsövek túlságosan terjedelmesek és nehezek ahhoz, hogy noteszgépekhez lehessen őket használni, ezért az ilyen képernyők számára más technológiai megoldásokra van szükség. A legelterjedtebb az **LCD (Liquid Crystal Display,**

folyadékkristályos kijelző) technológia. Nagyon összetett, sok változata létezik, és gyorsan változik, ezért a leírás is szükségszerűen rövid és nagymértékben leegyszerűsített lesz.

A folyadékkristályok nyúlós, ragadós szerves molekulák, amelyek úgy folynak, mint a folyadékok, de térbeli szerkezetük is van, mint a kristályoknak. Egy osztrák botanikus (Rheinitzer) fedezte fel 1888-ban, és először az 1960-as években használták fel kijelzőkhöz (például számológépekben, órákban). Ha az összes molekula azonos irányban helyezkedik el, a kristály optikai tulajdonsága a beeső fény irányától és polarizáltságától függ. Elektromos mező felhasználásával a molekulák elrendeződése, így az optikai tulajdonsága megváltoztatható. Konkrétan, ha a folyadékkristályra fényt bocsátunk, a kilépő fény intenzitása elektromosan szabályozható. Ezt a tulajdonságot kihasználva lapos megjelenítőket készíthetünk.

Egy LCD képernyő két párhuzamos üveglemezből áll, melyek közötti zárt térben folyadékkristály van. Átlátszó elektródákat erősítenek mindkét lemezhez. A hátsó lemez mögül érkező fény (akár természetes, akár mesterséges) világítja meg a képernyőt hátulról. A lemezekhez erősített elektródákkal hoznak létre elektromos teret a folyadékkristályban. A képernyő különböző területei különböző feszültséget kapnak, így jeleníthető meg a kívánt kép. A képernyő elejéhez és hátuljához polaroid szűrők vannak ragasztva, mert az alkalmazott technológia megkívánja a polarizált fény használatát. Az általános felépítés a 2.32. (a) ábrán látható.

Jóllehet sokféle LCD kijelzőt használnak, most egy konkrét fajtát fogunk tárgyalni, az ún. **TN (Twisted Nematic, elforgatott molekulájú)** kijelzőt. Ebben a



2.32. ábra. *(a) Egy LCD képernyő felépítése. (b) A hátsó és az első lemezek barázdái merőlegesek egymásra*

kijelző fajtában a hátsó lemez apró vízszintes barázdákat, az elülső lemez pedig apró függőleges barázdákat tartalmaz, ahogyan az a 2.32. (b) ábrán látható. Elektromos mező hiányában az LCD molekulái leginkább a barázdák irányában helyezkednek el. Mivel az elülső és a hátsó barázdák 90 fokos szöget zárnak be egymással, a molekulák elrendeződése (és ezért a kristály szerkezete is) csavarodik a két lemez között.

A képernyő hátulján egy vízszintes polaroid van, amely csak vízszintesen polarizált fényt enged át. A képernyő első felületén pedig egy függőleges polaroid van, amely csak függőlegesen polarizált fényt enged át. Ha nem volna folyadék a lemezek között, a hátul belépő vízszintesen polarizált fényt az első polaroid blokkolná, és a képernyő teljesen sötét lenne.

Az LCD molekuláinak csavart kristályszerkezete azonban áthaladáskor elforgatja a fényt, és úgy változtatja meg, hogy függőlegesen polarizáltan jön ki. Így aztán elektromos mező hiányában a képernyő egyenletesen megvilágított. A lemezek kiválasztott részeire feszültséget kapcsolva, a csavart szerkezet megszüntethető, elzárva ezzel a fény útját.

Kétféle módszert alkalmaznak a feszültség beállítására. Egy (olcsó) **passzív mátrixmegjelenítő**nél mindkét elektróda párhuzamos vezetékeket tartalmaz. Egy 640 × 480-as képernyőben például a hátsó elektróda 640 függőleges, az első elektróda pedig 480 vízszintes vezetéket tartalmazhat. Az egyik függőleges vezetékre feszültséget kapcsolva, majd impulzust bocsátva egy vízszintesre, egy kiválasztott pixelpozíció rövid időre elsötétíthető. Az impulzust ismételve a következő pixelekre, egy sötét vonal rajzolható a CRT analógiájára. Általában az egész képernyőt kirajzolják másodpercenként 60-szor, így a szem folytonos képet érzékel, megint csak ugyanúgy, mint a CRT esetén.

A másik széles körben elterjedt módszer az **aktív mátrixmegjelenítő**. Ez drágább, de jobb képet ad. A két merőleges vezetékháló helyett az egyik elektródában egy apró kapcsolóelem van minden pixelnél. Ezeket ki-be kapcsolgatva tetszőleges feszültségminta hozható létre a képernyőn, így tetszőleges képet lehet kirajzolni. A kapcsolóelemeket **vékonyfilm-tranzisztorok**nak, azokat a lapos megjelenítőket pedig, amelyek ilyeneket használnak **TFT megjelenítők**nek nevezik. A legtöbb hordozható noteszgép és az asztali számítógépek lapos megjelenítői manapság TFT technológiát használnak.

Eddig azzal foglalkoztunk, hogy hogyan működik egy monokróm megjelenítő. A színes megjelenítőkkel kapcsolatban elegendő annyit mondanunk, hogy az általános elvek ezeknél is ugyanazok, de a részletek sokkal bonyolultabbak. Optikai szűrőket használnak arra, hogy a fehér fényt minden pixelnél vörös, zöld és kék komponensekre bontsák, így ezeket egymástól függetlenül jeleníthetik meg. Minden szín felépíthető e három alapszín lineáris szuperpozíciójával.

Video RAM-ok

Mind a katódsugárcsöves, mind a vékonyfilm-tranzisztoros képernyők másodpercenként 60–100 alkalommal frissítik a képernyőjüket egy speciális memóriából, amelyet videomemóriának neveznek, és amely a képernyővezérlő kártyán található. Ez a memória egy vagy több, a képernyőt reprezentáló bittérképet tartalmaz. Egy olyan képernyő esetében, amelyen 1600 × 1200 képelem, ún. **pixel** található, a videomemória 1600 × 1200 értéket fog tartalmazni, minden pixelhez egyet. Valójában tartalmazhat több ilyen bittérképet is, ami lehetővé teszi, hogy gyorsan át lehessen kapcsolni egyik képről egy másikra.

A legmodernebb képernyők esetén minden pixelt egy 3 bájtos RGB érték reprezentál, amely a pixel piros, zöld és kék színkomponensének intenzitása. A fizika törvényeiből tudjuk, hogy minden szín előállítható a piros, zöld és kék színű fény lineáris szuperpozíciójával.

Egy videomemóriának, amely 1600 × 1200 3 bájtos pixelt tartalmaz, majdnem 5,5 MB méretűnek kell lennie a kép tárolásához, és tekintélyes mennyiségű CPU-időt igényel bármilyen művelet, amelyet vele végzünk. Ezért néhány számítógépben kompromisszumot kötnek, és egy 8 bites számot használnak a kívánt szín kiválasztásához. Ezt a számot indexként használják egy hardvertáblázathoz, amelyet **színpalettá**nak neveznek, és 256 elemet tartalmazhat, minden egyes elem egy-egy 24 bites RGB érték. Ezt az elgondolást **indexelt színelőállítás**nak nevezik, a videomemória-igényt kétharmadával csökkenti, azonban egyszerre csak 256 különböző színt enged meg a képernyőn. Mivel általában a képernyő minden ablaka saját leképezéssel rendelkezik, de csak egy hardverpaletta van, amikor több ablak van a képernyőn, gyakran csak az aktív ablak színei jelennek meg helyesen.

A grafikus videomegjelenítők nagy sávszélességet igényelnek. Egy teljes képernyős valódi színeket alkalmazó multimédiakép egy 1600 × 1200-as képernyőn 5,5 MB adat másolását jelenti a videomemóriába minden egyes fázishoz. Egy folyamatosan mozgó videóhoz, amelyhez legalább 25 fázis/s szükséges, ez 137,5 MB/s adatátviteli sebességet jelent. Ez a terhelés jóval nagyobb annál, mint amit az (E)ISA sín, és annál is több, mint amit az eredeti PCI sín kezelni tud (127,2 MB/s). Természetesen a kisebb képek kisebb sávszélességet igényelnek, de a sávszélesség így is fontos szempont.

Az Intel, hogy nagyobb sávszélességet biztosítson a CPU-tól a video RAM felé, a Pentium II-től kezdve, egy új sínt vezetett be a video RAM-hoz, az AGP (Accelerated Graphics Port) sínt, amely 32 bit adatot tud továbbítani 66 MHz sebességgel, így 252 MB/s adatátviteli sebességet nyújt. A további verziók 2-, 4-, sőt 8-szoros sebessége elegendő sávszélességet biztosít a nagy interaktivitású grafikához anélkül, hogy túlterhelné a fő PCI sínt.

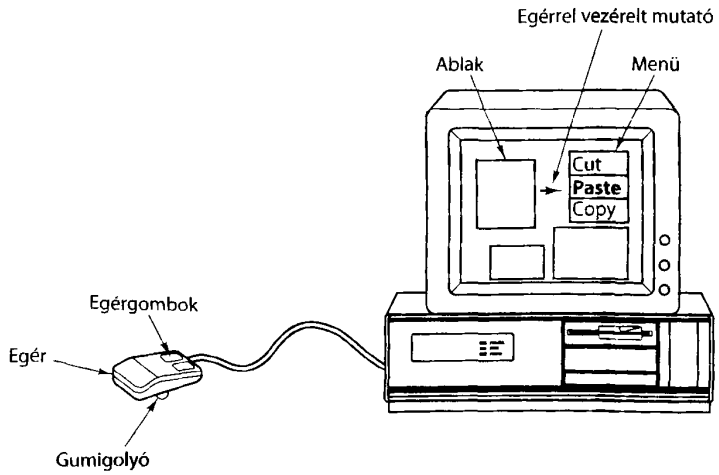
2.4.3. Egér

Egyre többen használják olyanok is a számítógépeket, akik kevés ismerettel rendelkeznek a működésükről. Az ENIAC gép korában a számítógépeket még az építői programozták, az 1950-es években pedig csak magasan képzett szakemberek használták őket. Ma sok olyan ember is használ számítógépet, akinek csak az a fontos, hogy munkáját el tudja végezni, és nem tud (esetleg nem is akar tudni) sokat arról, hogy a számítógépek hogyan működnek vagy hogyan programozhatók.

A régi időkben a legtöbb számítógépnek parancssoros interfésze volt, ahová a felhasználók parancsokat gépeltek be. Mivel a nem számítógépes szakemberek gyakran barátságtalannak vagy egyenesen ellenségesnek találják a parancssoros interfészeket, sok számítógépes cég – például a Macintosh és a Windows – rámutatással és kattintgatással működő interfészeket fejlesztett ki. Ezek használatához azonban arra van szükség, hogy képesek legyünk a képernyő egyes pontjaira rámutatni, aminek a leggyakoribb eszköze az egér.

Az **egér** egy kis műanyag doboz, amely a billentyűzet mellett helyezkedik el az asztalon. Ha mozgatjuk, a képernyőn látható kis jel szintén elmozdul, ezáltal a felhasználók rámutathatnak a képernyőn lévő objektumokra. Az egér tetején egy, kettő vagy három gomb van, ezekkel lehet a menükből választani. Sok vita folyt már arról, hogy hány gombja legyen az egérnek. A legegyszerűbb felhasználóknak az egy gomb a legjobb (nehéz rosszat megnyomni, ha csak egy van), a kifinomultabbak szeretik a több gombban rejlő lehetőségeket, amivel érdekes dolgok művelhetők.

Eddig háromféle egeret gyártottak: mechanikus, optikai és optomechanikus egeret. Az elsőnek két gumikerék áll ki az aljából, egymásra merőleges tengellyel. Ha ezt az egeret hosszirányban mozgatjuk, csak az egyik kerék mozog. Ha keresztirányban, akkor pedig csak a másik. A kerekek egy-egy változtatható ellenállást



2.33. ábra. *Menüelemek kiválasztása egérrel*

(potenciométer) vezérelnek. Az ellenállás változását mérve megállapítható, hogy az egyes kerekek mennyit mozdultak el, vagyis az egér mekkora utat tett meg az egyes irányokban. Az elmúlt években a fenti megoldást egy olyan váltotta fel, ahol a két kerék helyett egy kiálló golyót használnak; ez a 2.33. ábrán látható.

Az egerek másik fajtája az optikai. Ennek nincs se kereke, se golyója, ehelyett egy **LED (Light Emitting Diode, fénykibocsátó dióda)** és egy fénydetektor van az aljában. Az optikai egeret egy speciális, sűrű négyzetrácsos mintázatú műanyag lapon használják. Ahogy az egér mozog a rács felett, a fénydetektor érzékeli, ha vonal felett halad el, mert a LED-ből származó visszavert fény intenzitása megváltozik. Az egérben lévő elektronika számolja az egyes irányokban átlépett vonalak számát.

A harmadik fajta az optomechanikus egér. Az újabb mechanikus egérhez hasonlóan egy golyó két, egymáshoz képest 90 fokban elhelyezett tengelyt forgat. A tengelyek résekkel ellátott tárcsákhoz rögzítettek, a réseken a fény át tud haladni. Ahogy az egér mozog, a tengelyek forognak, és a réseken keresztül egy LED-ből fényimpulzusok jutnak el a hozzá tartozó fénydetektorba. A fényimpulzusok száma arányos a megtett úttal.

Jóllehet az egereket sokféleképpen be lehet állítani, a leggyakoribb az, hogy az egér egy 3 bájtból álló üzenetet küld a számítógépnek, ha megtesz egy bizonyos minimális távolságot (például 0,01 inchet), amelyet néha **mickey**-nek neveznek. Általában ezek a karakterek egy soros vonalon érkeznek bitenként. Az első bájt egy előjeles egész szám, amely megadja, hogy az egér mennyit mozdult x irányban az utolsó tizedmásodpercben. A második bájt ugyanezt az információt adja meg az y irányban. A harmadik bájt az egér gombjainak aktuális állapotát adja meg. Néha két bájtot használnak mindkét koordináta esetében.

A számítógépben egy alacsony szintű szoftver fogadja a beérkező jeleket, és az egér által küldött relatív mozgásadatokból előállítja az abszolút pozíciót. Ezt követően az egér elhelyezkedésének megfelelően egy nyilat jelenít meg a képernyőn. Ha a nyíl a megfelelő objektumra mutat, a felhasználó kattint egyet az egyik gombbal, ezután a számítógép a nyíl elhelyezkedéséből meg tudja állapítani, hogy melyik objektum lett kiválasztva.

2.4.4. Nyomtatók

Egy elkészült vagy a világhálóról letöltött dokumentumot a felhasználók gyakran ki akarnak nyomtatni, így minden számítógéphez csatlakoztatható nyomtató. Ebben az alfejezetben néhány gyakori monokróm (vagyis fekete-fehér) és színes nyomtatót tárgyalunk.

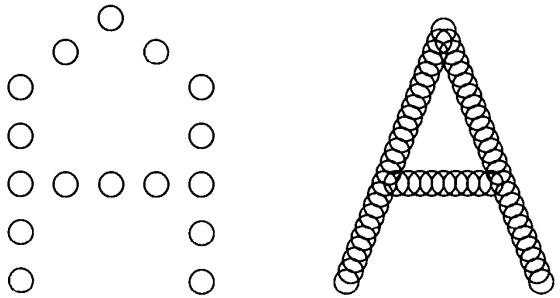
Monokróm nyomtatók

A legolcsóbb a **mátrixnyomtató**, amelyben egy 7–24 elektromágnesesen aktivizálható tűt tartalmazó nyomtatófej halad el minden nyomtatandó sorban. Az egyszerűbb fajtákban 7 tű van, és például soronként 80 karaktert tud 5 × 7-es pont-

mátrixokból előállítani. Valójában minden kinyomtatott sor 7 vízszintes sorból áll, mindegyikben 5 × 80 = 400 ponttal. A pontokat ki lehet nyomtatni, vagy sem, attól függően, hogy egy adott helyen milyen karakterre van szükség. A 2.34. (a) ábra egy 5 × 7-es mátrixba nyomtatott „A" betűt mutat be.

A nyomtatási minőség kétféleképpen javítható: több tű használatával vagy egymást átfedő körök alkalmazásával. A 2.34. (b) ábrán látható „A" betűt 24 átfedő kört produkáló tűvel lehet előállítani. Egy soron általában többször is végig kell menni, hogy az átfedő körök létrejöhessenek, így a jobb minőség együtt jár a nyomtatási sebesség csökkenésével. A legtöbb mátrixnyomtató többféle üzemmódban is tud működni, ezáltal választási lehetőséget nyújt a nyomtatási minőség és a sebesség között.

A mátrixnyomtatók olcsók (főleg a kellékeiket tekintve) és nagyon megbízhatók, habár lassúak, hangosak és gyengék a grafikus képességeik. A mai rendszerekben három fő felhasználási területük van. Először is, népszerűek a nagy (> 30 cm) előre nyomtatott formanyomtatványokhoz. Másodszor, jók a kis papírdarabokra való nyomtatáshoz, például pénztárblokkokhoz, ATM pénzkiadó automaták és hitelkártya-tranzakciók szelvényeihez vagy repülőtéri beszállókártyákhoz. Harmadszor, többpéldányos leporellókra való nyomtatáshoz ez a legolcsóbb technológia.



2.34. ábra. *(a) Az „A" betű 5 × 7-es mátrixon. (b) Az „A" betű 24 átfedő tűvel nyomtatva*

Olcsó otthoni nyomtatáshoz a **tintasugaras nyomtató** az egyik legkedveltebb. A mozgatható, tintapatront tartalmazó nyomtatófej vízszintesen végighalad a papír előtt, mialatt tintát permetez apró fúvókáiból. A tintacseppecskék térfogata körülbelül 1 pikoliter, ami azt jelenti, hogy 100 millió kényelmesen belefér egy cseppnyi vízbe.

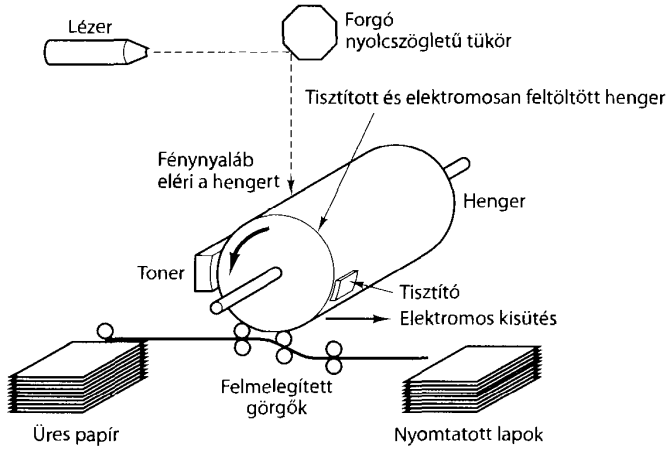
A tintasugaras nyomtatók két változatban kaphatók: piezoelektromos (az Epson használja) és a hővezérlésű (a Canon, a HP és a Lexmark használja). A piezoelektromos tintasugaras nyomtatókban egy speciális kristály van a tintapatron mellett. Amikor feszültséget kapcsolnak a kristályra, enyhén deformálódik, és egy tintacseppet présel ki a fúvókán. Minél nagyobb a feszültség, annál nagyobb a festékcsepp, így egy szoftver szabályozhatja a csepp méretét.

A hővezérlésű nyomtatókat általában **festékbuborékos nyomtató**nak (**bubblejet**) nevezik. Ezekben egy kis ellenállás van minden fúvókában. Amikor az ellenállásra feszültséget kapcsolnak, nagyon gyorsan felhevül, és a vele érintkező festéket azonnal felmelegíti egészen a forráspontig, a festék elpárolog és egy gázbuborékot képez. A gázbuborék térfogata nagyobb, mint a tintáé, amelyből létrejött, így nyomás keletkezik a fúvókában. A tinta csak egyetlen helyre mehet: a fúvókán keresztül a papírra. A fúvókát ezután lehűtik, és a keletkező vákuum egy újabb tintacseppet szív be a tintapatronból. A nyomtató sebességét az határozza meg, hogy a fűtés/hűtés ciklus milyen gyorsan ismételhető. Minden csepp azonos méretű, és általában kisebb, mint amilyet a piezoelektromos nyomtatók előállítanak.

A tintasugaras nyomtatók felbontása tipikusan legalább 1200 **dpi** (**dots per inch**), a legjobbaké eléri a 4800 dpi-t. Olcsók, csendesek és jó minőséget állítanak elő, de egyúttal lassúak is, és drága tintapatronokat használnak. Amikor a legjobb tintasugaras nyomtatóval egy nagy felbontású fényképet speciális bevonatú fotópapírra nyomtatnak, az eredmény nem különböztethető meg a hagyományos fényképtől még 20 × 25 cm-es méret esetében sem.

Azóta, hogy Johannes Gutenberg a XV században feltalálta a könyvnyomtatást, talán a **lézernyomtató** megjelenése volt a nyomtatás egyik legizgalmasabb eseménye. Ez az eszköz olyan tulajdonságokat egyesít magában, mint a kiváló minőségű kép, nagy rugalmasság, sebesség és elfogadható költség. A lézernyomtatók majdnem ugyanazt a technológiát használják, mint a fénymásolók. Valójában sok cég gyárt olyan eszközöket, amelyek kombinálni tudják a másolást és a nyomtatást (néha még a faxkezelést is).

A lézernyomtató működési elve a 2.35. ábrán látható. A nyomtató szíve egy fényérzékeny anyaggal bevont forgó precíziós henger (némely igényesebb rendszerben szalag). Egy-egy lap nyomtatása előtt körülbelül 1000 voltra feltöltik. Ezt követően egy lézer fénye pásztázza végig a hengert hosszában, hasonlóan a katódsugárcsöves



2.35. ábra. *A lézernyomtató működése*

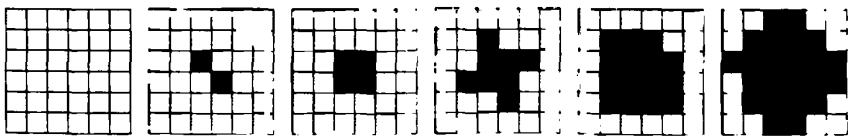
pásztázáshoz, csak itt elektromos feszültség helyett egy nyolcszögletű tükörrel irányítják a fényt a hengerre. A fényt modulálják, hogy világos és sötét pontokat kapjanak. Azok a pontok, ahol fény éri a hengert, elveszítik elektromos töltésüket.

Miután egy pontokból álló sor elkészült, a henger a fok egy törtrészével elfordul, és elkezdődhet a következő sor előállítása. Később az első sor eléri a toner- (festék-) kazettát, amely elektrosztatikusan érzékeny fekete port tartalmaz. A por hozzátapad a még feltöltött pontokhoz, így láthatóvá válik a sor. Tovább fordulva a bevont henger hozzányomódik a papírhoz, átadva a papírnak a festéket. A papír ezután felmelegített görgők között halad el, ezáltal a festék véglegesen hozzátapad a papírhoz, kialakul a kép. Később, a továbbforduló hengerről eltávolítják a töltést és letörlik a maradék port, ezzel a henger készen áll arra, hogy újra feltöltsék a következő laphoz.

Mondanunk sem kell, hogy ez az egész folyamat a fizikai, kémiai, mechanikai és optikai tervezés különösen bonyolult együttese. Ennek ellenére teljesen összeállított **nyomtatóművek** kaphatók több kereskedőnél is. A lézernyomtató-gyártók ezekhez az alapegységekhez saját elektronikájukat és szoftverüket adják, így állítva elő a kész nyomtatókat. Az elektronika egy gyors CPU-ból és néhány megabájt memóriából áll, amelyben elfér egy kinyomtatandó lap bittérképe és számos betűkészlet, ezek némelyike beépített, mások letölthetők. A legtöbb elfogad lapleíró parancsokat is (nemcsak a központi CPU által elkészített bittérképet). Ezek a parancsok olyan nyelvekhez tartoznak, mint a HP PCL-je vagy az Adobe PostScriptje.

A 600 dpi vagy annál nagyobb felbontású lézernyomtatók elfogadható fekete-fehér fényképeket képesek nyomtatni, de a technológia trükkösebb, mint ahogy azt elsőre gondolnánk. Tekintsünk egy 600 dpi felbontással szkennelt fényképet, amelyet egy 600 dpi-s nyomtatón szeretnénk kinyomtatni. A szkennelt kép 600 × 600 pixelt tartalmaz inchenként, mindegyikhez egy szürke árnyalat érték van rendelve 0-tól (fehér) 255-ig (fekete). A nyomtató szintén tud 600 dpi felbontással nyomtatni, de minden pixel vagy fekete lesz, vagy fehér. Szürkét nem lehet nyomtatni.

A szokásos megoldás a szürkeárnyalatos képek nyomtatására az ún. **halftoning** – a kereskedelemben kapható poszteréknél is ezt alkalmazzák. A képet halftone cellákra bontják, egy cella jellemzően 6 × 6 pixel. Minden cella 0 és 36 közötti fekete pixelt tartalmazhat. A szemünk a több fekete pixelt tartalmazó cellát sötétebbnek látja, mint a kevesebbet tartalmazót. A 0 és 255 közötti szürkeségi értékeket 37 zónába sorolják. A 0-tól 6-ig terjedő értékek vannak a 0-s zónában, 7-től 13-ig az 1-es zónában és így tovább (a 36-os zóna egy kicsit kisebb, mint a többi, mert a 256 nem



2.36. ábra. *Szürkeségi árnyalatokhoz tartozó halftone pontok. (a) 0–6. (b) 14–20. (c) 28–34. (d) 56–62. (e) 105–111. (f) 161–167*

osztható 37-tel maradék nélkül). Ha egy 0-s zónába eső szürkeségi értékkel találkozunk, az annak megfelelő halftone cella a papíron üresen marad [lásd 2.36. (a) ábra]. Az 1-es zónába eső pixelek helyére 1 fekete pixel kerül. A 2-es zónába eső értékek 2 fekete pixelként jelennek meg [lásd 2.36. (b) ábra]. A többi zóna értékekhez a 2.36. (c)–(f) ábrák tartoznak. Természetesen egy 600 dpi-s fénykép halftone változatának tényleges felbontása 100 cella/inch, amit **halftone képernyő-frekvenciának** neveznek és a hagyomány szerinti mértékegysége az **lpi (lines per inch)**.

Színes nyomtatók

Színeket kétféleképpen állíthatunk elő: színösszeadással és színkivonással. A színösszeadással készült képek, mint például a CRT monitorokon láthatók, a három additív alapszín, a vörös, a zöld és a kék lineáris szuperpozíciójával állíthatók elő. A színkivonással készült képek, mint például a színes fényképek és a népszerű képes folyóiratok képei, bizonyos hullámhosszúságú fényt elnyelnek, míg másokat visszavernek. Ezek a három szubtraktív alapszín, a cián (a vöröst teljesen elnyeli), a sárga (a zöldet teljesen elnyeli) és a bíborvörös (a kéket teljesen elnyeli) lineáris szuperpozíciójával állnak elő. Elméletileg minden szín előállítható a cián, a sárga és a bíborvörös tinta keverésével. Gyakorlatilag nehéz olyan teljesen tiszta tintákat készíteni, amelyek az összes fényt elnyelik, és a keverék feketének látszik. Emiatt majdnem mindegyik színes nyomtatási rendszer négy tintát használ: ciánt, sárgát, bíborvöröset és feketét. Ezeket a rendszereket **CMYK-nyomtatók**nak (Cyan, Magenta, Yellow, blacK) nevezzük. A monitorok ezzel szemben az additív alapszíneket és az RGB- (Red, Green, Blue) rendszert használják a színek előállítására.

Azoknak a színeknek az összességét, amelyeket egy monitor vagy nyomtató képes előállítani, **gamut**nak nevezzük. Egyetlen eszköz gamutja sem azonos a világ valóságos színeivel, mivel legjobb esetben is csak 256-féle intenzitással jelennek meg az alapszínek, ami 16 777 216 különböző színt ad. A technológia tökéletlenségei ezt tovább csökkentik, és a maradék sem mindig egyenletesen oszlik el a színskálán. Ezeken túl a színek érzékelése nemcsak a fény fizikai jellemzőin, hanem sokban azon is múlik, hogy a csapok és a pálcikák hogyan helyezkednek el az emberi retinán.

A fentiek következményeképpen egy képernyőn megfelelőnek látszó színes képet ugyanúgy kinyomtatni közel sem triviális dolog. Többek között az alábbi problémákkal kell szembenézni:

1. A színes monitorok kibocsátott fényt használnak; a színes nyomtatók visszavert fényt.
2. A CRT-k színenként 256-féle intenzitásra képesek; a színes nyomtatóknak halftoning technikát kell alkalmazniuk.
3. A monitoroknak sötét a háttere; a papír világos hátteret ad.
4. Az RGB és a CMYK gamutok különbözők.

A valóságossal (vagy éppen a monitorral) megegyező színes képek előállításához megfelelő eszközbeállításra, kifinomult szoftverre és nagy felhasználói tapasztalatra van szükség.

Színes nyomtatáshoz ötféle technológiát alkalmaznak a mindennapos használatban; ezek mindegyike a CMYK-rendszeren alapul. Az egyszerűbbek a színes tintasugaras nyomtatók, melyek ugyanúgy működnek, mint a monokróm tintasugaras nyomtatók, de egy helyett négy patronnal (cián, sárga, bíbor és fekete). Mérsékelt áron (a nyomtatók olcsók, de a tintapatronok nem) jó eredményeket adnak színes grafikák, és még úgy-ahogy megfelelőt fényképek számára.

A legjobb eredmények elérése érdekében speciális tintát és papírt kell használni. Kétféle tinta van. A **festékalapú tinák** folyékony hordozóanyagban oldott színes festékek. Élénk színeket adnak, és könnyen folynak. Fő hátrányuk, hogy ultraibolya fénynek, például napsütésnek kitéve fakulnak. A **pigmentalapú tinák** szilárd pigmentrészecskéket tartalmaznak folyékony hordozóanyagban, amely elpárolog a papírról, hátrahagyva a pigmenteket. Nem fakulnak, de nem is olyan élénk a színük, mint a festékalapú tintáknak, ezenkívül a pigmentek szeretik eltömíteni a fúvókákat, amelyek emiatt rendszeresen tisztításra szorulnak. Fényképek nyomtatásához bevonatos vagy fényes papírra van szükség, amelyeket speciálisan arra terveztek, hogy a tintacseppeket egyben tartsák, és ne hagyják szétterülni.

A tintasugaras nyomtatóknál egy fokkal jobbak a **szilárd tintás nyomtatók**. Ezekbe négy speciális, szilárd tintatömböt kell elhelyezni, amelyeket aztán felolvasztanak, és tartályokban tárolnak. Ezeknél a nyomtatóknál bekapcsolás után néha 10 percet is várni kell, amíg a tintatömbök felolvadnak. A forró tintát a papírra szórják, ahol megszilárdul, és két henger között átvezetve véglegesen a papíron marad.

A színes nyomtatók harmadik fajtája a színes lézernyomtató. Monokróm testvéréhez hasonlóan működik, kivéve, hogy a négy színnek megfelelő képeket egyenként viszik fel a hengerre, és mindegyiknél külön tonerből színezik meg a papírt. Mivel a teljes bittérkép általában előre elkészül, egy 1200 x 1200 dpi felbontású kép, amely 80 négyzetinchet foglal el a papíron, 115 millió pixelből áll. Pixelenként 4 bittel számolva, a nyomtatónak 55 MB memóriára van szüksége csak a bittérkép tárolására, nem számítva a belső processzorok, betűkészletek és egyebek memóriaszükségletét. Ezek a követelmények drágává teszik a színes lézernyomtatókat, de a nyomtatás gyors, a minőség nagyon jó, és a képek időtállók.

A negyedik fajta színes nyomtató a **viasznyomtató**. Egy széles, négyszínű viaszt tartalmazó szalag található benne, amely lapméretű szegmensekre osztott. Több ezer fűtőelem olvasztja meg a viaszt, ahogy a papír mozog alatta. A viasz a CMYK-rendszert használva pixelenként hozzátapad a papírhoz. A viasznyomtatók voltak a legelterjedtebb színes nyomtatóeszközök, de ezeket egyre inkább olyan nyomtatók váltják fel, amelyeknek olcsóbbak a kellékeik.

A színes nyomtatók ötödik fajtája a **festékszublimációs nyomtató**. Habár freudi felhangjai vannak, a szublimáció annak a folyamatnak a tudományos neve, amikor egy szilárd halmazállapotú anyag gázzá változik anélkül, hogy keresztülmenne a folyékony halmazállapoton. A szárazjég (fagyott szén-dioxid) egy jól ismert szublimáló anyag. A festékszublimációs nyomtatóban egy több ezer fűtőelemet tartalmazó nyomtatófej felett haladnak el a hordozóanyagban tárolt CMYK festékek. A festékek azonnal elpárolognak, és a közelben lévő speciális papírra kerülnek. Minden fűtőelem 256 különböző hőmérsékletet tud produkálni. Minél magasabb a hőmérséklet, annál több festék kerül a papírra, és annál élénkebb lesz a szín. A többi színes nyomtatóval ellentétben közel folytonos színátmeneteket lehet elérni, így a halftoning technikára nincs szükség. Kis fényképek készítésére használt nyomtatók gyakran ezen az elven működnek, és nagyon valósághű képeket tudnak produkálni speciális (és drága) papírra.

2.4.5. Telekommunikációs berendezések

A legtöbb számítógép napjainkban számítógép-hálózatba van bekapcsolva, legtöbbször az internetbe. E kapcsolat biztosításához speciális eszközökre van szükség. Ebben a fejezetben azt nézzük meg, hogyan működnek ezek az eszközök.

Modemek

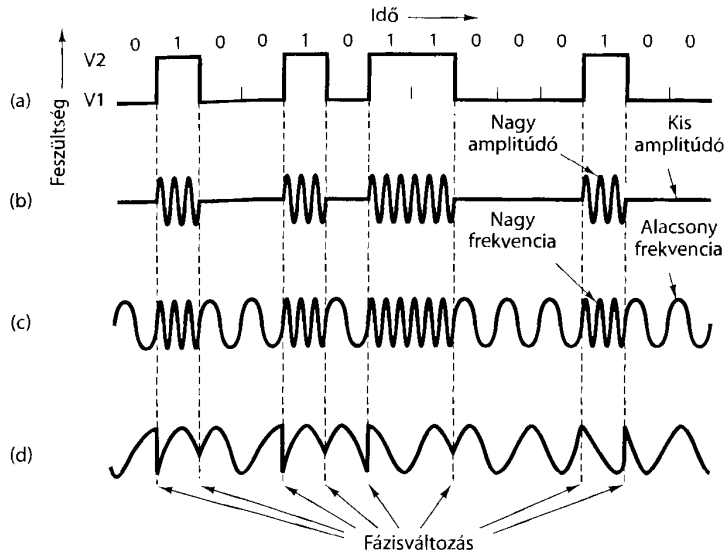
A számítógépek használatának növekedésével gyakran válik szükségessé, hogy egy számítógép egy másikkal kommunikáljon. Például sokan otthoni számítógépüket arra is használják, hogy munkahelyi gépükkel, egy internetszolgáltatóval vagy egy internetes banki rendszerrel kommunikáljanak. Sok esetben a telefonvonalak teszik lehetővé e kapcsolatok fizikai megvalósítását.

Egy sima telefonvonal azonban nem alkalmas a számítógép jeleinek továbbítására, mivel utóbbiak általában a 0-kat 0 volttal, az 1-eseket 3–5,5 volt közötti feszültséggel reprezentálják, ahogy az a 2.37. (a) ábrán látható. A kétszintű jelek jelentős torzulást szenvednek a hangátvitelre tervezett vonalon, ezzel átviteli hibákat eredményeznek. Egy 1000 és 2000 Hz közötti, **vivőhullám**nak nevezett tiszta szinuszos hullám azonban aránylag kis torzulással átvihető, így a legtöbb telekommunikációs rendszer ezt használja fel a működéséhez.

Mivel a szinusz jel hullámzása tökéletesen kiszámítható, egy tiszta szinusz jel nem hordoz semmilyen információt. Az amplitúdó, a frekvencia vagy a fázis változtatásával azonban átvihetjük 1-esek és 0-k sorozatát (lásd 2.37. ábra). Ezt a folyamatot **modulációnak** nevezik. Az **amplitúdómoduláció** [lásd 2.37. (b) ábra] két feszültségszintet használ, egyiket a 0-khoz, a másikat az 1-esekhez. Ha egy nagyon kis sebességű digitális adatátvitelbe belehallgatnánk, halk hangot hallanánk 0 esetén, hangosat 1 esetén.

A **frekvenciamoduláció** [lásd 2.37. (c) ábra] esetén a feszültségszint állandó, de a vivőhullám frekvenciája eltérő 1-esek és 0-k esetén. Ha belehallgatunk egy frekvenciamodulált digitális adatátvitelbe, különböző magasságú hangot hallunk a 0-k és az 1-ek esetében. A frekvenciamodulációt gyakran **frekvenciaeltolásos kódolás**nak **(frequency shift keying)** is nevezik.

Az egyszerű **fázismoduláció**nál [lásd 2.37. (d) ábra] az amplitúdó és a frekvencia nem változik, de a vivőhullám fázisa 180 fokkal eltolódik minden 0–1 vagy



2.37. ábra. *A 01001011000100 bináris szám bitenkénti átvitele telefonvonalon. (a) Kétszintű jel. (b) Amplitúdómoduláció. (c) Frekvenciamoduláció. (d) Fázismoduláció*

1–0 váltásnál. Kifinomultabb fázismodulációs rendszerekben minden oszthatatlan időintervallum kezdetén a vivőhullám fázisa hirtelen eltolódik 45, 135, 225 vagy 315 fokkal, ezzel az ún. **dibit** fáziskódolással 2 bitet lehet intervallumonként átvinni. Például a 45 fokos fáziseltolás 00-t jelenthet, a 135 fokos eltolás 01-et és így tovább. Más, intervallumonként 3 vagy több bitet kódolni képes módszerek is léteznek. Az időintervallumok száma (vagyis a lehetséges jelváltások száma másodpercenként) a **baud**. Intervallumonként 2 vagy több bit átvitele esetén a másodpercenként átvitt bitek száma nagyobb, mint a jelváltások száma. Nagyon sokan összekeverik ezt a két fogalmat.

Ha az átviendő adatok 8 bites karakterekből állnak, előnyös lenne egy olyan módszer, amellyel 8 bitet egyszerre tudnánk átvinni – vagyis 8 párhuzamos vezeték. Mivel a telefonvonal csak egy csatornát szolgáltat, ezért a biteket sorosan, egymás után kell elküldeni (vagy páronként, ha dibit kódolást használunk). Az az eszköz, amelyik kétszintű jelek formájában, bitenként karaktereket fogad el a számítógéptől, és a biteket egyesével vagy kettesével amplitúdó-, frekvencia- vagy fázismodulációt használva továbbítja: a modem. A karakterek elejének és végének jelzésére minden 8 bites karaktert rendszerint megelőz egy startbit és követ egy stopbit, ami összesen 10 bitet jelent.

A küldő modem az egy karakterhez tartozó biteket egyenlő időközönként küldi el. Például a 9600 baud azt jelenti, hogy 104 milliomod másodpercenként történik egy jelváltás. A fogadó oldalon egy másik modem konvertálja a modulált jelet bináris számmá. Mivel a bitek szabályos időközönként érkeznek, ha a fogadó meghatározta a karakter elejét, már csak az órajelet kell figyelnie, hogy mikor vegyen mintát a következő bitek meghatározásához. A mai modemek 28 800 bit/másodperc és 57 600 bit/másodperc sebességgel működnek, általában ennél jóval alacsonyabb baud értékek mellett. Több módszer kombinációját használják arra, hogy több bitet küldhessenek jelváltásonként, modulálják az amplitúdót, a frekvenciát és a fázist is. Majdnem mindegyik **full-duplex**, ami azt jelenti, hogy egyszerre tudnak mindkét irányban kommunikálni (különböző frekvenciát használva). **Half-duplex** a neve az olyan modemnek vagy átviteli vonalnak, amely egyszerre csak egy irányban tud adatokat átvinni (hasonlóan egy egyvágányú vasúthoz, amelyen észak felé és dél felé is mehetnek a vonatok, de nem egyszerre). A csak egy irányban működő vonalak neve **szimplex**.

Digitális előfizetői vonalak

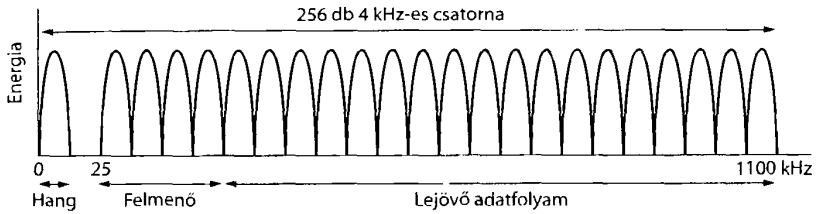
Amikor a telefonipar végre elérte az 56 Kbps sebességet, a jól végzett munka elismeréseképpen megveregette saját vállát. Eközben a kábeltévé-társaságok már 10 Mbps sebességet elérő vonalat kínáltak megosztott kábel felhasználásával, a műholdtársaságok pedig 50 Mbps feletti szolgáltatást tervezgettek. Ahogy az internet-hozzáférés biztosítása üzleti tevékenységük egyre fontosabb részévé vált, a **telefontársaságok** rájöttek, hogy a betárcsázásnál versenyképesebb szolgáltatásra van szükségük. Válaszképpen új digitális internetszolgáltatást kezdtek nyújtani. Azokat a szolgáltatásokat, amelyek a normál telefonvonalnál nagyobb sávszélességet biztosítanak, időnként **szélessávú**nak nevezik, bár ez a meghatározás inkább piaci, mint speciális technikai fogalom.

Kezdetben több egymást átfedő ajánlat jelent meg, amelyek az **xDSL** (Digital Subscriber Line, digitális előfizetői vonal) általános terminológia alá tartoznak (az *x* helyén különböző betűk állnak). A továbbiakban ismertetjük, hogy valószínűleg mivé válik majd ezek közül a legnépszerűbb, az **ADSL (Asymmetric DSL, aszimmetrikus digitális előfizetői vonal)**. Mivel az ADSL jelenleg is fejlődik, ezért a szabvány nem minden eleme van teljesen a helyén, az alábbiakban ismertetett néhány részlet idővel megváltozhat, az alapelképzelés azonban érvényes marad. Az ADSL-ről bővebben lásd (Summers, 1999; Vetter és társai, 2000).

A modemek lassúságának az az oka, hogy a telefont az emberi hang továbbítására találták ki, és a teljes rendszert erre a feladatra optimalizálták. Az adattovábbítás mindig is mostohagyermek volt. Az előfizetőktől a telefontársaság központjába futó vezetéket, amelyet **lokális hurok**nak **(local loop)** neveznek, tradicionálisan egy 3000 Hz-es szűrővel korlátozzák a telefontársaságnál. Ez az a szűrő, amely behatárolja az adattovábbítás sebességét. Szűrő nélkül a lokális huroknak a gyakorlatban elérhető sávszélessége függ a vezeték hosszától, de általában néhány kilométer távolságon elérheti az 1,1 MHz-et.

Az ADSL-kínálat legelterjedtebb megközelítését a 2.38. ábra mutatja be. Lényegében, azt teszik, hogy eltávolítják a szűrőt, és az elérhető 1,1 MHz spektrumot 256 független, egyenként 4312,5 Hz szélességű csatornára osztják fel. A 0-s csatornát az **egyszerű régi telefonszolgáltatás (Plain Old Telephone Service, POTS)**

céljaira használják. Az 1–5. csatornát nem használják, azért hogy megelőzzék a telefonhang és az adatjelek interferenciáját. A fennmaradó 250 csatornából egyet a felmenő, egyet a lejövő adatfolyam vezérlésére használnak. A többi csatorna pedig a felhasználók adatait továbbítja. Az ADSL olyan, mintha 250 modemünk lenne.

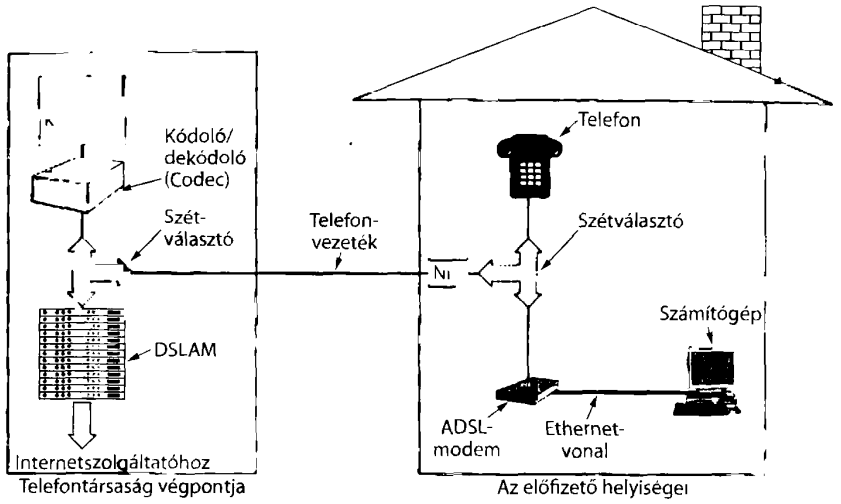


2.38. ábra. *Az ADSL működése*

Elvileg minden fennmaradó csatorna egy-egy full-duplex adatfolyam megvalósítására alkalmas, de a felharmónikusok, az áthallás és más effektusok a gyakorlati rendszereket jóval az elméleti korlát alatt tartják. A szolgáltató határozza meg, hogy a hány csatornát lehet feltöltésre és hányat letöltésre használni. Az 50-50% is megvalósítható lenne a felmenő és a lejövő csatornák számára, de a legtöbb szolgáltató a sávszélesség 80-90%-át a letöltésre allokálja, hiszen a legtöbb felhasználó inkább letölt adatokat, mint fel. Emiatt került az „A" betű az ADSL rövidítésbe. Egy gyakori megosztás: 32 csatorna a feltöltésre, és az összes többi pedig a letöltésre.

Az egyes csatornákon állandóan figyelik a vonalminőséget, és ha szükséges, megváltoztatják az adatsebességet, így a különböző csatornákon különböző lehet az átviteli sebesség. Az adatokat a valóságban az amplitúdó- és fázismoduláció kombinációjával továbbítják, így elérhetik a 15 bitet baudonként. Mondjuk, 224 lejövő csatornánál és 15 bit/baudnál, 4000 baud esetén a lejövő teljes sávszélesség 13,44 Mbps. A gyakorlatban a jel-zaj viszony sosem elég jó ahhoz, hogy ezt a sebességet el lehessen érni, de 4–8 Mbps elérhető, jó minőségű hurkokkal és rövid vezetékszakaszokkal.

Egy tipikus ADSL-elrendezés látható a 2.39. ábrán. A diagram azt mutatja be, hogy a felhasználó vagy a telefontársaság szerelőjének egy **hálózati interfészt** (**Network Interface Device, NID**) kell felszerelnie az előfizető telephelyén. Ez a kis műanyag doboz jelöli a telefontársaság birtokának a végét és az előfizető birtokának a kezdetét. A hálózati interfészhez közel (vagy néha azzal egy készülékben) található a **szétválasztó (splitter)**; ez egy analóg szűrő, amely leválasztja az adathálózatról a 0–4000 Hz-es sávot a POTS számára. A POTS jelet egy meglévő telefonra vagy telefaxra, az adatjeleket pedig egy ADSL-modemre irányítják. Az ADSL-modem tulajdonképpen egy digitális jelfeldolgozó eszköz, amely úgy van összekapcsolva, mintha 250 modem működne párhuzamosan különböző frekvenciákon. Mivel a legtöbb ADSL-modem külső, ezért a számítógépeket nagy sebességgel kell hozzákapcsolni. Általában ezt úgy oldják meg, hogy egy Ethernet-hálózati kártyát tesznek a számítógépbe, és egy két állomásból álló, nagyon kis hálózatot hoznak létre, amely csak a számítógépből és az ADSL-modemből áll. (Az Ethernet-hálózat egy népszerű és olcsó lokális hálózati szabvány.) Néha USB portot használnak Ethernet-hálózat helyett. A jövőben belső ADSL-modemkártyák is lesznek, ehhez kétség sem férhet.



2.39. ábra. *Egy tipikus ADSL-berendezés konfigurációja*

A vezeték másik végén, a telefontársaság végpontján egy megfelelő szétválasztót kell felszerelni. A jel hang részét itt is ki kell szűrni, és a normál telefonközpontba küldeni. A 26 kHz feletti jeleket egy újfajta készülékbe, a **DSLAM**-ba (**Digital Subscriber Line Access Multiplexer, digitális előfizetői vonal hozzáférési multiplexer**) irányítják, ami ugyanolyan digitális jelfeldolgozó egység, mint az ADSL-modem. Amint a digitális jelekből a bitsorozatokat előállították, csomagokat alakítanak ki belőlük, és az internetszolgáltatóhoz küldik tovább.

Kábeles internet

Sok kábeltévé-társaság kínál manapság internet-hozzáférést kábelein keresztül. Mivel ez a techológia jelentősen különbözik az ADSL-től, érdemes röviden foglalkozni vele. A kábelszolgáltatók minden városban fő telephellyel rendelkeznek, valamint rengeteg, elektronikával zsúfolt dobozzal szerte a működési területükön, amelyeket **fejállomásoknak** (**headend**) neveznek. A fejállomások nagy sávszélességű kábelekkel vagy üvegkábelekkel kapcsolódnak a fő telephelyhez.

Minden fejállomásról egy vagy több kábel indul el, otthonok és irodák százain halad keresztül. Minden előfizető ehhez a kábelhez csatlakozik, ahol az átlépi az előfizető telephelyének határát. Így felhasználók százai osztoznak egy a fejállomáshoz vezető kábelen. Rendszerint a kábelek sávszélessége 750 MHz körül van.

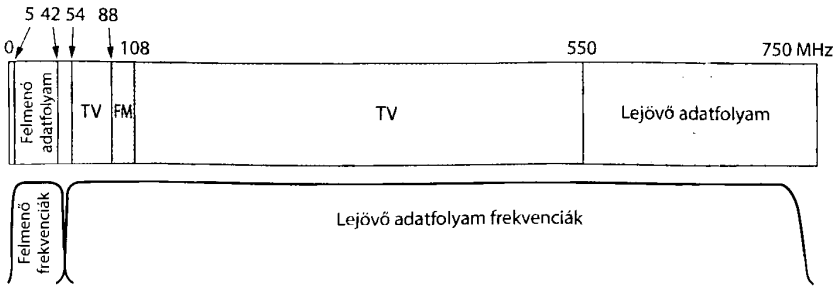
Ez a rendszer teljesen különbözik az ADSL-től, mivel ott minden előfizetőnek saját vezetéke van a telefontársasághoz. Bár a gyakorlatban egy saját 1,1 MHz sávszélességű csatorna nem nagyon különbözik a fejállomáshoz vezető kábel 200 MHz-es részének 400 felhasználó számára történő felosztásától, akiknek a fele nem használja a rendszert az adott pillanatban. Ez azonban azt jelenti, hogy hajnali 4 órakor jobb a kiszolgálás, mint délután 4-kor, míg az ADSL egész nap állandó. Azok, akik optimális kábeles internetszolgáltatásra vágynak, meggondolhatják, hogy gazdag vagy szegény környékre költözzenek (előbbiben a házak távol vannak egymástól, így kevés előfizető jut egy kábelre, utóbbiban pedig senki sem engedheti meg magának az internet-előfizetést).

Mivel a kábel egy megosztott kommunikációs lehetőség, fontos kérdés annak meghatározása, hogy ki mikor és melyik frekvencián adhat. Hogy ezt jobban megértsük, vizsgáljuk meg röviden, hogyan működik a kábeltévé. A kábeltévé-csatornák Észak-Amerikában rendesen az 54–550 MHz tartományt foglalják el (kivéve az FM rádió 88–108 MHz közötti sávját). Ezek a csatornák 6 MHz szélességűek, ebben már az elválasztó sávok is benne vannak, amelyek a csatornák közötti átszivárgást hivatottak megakadályozni. Európában a sáv alsó határa 65 MHz, a csatornák szélessége pedig 6–8 MHz a PAL és a SECAM rendszer nagyobb felbontása miatt, egyébként a csatornakiosztás hasonló.

Amikor a kábeles internetet bevezették, a társaságoknak két problémát kellett megoldaniuk:

1. Hogyan bővítsék a rendszerüket az internetszolgáltatással úgy, hogy a tévéműsorok vételét ez ne befolyásolja.
2. Hogyan valósítsanak meg kétirányú forgalmat, amikor az erősítők alapvetően egyirányúak.

A kiválasztott megoldás a következő. A modern kábelek jóval 550 MHz felett is működnek, gyakran 750 MHz-ig vagy még tovább. A feltöltésre szolgáló (a felhasználótól a fejállomás felé adatokat továbbító) csatornák az 5–42 MHz sávban vannak (Európában kicsit magasabb frekvencián), a letöltésre szolgáló csatornák (a fejállomástól a felhasználóhoz) forgalma a frekvenciatartomány felső részére esik, ahogyan az a 2.40. ábrán látszik.



2.40. ábra. *Egy tipikus kábeltévés internetszolgáltatás frekvenciakiosztási diagramja*

Vegyük észre, hogy a televíziójelek mind lejövő jelek, ezért lehetőség van arra, hogy a felmenő sáv erősítői csak az 5–42 MHz tartományban működjenek, a lejövő sáv erősítői pedig csak az 54 MHz-nél magasabb frekvenciákon, ahogyan ez az ábrán is látszik. A felmenő és a lejövő adatfolyam sávszélességében aszimmetria van, mivel a TV-sáv felett nagyobb frekvenciatartomány áll rendelkezésre, mint alatta. Másfelől viszont a forgalom nagyobb része valószínűleg letöltés, ezért a kábeltársaságok nem vették zokon ezt a körülményt. Ahogyan korábban láttuk, a telefontársaságok is aszimmetrikus DSL-szolgáltatást kínálnak, pedig ennek igazán nincs technikai indoka.

Az internet-hozzáféréshez egy kábelmodemre van szükség, amelyben két interfész van, egyik a számítógéphez, a másik a kábeltévé-hálózathoz. A számítógép és a kábelmodem közötti interfész kézenfekvő. Ez általában egy Ethernet-kártya, mint az ADLS-nél. A jövőben lehetséges, hogy az egész modem a számítógépbe dugott kisméretű kártya lesz, éppúgy, mint a V9.x belső modemeknél.

A másik oldal már bonyolultabb. A kábelszabvány nagyobb része rádióelektronikával foglalkozik, ami jóval túlmutat e könyv keretein. Az egyetlen említésre méltó részlet, hogy a kábelmodemek ugyanúgy, mint az ADSL-modemek állandóan be vannak kapcsolva. Akkor veszik fel a kapcsolatot, amikor bekapcsolják, és folyamatosan kapcsolatban maradnak, amíg ki nem kapcsolják, mivel a kábeltársaságok nem a kapcsolat fennállásának ideje alapján számítják a díjat.

Hogy jobban megértsük a működésüket, nézzük meg mi történik, amikor a kábelmodemet csatlakoztatjuk és bekapcsoljuk. A modem végigellenőrzi a lejövő csatornákat, és olyan speciális csomagokat keres, amelyeket a fejállomás periodikusan küld, és a rendszerparamétereket tartalmazzák az újonnan bekapcsolódó modemek számára. Amint a modem megtalálta ezt a csomagot, jelzi a jelenlétét az egyik felmenő csatornán. A fejállomás azzal válaszol, hogy kijelöli a modem felmenő és lejövő csatornáit. Ez a kijelölés később megváltozhat, ha a fejállomás a terhelés kiegyenlítése miatt szükségesnek ítéli.

Ezután a modem meghatározza a fejállomástól mért távolságát egy speciális csomag küldésével, úgy, hogy leméri, mennyi ideig tart a válasz megérkezése. Ezt a folyamatot **távolságbehatárolás**nak (**ranging**) hívják. A modem számára fontos, hogy tudja a távolságát, hogy ehhez tudja igazítani a felmenő csatornák működtetését, és az időzítését helyesen állítsa be. Az adatforgalmat rövid idejű szeletekre, ún. **minislot**okra osztják. Minden felfelé haladó csomagnak egy vagy több minislotba kell belefférnie. A fejállomás ad jelet egy újabb minislot-periódusra, azonban a startpisztolyt nem egyszerre hallja meg minden modem a kábel mentén mért terjedési idő miatt. A modemek – ha tudják, milyen messze vannak a fejállomástól – ki tudják számítani, hogy valójában mennyi ideje indult el az előző minislot-periódus. A minislot hossza a hálózattól függ. A hasznos méret jellemzően 8 bájt.

Az inicializálási fázisban a fejállomás minden egyes modem számára biztosít egy minislotot, amellyel felmenő sávszélességet kérhetnek. A szabály az, hogy több modemet is rendelhetnek ugyanahhoz a minislothoz, ami versenyhelyzetet teremt. Amikor a számítógép csomagot szeretne küldeni, a csomagot elküldi a modemhez, amely megkéri a szükséges számú minislotot a továbbításához. Ha a kérést a fejállomás elfogadta, nyugtázást küld a lejövő csatornán, és megadja, hogy

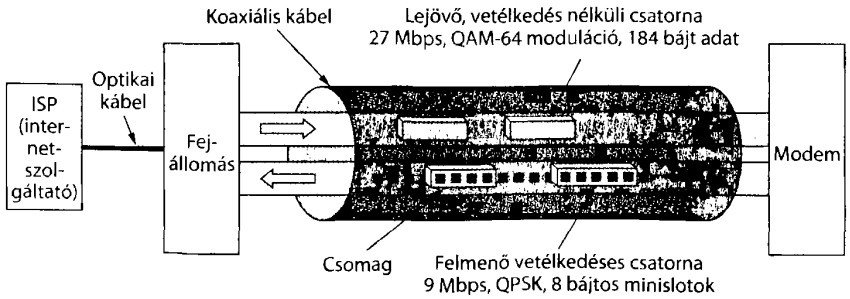
mely minislotokat foglalta le a modem számára. Azután a modem elküldi a csomagot, kezdve az első lefoglalt minislottal. További csomagokat a fejlécben lévő mező használatával lehet kérni.

Ha azonban versenyhelyzet van a minislotok kérésénél, a nyugtázás elmarad, ezért a modem vár egy ideig, majd újra próbálkozik. Minden egyes, ismételten bekövetkező hibánál a véletlen várakozás idejét duplázzák, hogy ha nagy a forgalom, szétterítsék a terhelést.

A lejövő csatornákat a felmenő csatornáktól eltérően irányítják. Az első dolog, hogy csak egy küldő van (a fejállomás), így nincs versenyhelyzet, és nincs szükség minislotokra, ami tulajdonképpen statisztikai alapú időosztásos multiplexelés. A másik, hogy a lejövő adatforgalom általában sokkal nagyobb, mint felmenő, így rögzített, 204 bájt méretű csomagokat használnak. Ennek egy része egy Reed–Solomon-féle hibajavító kód, és más járulékos dolgok, így a felhasználó számára 184 bájt hasznos adatmennyiség marad. Ezeket a számokat a MPEG-2 szabványú digitálistelevízió-szabvánnyal való kompatibilitás miatt választották, így a tévé és a lejövő csatornák formátuma azonos. A logikai kapcsolatokat a 2.41. ábra mutatja be.

Visszatérve a modem inicializáláshoz, amint a modem behatárolta a távolságát, megkapta a felmenő és lejövő csatornáit, valamint a minislotkiosztását, elkezdheti a csomagok küldését. Ezek a csomagok a fejállomáshoz futnak be, amely továbbítja egy dedikált csatornán a kábeltársaság fő telephelyére, majd az internetszolgáltatóhoz (amely lehet maga a kábeltársaság is). Az **internetszolgáltató (Internet Service Provider, ISP)** felé menő első csomag a hálózaticím-kérés (technikailag egy IP szám), amelyet dinamikusan osztanak ki. Le szokták kérdezni, és megküldik a napi pontos időt is.

A következő lépés a biztonsághoz kötődik. Mivel a kábel osztott kommunikációs lehetőség, bárki bajba kerülhet azáltal, hogy valaki más elolvassa a rajta keresztülhaladó forgalmat. Annak megakadályozására, hogy mindenki a szomszédjai után szimatoljon (képletesen szólva), a forgalom mindkét irányban titkosított.



2.41. ábra. *A felmenő és a lejövő csatornák tipikus részletei Észak-Amerikában. A QAM-64 (Quadrature Amplitude Modulation, kvadratúra amplitúdómoduláció) 6 bit/Hz-et enged meg, de magasabb frekvenciákon is működik. A QPSK (Quadrature Phase Shift Keying, kvadratúra fáziseltolásos kódolás) alacsony frekvenciákon működik, és csak 2 bit/Hz-et enged meg*

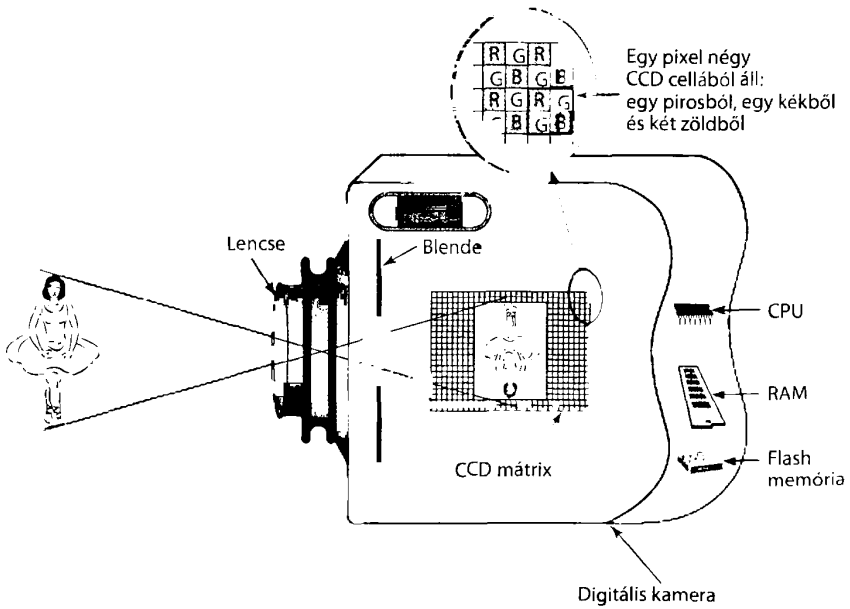
Az inicializálási folyamat egyik része a titkosítási kulcsok beállítása. Első pillantásra képtelenségnek tűnik, hogy két fél, a fejállomás és a modem, megállapodnak egy titkos kulcsban fényes nappal, ezer ember szeme láttára. Kiderült azonban, hogy nem, de az alkalmazott technika – a Diffie–Hellman-algoritmus – ismertetése kívül esik e könyv keretein. [A technika ismertetését lásd (Kaufman és társai, 2002).]

Végül, a modemnek be kell jelentkeznie, és meg kell adnia saját egyedi azonosítóját a titkos csatornán. Ekkor az inicializálás véget ér. A felhasználó bejelentkezhet az ISP-hez, és elkezdhet dolgozni.

Sokkal többet is el lehetne még mondani a kábelmodemekről. Néhány fontos szakirodalom: (Adams és Dulchinos, 2001; Donaldson és Jones, 2001; Dutta-Roy, 2001).

2.4.6. Digitális kamerák

A számítógépek egyre népszerűbb alkalmazási területe a digitális fényképezés, amely a digitális kamerákat a számítógép-perifériák egyik fajtájává tette. Nézzük meg röviden, hogyan működik. Minden kamerának van egy lencséje, amely a tárgyakról a kamera hátoldalán egy képet hoz létre. A hagyományos fényképezőgépekben a hátoldalon egy film halad, amelyen rejtett kép alakul ki, ha fény éri. A rejtett kép láthatóvá tehető bizonyos vegyszerek alkalmazásával a filmelőhívóban.



2.42. ábra. *A digitális kamera*

A digitális kamera ugyanígy működik attól eltekintve, hogy a filmet egy téglalap alakú **CCD (Charge Coupled Device, töltéscsatolt eszköz)** mátrixra cserélték, amely fényérzékeny. (Bizonyos kamerák CMOS-t használnak, de mi most az elterjedtebb CCD-kre koncentrálunk.)

Amikor a CCD-t fény éri, elektromosan feltöltődik. Minél erősebb a fény, annál több töltést kap. A töltés nagysága analóg-digitális átalakítóval 0 és 255 közötti (olcsó fényképezőgépeknél) vagy 0 és 4095 közötti (a digitális tükörreflexes fényképezőgépeknél) egész számként olvasható le. Az alapelrendezés a 2.42. ábrán látható.

Minden egyes CCD egyetlen értéket szolgáltat, függetlenül a fény színétől, amely érte. Színes képek készítéséhez a CCD-ket négyesével csoportosítják. A CCD tetejére egy Bayer-szűrőt tesznek, amely minden csoportból csak a piros fényt engedi át az egyik CCD-hez a négyből, a kéket egy másikhoz, a zöldet pedig a megmaradt kettőhöz. Két zöldet azért használnak, mert sokkal kényelmesebb négy CCD-vel reprezentálni egy pixelt, mint hárommal, továbbá a szem érzékenyebb a zöld színre, mint a pirosra és a kékre. Amikor egy fényképezőgépgyár azt állítja, hogy mondjuk 6 millió pixeles a fényképezőgép, akkor nem mond igazat. A kamera valóban 6 millió CCD-t tartalmaz, ami azonban négyesével csak 1,5 millió pixelt jelent. A képet 2828 × 2121 pixel méretű mátrixként (az olcsó fényképezőgépeknél) vagy 3000 × 2000 pixel méretű mátrixként (a digitális tükörreflexes fényképezőgépeknél) lehet kiolvasni úgy, hogy a többletpixeleket a kamerában lévő szoftver interpolációval állítja elő.

Amikor a kioldógombot a kamerán megnyomjuk, a kamerában futó szoftver három dolgot tesz: beállítja a fókuszt, megállapítja az expozíciós időt, és beállítja a fehéregyensúlyt. Az automatikus fókuszálás a kép nagy frekvenciájú összetevőit (finom részleteit) analizálja, és addig mozgatja a lencsét, amíg a leginkább részletgazdag képet adja. Az expozíciós idő meghatározása a CCD-kre eső fény erősségének megmérésével kezdődik, ezt követi a lencsenyílás és az expozíciós idő beállítása úgy, hogy a fényerő a CCD érzékenységének középtartományába essen. A fehéregyensúly beállítása a beeső fény spektrumának mérésével történik, hogy később a szükséges színkorrekciót el lehessen végezni.

Ezután a képet a gép a CCD-kről leolvassa, és a fényképezőgép belső RAM-jában pixelmátrixként tárolja. A fotóriporterek által használt legjobb minőségű digitális tükörreflexes fényképezőgépek nyolc nagy felbontású képet tudnak készíteni 5 másodperc alatt, és körülbelül 1 GB memóriára van szükségük, hogy a képeket feldolgozás előtt ideiglenesen, illetve azután hosszabb időre tárolják. Az olcsó kamerák kevesebb, de éppen elég RAM-ot tartalmaznak.

Az exponálás utáni fázisban a kamera szoftvere alkalmazza a fehéregyensúly színkorrekciós beállításokat, hogy kompenzálja a tárgyról jövő pirosas vagy kékes fényt (például árnyékban levő személy/tárgy vagy vakuhasználat esetén). Ezután egy zajcsökkentő algoritmust futtat, majd egy másikat, amely a hibás CCD-k hatását kompenzálja. Ezt követően megkísérli a képet élesíteni az élek megkeresésével és körülöttük a fényerőváltozás növelésével (ha ez a funkció nincs letiltva).

Végül, a képet tömöríteni is lehet, hogy a szükséges tárhelyet csökkenteni lehessen. Elterjedt formátum a **JPEG (Joint Photographic Experts Group)**, amelyben kétdimenziós térbeli Fourier-transzformációt alkalmaznak, és bizonyos magas frekvenciájú komponenseket elhagynak. A transzformáció eredménye, hogy a kép tárolása kevesebb bitet igényel, azonban finom részletek elvesznek.

Amikor az összes kamerán belüli feldolgozás befejeződött, a képet egy tárolóeszközre írják, amely lehet egy flash memória vagy kicsi kivehető mágneslemez-egység; ez utóbbit **mikromeghajtó**nak nevezik. Az utófeldolgozás és a felírás képenként néhány másodpercet vehet igénybe.

Amikor a felhasználó hazaérkezik, a fényképezőgépet csatlakoztathatja egy számítógéphez, általában például USB vagy FireWire kábellel. A képek ezután átmásolhatók a fényképezőgépről a számítógép mágneslemezére. Speciális szoftverrel, mint például az Adobe Photoshop, a felhasználó vághatja a képeket, beállíthatja a fényerősséget, a kontrasztot, a színegyensúlyt, élesítheti, elhomályosíthatja vagy eltávolíthatja a képek egy részletét, és számos szűrőt alkalmazhat. Amikor a felhasználó elégedett az eredménnyel, a képfájlokat színes nyomtatón kinyomtathatja, elküldheti interneten egy fotókidolgozással foglalkozó céghez vagy felírhatja CD-ROM-ra vagy DVD-re megőrzés vagy későbbi nyomtatás céljából.

A számítási kapacitás, RAM, mágneslemez-terület és a szoftver mennyisége egy digitális tükörreflexes fényképezőgépben megdöbbentő. A számítógépnek nemcsak el kell végeznie az összes fentebb leírt tevékenységet, hanem kommunikálnia kell a lencsébe épített CPU-val, a vakuba épített CPU-val, frissítenie kell az LCD megjelenítőn a képet, kezelnie kell az összes nyomógombot, forgatógombot, fényt, kijelzőt és egyéb ketyeréket a kamerán valós időben. Ez egy különösen erőteljes beágyazott rendszer, gyakran egy néhány évvel korábbi asztali számítógép versenytársa is lehet.

2.4.7. Karakterkódok

Minden számítógép használ valamilyen karakterkészletet. Ez legalább (az angol) 26 nagybetűt, 26 kisbetűt, 0-tól 9-ig a számjegyeket és egy sor speciális szimbólumot, mint például szóközt, pontot, mínuszjelet, vesszőt és kocsi vissza jelet tartalmazza.

Ahhoz, hogy a számítógépben tárolni tudjuk ezeket a karaktereket, mindegyikhez hozzá kell rendelni egy számot: például a = 1, b = 2, ..., z = 26, + = 27, – = 28. A karakterek számokra történő leképezését **karakterkód**nak nevezzük. Alapvető fontosságú, hogy az egymással kommunikáló számítógépek ugyanazt a kódot használják, különben nem értik meg egymást. Ezért szabványokat dolgoztak ki. Az alábbiakban megvizsgáljuk a két legfontosabbat.

ASCII

Széles körben elterjedt kód az **ASCII (American Standard Code for Information Interchange, az információcsere amerikai szabványos kódrendszere)**. Minden ASCII karakter 7 bites, vagyis összesen 128 karakter lehet. A 2.43. ábra tartalmazza az ASCII kódot. A 0 és 1F (hexadecimális) közötti kódok vezérlőkarakterek, amelyek nem nyomtathatók.

Sok ASCII vezérlőkaraktert eredetileg adatátvitelre szántak. Például egy üzenet állhat egy SOH (Start of Header, fejléc kezdete) karakterből, fejlécből, az üzenet szövegéből, egy ETX (End of Text, szöveg vége) karakterből, végül egy EOT (End of Transmission, átvitel vége) karakterből. A gyakorlatban azonban a telefonvonalakon és hálózatokon keresztül küldött üzeneteket egészen máshogy állítják össze, így az ASCII vezérlőkaraktereket már nem nagyon használják.

Hexadecimális kód	Név	Jelentés
0	NUL	Null
1	SOH	Start Of Heading (Fejléc kezdete)
2	STX	Start Of Text (Szöveg kezdete)
3	ETX	End Of Text (Szöveg vége)
4	EOT	End Of Transmission (Átvitel vége)
5	ENQ	Enquiry (Tudakozódás)
6	ACK	ACKnowledgement (Nyugta)
7	BEL	Bell (Csengő)
8	BS	BackSpace (Törlés)
9	HT	Horizontal Tab (Vízszintes tabulátor)
A	LF	Line Feed (Soremelés)
B	VT	Vertical Tab (Függőleges tabulátor)
C	FF	Form Feed (Lapdobás)
D	CR	Carriage Return (Kocsi vissza)
E	SO	Shift Out (Váltókikapcsolás)
F	SI	Shift In (Váltóbekapcsolás)
10	DLE	Data Link Escape (Vezérlőkarakter)
11	DC1	Device Control 1 (Eszközvezérlő 1)
12	DC2	Device Control 2 (Eszközvezérlő 2)
13	DC3	Device Control 3 (Eszközvezérlő 3)
14	DC4	Device Control 4 (Eszközvezérlő 4)
15	NAK	Negative ACKnowledgement (Negatív nyugta)
16	SYN	SYNchronous idle (Szinkronjel)
17	ETB	End of Transmission Block (Átviteli blokk vége)
18	CAN	CANcel (Visszavonás)
19	EM	End of Medium (Adathordozó vége)
1A	SUB	SUBstitute (Helyettesítés)
1B	ESC	ESCape (Vezérlőkód)
1C	FS	File Separator (Fájlelválasztó)
1D	GS	Group Separator (Csoportelválasztó)
1E	RS	Record Separator (Rekordelválasztó)
1F	US	Unit Separator (Egységelválasztó)

2.43. ábra. *Az ASCII karakterkészlet*

Hexa-dec. kód	Karakter	Hexa-dec. kód	Karakter	Hexa-dec. kód	Karakter	Hexa-dec. kód	Karakter	Hexa-dec. kód	Karakter	Hexa-dec. kód	Karakter
20	(Szóköz)	30	0	40	@	50	P	60		70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

2.43. ábra. *Az ASCII karakterkészlet (folytatás)*

Az ASCII nyomtatható karakterek egyértelműek. Szerepelnek közöttük az angol nagybetűk, kisbetűk, számjegyek, írásjelek és néhány matematikai szimbólum.

UNICODE

A számítógépipar nagyrészt az Egyesült Államokban fejlődött ki, ami az ASCII karakterkészlethez vezetett. Az ASCII kiváló az angol nyelvterületen, de kevésbé jó más nyelvekhez. A franciához ékezetek kellenek (például *système*); a némethez diakritikus jelek – mellékjel, hangsúly, ékezet – kellenek (például *für*) és így tovább. Némelyik európai nyelvben vannak olyan betűk, amelyek hiányoznak az ASCII-ből, mint például a német ß és a dán ø. Néhány nyelvnek teljesen eltérő ábécéje van (például az orosz, az arab), bizonyos nyelveknek pedig nincs is ábécéjük (például a kínai). Ahogy a számítógépek elterjedtek a világon, és a szoftverfejlesztők olyan helyeken is el akarták adni az árujukat, ahol a legtöbb felhasználó nem beszélte az angolt, más karakterkészletre volt szükség.

Az ASCII kibővítésére tett első próbálkozás az IS 646 volt, amikor is újabb 128 karaktert adtak az ASCII-hez, ez a 8 bites **Latin-1** kód volt. Az új karakterek többnyire ékezetekkel és diakritikus jelekkel ellátott latin betűk voltak. A következő próbálkozás az IS 8859 volt, amely bevezette a **kódlap** fogalmát, ami egy 256 karakterből álló készlet egy bizonyos nyelv vagy nyelvcsoport számára. Az IS 8859-1 a Latin-1. Az IS 8859-2 kezeli a latinalapú szláv nyelveket (például a cseh, a lengyel) és a magyart. Az IS 8859-3 tartalmazza a török, máltai, eszperantó, galíciai

és más nyelvekhez szükséges karaktereket. A kódlapos megközelítés problémája, hogy a programoknak számon kell tartaniuk, melyik kódlap az aktív, nem lehet keverni a különböző kódlaphoz tartozó karaktereket, további probléma, hogy a japán, valamint a kínai nyelvet teljesen kihagyták.

Egy számítógépes cégekből álló csoport elhatározta, hogy megoldja a problémát, és életre hív egy konzorciumot egy új rendszer, a **UNICODE** létrehozására és nemzetközi szabvánnyá nyilvánítására (IS 10646). A UNICODE-ot már támogatja néhány programozási nyelv (például Java), néhány operációs rendszer (például Windows XP) és sok alkalmazás is. Valószínűleg egyre jobban elfogadottá válik, ahogy a számítógépipar globalizálódik.

A UNICODE alapötlete az, hogy minden karakterhez és szimbólumhoz egy állandó, 16 bites értéket kell rendelni, amit **kódpozíció**nak neveznek. Nincsenek sokbájtos karakterek és speciális vezérlőkódok. Megkönnyíti a programok írását, hogy minden szimbólum 16 bites.

16 bites szimbólumokkal a UNICODE 65 536 kódpozícióval rendelkezik. Mivel a világ nyelvei együttesen körülbelül 200 000 szimbólumot használnak, a kódpozíciók értékes kincsek, amelyeket nagy körültekintéssel kell kiosztani. Körülbelül a kódpozíciók fele már ki van osztva, és a UNICODE konzorcium folyamatosan tekinti át a maradék felosztására vonatkozó javaslatokat. A UNICODE elfogadásának felgyorsítására törekedve a konzorcium a Latin-1 kódokat bölcsen a 0-tól 255-ig terjedő kódpozíciókra helyezte, ezáltal az ASCII és a UNICODE közötti konverzió könnyen elvégezhető. A kódpozíciók elvesztegetésének megakadályozása érdekében minden diakritikus szimbólumnak saját kódpozíciója van. A szoftver feladata, hogy kombinálja a diakritikus szimbólumot a szomszédos karakterrel, hogy egy újabbat állítson elő.

A kódpozíciók 16-osával blokkokra vannak osztva. Minden jelentősebb ábécé néhány egymás utáni zónát foglal el. Néhány példa (és a lefoglalt kódpozíciók száma): latin (336), görög (144), cirill (256), örmény (96), héber (112), devanagari (128), gurmukhi (128), oriya (128), telugu (128) és kannada (128). Figyeljük meg, hogy mindegyik nyelvhez több kódpozíciót rendeltek, mint ahány betűje van. Ez részben azért van, mert sok nyelvben egy betűnek több formája is van. Például az angolban minden betűnek két formája van – kisbetű és NAGYBETŰ. Némely nyelvben három vagy több forma is van, esetleg attól függően, hogy a betű a szó elején, közben vagy a végén van-e.

Az ábécéken kívül kódpozíciókat rendeltek a diakritikus jelekhez (112), írásjelekhez (112), alsó és felső indexekhez (48), pénznemek jeleihez (48), matematikai szimbólumokhoz (256), geometriai formákhoz (96) és egyéb érdekes jelekhez (192).

Ezek után következnek a kínai, japán és koreai nyelvhez szükséges szimbólumok. Először 1024 fonetikus szimbólum (például a katakana és a bopomofo), majd a kínaiban és a japánban használt egyesített Han ideogramok (20 992), végül a koreai Hangul szótagok (11 156).

A felhasználók kitalálhatnak speciális karaktereket, és saját céljaikra használhatnak 6400 kódpozíciót.

Bár a UNICODE a nemzetközi használattal kapcsolatos sok problémát megold, nem vállalkozik a világ összes problémájának megoldására. Például, míg a latin ábécé sorrendben van, a Han ideogramok nincsenek ábécé szerint rendezve. Ennek következménye, hogy egy angol program ábécérendbe állíthatja a „cat" és „dog" szavakat az első betűjük UNICODE-jának összehasonlításával. Egy japán programnak külső táblázatokat kell használnia annak eldöntésére, hogy két szimbólum közül melyik van előbb az ábécében.

Egy másik probléma, hogy folyamatosan új szavak jönnek létre. Ötven évvel ezelőtt senki sem használta az applet, kibertér, gigabájt, lézer, modem, smiley és videokazetta szavakat. Az angolban az új szavak nem igényelnek új kódpozíciókat. A japánban igen. Az új technikai szavak mellett igény van legalább 20 000 új (főként kínai) személy- és helységnév felvételére. A vakok szerint a Braille-írásjeleket is fel kellene venni, valamint különféle érdekcsoportok is igényt tartanak a szerintük jogosan nekik járó kódpozíciókra. A UNICODE konzorcium megtárgyalja, és dönt az összes új javaslatról.

A UNICODE ugyanazt a kódpozíciót használja olyan karakterekre, amelyek a japánban és a kínaiban majdnem egyformán néznek ki, de mást jelentenek, vagy kicsit különbözőképpen írják a két nyelvben (mintha az angol szövegszerkesztők a „blue" helyett mindig „blew"-t javasolnának, mivel ugyanúgy kell őket kiejteni). Némelyek ezt úgy tekintik, mint az értékes kódpozíciókkal való takarékoskodást; mások szerint ez az angolszász kulturális imperializmus újabb megnyilvánulása (van még valaki, aki azt gondolja, hogy 16 bites karakterkódok kiosztása nem politikai kérdés?). Még tovább rontja a helyzetet, hogy a teljes japán szótárban (a neveket nem számítva) 50 000 kanji írásjel van, így a Han ideogramokhoz rendelkezésre álló 20 992 kódpozíció mellett kényelmetlen döntéseket kell hozni. Nem minden japán gondolja úgy, hogy egy számítógépes cégekből álló konzorcium (még ha van köztük néhány japán is) az ideális fórum ezeknek a döntéseknek a meghozatalára.

2.5. Összefoglalás

A számítógépes rendszerek háromféle típusú alkotóelemből állnak: processzorok, memóriák és B/K eszközök. A processzor feladata, hogy az utasításokat a memóriából egyenként beolvassa, dekódolja és végrehajtsa. A betöltő-dekódoló-végrehajtó ciklus mindig leírható egy algoritmussal, és valójában ezt néha egy alacsonyabb szinten futó értelmező program végzi. A sebesség növelése érdekében ma sok számítógép egy vagy több csővezetéket tartalmaz, vagy szuperskaláris felépítésű, amelyben több, párhuzamosan működő funkcionális egységet találunk.

Egyre gyakoribbak a több processzort tartalmazó rendszerek. A párhuzamos számítógépek között találjuk a tömbprocesszorokat, amelyekben ugyanazt a műveletet lehet elvégezni egyszerre több adathalmazon, a multiprocesszorokat, amelyekben több CPU egy közös memórián osztozik, és a multiszámítógépeket, amelyek üzenetküldéssel kommunikáló, külön memóriával rendelkező számítógépek.

A memóriákat két csoportba soroljuk: központi memória és háttérmemória. A központi memória tárolja a végrehajtás alatt álló programot. Elérési ideje rövid

– legfeljebb néhányszor tíz nanoszekundum –, függetlenül az elérni kívánt címtől. A gyorsítótárak még tovább csökkenthetik az elérési időt. Bizonyos memóriák a megbízhatóság növelése érdekében hibajavító kódolással működnek.

Ezzel ellentétben a háttérmemóriák elérési ideje sokkal hosszabb (ezredmásodperc, esetleg még több), ami függ az olvasni vagy írni kívánt adat helyétől. A szalagok, mágneslemezek és optikai lemezek a leggyakoribb háttérmemóriák. A mágneslemezeknek sok fajtája van, például a hajlékonylemez, a Winchester, IDE-lemez, SCSI-lemez és a RAID. Optikai lemez a CD-ROM, a CD-R és a DVD.

A B/K egységek arra szolgálnak, hogy a számítógépből adatokat vigyenek ki, illetve a számítógépbe adatokat vigyenek be. A processzorhoz és a memóriához egy vagy több sín segítségével kapcsolódnak. Példaként említettük a terminált, az egeret, a nyomtatót és a modemet. A legtöbb B/K eszköz az ASCII karakterkódot használja, habár a UNICODE elfogadottsága a számítógépipar globalizálódásával egyre gyorsabban növekszik.

2.6. Feladatok

1. Tekintsük egy olyan gép működését, amelynek az adatútja a 2.2. ábrán látható. Tegyük fel, hogy az ALU bemeneti regiszterek feltöltése 5 ns-ot vesz igénybe, az ALU 10 ns alatt végzi el a feladatát, az eredmények regiszterekbe töltéséhez pedig további 5 ns kell. Mennyi a maximális MIPS érték, amit ez a gép csővezeték nélkül el tud érni?
2. Mi a célja a 2.1.2. alfejezet felsorolásában a 2. lépésnek? Mi történne, ha ezt a lépést kihagynánk?
3. Az 1. számítógépen minden utasítás végrehajtása 10 ns ideig tart. A 2. számítógépen 5 ns-ig. Állíthatjuk-e teljes bizonyossággal, hogy a 2. számítógép gyorsabb? Fejtse ki bővebben.
4. Tegyük fel, hogy Ön tervezett egy beágyazott rendszerben használható egyetlen lapkából álló számítógépet. Az összes memória a processzorral együtt a lapkán van, ugyanolyan sebességgel működik, és az adatok elérésekor nincs késleltetés. Vizsgálja meg egyenként a 2.1.4. rész tervezési elveit, és döntse el, hogy még mindig olyan fontosak-e (feltételezve, hogy a nagy teljesítmény továbbra is kívánatos).
5. Egy bizonyos számítás erősen szekvenciális – vagyis minden lépés függ az előzőtől. Egy tömbprocesszor vagy egy csővezetéket használó processzor lenne megfelelőbb ehhez a számításhoz? Magyarázza el.
6. Az újonnan feltalált könyvnyomtatással versenyre kelve, egy középkori kolostorban elhatározták, hogy kézzel írt papírkötésű könyvek tömeggyártásába fognak úgy, hogy nagyon sok írnokot összegyűjtenek egy hatalmas terembe. A főírnok hangosan felolvassa a másolandó könyv első szavát, amelyet az összes írnok leír. Ezután a főírnok felolvassa a második szót, amelyet az összes írnok leír. Ezt ismétlik addig, amíg az egész könyvet hangosan fel nem olvasták, és le nem másolták. A 2.1.6. rész melyik párhuzamos processzorokból álló rendszerére emlékeztet legjobban a fenti módszer?
7. A könyvben tárgyalt ötszintű memóriahierarchiában lefelé haladva az elérési idő növekszik. Adjon megalapozott becslést az optikai lemez és a regiszterek elérési idejének arányára. Feltételezhetjük, hogy a lemez a meghajtóban van.
8. A szociológusok egy kérdőívben feltett tipikus kérdésre, mint például „Hisz-e ön a fogtündérben?" három lehetséges választ kaphatnak: igen, nem vagy nem tudom. Ezt figyelembe véve, a Szociomágneses Számítógépgyár elhatározta, hogy számítógépet épít kérdőívek feldolgozására. Ennek a számítógépnek háromállapotú memóriája van – azaz minden egyes bájt (trájt?) 8 tritet tartalmaz, minden egyes trit egy 0, 1 vagy 2 értéket tárol. Hány tritre van szükség egy 6 bites szám reprezentálásához. Adjon még egy képletet, amely megadja egy n bites szám ábrázolásához szükséges tritek számát.
9. Számítsa ki az emberi szem adatátviteli kapacitását a következő információk segítségével. A látómező körülbelül 10^6 elemből (pixelből) áll. Minden pixel redukálható a három alapszín szuperpozíciójára, mindhárom 64 különböző intenzitásértéket vehet fel. A feldolgozáshoz 100 msec áll rendelkezésre.
10. Számítsa ki az emberi fül adatátviteli kapacitását a következő információkból. Az emberi hallás 22 kHz frekvenciáig terjed. Ahhoz, hogy egy 22 kHz-es hangjelből minden információt kinyerjünk, a hangból mintát kell vennünk a határfrekvencia kétszeresével, 44 kHz-cel. Egy 16 bites intenzitásérték valószínűleg elég a legtöbb hallható információ kinyerésére (azaz a fül nem tud 65 535-nél több intenzitásszintet megkülönböztetni).
11. Minden élőlény genetikai információja DNS-molekulákban tárolódik. Egy DNS-molekula a négy alapnukleotid (A, C, G és T) lineáris sorozata. Az emberi génállomány megközelítőleg 3×10^9 nukleotidból áll, körülbelül 30 000 gén formájában. Mennyi információ van (bitekben mérve) a teljes génállományban? Mekkora egy átlagos gén maximális információkapacitása (bitekben mérve)?
12. Egy bizonyos számítógépbe 268 435 456 bájtnyi memória tehető. Miért választ egy gyártó ilyen furcsa számot egy olyan könnyen megjegyezhető helyett, mint például a 250 000 000?
13. Tervezzen egy páros paritásos Hamming-kódot a 0-tól 9-ig terjedő számjegyekhez.
14. Tervezzen a 0, ..., 9 számjegyekhez olyan kódolást, amelynek 2 a Hamming-távolsága.
15. Egy Hamming-kódolásban bizonyos bitek „feleslegesek" abban az értelemben, hogy ellenőrzésre használjuk őket, és nem információtárolásra. Mennyi a felesleges bitek százalékos aránya azoknál az üzeneteknél, amelyek teljes hossza (adat + ellenőrző bitek száma) $2^n - 1$? Számítsa ki a kifejezés értékét 3 és 10 közé eső n értékekre.
16. A 2.19. ábrán látható mágneslemeznek sávonként 1024 szektora van, és 7200 RPM a forgási sebessége. Mennyi a lemez folyamatosan fenntartható adatátviteli sebessége egy sávon belül?

17. Egy számítógépnek 5 ns ciklusidejű sínje van, ennek során egy 32 bites szót tud a memóriába beírni vagy onnan kiolvasni. A számítógépben van egy Ultra4-SCSI-lemez, amely használja a sínt, és 160 MB/s az adatátviteli sebessége. A CPU normál körülmények között 10 ns-onként beolvas és végrehajt egy 32 bites utasítást. Mennyire lassítja le a lemez a CPU-t?
18. Tegyük fel, hogy egy operációs rendszer lemezkezelő részét Ön készíti. Logikailag a lemezt blokkok sorozataként reprezentálja, a legbelső blokk a 0-s, a többi egyesével növekvő sorszámokat kap a legkülsőig. A fájlok létrehozásához üres szektorokat kell lefoglalni. Ezt meg lehet tenni kívülről befelé vagy belülről kifelé. Lényeges-e, hogy melyik módszert választja? Indokolja meg a választ.
19. Mennyi ideig tart egy olyan lemezt végigolvasni, amelynek 10000 cilindere, cilinderenként 4 sávja és sávonként 2048 szektora van? Először a 0. sáv összes szektorát kell leolvasni a 0.-tól kezdődően, aztán az 1. sáv összes szektorát a 0.-tól kezdődően és így tovább. A körülfordulási idő 10 ms, egy szomszédos cilinder megkeresése 1 ms, a legrosszabb esetben egy cilinder megkeresése 20 ms. Egy cilinder sávjai közötti váltás időveszteség nélkül megvalósítható.
20. A 3-as szintű RAID képes javítani az egyetlen bitet érintő bithibákat egy paritásmeghajtó segítségével. Mi az értelme a 2-es szintű RAID-nek? Végül is, az is csak egyetlen bitre kiterjedő hibákat tud javítani, de ehhez több meghajtót használ.
21. Mennyi egy 2-es módú CD-ROM pontos kapacitása (bájtokban), ha a szokásos 80 percnyi adat van rajta? Mekkora a felhasználó számára elérhető kapacitás az 1-es módban?
22. Egy CD-R felírásához a lézernek nagy sebességgel kell ki-be kapcsolnia. Ha 10× sebességgel 1-es módban dolgozik, mennyi az impulzusok hossza nanoszekundumban?
23. Jelentős arányú tömörítésre van szükség ahhoz, hogy egy egyoldalas, egyrétegű DVD-re ráférjen 133 percnyi videofilm. Számítsa ki a szükséges tömörítési arányt. Tételezze fel, hogy 3,5 GB áll a videofilm rendelkezésére, a képfelbontás 720 × 480 pixel 24 bites színekkel, és 30 képet vetítünk másodpercenként.
24. A Blu-Ray 4,5 MB/s sebességgel működik és a kapacitása 25 GB. Mennyi ideig tart elolvasni egy teljes lemezt?
25. Egy CPU és a hozzá tartozó memória közötti átviteli sebesség nagyságrendekkel nagyobb, mint a mechanikus B/K átviteli sebesség. Miért rontja ez a kiegyensúlyozatlanság a hatékonyságot? Hogyan enyhíthető a probléma?
26. Egy gyártó reklámjában azt állítja, hogy színes bittérképes terminálja 2^{24} különböző színt tud megjeleníteni. A berendezésben azonban minden pixelhez csak 1 bájt tartozik. Hogy lehetséges ez?
27. Egy bittérképes terminál felbontása 1600 × 1200. A képernyő másodpercenként 75-ször rajzolódik újra. Milyen hosszú az egyetlen pixelhez tartozó impulzus?
28. Egy bizonyos betűkészlettel egy monokróm lézernyomtató egy lapra 50 sort, soronként 80 karaktert tud nyomtatni. Egy átlagos karakter egy 2 × 2 mm méretű négyzetet foglal el, amelynek körülbelül 25%-át a tonerből származó festék fedi. A fennmaradó rész fehéren marad. A festékréteg vastagsága 25 mikron. A nyomtató tonerkazettája 25 × 8 × 2 cm méretű. Hány lapot lehet kinyomtatni a tonerkazettával?

29. Ha egy páratlan paritásos ASCII szöveget egy 56 000 bps sebességű modemmel aszinkron módon 5600 karakter/s sebességgel elküldünk, a vett bitek hány százaléka a tulajdonképpeni adat (a töltelékbitekkel szemben)?
30. A Hi-Fi Modem Gyár elkészített egy új frekvenciamodulációs modemet, amely 2 helyett 64 frekvenciát használ. Minden másodperc n egyenlő hosszúságú intervallumra van osztva, ezek mindegyike a 64 lehetséges hang egyikét tartalmazza. Szinkron átviteli módban hány bitet tud ez a modem elküldeni másodpercenként?
31. Egy internetfelhasználó előfizetett egy 2 Mbps sebességű ADSL-szolgáltatásra. A szomszédja kábeles internetre fizetett elő, amelynek az osztott sávszélessége 12 MHz. Az alkalmazott modulációs séma a QAM-64. Tegyük fel, hogy n ház van a kábelre kötve, és mindegyikben van egy-egy számítógép. Egy adott időpontban ezeknek a számítógépeknek f része online. Milyen feltételek esetén kap jobb szolgáltatást a kábeles internet előfizetője az ADSL-előfizetővel szemben?
32. Egy digitális kamera felbontása 3000 × 2000 pixel, 3 bájt/pixeles RGB színekkel. A kamera gyártója 5× tömörítéssel szeretné a flash memóriába írni a képet 2 másodperces belül. Milyen adatátviteli sebesség szükséges ehhez?
33. Egy kiváló minőségű digitális kamerának 16 millió pixeles érzékelője van, mindegyik pixel 3 bájtos. Hány képet tud tárolni egy 1 GB-os flash memóriakártya, ha a tömörítési tényező 5×? Tegyük fel, hogy 1 GB 2^{30} bájt.
34. Becsülje meg, hogy a szóközöket is beleértve egy tipikus számítástudományi témájú könyv hány karaktert tartalmaz. Hány bitre van szükség ahhoz, hogy a könyvet paritással együtt ASCII kódban tároljuk? Hány CD-ROM-ra van szükség ahhoz, hogy egy 10000 kötetes számítástudományi könyvtárat tároljunk? Hány kétoldalas, kétrétegű DVD-re van szükség ugyanehhez?
35. Írjon egy *hamming*(*ascii*, *encoded*) eljárást, amely az *ascii* alsó hét bitjét egy 11 bites, kódszóvá alakítja, és az *encoded* változóba tárolja.
36. Írjon egy *distance*(*code*, *n*, *k*) függvényt, amely az n darab k bites karakterből álló *code* tömböt dolgozza fel, és eredményként a karakterhalmaz távolságát adja vissza.

3. A digitális logika szintje

Az 1.2. ábrán látható hierarchia alján találjuk a digitális logika szintjét: a számítógépek igazi hardverét. Ebben a fejezetben a digitális logikát több szempont szerint fogjuk vizsgálni. Ez egy alapozó blokk a magasabb szintek tanulmányozásához, amelyeket a későbbi fejezetekben fogunk tárgyalni. A téma a számítástudomány és a villamosmérnöki munka határán van, de mivel az anyag önmagában is érthető, követéséhez nem szükséges előzetes hardver vagy mérnöki tapasztalat.

Az alapelemek, amelyekből minden digitális számítógép felépül, meglepően egyszerűek. Vizsgálatainkat azzal kezdjük, hogy bemutatjuk ezeket az alapelemeket, továbbá egy speciális kétértékű algebrát (Boole-algebrát) is, amelyet az alapelemek elemzésére használunk. Ezután megvizsgálunk néhány alapvető áramkört (kapu), melyek egyszerű kombinációjával további áramköröket tudunk felépíteni; olyan áramköröket is, amelyekkel aritmetikai műveleteket hajthatunk végre. A következő fejezet arról szól, hogy milyen módon kombinálhatjuk a kapukat információtárolás céljából, azaz hogyan épül fel a memória. Ezt követően a központi egységek témáját tárgyaljuk, és főleg azt elemezzük, hogy hogyan kapcsolódnak össze az egylapkás központi egységek a memóriával és a perifériális eszközökkel. A fejezet végén számos példát mutatunk be az ipari alkalmazásokból.

3.1. Kapuk és Boole-algebra

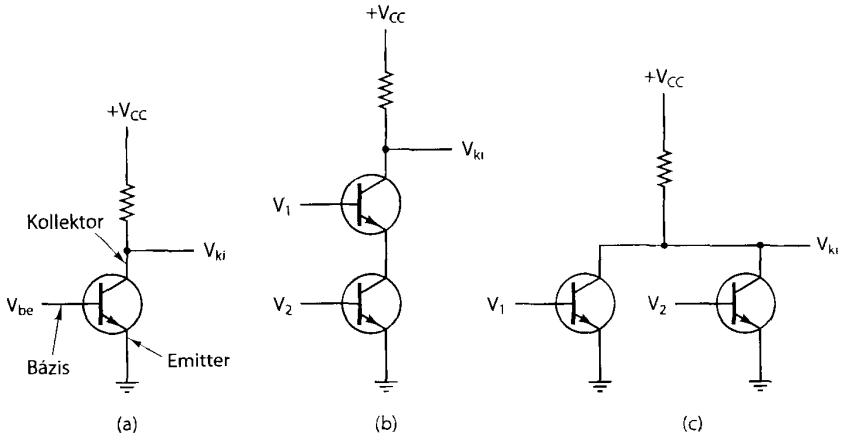
Digitális áramköröket készíthetünk néhány nagyon egyszerű elem különböző kombinálásával. A következő részekben ezeket az alapelemeket írjuk le, és megmutatjuk, hogyan kombinálhatók, valamint bevezetünk egy hatékony matematikai technikát, amelyet viselkedésük elemzésére használunk.

3.1.1. Kapuk

A digitális áramkör egy olyan áramkör, amelyben csak két logikai érték van jelen. Tipikusan a 0 és 1 volt (V) közötti jel reprezentál egy értéket (például a bináris 0-t), és a 2 és 5 volt közötti egy másik értéket (például a bináris 1-et). Ezen a két intervallumon kívüli feszültségértékek nem megengedettek. A pici elektromos eszközök, amelyeket **kapu**nak (**gates**) hívunk, kétértékű jelek különböző függvényeit tudják kiszámítani. Ezek a kapuk alkotják azt a hardverbázist, amelyből minden digitális számítógép felépül.

A kapuk belső működésének részletezése túlmutat ennek a könyvnek a terjedelmén, az **eszközszint**hez (**device level**) tartozik, amely a mi nullás (0.) szintünk alatt van. Mindezek ellenére elkalandozunk, hogy egy rövid áttekintést adjunk erről a nem nagyon bonyolult alapműködésről. Minden modern digitális logika végül is azon a tényen alapul, hogy egy tranzisztor úgy tud működni, mint egy nagyon gyors bináris (kétállapotú) kapcsoló. A 3.1. (a) ábrán bemutatunk egy bipoláris tranzisztort (a kör), amely be van ágyazva egy egyszerű áramkörbe. A tranzisztornak három kapcsolata van a külvilággal: a **kollektor**, a **bázis** és az **emitter**. Amikor a bemenő feszültség, V_{be} bizonyos érték alatt van, a tranzisztor zárt állapotban van, és úgy viselkedik, mint egy végtelen ellenállás. Ez azt okozza, hogy az áramkör outputja, V_{ki} közel van a V_{cc} értékéhez, ami egy külsőleg vezérelt feszültség, tipikusan +5 volt az ilyen típusú tranzisztornál. Amikor V_{be} meghaladja a kritikus értéket, a tranzisztor kinyit, és úgy viselkedik, mint egy vezeték, és ez azt eredményezi, hogy a V_{ki}-t lehúzza a földhöz (lecsökkenti a megállapodás szerinti 0 voltra).

Nagyon fontos dolog megjegyezni, hogy amikor a V_{be} alacsony, akkor a V_{ki} magas, és fordítva. Ez az áramkör így egy fordító (inverter), amely a logikai 0-t logikai 1-gyé konvertálja, és a logikai 1-et pedig logikai 0-vá. Az ellenállás (a cikcak-



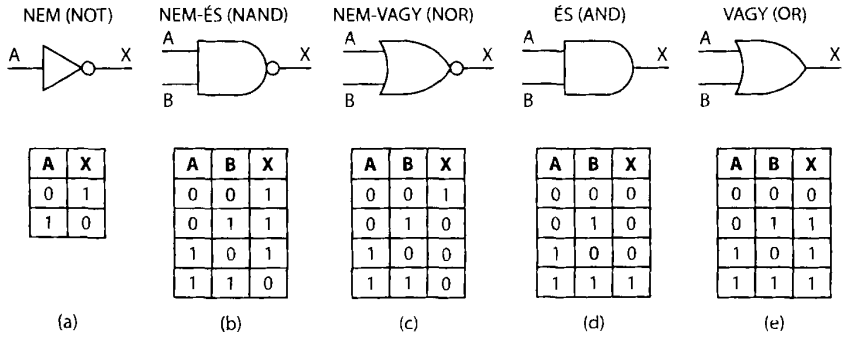
3.1. ábra. *(a) Tranzisztoros fordító [inverter, NEM, (NOT) kapu]. (b) NEM-ÉS (NAND) kapu. (c) NEM-VAGY (NOR) kapu*

kos vonal jelzi) azért szükséges, hogy behatárolja az áram mennyiségét, amelyet a tranzisztor levezet, hogy ne égjen ki. Az egyik állapotból a másik állapotba szükséges kapcsolási idő tipikusan néhány nanoszekundum.

A 3.1. (b) ábrán két tranzisztor van egymás után sorba kötve. Ha V_1 és V_2 is magas, mindkét tranzisztor vezetni fog, és V_{ki} alacsony lesz. Ha valamelyik bemenet alacsony, a megfelelő tranzisztor zárva lesz, a kimenet pedig magas. Más szavakkal: a V_{ki} akkor és csak akkor lesz alacsony, ha mind a V_1, mind a V_2 magas.

A 3.1. (c) ábra két tranzisztor párhuzamos kapcsolását tartalmazza a soros helyett. Ebben a felépítésben, ha valamelyik bemenet magas, akkor a megfelelő tranzisztor vezet, és a kimenetet földeli. Ha a két bemenet alacsony, a kimenet magas marad.

Ez a három áramkör vagy azonos megfelelőik (ekvivalenseik) alkotják a három legegyszerűbb kaput. Nevük rendre: NEM (NOT), NEM-ÉS (NAND) és NEM-VAGY (NOR) kapu. A NEM kapukat gyakran hívjuk **invertereknek (fordítók)**; a két megnevezést felcserélhető módon fogjuk használni. Ha bevezetjük azt a konvenciót, hogy a „magas" (V_{cc}) a logikai 1, és az „alacsony" (föld) a logikai 0, a kimenet értékét a bemeneti értékek egy függvényeként fejezhetjük ki. A három kapu leírásához használt szimbólumokat a 3.2. (a)–(c) ábrák mutatják az egyes áramkörök működésével együtt. Ezeken az ábrákon A és B a bemenet, X a kimenet. Minden sor meghatározza a kimenetet a bemenetek egy-egy kombinációjára.



A	X
0	1
1	0

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

3.2. ábra. *Az öt alapkapu szimbóluma és működése*

Ha a 3.1. (b) ábra kimenő jelét bevezetjük egy invertáló áramkörbe, egy másik áramkört kapunk, amely pontosan az ellenkezője a NEM-ÉS kapunak – azaz egy olyan áramkör, amelynek a kimenete akkor és csak akkor 1, ha mindkét bemenete 1. Egy ilyen áramkört ÉS (AND) kapunak hívunk, szimbólumát és működését a 3.2. (d) ábrán adjuk meg. Hasonlóan, ha a NEM-VAGY kaput összekötjük egy inverterrel, egy olyan áramkört kapunk, ahol a kimenet akkor 1, ha legalább az egyik bemenet 1, és 0, ha mindkét bemenet 0. Ennek az áramkörnek, amelyet VAGY (OR) kapunak hívunk, a szimbóluma és működése a 3.2. (e) ábrán található. A kis köröket, amelyeket a szimbólumok részeként használunk az inverter jelölésére a NEM-ÉS és a NEM-VAGY kapuknál, **inverziós gömböknek (inversion bubbles)** nevezzük. Gyakran használjuk ezeket más környezetben is, egy ellenkezőjére váltott jel jelölésére.

A 3.2. ábrán szereplő öt kapu adja a digitális logika szintjének alapvető építőelemeit. Az eddigi tárgyalásból világos, hogy a NEM-ÉS és a NEM-VAGY kapuk két tranzisztort igényelnek, míg az ÉS és a VAGY kapuk egyenként hármat. Ezen okból sok számítógép inkább a NEM-ÉS és NEM-VAGY kapukra alapozódik, és nem a jobban ismert ÉS és VAGY kapukra. (A gyakorlatban minden kapu megvalósítható különböző módon, de a NEM-ÉS és a NEM-VAGY kapuk egyszerűbbek, mint az ÉS és VAGY kapuk.) Mellékesen érdemes megjegyeznünk, hogy a kapuknak több mint két bemenete is lehet. Például a NEM-ÉS kapunak tetszőlegesen sok bemenete lehet, de a gyakorlatban nyolcnál több nem használatos.

Bár annak a részletei, hogy hogyan építik meg a kapukat, az eszközszinthez tartozik, megemlítünk két fő gyártási technológiai családot, amelyekre a leggyakrabban hivatkoznak. A két fő technológia a **bipoláris** és a **MOS (Metal Oxide Semiconductor) fém-oxid félvezetős**. A bipoláris technológia fő típusa a **TTL (Transistor-Transistor Logic) tranzisztor-tranzisztor logika**, amely évek óta az egyik fő hordozója, „munkalova" a digitális elektronikának, és az **ECL (Emitter-Coupled Logic) emitter csatolású logika**, amelyet akkor használnak, ha nagyon nagy sebességű műveletek végrehajtására van szükség. A **MOS**-technológiát manapság széles körben használják számítógépes áramkörökben.

A MOS-kapuk lassabbak, mint a TTL és az ECL, de kevesebb áramot igényelnek, és kevesebb helyet foglalnak el, és ezért sokkal nagyobb számú elem rakható szorosan egymás mellé. A MOS-nak nagyon sok változata van, beleértve a PMOS-t, az NMOS-t és a CMOS-t. Bár a MOS-tranzisztorokat a bipoláris tranzisztoroktól eltérő módon készítik, az elektronikai kapcsolási képességük azonos. A legtöbb modern központi egység és memória CMOS-technológiát használ, amely +3,3 volt feszültséget igényel. Ez az összes, amit el szerettünk volna mondani az eszközszintről. (További tanulmányozáshoz lásd 9. fejezet: Ajánlott olvasmányok és irodalomjegyzék.)

3.1.2. Boole-algebra

A kapuk kombinációjából felépíthető áramkörök leírásához egy új típusú algebra szükséges, amelynek változói és függvényei csak a 0 és 1 értéket vehetik fel. Egy ilyen algebrát **Boole-algebrának** hívnak, felfedezőjéről, George Boole (1815–1864) angol matematikusról elnevezve. Pontosabban szólva a Boole-algebra egy specifikus típusára, a **kapcsolóalgebrára (switching algebra)** fogunk hivatkozni, de a Boole-algebra kifejezés olyan széles körben használatos kapcsolóalgebra jelentéssel, hogy ebben nem teszünk különbséget.

Csakúgy, mint a „hagyományos" algebrában (azaz a középiskolai algebrában) függvényeket értelmezünk. A *Boole-függvényeknek* egy vagy több bemeneti változója van, és egy eredményt szolgáltat, amely csak e változók értékétől függ. Egy egyszerű f függvényt, úgy definiálhatunk, hogy megmondjuk: $f(A)$ 1, ha A 0 és $f(A)$ 0, ha A 1. Ez a függvény a 3.2. (a) ábra NEM függvénye.

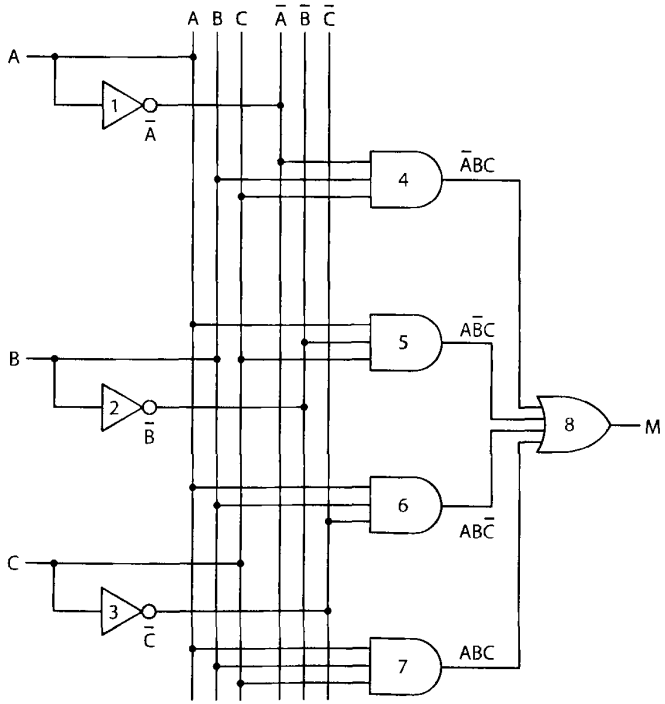
Mivel az n változós Boole-függvény változóinak csak 2^n lehetséges kombinációja van, a függvényt teljesen leírhatjuk egy 2^n sorral rendelkező táblázattal, ahol

egy-egy sor megmondja a bemeneti értékek adott kombinációja mellett a függvény értékét. Ezt a táblázatot **igazságtáblázat**nak **(truth table)** nevezzük. A 3.2. ábra minden táblázata igazságtáblázat. Ha megegyezünk abban, hogy egy igazságtáblázat sorait mindig numerikus (2-es alapú) számsorrendben adjuk meg, azaz a két változónál a sorrend 00, 01, 10 és 11, akkor a függvényt teljesen leírhatjuk egy 2^n bit hosszú bináris számmal, amelyet az igazságtáblázat eredményoszlopát függőlegesen olvasva kapunk meg. Ez a NEM-ÉS-nél 1110, a NEM-VAGY-nál 1000, az ÉS-nél 0001 és a VAGY-nál 0111. Nyilvánvaló, hogy két változónak csak 16 Boole-függvénye létezik, amelyek megfelelnek a 16 lehetséges 4 bit hosszúságú eredménysorozatoknak. Ezzel ellentétben a hagyományos algebra végtelen sok kétváltozós függvényt tartalmaz, ezek egyike sem írható le az összes bemenethez tartozó igazságtáblázattal, mert mindegyik változónak végtelen sok lehetséges értéke van.

A 3.3. (a) ábra mutatja egy háromváltozós Boole-függvény igazságtáblázatát: $M = f(A, B, C)$. Ez a függvény a többségi logikai függvény, azaz 0-val egyenlő, ha a bemenetek többsége 0, és 1, ha a bemenetek többsége 1. Bár tetszőleges Boole-függvényt megadhatunk az igazságtáblázatával, a változók számának növekedésével ez a jelölés egyre kényelmetlenebb. Ehelyett gyakran más jelölést alkalmaznak.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

3.3. ábra. *(a) A háromváltozós többségi függvény igazságtáblázata. (b) Áramkör (a) megvalósítására*

Ahhoz, hogy megértsük, honnan ered a másik jelölés, vegyük észre, hogy bármely Boole-kifejezés meghatározható, ha megmondjuk, hogy a bemenő változók mely kombinációi adnak 1-es kimeneti értéket. A 3.3. (a) ábra függvénye a változók négy bemeneti kombinációjára ad M-re 1 értéket. Megállapodás szerint egy bemeneti változó fölé felülvonást helyezünk, ha jelezni akarjuk, hogy az értéke invertált. A felülvonás hiánya azt jelenti, hogy nem invertált. Implicit jelölésű szorzást vagy egy pontot használunk a Boole ÉS (AND, logikai szorzás) és +-t a Boole VAGY (OR, logikai összeadás) függvény jelölésére. Így például az $A\bar{B}C$ azt jelenti, hogy ennek az értéke csak akkor 1, ha $A = 1$, $B = 0$ és $C = 1$. Ugyanígy $AB + B\bar{C}$ csak akkor 1, ha ($A = 1$ és $B = 0$) vagy ($B = 1$ és $C = 0$). A 3.3. (a) ábrának négy sora eredményez 1-es bitet a kimeneten: $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ és ABC. Az M függvény igaz (azaz 1 értékű), ha a négy eset valamelyike igaz; így felírhatjuk, hogy

$$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

ami az igazságtáblázat egy tömör formája. Az n változós függvényt úgy tudjuk leírni, mint legfeljebb 2^n darab n változós (logikai) szorzat (logikai) összegét. Ahogy rövidesen látni fogjuk ez a formula különösen azért fontos, mert közvetlenül megmutatja, hogy valósítható meg a függvény szabványos kapuk használatával.

Fontos, hogy ne felejtsük el a különbséget az absztrakt Boole-függvény és az elektronikus áramkörrel történő megvalósítása között. A Boole-függvény változókból, mint A, B és C, és Boole-műveletekből áll, mint ÉS, VAGY és NEM. A Boole-függvényeket igazságtáblázattal vagy olyan Boole-kifejezésekkel írhatjuk le, mint például

$$F = A\bar{B}C + A\bar{B}\bar{C}$$

Egy Boole-függvényt megvalósíthatunk elektronikus áramkörrel (gyakran különböző módokon is), ha a bemenő és kimenő változókat elektromos jelekkel reprezentáljuk, a műveleteket pedig kapukkal, mint az ÉS, VAGY és NEM. Általában az ÉS, VAGY és NEM jelöléseket használjuk, amikor a Boole-operátorokra, és az ÉS, VAGY és NEM jelöléseket, ha a kapukra gondolunk, de ez gyakran elég bizonytalan.

3.1.3. A Boole-függvények megvalósítása

Ahogy az előbb említettük, a Boole-függvény szorzatok legfeljebb 2^n tagú összegével történő leírása közvetlenül elvezet egy lehetséges megvalósításhoz. A 3.3. ábrát használva példaként, láthatjuk, hogyan hajtható végre a megvalósítás. A 3.3. (b) ábrán az A, B és C bemeneteket a bal szélen, a kimeneti függvényt, M-et, a jobb szélen ábrázoltuk. Mivel szükségünk van a bemeneti változók invertált értékére (komplemensére), a bemenetek elágaztatásával, az 1-gyel, 2-vel és 3-mal jelzett inverteren átvezetve előállítjuk ezeket. Annak érdekében, hogy ne legyen zavaros az ábra, berajzoltunk hat függőleges vonalat, amelyből három össze van kötve a bemeneti változókkal, a másik három ezek komplemenseivel. Ezek a vonalak

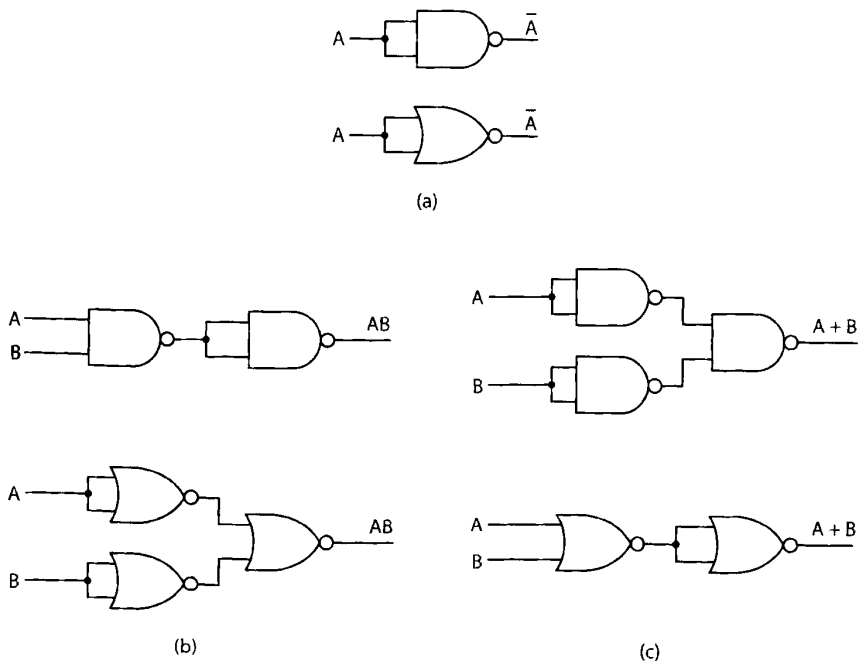
szolgáltatják a bemeneteket a kapuk számára. Például az 5, 6, 7 kapuk mindegyike használja A-t bemenetként. Egy tényleges áramkörben ezek a kapuk valószínűleg közvetlenül hozzá vannak kötve A-hoz anélkül, hogy közbülső „függőleges" vezetéket használnának.

Az áramkör négy ÉS kaput tartalmaz, az M függvény minden tagjához egyet (az igazságtáblázat minden olyan sorához, ahol 1-es bit van az eredményoszlopban). Minden ÉS kapu kiszámolja az igazságtáblázatnak egy sorát, ahogy jeleztük. Végül a szorzatok eredményei VAGY-olásra kerülnek, és ez adja a végeredményt.

A 3.3. (b) ábrán látható áramkör egy olyan megállapodást használ, amelyre többször szükségünk lesz az egész könyvben: amikor két vonal metszi egymást, nincs kapcsolat közöttük, hacsak egy nagy pont nem jelzi ezt a kereszteződésben. Például a 3-as kapu kimenete keresztezi mind a hat függőleges vonalat, de csak a C-hoz csatlakozik. Legyünk óvatosak, mert sok szerző más jelölést használ.

A 3.3. ábrán megadott példából világosnak kell lennie, hogy miként valósítható meg áramkörrel bármilyen Boole-függvény:

1. Írjuk fel a függvény igazságtáblázatát.
2. Biztosítsunk NEM kapukat minden bemenet komplemensének előállításához.
3. Rajzoljunk ÉS kaput minden sorhoz, amelynek az eredményoszlopában 1 van.



3.4. ábra. *(a) NEM, (b) ÉS, (c) VAGY kapuk csak NEM-ÉS vagy csak NEM-VAGY kapukkal történő megvalósítása*

4. Kapcsoljuk össze az ÉS kapukat a megfelelő bemenetekkel.
5. Az összes ÉS kapu kimenetét tápláljuk be egy VAGY kapuba.

Bár megmutattuk, hogyan lehet bármilyen Boole-függvényt megvalósítani NEM, ÉS és VAGY kapukkal, gyakran kényelmesebb az áramköröket csak egyfajta kapuval megvalósítani. Szerencsére egyszerű módja van annak, hogy az előző algoritmus által létrehozott áramkört átalakítsuk tisztán NEM-ÉS vagy tisztán NEM-VAGY formájúvá. Ahhoz hogy megtegyünk egy ilyen átalakítást, csak arra van szükség, hogy a NEM, ÉS és VAGY kaput egyetlen kaputípus felhasználásával valósítsuk meg. A 3.4. ábra felső sora mutatja, hogyan valósíthatjuk meg mindhárom kaput csak NEM-ÉS kapukkal; az alsó sor mutatja, hogyan tehetjük meg csak NEM-VAGY kapukat használva. (Ez így egyszerű, de más mód is lehetséges.)

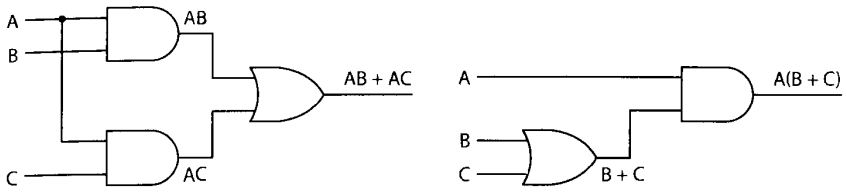
A Boole-függvények megvalósításának egyik módja, ha csak NEM-ÉS vagy csak NEM-VAGY kaput akarunk használni, hogy először a fenti megvalósítási lépéseket végezzük el NEM, ÉS és VAGY kapukkal. Ezután cseréljük ki a többszörös bemenettel rendelkező kapukat két bemenetű kapukból álló ekvivalens áramkörökkel. Például: $A + B + C + D$-t kiszámíthatjuk, mint $(A + B) + (C + D)$ úgy, hogy három darab kétbemenetes VAGY kaput használunk. Végül a NEM, ÉS és VAGY kapukat kicseréljük a 3.4. ábra áramköreivel.

Bár ez az eljárás nem vezet optimális áramkörhöz abban az értelemben, hogy minimális számú kaput használjon, de az kimutatható, hogy ez a megoldás mindig megvalósítható. A NEM-ÉS és a NEM-VAGY kapukról azt mondjuk, hogy **teljesek (complete)**, mert bármely *Boole-függvény kiszámítható* ezek bármelyikének kizárólagos felhasználásával. Más kapunak nincs meg ez a tulajdonsága, ami miatt gyakran előnyben részesítik ezeket az áramköri blokkok tervezésénél.

3.1.4. Áramköri ekvivalencia

A hálózattervezők gyakran próbálják csökkenteni termékeikben a kapuk számát, hogy csökkentsék az alkatrészek árát, a nyomtatott áramköri lap nagyságát, az áramfogyasztást és így tovább. Az áramkör bonyolultságának csökkentéséhez, a tervezőknek találni kell egy olyan áramkört, amely ugyanazt a függvényt számolja ki, mint az eredeti, de kevesebb kapuból áll (vagy egyszerűbb kapukból, például kétbemenetes kapukból a négybemenetesek helyett). Az ekvivalens áramkörök keresésében a Boole-algebra nagyon értékes eszköz.

A Boole-algebra felhasználására példaként tekintsük a 3.5. (a) ábrán bemutatott $AB + AC$ áramkört és igazságtáblázatot. Bár még nem tárgyaltuk, de a hagyományos algebra nagyon sok szabályát megtartja a Boole-algebra is. Speciálisan, az $AB + AC$ a disztribúciós szabály használatával felírható $A(B + C)$ alakban is. A 3.5. (b) ábra mutatja az $A(B + C)$ áramkörét és igazságtáblázatát. Mivel két függvény akkor és csak akkor ekvivalens, ha az összes lehetséges bemenetre a két függvény ugyanazt a kimenetet adja; nagyon egyszerűen ellenőrizhetjük a 3.5. ábra igazságtáblázataiból, hogy $A(B + C)$ ekvivalens $AB + AC$-vel. Az ekvivalencia ellenére világosan látható, hogy a 3.5. (b) áramkör jobb, mint a 3.5. (a), mert kevesebb kaput tartalmaz.



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

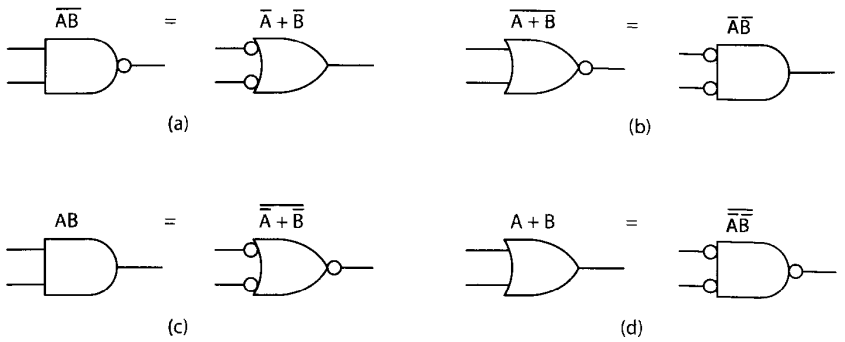
3.5. ábra. *Két ekvivalens függvény. (a) AB + AC. (b) A(B + C)*

Általában az áramkörtervezők Boole-függvénnyel kezdenek, és aztán alkalmazzák a Boole-algebra szabályait, és próbálnak egyszerűbb, de ekvivalens függvényt találni. A végleges formából azután létrehozzák az áramkört.

Ahhoz, hogy ezt a megközelítést használjuk, szükségünk van a Boole-algebra néhány azonosságára. A 3.6. ábra mutatja a legfontosabbakat. Érdekes megjegyezni, hogy minden szabálynak két formája van, amelyek egymásnak **duál**jai. Az ÉS és VAGY, valamint a 0 és az 1 egyidejű cseréjével bármelyik forma előállítható a duálisából. Az összes szabály könnyen bizonyítható igazságtáblázatuk megalko-

Név	ÉS forma	OR forma
Identitásszabály	$1A = A$	$0 + A = A$
Nullszabály	$0A = 0$	$1 + A = 1$
Idempotens szabály	$AA = A$	$A + A = A$
Inverz szabály	$A\overline{A} = 0$	$A + \overline{A} = 1$
Kommutatív szabály	$AB = BA$	$A + B = B + A$
Asszociatív szabály	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Disztribúciós szabály	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Abszorpciós szabály	$A(A + B) = A$	$A + AB = A$
De Morgan-szabály	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

3.6. ábra. *A Boole-algebra néhány azonossága*



3.7. ábra. *Néhány kapu alternatív jelölése: (a) NEM-ÉS. (b) NEM-VAGY. (c) ÉS. (d) VAGY*

tásával. A De Morgan-szabály, az abszorpciós szabály és a disztribúciós szabály ÉS formája kivételével az eredmények meglehetősen intuitívek. A De Morgan-szabályt kiterjeszthetjük több mint két változóra is, például $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$.

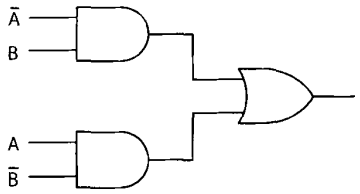
A De Morgan-szabály egy alternatív jelölést sugall. A 3.7. (a) ábra az ÉS alakot mutatja, a negációt inverziós gömb jelöli mind a bemeneten, mind a kimeneten. Így egy VAGY kapu invertált bemenettel ekvivalens egy NEM-ÉS kapuval. A 3.7. (b) ábrából, a De Morgan-szabály duál formájából világos, hogy egy NEM-VAGY kapu megrajzolható egy invertált bemenetekkel rendelkező ÉS kapuval. A De Morgan-szabály mindkét formájának negálásával jutunk el a 3.7. (c) és (d) ábrához, amelyek az ÉS és VAGY kapuk ekvivalens reprezentációit mutatják. Analóg szimbólumok léteznek a De Morgan-szabály többváltozós formáira (például egy n bemenettel rendelkező NEM-ÉS kapuból egy n invertált bemenettel rendelkező VAGY kapu lesz).

A 3.7. ábra azonosságait és a hasonló több bemenetű kapukat használva könnyű átalakítani egy igazságtáblázat összeg-szorzat ábrázolását tisztán NEM-ÉS vagy tisztán NEM-VAGY formájúvá. Példaként tekintsük a 3.8. (a) ábra KIZÁRÓ-VAGY (EXCLUSIVE OR, XOR) függvényét. A szabvány szorzat-összeg áramkört a 3.8. (b) ábra mutatja. Azért, hogy NEM-ÉS formájúvá konvertáljuk, a vonalakat, amelyek az ÉS kapuk kimenetét kötik össze a VAGY kapuk bemenetével, újra kell rajzolni két inverziós gömbbel, ahogy a 3.8. (c) ábra mutatja. Végül a 3.7. (a) ábrát használva eljutunk a 3.8. (d) ábrához. Az A és $\overline{B}$ változókat A és B-ből NEM-ÉS és NEM-VAGY kapukkal úgy generálhatjuk, hogy a bemeneteket összekötjük. Megjegyezzük, hogy az inverziós gömbök oda mozdíthatók a vonalak mentén, ahova akarjuk, például a 3.8. (d) ábra bemeneti kapuinak kimeneteiről elmozdíthatjuk a kimeneti kapu bemeneteihez.

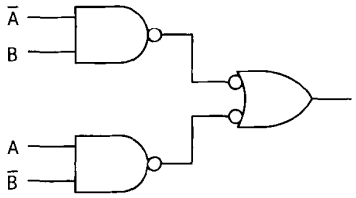
Végezetül az áramköri ekvivalenciával kapcsolatban egy meglepő eredményt fogunk bemutatni, mely szerint azonos fizikai kapukkal különböző függvényt tudunk kiszámolni, attól függően, hogy milyen konvenciót használunk. A 3.9. (a) ábrán bemutatjuk egy bizonyos F kapu kimenetét különböző bemeneti kombinációkkal. Mind a bemenetet, mind a kimenetet feszültségben (volt) adjuk meg. Ha azt a konvenciót használjuk, hogy a 0 volt logikai 0-nak és a 3,3 volt vagy az 5 volt

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

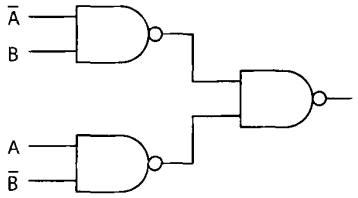
(a)



(b)



(c)



(d)

3.8. ábra. *(a) KIZÁRÓ-VAGY (XOR) függvény igazságtáblázata. (b)–(d) Három áramkör ezen függvény kiszámítására*

a logikai 1-nek felel meg, ezt **pozitív logikának** hívjuk, és a 3.9. (b) ábrán látható igazságtáblázatot, az ÉS függvényt kapjuk. Ha azonban a **negatív logikát** alkalmazzuk, ahol a 0 volt a logikai 1 és a 3,3 volt vagy az 5 volt a logikai 0, a 3.9. (c) ábra igazságtáblát, a VAGY függvényt kapjuk.

Így kritikus az a konvenció, amelyet a feszültségek logikai értékekre történő leképezésére használunk. Ha nem mondunk mást, ettől kezdve pozitív logikát fogunk használni, így az 1-es logikai érték fogalma, az igaz és a magas szinonimák; ugyanígy szinonima a logikai 0, a hamis és az alacsony.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

3.9. ábra. *(a) Az eszköz elektromos jellemzői. (b) Pozitív logika. (c) Negatív logika*

3.2. Alapvető digitális logikai áramkörök

Az előző szakaszokban megmutattuk, hogyan valósítjuk meg az igazságtáblázatokat és más egyszerű áramköröket egyedi kapuk használatával. Mostanában a gyakorlatban kevés áramkört alakítanak ki kapu alapon, bár valamikor ez volt az általános. Napjainkban a szokásos építőblokkok a modulok, amelyek számos kaput tartalmaznak. A következő szakaszokban megvizsgáljuk ezeket az építőblokkokat közelebbről, megnézzük, hogyan használják őket, és hogyan szerkeszthetők meg egyedi kapukból.

3.2.1. Integrált áramkörök

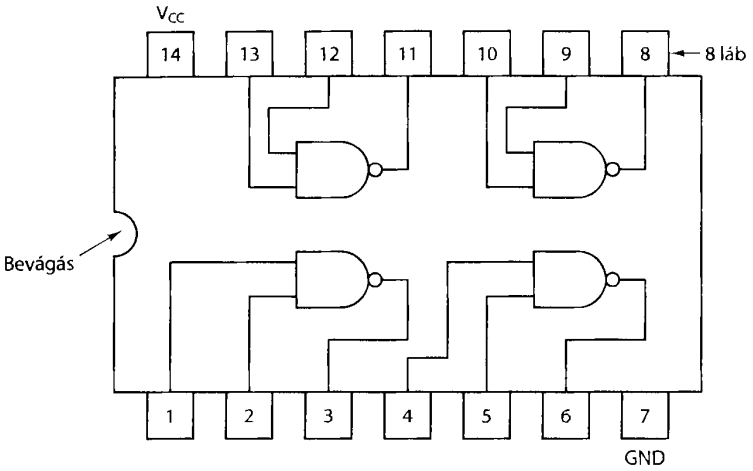
A kapukat nem egyedileg gyártják és árulják, hanem egységekben, ún. **integrált áramkörök**ben, amelyeket **IC**-knek **(Integrated Circuits)** vagy **lapkák**nak **(chips)** hívnak. Egy IC körülbelül egy 5 × 5 mm-es négyzetes szilíciumdarab, amelyen néhány kaput helyeznek el. A kis IC-ket szokásosan egy derékszögű műanyag vagy kerámialapon (tokban) helyezik el, amely 5–15 mm széles és 20–50 mm hosszú. A hosszú élek mentén 5 mm hosszú lábaknak két párhuzamos sora van, amely behelyezhető egy foglalatba, vagy nyomtatott áramköri táblára forraszthető. Minden láb egy-egy kapunak a bemenete vagy kimenete, vagy áram, vagy pedig föld bemenet. A kívül kétsoros lábazást és a belső integrált áramkört együtt **DIP**-nek **(Dual Inline Packages, kétlábsoros tokozás)** nevezik, de mindenki lapkának (chipnek) hívja, elkenve a különbséget a szilíciumdarab és a tokozása között. A legtöbb ismert tokozásnak 14, 16, 18, 20, 22, 24, 28, 40, 64 vagy 68 lába van. A nagy lapkánál gyakran használatos a négyzetes tokozás, ahol vagy mind a négy oldalon, vagy pedig az alsó részen találhatók lábak.

A lapkákat (nem pontos) osztályokba sorolhatjuk a kapuk száma alapján. Ez az osztályozás nyilvánvalóan eléggé durva, de gyakran hasznos.

SSI (Small Scale Integrated – kis integráltságú) áramkör: 1–10 kapu;
MSI (Medium Scale Integrated – közepes integráltságú) áramkör: 10–100 kapu;
LSI (Large Scale Integrated – nagy integráltságú) áramkör: 100–100 000 kapu;
VLSI (Very Large Scale Integrated – nagyon nagy integráltságú) áramkör:
>100 000 kapu.

Ezek az osztályok különböző tulajdonságokkal rendelkeznek, és különböző módon használhatók.

Egy SSI lapka tipikusan 2–6 független kaput tartalmaz, amelyeket külön-külön használhatunk az előző fejezetekben meghatározott módon. A 3.10. ábra egy közönséges SSI lapka vázlatos rajzát mutatja be, amely négy NEM-ÉS kaput tartalmaz. Minden kapunak két bemenete és egy kimenete van, ez a négy kapuhoz összesen 12 lábat igényel. Ráadásul a lapkának még szüksége van áramra (V_{cc} feszültség) és földre (GND), amelyeken mind a négy kapu osztozik. Általában a tokozás tartalmaz egy bevágást az 1-es láb közelében, hogy jelezze a lapka orientációját. Hogy



3.10. ábra. *SSI lapka négy kapuval*

elkerüljük a zavart az áramköri diagramokon, az áramot, a földet és a nem használt kapukat szokás szerint nem jelöljük.

Hasonló lapkák már darabonként néhány centért megkaphatók. Minden SSI lapka egy „maréknyi" kaput és kb. 20 lábat tartalmaz. A 70-es években a számítógépeket nagyszámú ilyen lapkából építették fel, manapság egy teljes CPU és egy jelentős méretű (gyorsító) memória egy lapkába van „bemaratva".

Céljainkhoz föltehetjük, hogy minden kapu ideális abban az értelemben, hogy a kimenet azonnal előáll, amint a bemenetet alkalmaztuk. A valóságban a lapkáknak véges **kapukésleltetésük (gate delay)** van, amely tartalmazza mind a jel terjedését a lapkán keresztül, mind a kapcsolási időt. A tipikus késleltetés 1 és 10 ns között van.

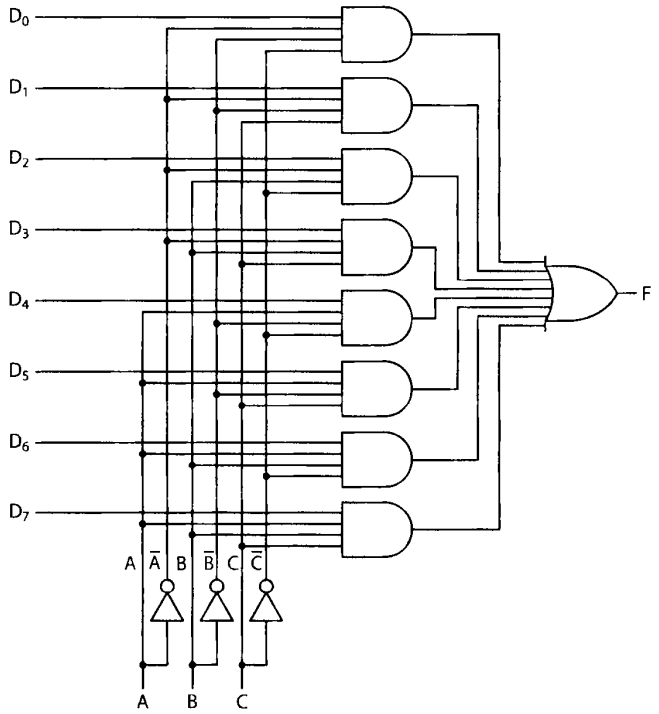
A jelenlegi technikai színvonal azt biztosítja, hogy hozzávetőlegesen 10 millió tranzisztort helyezzenek el egy lapkán. Mivel mindegyik áramkör felépíthető NEM-ÉS kapukból, azt gondolhatjuk, hogy egy gyártó olyan általános lapkát tud készíteni, amely 5 millió NEM-ÉS kapuból áll. Sajnálatos, hogy egy ilyen lapkához 15 000 002 lábra lenne szükség. Szabványos, 0,1 inch-es lábtávolsággal ez a lapka 18 km-nél is hosszabb lenne, ami hátrányos a piacon. Világos, hogy a technológiai fejlődés kihasználásának egyetlen módja, hogy nagy kapu/láb arányú áramköröket tervezünk. A következő szakaszokban olyan egyszerű MSI áramköröket vizsgálunk meg, amelyek sok kaput tartalmaznak, hasznos funkciókat biztosítanak, és csak korlátozott számú külső kapcsolattal (lábbal) rendelkeznek.

3.2.2. Kombinációs áramkörök

A digitális logika nagyon sok alkalmazása megkívánja, hogy egy áramkör többszörös bemenettel és többszörös kimenettel rendelkezzen, és a kimeneteit határozzák meg a pillanatnyi bemenetei. Az ilyen áramkört **kombinációs áramkör**nek **(combinational circuit)** hívjuk. Nem minden áramkörnek van meg ez a tulajdonsága. Például, memóriaelemeket tartalmazó áramkör tud olyan kimeneteket generálni, amelyek függnek a tárolt értékektől és természetesen a bemenő változóktól is. Egy áramkör, amely olyan igazságtáblát valósít meg, mint például a 3.3. (a) ábra táblája, a kombinációs áramkör tipikus példája. Ebben a szakaszban megvizsgálunk néhány gyakran használt kombinációs áramkört.

Multiplexerek

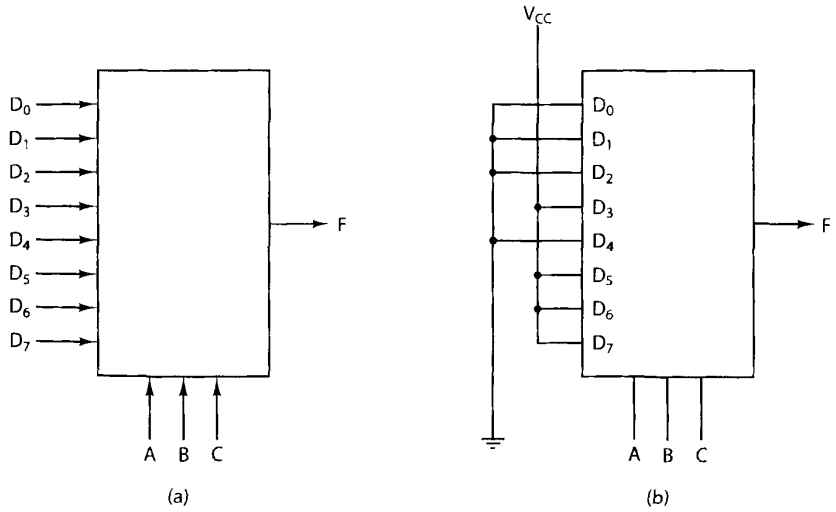
A digitális logika szintjén a **multiplexer** olyan áramkör, amely 2^n adatbemenettel, 1 adatkimenettel és n vezérlőbemenettel rendelkezik, mely utóbbiak egy adatbemenetet választanak ki. A kiválasztott adatbemenetre azt mondjuk, hogy a kime-



3.11. ábra. *Nyolcbemenetes multiplexer áramkör*

netre irányított vagy „kapuzott" (gated). A 3.11. ábra egy nyolcbemenetes multiplexer vázlatos ábráját tartalmazza. A három vezérlővonal az *A*, *B* és *C* egy 3 bites számot kódol, amely meghatározza, hogy a nyolc bemenő vonal közül melyiket kapuzzuk a VAGY kapura és ennek megfelelően a kimenetre. Bármilyen érték van a vezérlővonalakon, hét ÉS kapunak mindig 0 lesz a kimenete; a nyolcadik kapué pedig a kiválasztott bemeneti vonal értékétől függően vagy 0, vagy 1 lesz. A vezérlőbemenetek megfelelő kombinációjával bármelyik ÉS kapu kiválasztható. A multiplexer áramkör a 3.11. ábrán látható. Ha az áram- és a földlábakat is hozzávesszük, egy 14 lábas tokban helyezhető el.

Multiplexert használva megvalósíthatjuk a 3.3. (a) ábra többségi függvényét, ahogyan ezt a 3.12. (b) ábra mutatja. Az *A*, *B* és *C* mindegyik kombinációjára egy adatbemenet választódik ki. Mindegyik bemenet hozzá van kötve vagy a V_{cc}-hez (logikai 1), vagy a földhöz (logikai 0). A bemenetek huzalozási algoritmusa nagyon egyszerű: a D_i bemenet értéke megegyezik az igazságtábla *i*-ik sorának értékével. A 3.3. (a) ábra 0., 1., 2. és 4. sora egyenlő 0-val, így a megfelelő bemenetek földeltek, míg a maradék sorok 1-gyel egyenlők, így ezeket a logikai 1-re kell kötni. Ezen a módon bármelyik háromváltozós igazságtábla implementálható a 3.12. (a) ábrán ábrázolt lapkával.



3.12. ábra. *(a) MSI-multiplexer. (b) Ugyanaz a multiplexer úgy huzalozva, hogy többségi függvényt számoljon*

Már bemutattuk, hogyan használhatunk egy multiplexer lapkát arra, hogy néhány bemenetből kiválasszunk egyet, és hogyan kell megvalósítani egy igazságtáblát. Egy másik alkalmazás: párhuzamosból sorosba történő átalakítóként való használat. A 8 bites adatot elhelyezzük a bemeneti vonalakon, és ezután a vezérlővonalakkal sorban lépegetünk 000-tól 111-ig (binárisan), ezzel a 8 bitet egymás után

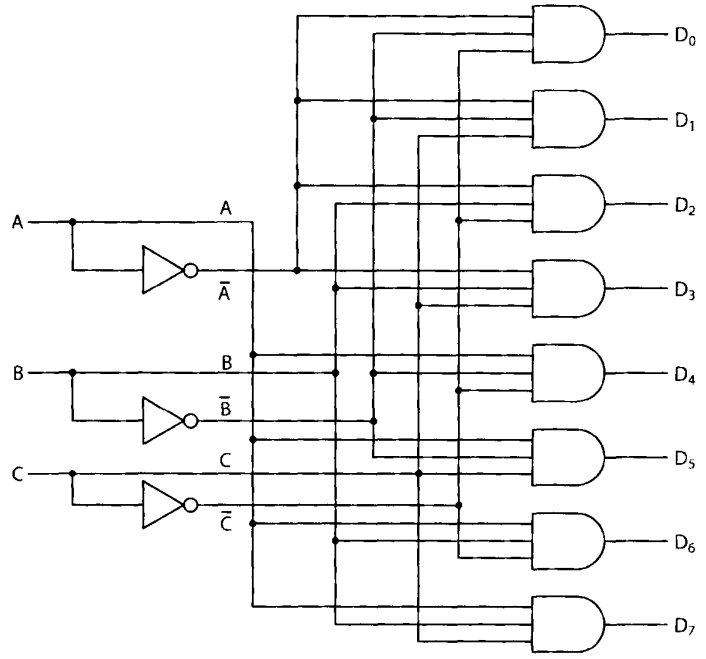
helyeztünk el az egyetlen kimenetre. A párhuzamos-soros konverzió tipikus felhasználása történik a klaviatúrában, ahol mindegyik billentyű leütése implicit módon 7 vagy 8 bites számot jelent, és ezt sorosan kell átküldeni egy telefonvonalon.

A multiplexer fordítottja a **demultiplexer**, amely egy egyedi bemenő jelet irányít a 2^n kimenet valamelyikére az *n* vezérlővonal értékétől függően. Ha a vezérlővonalak bináris értéke *k*, a *k*-adik kimenet a kiválasztott kimenet.

Dekódolók

Második példaként nézzünk meg most egy olyan áramkört, amely *n* bites számot használ bemenetként, és pontosan egyet kiválaszt a 2^n kimenet közül (1-re állítja). Az ilyen áramkört **dekódoló**nak nevezzük, ezt mutatja *n* = 3-ra a 3.13. ábra.

Ahhoz, hogy megnézzük, hogy a dekódoló hol lehet hasznos, képzeljünk el egy memóriát, amely nyolc lapkából (lapka-0, lapka-1, ..., lapka-7) áll, melyek mindegyike 1 MB-ot tartalmaz. A lapka-0-nak a címtartománya 0-tól 1 MB-ig, a lapka-1-nek 1 MB-tól 2 MB-ig és így tovább. Amikor egy címet akarunk a memóriában használni, a felső 3 bitet használjuk arra, hogy kiválasszuk a nyolc lapka egyikét. A 3.13. ábra áramkörét követve ez a 3 bit a három bemenet, *A*, *B* és *C*. A bemenetektől függően D_0, ..., D_7-ből pontosan egy kimenet egyenlő 1-gyel; az összes



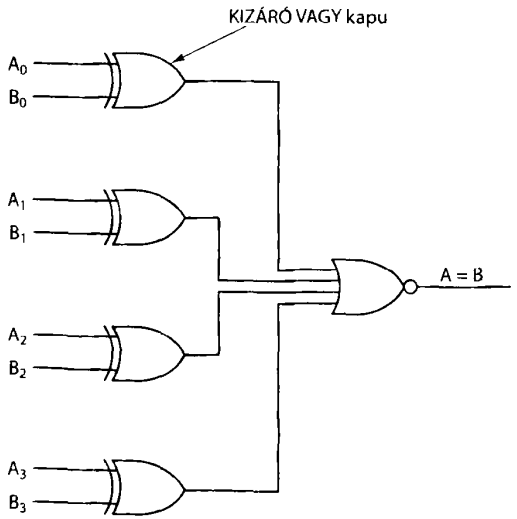
3.13. ábra. *3-ról 8-ra dekódoló áramkör*

több 0. Mindegyik kimenet a nyolc memórialapka valamelyikének a használatát engedélyezi. Mivel csak egy kimenet van 1-re állítva, így egyetlen lapka használata engedélyezett.

A 3.13. ábrán lévő áramkör működése nagyon egyszerű. Minden ÉS kapunak három bemenete van, amelyek közül az első: vagy A, vagy $\overline{A}$, a második: vagy B, vagy $\overline{B}$, a harmadik pedig: vagy C, vagy $\overline{C}$. Különböző bemenetkombinációkkal mindegyik kapu kiválasztható: D_0 az A, B, C-tal, a D_1 az A, $\overline{B}$, C-vel és így tovább.

Összehasonlítók

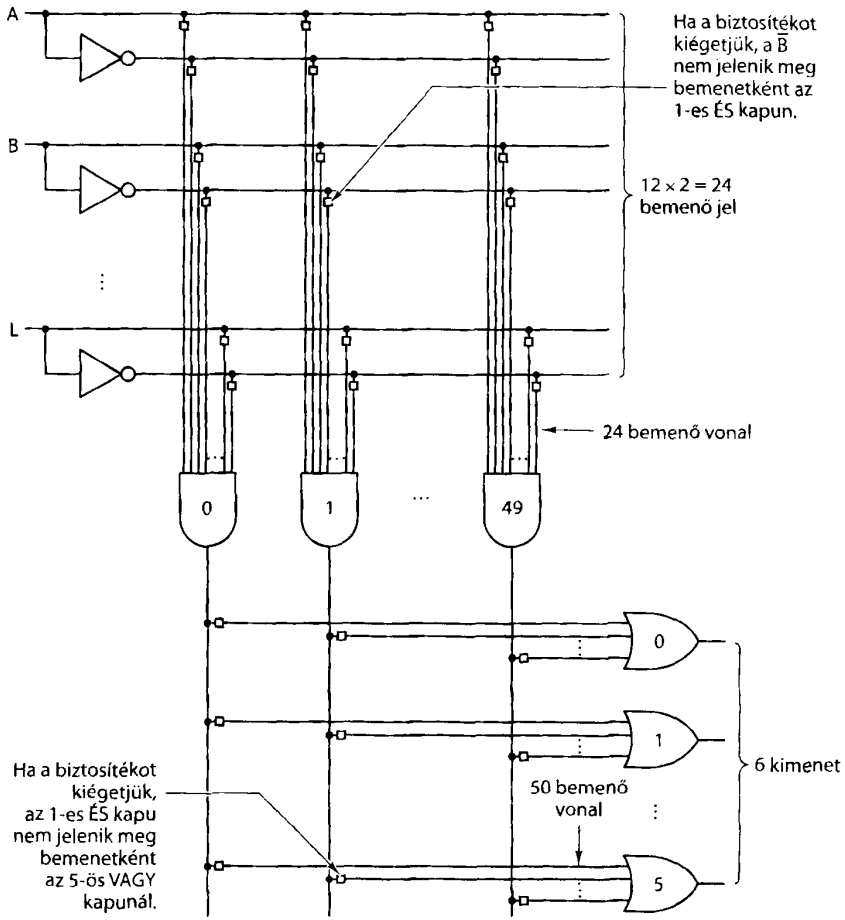
Egy másik hasznos áramkör az **összehasonlító (comparator)**, amely két bemeneti szót hasonlít össze. A 3.14. ábrán ábrázolt egyszerű összehasonlító két bemenettel dolgozik, A és B, mindegyikük 4 bit hosszúságú, és 1-et ad eredményül, ha a bemenetek azonosak, és 0-t, ha különbözők. Az áramkör a KIZÁRÓ-VAGY (EXCLUSIVE OR, XOR) kapun alapul, amely 0-t ad, ha a bemenetek egyenlők és 1-et, ha nem egyenlők. Ha a két bemeneti szó egyenlő, akkor mind a négy KIZÁRÓ-VAGY kapu 0-t kell adjon a kimenetén. Ezt a négy jelet VAGY művelettel össze tudjuk kapcsolni; ha az eredmény 0, a bemeneti szavak azonosak, különben nem. Példánkban NEM-VAGY kaput használtunk az utolsó áramköri szakaszban azért, hogy a teszt jelentése fordított legyen: 1 jelentse az egyenlőt, 0 pedig a nem egyenlőt.



3.14. ábra. *Egyszerű 4 bites összehasonlító*

Programozható logikai tömbök

Korábban láttuk, hogy tetszőleges függvények (igazságtáblázatok) megszerkeszthetők azáltal, hogy ÉS kapukkal logikai szorzatokat számolunk ki, és azután a szorzatokat VAGY-oljuk. Egy nagyon általános lapka, a **programozható logikai tömb** vagy **PLA (Programmable Logic Array)** szolgál a logikai szorzat-összeg képzésére, a 3.15. ábra egy egyszerű példát mutat be. Ennek a lapkának 12 változó számára van bemeneti vonala. Mindegyik bemenetnek a komplemense a lapkán belül kép-



3.15. ábra. *12 bemenetű, 6 kimenetű programozható logikai tömb. A kis négyzetek a biztosítékokat jelzik, amelyek kiégetésével meghatározható a végrehajtandó funkció. Az olvadó biztosítékokat két mátrixban helyezik el: a felső az ÉS kapuk, az alsó pedig a VAGY kapuk számára szolgál*

ződik, így végül 24 bemeneti jelet kapunk. Az áramkör szíve egy 50 és kapuból álló tömb, ezek mindegyikének a bemenete a 24 bemenő jel bármelyik részhalmaza lehet. Egy 24 × 50 bites mátrix határozza meg, hogy melyik bemenő jel melyik és kapura kapcsolódik. Ezt a mátrixot a felhasználó állítja be. Az 50 és kapu minden bemeneti vonala egy olvadó biztosítékot tartalmaz. Amikor a gyárból elküldik a PLA-t, mind az 1200 olvadó biztosíték sértetlen. A mátrix programozása során a felhasználó nagy árammal a lapkában kiégeti a kiválasztott biztosítékokat.

Az áramkör kimeneti része hat VAGY kapuból áll, mindegyik 50 bemenő jellel rendelkezik, amelyek megegyeznek az ÉS kapuk 50 kimenetével. Itt szintén egy felhasználó által meghatározott (50 × 6-os) mátrix mondja meg, hogy ténylegesen melyik kapcsolat létezzen. A lapkának 12 bemenő lába, 6 kimenő lába, tápfeszültség és föld kapcsolata van, ez összesen 20 láb.

A PLA használatára példaként tekintsük meg a 3.3. (b) ábrát újra. Három bemenetből, négy ÉS kapuból, egy VAGY kapuból és három inverterből áll. A megfelelő belső kapcsolatokkal a PLA ki tudja számolni ugyanezt a függvényt úgy, hogy a 12 bemenő jelből hármat használunk, az 50 ÉS kapuból négyet, a 6 VAGY kapuból pedig egyet. (A négy ÉS kapunak sorrendben az $A\bar{B}C$-t, $AB\bar{C}$-t, $A\bar{B}\bar{C}$-t és $\bar{A}\bar{B}\bar{C}$-t kell kiszámolnia; a VAGY kapu ennek a négy szorzatkifejezésnek az értékét használja bemenő jelként.) Valójában egyetlen PLA beprogramozható úgy, hogy szimultán módon összesen négy hasonló bonyolultságú függvényt értékeljen ki. Ezekre az egyszerű függvényekre a bemenő változók száma a korlátozó tényező; a bonyolultabb esetekben maguk az ÉS vagy VAGY kapuk jelenthetnek korlátot.

Bár a mező-programozású (field-programmable) PLA-t, amelyet az előbb mutattunk be, még használják, de nagyon sok alkalmazásnál a felhasználó által meghatározott (custom-made) PLA-k a használatosabbak. Ezek nagy részét a felhasználó tervezi meg, és a gyártó legyártja a felhasználó specifikációja alapján. Az ilyen PLA-k olcsóbbak, mint a mező-programozásúak.

Most összehasonlítjuk a három különböző módot, amelyet a 3.3. (a) ábrán lévő igazságtáblázat megvalósítására tárgyaltunk. Ha SSI-komponenseket használunk, négy lapkára van szükségünk. Alternatívaként tudjuk helyettesíteni egy MSI-multiplexer lapkával, ahogy ezt a 3.12. (b) ábra mutatja. Végül tudunk használni egy negyed PLA lapkát. A PLA nyilvánvalóan hatékonyabb, mint az előző két módszer, ha sok függvény szükséges. Az egyszerű áramkörök megvalósítására az olcsó SSI és MSI lapkák lehetnek előnyösek.

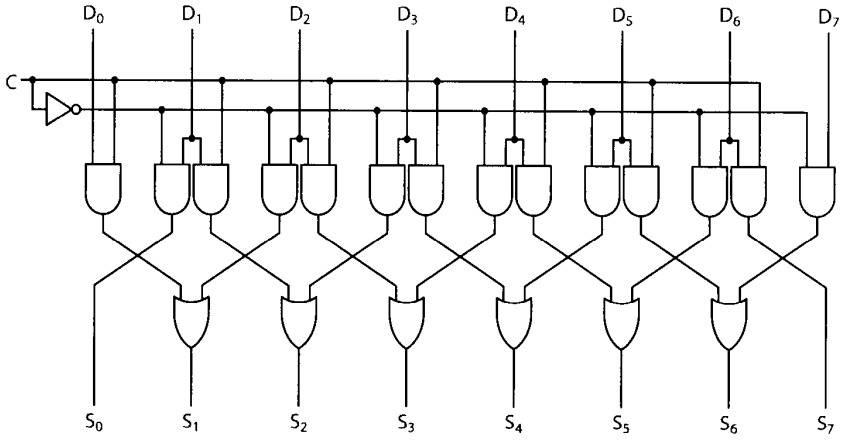
3.2.3. Aritmetikai áramkörök

Itt az ideje, hogy az előbb vizsgált általános célú MSI-áramköröktől, elmozduljunk az MSI kombinációs áramkörök felé, amelyek az aritmetikai műveleteket végzik. Egy 8 bites egyszerű léptetővel kezdünk, aztán megmutatjuk, hogy lehet összeadót szerkeszteni, és végül megvizsgáljuk az aritmetikai-logikai egységet, amely központi szerepet játszik minden számítógépben.

Léptető

Az első aritmetikai MSI áramkörünk 8 bemenettel, 8 kimenettel rendelkező léptető (shifter) (lásd 3.16. ábra). A nyolc bemenő bitet a D_0, ..., D_7-es vonal jelzi. A kimenetek, amelyek pontosan a bemenő jelek 1 bittel való eltolását jelentik, az S_0, ..., S_7 vonalakon érhetők el. A C vezérlővonal értéke meghatározza a léptetés irányát: 0 esetén balra, 1 esetén jobbra.

Nézzük, hogyan működik az áramkör, figyeljük meg, hogy a szélsők kivételével az összes bithez ÉS kapupárok kapcsolódnak. Amikor $C = 1$, minden kapupár jobb oldali tagja tudja a megfelelő bemenő bitet a kimenetre küldeni. Mivel a jobb oldali ÉS kapu kimenete össze van kötve a tőle jobbra eső VAGY kapu bemenetével, jobbra léptetés hajtódik végre. Amikor $C = 0$, az ÉS kapupár bal oldali tagja továbbítja a jelet, és balra léptetés történik.



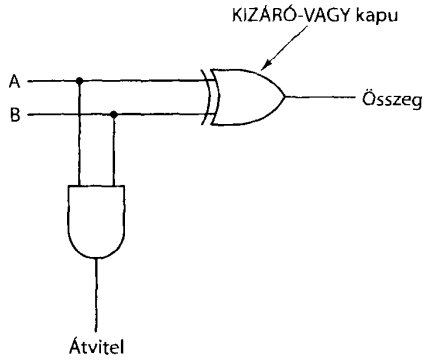
3.16. ábra. *1 bittel balra/jobbra léptető*

Összeadók

Majdnem elképzelhetetlen egy olyan számítógép, amely nem tud egész számokat összeadni. Következésképpen nagyon lényeges része minden CPU-nak (központi egységnek) egy összeadást végrehajtó áramkör. Az 1 bites egészek összeadásának igazságtáblázatát láthatjuk a 3.17. (a) ábrán. Két kimenet van feltüntetve: az A és B bemenő jelek összege, valamint az átvitel (carry) a következő (balra lévő) pozícióba. Az összeget és az átvitelt kiszámító áramkör a 3.17. (b) ábrán látható. Ez az egyszerű áramkör széles körben **fél összeadóként (half adder)** ismert.

Bár egy fél összeadó megfelelő két több-bites bemenő szó alsó bitjeinek összeadására, de nem jól működik a szavak középső bitpozícióin, mert nem kezeli a jobbról érkező átvitelt. Helyette a 3.18. ábrán ábrázolt **teljes összeadó (full adder)**

A	B	Összeg	Átvitel
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



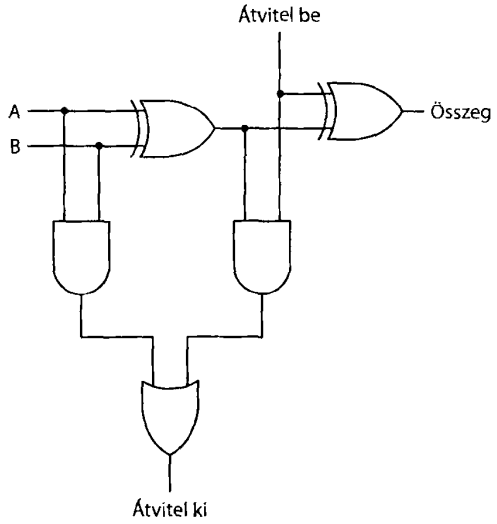
3.17. ábra. *(a) 1 bites összeadás igazságtáblája. (b) Fél összeadó áramkör*

szükséges. Az áramkör ábrájából világosan látszik, hogy egy teljes összeadó két fél összeadóból épül fel. Az *Összeg* kimenő vonal 1, ha az *A*, *B* és *Átvitel be* bemeneteken páratlan az 1-esek száma. Az *Átvitel ki* akkor 1, ha *A* és *B* is 1 (a VAGY kapu bal bemenete), vagy pontosan az egyikük 1-es, és az *Átvitel be* szintén 1. A két fél összeadó együtt számolja ki az összeg- és az átvitelbiteket.

Ahhoz, hogy mondjuk két 16 bites szó összeadásához összeadót építsünk, 16-szor meg kell ismételnünk a 3.18. (b) ábrán látható áramkört. Az *Átvitel ki* bitet a

A	B	Átvitel be	Összeg	Átvitel ki
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

3.18. ábra. *(a) A teljes összeadó igazságtáblázata. (b) Teljes összeadó áramkör*

bal oldali szomszédnál az *Átvitel be* ághoz kötjük. A jobb szélső bitnél az *Átvitel be* bitet 0-nak vesszük. Az ilyen típusú összeadót **átvitelt tovább terjesztő összeadó**nak (**ripple carry adder**) nevezzük. Mivel a legrosszabb esetben, ha 1-et hozzáadunk az 111...111 bináris számhoz, az összeadás mindaddig nem lesz kész, míg az átvitel nem halad végig az egész bitsoron a bal szélső bittől a jobb szélső bitig. Léteznek olyan összeadók, amelyeknek nincs meg ez a késleltetésc, emiatt gyorsabbak, és ezeket általában előnyben részesítik.

Az ilyen gyors összeadó szemléltetéséhez bontsunk két részre egy 32 bites összeadót: egy 16 bites alsó és 16 bites felső részre. Amikor az összeadás elkezdődik, a felső összeadó még nem tud dolgozni, mert nem tudja, hogy mi lesz az *Átvitel be* a felső 16 bit összeadása után.

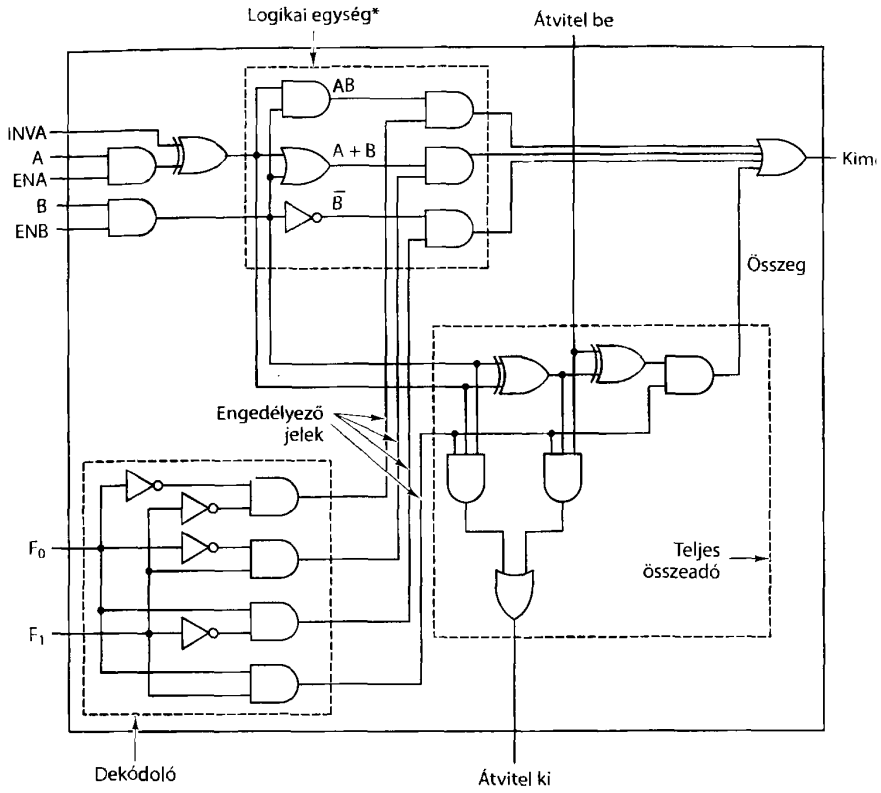
Tekintsük a következő módosítást. Ahelyett hogy a felső rész összeadásához egyetlen összeadót építenénk, két párhuzamosan működő összeadót építünk, megduplázzuk a felső rész hardverét. Így az áramkör most három 16 bites összeadóból áll: az alsó részből és az *U0*, *U1* két felső részből, ezek párhuzamosan működnek. *Átvitel be* értékeként 0-t töltünk *U0*-ba és 1-et *U1*-be. Most mind a kettőt el tudjuk indítani ugyanakkor, amikor az alsó fél is indult, de csak az egyik végeredménye lesz helyes. Amikor a 16 bites összeadás már megtörtént, ismert az *Átvitel be* a felső fél számára, akkor a megfelelő felső felet ki tudjuk választani. Ez a trükk felére redukálja az összeadási időt. Az ilyen összeadót **átvitelkiválasztó összeadó**nak (**carry select adder**) nevezzük. Ezt a trükköt ismételhetjük úgy, hogy mindegyik 16 bites összeadót 8 bites összeadók duplikálásával építjük fel és így tovább.

Aritmetikai-logikai egységek

A legtöbb számítógép egyetlen áramkört tartalmaz az ÉS, VAGY végrehajtására és két gépi szó összeadására. Ez az áramkör tipikusan *n* bites szavakra készül, és *n* azonos áramkört tartalmaz az egyes bit pozíciókra. A 3.19. ábra egy ilyen áramkört mutat be, amelyet **aritmetikai-logikai egység**nek (**Arithmetic Logic Unit**) vagy **ALU**-nak nevezünk. Ez az áramkör négy funkció bármelyikét végre tudja hajtani – nevezetesen, attól függően, hogy a funkciót kiválasztó F_0 és F_1 bemenő vonalak bináris értéke 00, 01, 10 vagy 11. A négy funkció: *A* és *B*, *A* VAGY *B*, *B* és $A + B$. Megjegyezzük, hogy $A + B$ egy aritmetikai összeadást, és nem Boole ÉS-t jelent.

A 3.19. ábra bal alsó sarkában egy 2 bites dekódoló van, hogy a négy művelet számára létrehozza az F_0 és F_1 vezérlőjeleken alapuló engedélyező jeleket. F_0 és F_1 értékétől függően a négy engedélyező vonal közül pontosan egy kerül kiválasztásra. Ez a vonalkiválasztás teszi lehetővé, hogy a kiválasztott funkció eredménye áthaladjon az utolsó VAGY kapun a kimenetre.

A bal felső sarokban az a logika szerepel, amely végrehajtja az *A* AND *B*, *A* OR *B* és $\bar{B}$ műveleteket, de ezek közül legfeljebb egy eredménye haladhat át az utolsó VAGY kapun, attól függően, hogy a dekódolóból kijövő engedélyező vonalak melyiket engedélyezik. Mivel a dekódolónak pontosan egy kimenete lesz 1, ezért a VAGY kapu bemenetét képező négy ÉS kapuból csak egy fog működni, a másik három ÉS kapu kimenete 0 lesz, függetlenül attól, hogy mi *A* és *B* értéke.

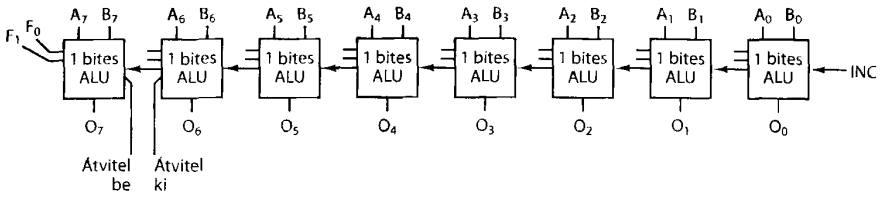


3.19. ábra. *1 bites ALU*

Azonkívül, hogy A-t és B-t logikai vagy aritmetikai műveletek bemeneteként használhatjuk, az is lehetséges, hogy bármelyiket 0-ba kényszerítsük az ENA vagy ENB negálásával. $\overline{A}$ beállítása is lehetséges INVA segítségével. A 4. fejezetben látni fogjuk INVA, ENA és ENB használatát. Normális feltételek mellett ENA és ENB is 1, hogy engedélyezze a bemeneteket, INVA pedig 0. Ebben az esetben A és B módosítás nélkül kerül az ALU-ba.

A 3.19. ábra a jobb alsó sarokban tartalmazza a teljes összeadót, amely A és B összegét számolja ki, beleértve az átvitel kezelését is, a valóságban több ilyen áramkör van összekötve, hogy teljes szóhosszúságú műveleteket lehessen végrehajtani. A 3.19. ábrán láthatóhoz hasonló áramkörök ténylegesen léteznek, és **bitszelet (bit slices)** néven ismertek. Ezek lehetővé teszik, hogy a számítógép-tervezők bármilyen kívánt bitszélességű ALU-t építsenek. A 3.20. ábra egy 8 bites

* Az ábrán a + jel a (VAGY) logikai összeadást jelenti. *(A lektor)*



3.20. ábra. *Nyolc 1 bites ALU-szelet összekapcsolása 8 bites ALU-vá. Az engedélyező és invertáló jeleket az egyszerűség kedvéért nem ábrázoljuk*

ALU-t mutat be, amely 8 darab 1 bites ALU-szeletből épül fel. Az INC bemenet csak az összeadásnál jut szerephez. INC jel hatására 1-gyel növekszik az eredmény, és így ki tudjuk számolni A + 1-et és $A + B$ + 1-et.

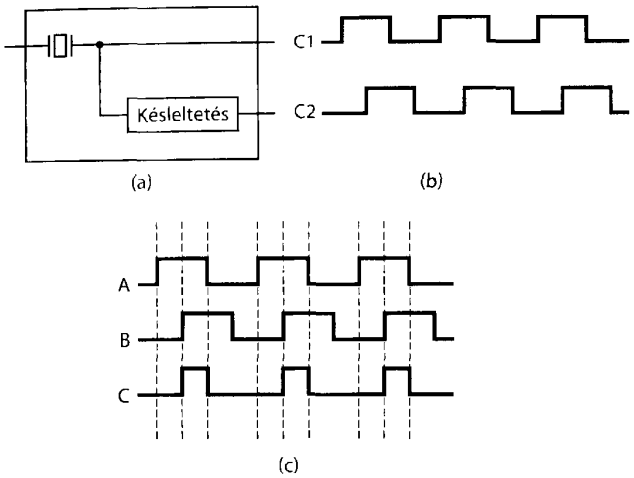
3.2.4. Órák

Sok digitális áramkörben az események, történésének sorrendje nagyon kritikus. Néha egy eseménynek meg kell előznie egy másikat, néha két eseménynek kell egy időben történnie. Annak érdekében, hogy a tervezők el tudják érni a kívánt időzítéseket, nagyon sok digitális áramkör órát használ a szinkronizáció biztosítására. Az **óra (clock)** ebben az értelemben egy áramkör, amely pontosan meghatározott szélességű pulzusok sorozatát bocsátja ki, és nagyon precízen meghatározott a két egymás utáni pulzus közötti időintervallum is. A két egymást követő pulzus élei közötti időintervallumot az óra **ciklusidejének (clock cycle time)** nevezzük. A pulzus frekvenciája általában 1 és 500 MHz között van, ennek megfelelően az órajel 1000 ns-tól 2 ns-ig terjedhet. A nagy pontosság eléréséhez az óra frekvenciáját általában egy kristályoszcillátor vezérli.

Egy számítógépben több esemény történhet egyetlen órajel alatt. Ha ezeknek az eseményeknek egy speciális sorrendben kell bekövetkezniük, az órajelet alciklusokra kell osztanunk. Egy finomabb felbontás általános megoldása, hogy az alap órajelet megcsapoljuk, és beszúrunk egy ismert késleltetésű áramkört, így egy másodlagos órajelet készítünk, amely fáziseltolással keletkezik az elsődlegesből, ahogy a 3.21. (a) ábra is mutatja. A 3.21. (b) ábrán lévő időzítési diagram négy időzítési referenciát biztosít a diszkrét események számára:

1. C1 felfutó éle;
2. C1 lefutó éle;
3. C2 felfutó éle;
4. C2 lefutó éle.

Különböző eseményeknek a különböző élekhez való hozzárendelésével tulajdonképpen elérjük a kívánt sorrendet. Ha több mint négy időzítési referencia szüksé-



3.21. ábra. *(a) Óra. (b) Időzítési diagram. (c) Aszimmetrikus óra generálása*

ges egy adott időzítési ciklusban, akkor több különböző késleltetésű másodlagos vonalat kell csatlakoztatnunk az elsődlegeshez.

Néhány áramkörben az időzítési intervallum az érdekesebb a diszkrét időpillanatoknál. Például, néhány esemény akkor történhet meg, amikor a C1 időzítő magas, nem pontosan a felfutó élnél. Egy másik esemény csak akkor történhet meg, amikor C2 magas. Ha több mint két intervallum szükséges, akkor több órajelet kell biztosítani, vagy a két órajel magas állapotának részlegesen át kell lapolnia egymást. Az utóbbi esetben négy különböző intervallumot különböztethetünk meg: C1 és C2, C1 és C2, C1 és C2 és C1 és C2.

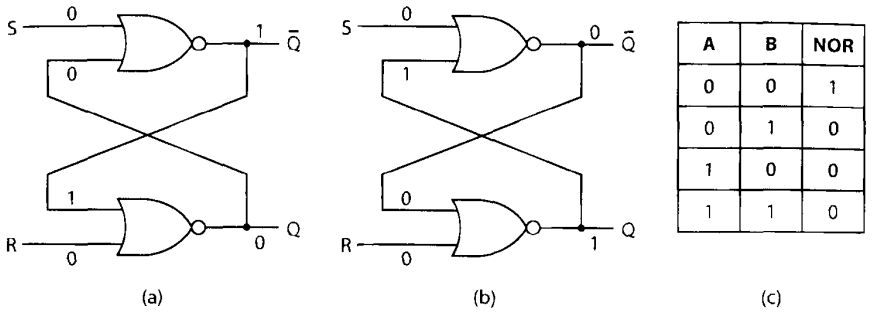
Mellesleg az órák szimmetrikusak, a magas állapotban eltöltött idő megegyezik az alacsony állapotban eltöltött idővel, ahogy azt a 3.21. (b) ábra mutatja. Aszimmetrikus pulzuslánc létrehozásához az alapórát egy késleltető áramkörrel lehet eltolni, és ÉS-elni az eredeti jellel, ahogy a 3.21. (c) ábrán látható a C sorban.

3.3. Memória

Minden számítógép lényeges eleme a memória. Ismereteink szerint memória nélkül már nincs számítógép. A memóriát használjuk mind a végrehajtandó utasítások, mind az adatok tárolására. A következő szakaszokban megvizsgáljuk a memória alapelemeit a kapuszinttől kezdve, hogy lássuk, hogyan működnek és hogyan kombinálhatók össze nagyméretű memóriák kialakításához.

3.3.1. Tárolók

Az 1 bites memória készítéséhez szükségünk van olyan áramkörre, amelyik valahogyan „visszaemlékszik" az előző bemeneti értékekre. A 3.22. (a) ábrán bemutatott módon készíthetünk ilyen áramkört két NEM-VAGY kapuból. Hasonló áramköröket építhetünk NEM-ÉS kapukból. A továbbiakban nem fogjuk ez utóbbiakat emlegetni, mert koncepcionálisan azonosak a NEM-VAGY verziókkal.



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

3.22. ábra. *(a) NEM-VAGY tároló 0-s állapotban. (b) NEM-VAGY tároló 1-es állapotban. (c) NEM-VAGY igazságtáblázata*

A 3.22. (a) ábra áramkörét **SR-tárolónak** (**Set Reset latch**) hívjuk. Két bemenete van: S (set) a tároló beállítására, R (reset) pedig a törlésére szolgál. Két kimenete van: a Q és $\bar{Q}$, amelyek egymás fordítottjai. A kombinációs áramkörökkel ellentétben a tároló kimenetei nem csupán az aktuális bemenetektől függnek.

Lássuk, hogyan is történik mindez. Tételezzük fel, hogy mind az S, mind az R 0 az idő nagy részében. Az érvelés kedvéért tegyük fel a továbbiakban, hogy $Q = 0$. Mivel Q vissza van vezetve a felső NEM-VAGY kapuba, annak mindkét bemenete 0, így a kimenet: $\bar{Q} = 1$. Az 1 érték vissza van vezetve az alsó kapuba, amelynek a bemenetei 1 és 0, ebből $Q = 0$ következik. Ez a 3.22. (a) ábrán látható állapot tehát állandó, stabil.

Most képzeljük el, hogy Q nem 0, hanem 1, R és S még mindig 0. A felső kapunak a két bemenete 0 és 1, a kimenete $\bar{Q} = 0$, amit visszavezetünk az alsó kapuba. Ezt az állapotot a 3.22. (b) ábra mutatja, ez szintén stabil. Az az állapot, amelynél mindkét kimenet egyenlő 0-val, nem stabil, mert akkor mindkét kapu mindkét bemenete 0, ezért mindkét kapu kimenete 1-re változik. Hasonlóan lehetetlen, hogy mindkét kimenet 1 legyen, mert 0 és 1 bemenetek hatására a kimenet 0-ra változik. Megállapításunk egyszerű: az $R = S = 0$ esetben a tárolónak két stabil állapota van, amelyeket mi 0-snak és 1-esnek nevezünk Q-tól függően.

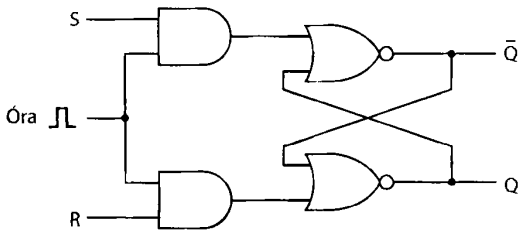
Most vizsgáljuk meg a bemenetek hatását a tároló állapotára. Tegyük fel, hogy $S = 1$, és $Q = 0$. A bemenetek a felső kapunál így 1 és 0, és a Q-t kimenetet 0-ba állítja. Ez a változás az alsó kapu mindkét bemenetén 0-t, és a kimeneten 1-et eredményez. Tehát S beállításával (azaz 1-et adunk neki értékül) a tároló állapotát

átkapcsolja a 0-s állapotból az 1-es állapotba. Az R bemenet 1-re való beállításának nincs hatása, amikor a tároló a 0-s állapotban van, mert az alsó NEM-VAGY kapu kimenete 0 a 10-s és az 11-es bemenetekre.

Hasonló meggondolással könnyű látni, hogy S 1-re állításának nincs hatása, amikor az állapot $Q = 1$, de R beállítása a tárolót $Q = 0$ állapotba viszi. Összefoglalásképpen, amikor S 1-re van állítva, akkor a tároló a $Q = 1$ állapotba kerül függetlenül attól, hogy milyen állapotban volt előzőleg. Hasonlóan, ha az R-t állítjuk 1-re, akkor a tároló $Q = 0$-ba kerül. Az áramkör „emlékszik", hogy S vagy R jelet kapott utoljára. Ezt a tulajdonságát felhasználva tudunk számítógép-memóriákat építeni.

Időzített SR-tároló

Gyakran fontos, hogy a tároló állapotváltozásai csak bizonyos meghatározott pillanatban történjenek. E cél eléréséhez kicsit módosítjuk az áramkört, ahogy a 3.23. ábrán látható, így megkapjuk az **időzített SR-tárolót (clocked SR latch)**.



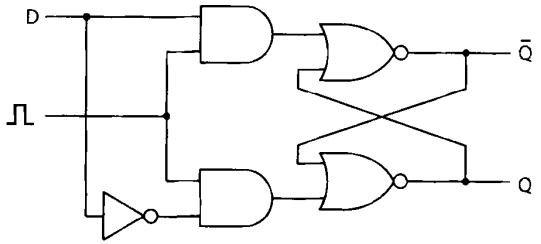
3.23. ábra. *Időzített SR-tároló*

Ez az áramkör egy további bemenettel rendelkezik, az órajellel, amely alaphelyzetben 0. A 0-s órajel hatására az ÉS kapu kimenete 0, függetlenül S-től és R-től, tehát a tároló nem változtatja meg az állapotát. Amikor az órajel 1, az ÉS kapuk hatása megszűnik, és a tároló érzékeny lesz S-re és R-re. A nevével ellentétben, az órajelet nem szükséges órával vezérelni. Az **érvényes (enable)** és **kapuzójel (strobe)** kifejezések széles körben használtak, ami azt jelenti, hogy ha az órabemeneten a jel 1, az áramkör érzékeny az S és R állapotokra.

Eddig gondosan a szőnyeg alá sepertük azt a problémát, hogy mi történik, amikor mind az S, mind az R 1. Jó okunk volt rá: az áramkör kimenete determinálatlanná válik, ha végül mind az R, mind az S 0-ba megy át. $S = R = 1$ bemenet hatására az egyetlen stabil állapot a $Q = \bar{Q} = 0$, de amint mindkét bemeneti jel 0 lesz, a tároló átugrik a két stabil állapot közül az egyikbe. Ha valamelyik bemenet hamarabb ér 0-ba, mint a másik, a lassabb nyer, mert a lassabb bemenet még 1, és az ennek megfelelő állapot áll be. Ha mindkét bemenet egyszerre tér vissza 0-ba (ami nagyon valószínűtlen), a tároló véletlenszerűen valamelyik stabil állapotába ugrik.

Időzített D-tárolók

Jó módszer az SR tároló $S = R = 1$ által okozott bizonytalanságának feloldására, hogy ennek az előfordulását megakadályozzuk. A 3.24. ábra egy egyetlen D bemenettel rendelkező tároló áramkört mutat. Mivel az alsó és kapu bemenete mindig komplemense a felső kapu bemenetének, a két 1-es bemenet problémája nem fordul elő. Amikor $D = 1$ és az órajel 1, a tároló a $Q = 1$ állapotba kerül. Amikor $D = 0$ és az órajel 1, a $Q = 0$ állapotba kerül. Más szavakkal, amikor az órajel 1, a D pillanatnyi értéke mintának tekinthető, és ezt tároljuk a tárolóban. Ezt az áramkört **időzített D-tárolónak (clocked D latch)** hívjuk, amely egy igazi 1 bites memória. A tárolt érték Q-ban mindig elérhető. A D aktuális értékének a memóriába töltéséhez egy pozitív pulzust kell adni az órajelbemenetre.



3.24. ábra. *Időzített D-tároló*

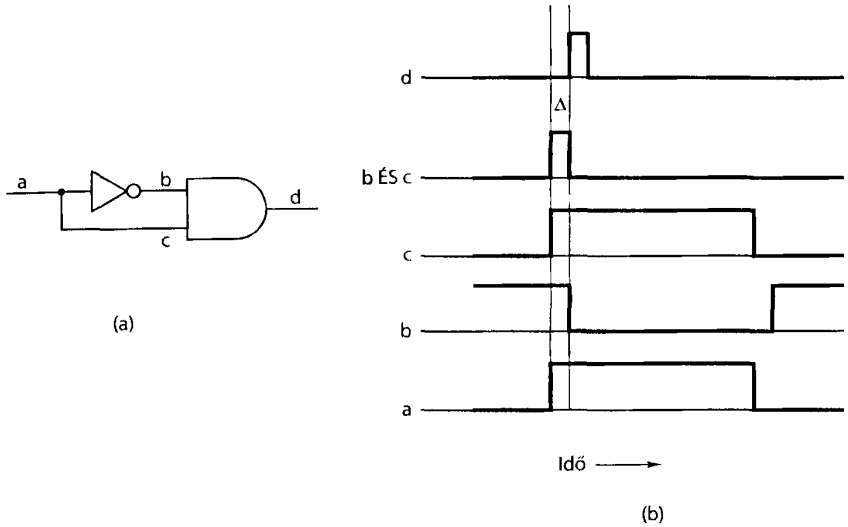
Ez az áramkör 11 tranzisztort igényel. Kicsit mesterkélt (de kevésbé nyilvánvaló) áramkörök hat tranzisztorral is tudnak tárolni 1 bitet. A gyakorlatban általában az ilyen áramköröket használják.

3.3.2. Flip-flopok

Sok áramkörnél szükséges lehet, hogy egy meghatározott időpontban vegyen mintát bizonyos vonalon levő értékről, és tárolja azt az értéket. Ezt a változatot **flip-flopnak (flip-flop, billenőkör)** nevezzük. Flip-flop esetén az órajel 1-es állásánál nem fordul elő állapotváltozás, hanem csak akkor, amikor az órajel átmegy 0-ból 1-be (felfutó él) vagy 1-ből a 0-ba (lefutó él). Így az órajel hossza nem lényeges, ha elég gyors az átmenet.

A nyomaték kedvéért megismételjük a flip-flop és a tároló közti különbséget. A flip-flop **élvezérelt (edge triggered)**, míg a tároló **szintvezérelt (level triggered)**. Legyünk óvatosak, mert az irodalomban ezeket a definíciókat gyakran zavarosan használják. Sok szerző „flip-flop"-ot használ, amikor tárolóra hivatkozik, és fordítva.

A flip-flop tervezésének különböző megközelítései vannak. Például, ha valamilyen módon létrehozunk egy nagyon rövid impulzust az órajel felmenő élénél, akkor ezt az impulzust betáplálhatjuk egy D-tárolóba. Ez egy ténylegesen alkalmazott megoldás, a 3.25. (a) ábrán egy ilyen áramkört láthatunk.

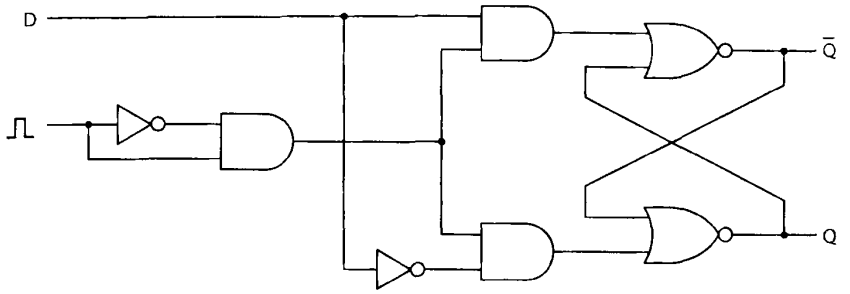


3.25. ábra. *(a) Pulzusgenerátor. (b) Az áramkör négy pontjának idődiagramja*

Első pillanatban úgy tűnhet, hogy az ÉS kapu kimenete mindig 0, mivel bármely jelnek 0 az ÉS kapcsolata a saját ellentettjével, de a helyzet ennél kicsit bonyolultabb. Az inverternek van egy pici, de nem nulla késleltetési ideje, és ez a késleltetés az, amely az áramkört megdolgoztatja. Tegyük fel, hogy négy ponton mérjük a feszültséget, legyen ez *a*, *b*, *c* és *d*. A bemenő jel, amit az *a*-n mérünk, egy hosszú órajelimpulzus, amint azt a 3.25. (b) ábra mutatja az alsó sorban. A *b* jel fölötte van. Megjegyezzük, hogy ez a jel invertált, és egy kicsit késleltetett, a késleltetés tipikusan néhány ns attól függően, hogy milyen típusú invertert használunk.

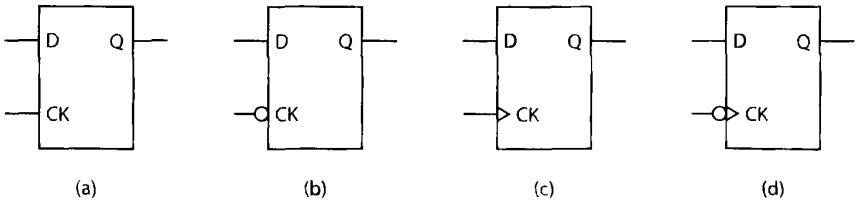
A *c* jel szintén késleltetve van, de csak a jel terjedési idejével (ez a fény terjedési sebessége). Ha a fizikai távolság az *a* és *c* között például 20 mikron, a késleltetési idő 0,0001 ns, ami elhagyható az inverteren való áthaladás késleltetéséhez képest. Így minden szándék és cél tekintetében a *c*-nél levő jel ugyanolyan jó, mint az *a*-nál levő eredeti jel.

Amikor az ÉS kapuban a *b* és *c* bemeneteket egybe ÉS-eljük, az eredmény egy rövid impulzuslökés, amint azt a 3.25. (b) ábra mutatja, ahol az impulzusnak a szélessége Δ, ami megegyezik az inverter kapu késleltetésével, tipikusan 5 ns vagy kisebb. Az ÉS kapu kimenete éppen ilyen impulzus, a kapu késleltetésével eltolva, ahogy a 3.25. (b) ábra tetején látható. Ez az eltolási idő pontosan azt jelenti, hogy a D-tároló az órajel felemelkedő éle után egy fix késleltetéssel később aktiválódik, de ez nincs hatással a pulzus szélességére. Egy 50 ns-os ciklusidővel rendelkező memóriában egy 5 ns széles mintavételező impulzus elegendően rövid lehet. Ebben az esetben egy teljes áramkör olyan lehet, mint a 3.26. ábrán látható. Érdemes megjegyeznünk, hogy ez a flip-flop terv szép, mert könnyű megérteni, de a gyakorlatban gyakran sokkal bonyolultabb flip-flopok használatosak.



3.26. ábra. *D-flip-flop*

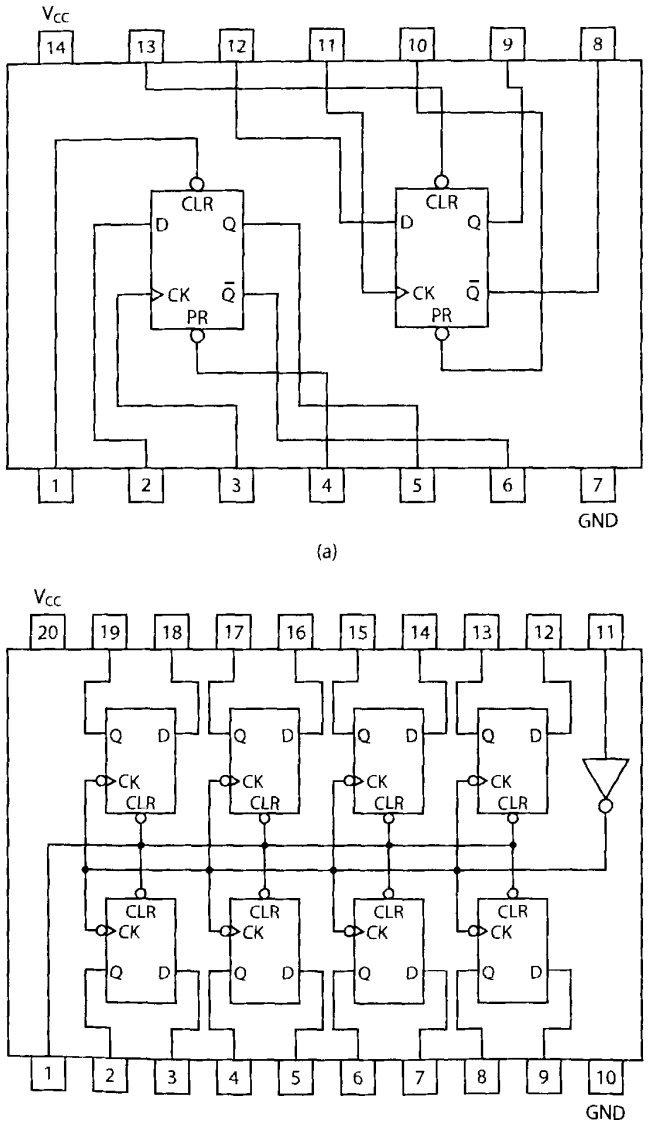
A flip-flopok és a tárolók szabványos jelöléseit a 3.27. ábra mutatja. A 3.27. (a) ábrán egy tároló jelölése látható, amely akkor tölti be *D* állapotát, amikor a *CK* órajel 1, ellentétben a 3.27. (b) ábrával, ahol a tároló órajele alapesetben 1, de *D* állapotának betöltéséhez 0-ra vált egy rövid időre. A 3.27. (c) és (d) ábrák flip-flopok és nem tárolók, amelyeket egy nyíl jellel jelölünk az óra bemenetén. A 3.27. (c) ábrán a beírás az órajel fölfutó (0-ból 1-be menő) élénél, amíg a 3.27. (d) ábrán a lefutó (1-ből 0-ba menő) élénél történik. Sok, de nem mindegyik tárolónak és flip-flopnak van $\overline{Q}$ kimenete, és néhánynak két további bemenete van, a *Beállító (Set)* vagy *Előre beállító (Preset)* (a $Q = 1$ állapotot állítja be) és a *Törlő (Reset* vagy *Clear)* (a $Q = 0$ állapotot állítja be).



3.27. ábra. *D-tárolók és -flip-flopok*

3.3.3. Regiszterek

A flip-flopok különböző konfigurációkban állnak rendelkezésre. Egy egyszerű lapka két független D-flip-flopot tartalmaz, a *Törlő* (CLR) és az *Előre beállító* (PR) jelekkel, ilyet mutat be a 3.28. (a) ábra is. Bár egybe van tokozva egy 14 lábú lapkában, a két flip-flop nincs kapcsolatban egymással. Ettől teljesen különböző elrendezés látható a 3.28. (b) ábrán, egy nyolcas flip-flop. Ebben nyolc D-flip-flop van (ezért használjuk a nyolcas kifejezést), ahol nemcsak a $\overline{Q}$ és az *Törlő* vonal hiányzik, hanem az órajelek is csoportosítva vannak, és a 11-es láb adja az órajelet. Maguk a flip-flopok megegyeznek a 3.27. (d) ábrán bemutatott típussal, de a flip-



3.28. ábra. *(a) Duális D-flip-flop. (b) Nyolcas flip-flop*

flopok (CK-nál lévő) inverziós gömbjeit hatástalanítja a 11-es láb invertere, így a flip-flopok a felfutó élnél tárolnak. A nyolc törlőjelet szintén csoportosították, így amikor az 1-es láb 0-ba megy, az összes flip-flop 0 állapotba kerül. Felmerül a

kérdés, hogy a 11-es láb miért van invertálva a bemeneti oldalon, és azután újból invertálva minden CK jelnél. Ez azért van, mert a bemeneti jelnek esetleg nincs elég árama, hogy vezérelje mind a nyolc flip-flopot; a bemeneti invertert valójában erősítőként használjuk.

Noha a 3.28. (b) ábrán ábrázolt óra és törlővonalak csoportosításának egyik oka az, hogy csökkentsék a lábak számát, ebben a konfigurációban a lapka másképp is felhasználható, nemcsak mint nyolc független flip-flop. Egyszerű 8 bites regiszternek is használható. További lehetőség: két ilyen lapkát párhuzamosan használva ki tudunk alakítani egy 16 bites regisztert az 1-es és a 11-es lábak összekötésével. A regiszterekről és használatukról a 4. fejezetben bővebben lesz szó.

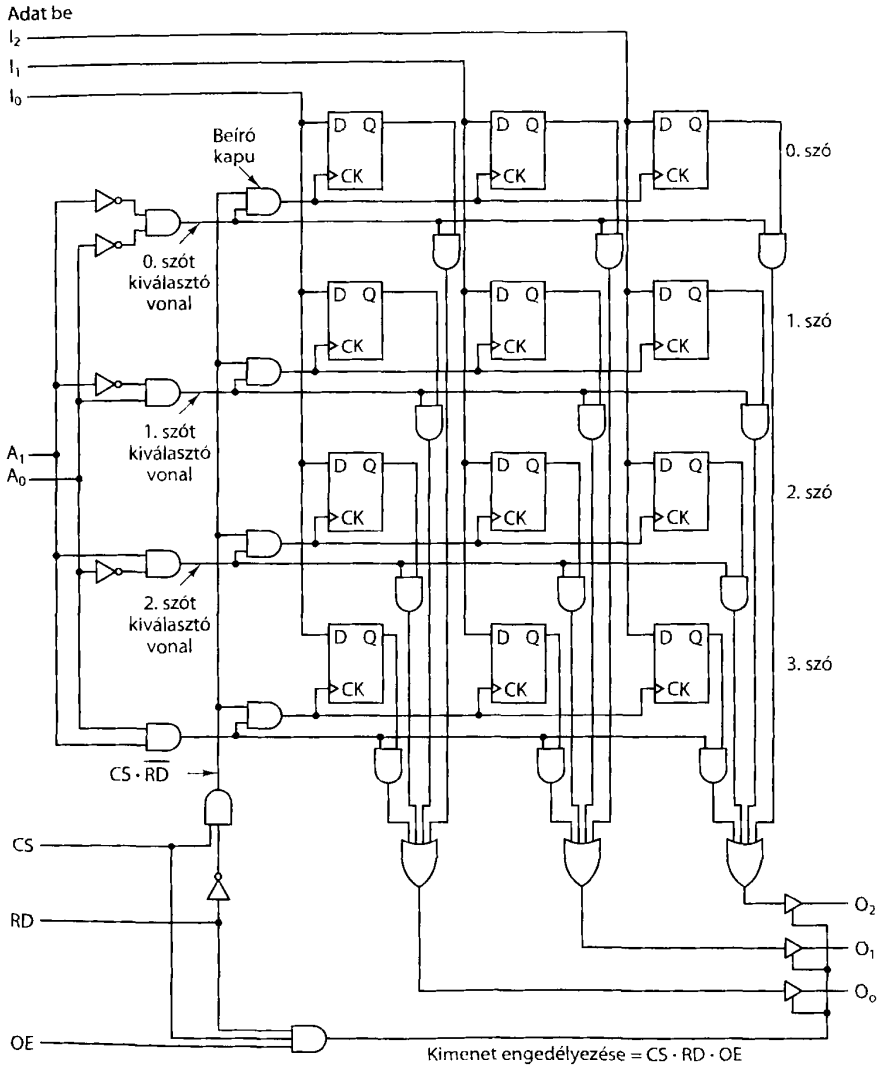
3.3.4. Memóriaszervezés

Bár most már elértünk a 3.24. ábrán bemutatott egyszerű 1 bites memóriától a 3.28. (b) ábrán ábrázolt 8 bites memóriáig, a nagyobb memóriák építéséhez másfajta szervezés szükséges; olyan, amelyben egyedi szavakat tudunk megcímezni. Széles körben használatos memóriaszervezést láthatunk a 3.29. ábrán, amely teljesíti ezt az elvárást. Ez a példa négy 3 bites szóból álló memóriát mutat be. Minden művelet teljes 3 bites szavakat olvas vagy ír. Amíg a memória 12 bitnyi kapacitása alig több, mint a nyolcas flip-flopé, de kevesebb lábat kíván, és ami a legfontosabb: ez a tervezés könnyen kiterjeszthető nagy memóriákra.

A 3.29. ábrán látható memória első ránézésre komplikáltnak tűnik, de valójában nagyon egyszerű a szabályos struktúrája miatt. Nyolc bemenő és három kimenő vonala van. A három adat: I_0, I_1 és I_2; a két a cím: A_0 és A_1; és a három a vezérlőbemenet: CS (Chip Select) a lapka kiválasztásához, RD (Read) az olvasás és írás megkülönböztetésére és OE (Output Enable) a kimenet engedélyezésére. A három adatkimenet: O_0, O_1 és O_2. Elvileg ezt a memóriát egy 14 lábas tokozásba el lehet helyezni, beleértve a tápfeszültséget és a földet, szemben a 20 lábas oktális flip-floppal.

A memórialapka kiválasztásához a külső logikának be kell állítani CS-t magasra (logikai 1), és olvasás esetén RD-t is magasra, írás esetén pedig alacsonyra (logikai 0). A két címvonalat be kell állítani annak jelzésére, hogy a négy 3 bites szó közül melyiket szeretnénk írni vagy olvasni. Olvasáskor az adatbemeneti vonalakat nem használjuk, a kimenő vonalakon viszont megjelenik a kiválasztott szó. Írásnál az adatbemeneti vonalakon lévő bitek betöltődnek a kiválasztott memóriaszóba, a kimenő adatvonalakat nem használjuk.

Most nézzük meg részletesen 3.29. ábrát, hogy lássuk a működését. A négy szót kiválasztó ÉS kapu (a bemeneteivel együtt) a memória bal oldalán egy dekódolót alkot. A bemeneti inverterek úgy vannak elhelyezve, hogy minden kapu különböző cím esetén ad ki engedélyező (magas) jelet. Minden kapu egy szókiválasztó vonalat vezérel, fentről lefelé haladva a 0., 1., 2. és 3.-at. Amikor a lapka írásra van választva, a CS · $\overline{\text{RD}}$-vel jelölt függőleges vonal lesz magas, ezzel engedélyezi a négy író kapu valamelyikét, attól függően, hogy melyik szókiválasztó vonal a magas. Az író kapu kimenete a kiválasztott szóhoz tartozó összes CK jelet meghajtja, hogy be-



3.29. ábra. *4 × 3-as memória logikai diagramja. Minden sor a négy 3 bites szó egyikét tartalmazza. Az olvasás és írás műveletek mindig a teljes szót olvassák vagy írják*

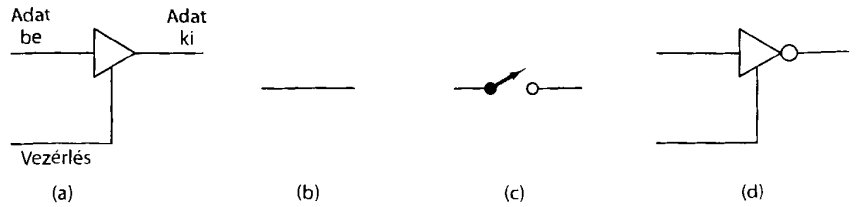
töltse a bemeneti adatokat (adat be) a szóhoz tartozó flip-flopokba. Az írás csak akkor történik meg, ha a CS magas és az RD alacsony, és csak az A_0 és A_1-gyel kiválasztott szóba történik írás; az összes többi szó változatlanul marad.

Az olvasás hasonló az íráshoz. A cím dekódolása pontosan ugyanúgy történik, mint az írásnál. Csak most a CS · $\overline{\text{RD}}$ vonal alacsony, így az összes író kapu letiltódik, és egyetlenegy flip-flop sem módosul. Ehelyett a megcímzett szókiválasztó vonal érvényesíti azokat az és kapukat, amelyek a kiválasztott szó Q bitjeivel vannak összekötve. Így a kiválasztott szó küldi az adatait az ábra alján elhelyezett négybemenetű VAGY kapukra, míg a másik három szó 0-t küld. Következésképpen a VAGY kapuk kimeneteinek értékei azonosak a kiválasztott szóban tárolt értékekkel. A három ki nem választott szó nem játszik szerepet a kimenetben.

Láthatjuk, hogy a tervünkön a bemeneti és a kimeneti adatvonalak különbözők, ezzel ellentétben a valódi memóriáknál azonosak. Ha közös be- és kimenő vonalak esetén összekötnénk a VAGY kapukat a kimeneti vonalakkal, akkor a lapka írás idején is megpróbálna adatot küldeni, azaz mindegyik vonalra egy meghatározott értéket tenni, és ez összeütközésbe kerülne a bemeneti adatokkal. Ezért kívánatos, hogy olyan megoldást alkalmazzunk, amely olvasás esetén összeköti a VAGY kapukat a kimeneti vonalakkal, de írás esetén teljesen leválasztja. Ehhez szükségünk van egy elektronikus kapcsolóra, amely egy kapcsolatot néhány ns alatt fel tud építeni és meg tud szakítani.

Szerencsére van ilyen kapcsoló. A 3.30. (a) ábrán egy ilyen kapcsoló, a **nem invertáló puffer (noninverting buffer)** jelét láthatjuk. Egy adatbemenete, egy adatkimenete és egy vezérlőbemenete van. Amikor a vezérlőbemenet magas, a puffer úgy viselkedik, mint egy huzal (össze van kötve), ahogy ezt a 3.30. (b) ábra mutatja. Amikor a vezérlőbemenet alacsony, a puffer úgy viselkedik, mint egy megszakított áramkör (lásd 3.30. (c) ábra). Ez olyan, mintha valaki egy drótvágóval levágta volna az adatkimeneteket az áramkör többi részéről. Azonban, ellentétben a drótvágós módszerrel, a kapcsolat néhány ns után vissza tud állni a vezérlőjel újbóli magasra állításával.

A 3.30. (d) ábra az **invertáló puffert (inverting buffer)** mutatja, amely úgy viselkedik, mint egy normális inverter, ha a vezérlőjel magas, és amikor a vezérlőjel alacsony, lekapcsolja a kimenetet az áramkörről. A pufferek mindkét típusa **háromállapotú eszköz (tri-state devices)**, mert 0-t, 1-et vagy semmit (nyitott áramkör) tudnak kibocsátani. A pufferek erősítik is a jeleket, így nagyon sok bemenetet tudnak egyszerre meghajtani. Ebből a célból néha még akkor is használjuk ezeket az áramkörben, ha a kapcsolási tulajdonságuk nem szükséges.



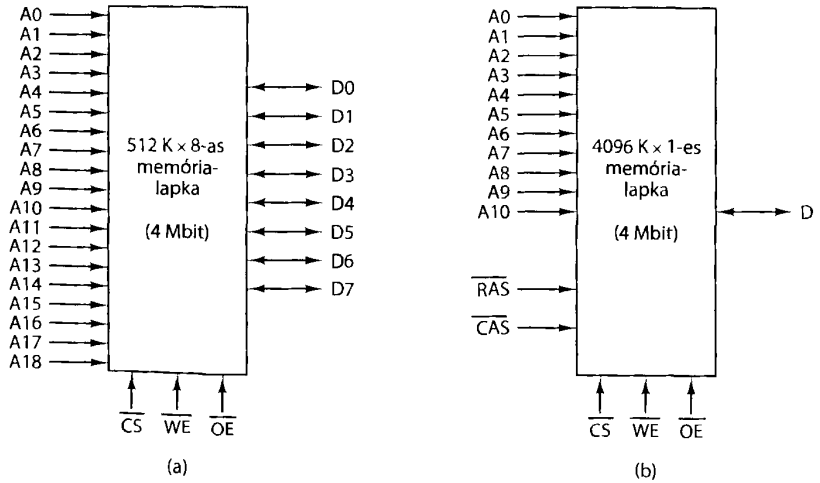
3.30. ábra. *(a) Nem invertáló puffer. (b) Működése magas értékű vezérlőbemenet esetén. (c) Működése alacsony értékű vezérlőbemenet esetén. (d) Invertáló puffer*

Visszatérve a memória-áramkörhöz, világos a három nem invertáló puffer szerepe a kimeneti vonalakon. Amikor CS, RD és OE mind magas, a kimenetet engedélyező vonal is magas, ekkor engedélyezve vannak a pufferek (vezetnek), és elhelyezik a szót a kimeneti vonalakon. Amikor CS, RD vagy OE valamelyike alacsony, az adatkimenetek lekapcsolódnak az áramkör többi részéről.

3.3.5. Memórialapkák

Az a szép a 3.29. ábrán bemutatott memóriában, hogy könnyen kiterjeszthető nagyobb méretekre. Ahogy rajzoltuk, a memória 4 × 3-as, azaz négy szóban egyenként 3 bit van. Ahhoz, hogy kiterjesszük 4 × 8-ra, csak az szükséges, hogy hozzáadjunk öt oszlopot négy-négy flip-floppal, öt bemeneti és öt kimeneti vonalat. Ahhoz, hogy 4 × 3-ról 8 × 3-ra terjesszük ki, még további négy sort adunk hozzá három-három flip-floppal és egy A_2 címvonalat. Az ilyen típusú struktúránál, a maximális hatékonyság érdekében a memóriában lévő szavak számának kettő hatványának kell lennie, de a szavakban lévő bitek száma bármennyi lehet.

Mivel az integrált áramkörök technológiája nagyon alkalmas arra, hogy olyan lapkákat állítson elő, amelyek belső struktúrája egy ismétlődő kétdimenziós minta, a memórialapkák ennek ideális alkalmazásai. Ahogy a technológia fejlődik, az egy lapkába elhelyezett bitek száma növekszik, ez tipikusan egy kettes szorzót jelent minden 18 hónapban (Moore törvénye). A nagyobb lapkák nem mindig kompatibilisek a kisebbekkel, a különböző kereskedelmi tulajdonságok a kapacitás, a sebesség, a teljesítmény, az ár és a kapcsolódási felületek tekintetében. Általában az aktuálisan kapható legnagyobb lapkákat felárral árusítják, és így drágábbak egy bitre vonatkozóan, mint a régebbi kisebbek.



3.31. ábra. *4 Mbit-es memórialapka két lehetséges szervezése*

Bármely adott méretű memórialapka különböző módon szervezhető. A 3.31. ábra két lehetséges szervezését mutatja egy régebbi 4 Mbit-es lapkának: 512 K × 8-as és 4096 K × 1-es szervezését. (Mellékesen megjegyezzük, a memórialapkák nagyságát általában inkább bitekben jelzik, mint bájtokban, így mi is követjük ezt a konvenciót.) A 3.31. (a) ábrán 19 címvonal szükséges a 2^{19} bájt bármelyikének megcímzéséhez, és nyolc adatvonal szükséges, hogy olvassa vagy írja a kiválasztott bájtot.

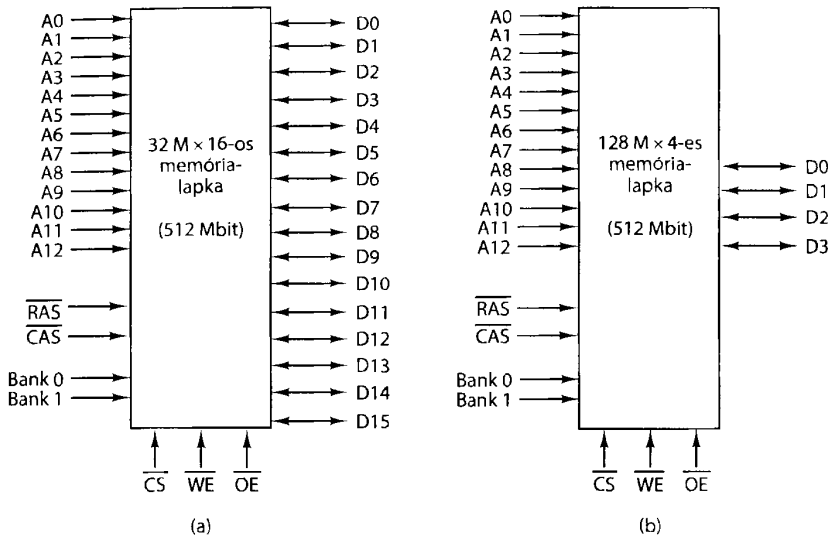
A rend kedvéért itt egy megjegyzést teszünk a terminológiához. Bizonyos lábakon a magas feszültség, másoknál az alacsony feszültség okoz akciót. Azért, hogy elkerüljük a zavart, következetesen azt fogjuk mondani, hogy egy jel **beállított (asserted)**; ahelyett, hogy magas vagy alacsony, és ez azt jelenti, hogy úgy van beállítva, hogy valamilyen akciót váltson ki. Így bizonyos lábakra a beállítás azt jelenti, hogy magasra, míg másoknál azt jelenti, hogy a lábak alacsonyra vannak állítva. Az alacsony jellel beállított lábakat a jel nevének felülvonásával jelezzük. Így a CS nevű jel akkor van beállítva, ha magas, a CS pedig akkor, ha alacsony. A beállított jel ellentettje a **negált jel**. Ha semmi különös nem történik, a lábak negáltak.

Most térjünk vissza a 3.31. ábrán lévő lapkákhoz. Mivel a számítógép általában több memórialapkával rendelkezik, kell egy jel, amely kiválasztja az aktuálisan szükséges lapkát, ez fog működni, az összes többi pedig nem. A $\overline{\text{CS}}$ (Chip Select) jel szolgál erre a célra. Ezt kell beállítani a lapka érvényesítéséhez. Valahogy meg kell különböztetnünk az írást az olvasástól. Erre szolgál a $\overline{\text{WE}}$ (Write Enable) jel. Végül az $\overline{\text{OE}}$ (Output Enable) jellel a kimeneti jeleket vezéreljük. Amikor nincs beállítva, a lapka kimenete le van kapcsolva az áramkörről.

A 3.31. (b) ábrán különböző címzési sémákat használunk. Belül ez a lapka egy 2048 × 2048-as 1 bites cellákból álló mátrix, amely összesen 4 Mbit. A lapkán belüli címzéshez először egy sort jelölünk ki a 11 címlábon a bitek beállításával. Ezután a $\overline{\text{RAS}}$ (Row Address Strobe, sorcím) kapuzójelet állítjuk be. Ezt követi az oszlop címének beállítása a címlábakon, végül a $\overline{\text{CAS}}$ (Column Address Strobe, oszlopcím) kapuzójel beállítása következik. A lapka egy adatbit befogadásával vagy kiadásával válaszol.

Hatalmas memórialapkákat gyakran $n \times n$-es mátrixként építenek föl, amelyek címzése sor és oszlop megadásával történik. Ez a szervezés lecsökkenti a szükséges lábak számát, de egyben a lapka címzését lassabbá teszi: két címzési ciklus szükséges, egy a sor és egy az oszlop címzéséhez. Hogy az ilyen tervezésnél valamennyit visszanyerjünk az elvesztett sebességből, néhány memórialapkánál a sorcímzést az oszlopcímek egy sorozata követheti, hogy egy adott sorban lévő egymás utáni biteket gyorsabban érhessük el.

Évekkel ezelőtt a legnagyobb memórialapkákat gyakran a 3.31. (b) ábrához hasonlóan szervezték. Ahogy a memóriaszavak 8 bitről 32 vagy még több bitesre nőttek, az 1 bit széles lapkák kezdtek kényelmetlenné válni. 32 lapka szükséges ahhoz, hogy egy 32 bites szavakból álló memóriát építsünk 4096 K × 1 lapkákból. Ennek a 32 lapkának a teljes kapacitása 16 MB, ha pedig 512 K × 8 lapkából csak négyet kell párhuzamosan használnunk, ez 2 MB-os memóriát jelent. Annak érdekében, hogy elkerülhető legyen 32 lapka használata a memóriához, a legtöbb lapkagyártónak jelenleg már saját 1, 4, 8 és 16 bit szélességű memórialapka-családja van. Természetesen a 64 bites szavaknál a helyzet még rosszabb.



3.32. ábra. *512 Mbit-es memórialapka két lehetséges szervezése*

A 3.32. ábrán két példát mutatunk modern 512 Mbit-es lapkákra. Ezeken a lapkákon négy, egyenként 128 Mbit-es memóriamodul található, ehhez két modulkiválasztó vonal szükséges. A 3.32. (a) ábrán szereplő terv egy 32 M × 16-os szervezésű, 13 vezeték a RAS jelhez, 10 vonal a CAS jelhez és 2 vonal a modulkiválasztáshoz. Ez a 25 jel lehetővé teszi 2^{25} belső 16 bites cella megcímzését. Ezzel szemben a 3.32. (b) ábra 128 M × 4-es tervezésű, 13 vezeték a RAS jelhez, 12 vezeték a CAS jelhez és 2 vezeték a modulkiválasztáshoz. Ebben az esetben a 27 jel tudja a 2^{27} darab belső 4 bites cella bármelyikét kiválasztani. Mérnöki szempontok határozzák meg azt a döntést, hogy hány sora és hány oszlopa legyen a lapkának. A mátrixnak nem szükséges négyzetesnek lennie.

Ezek a példák két eltérő és egymástól független memórialapka tervezési elvet mutatnak be. Az első a kimeneti szélesség (bitekben): 1, 4, 8, 16 vagy hány bitet továbbít egyszerre a lapka. A második, hogy az összes címbit megjelenik-e egyszerre a lábakon, vagy a sorok és oszlopok egymás után jelennek meg, amint a 3.32. ábrákon láttuk. A tervezőknek mindkét kérdésre válaszolniuk kell, még mielőtt megkezdik a lapka tervezését.

3.3.6. RAM-ok és ROM-ok

Az eddig megismert memóriák mindegyikét lehetett írni és olvasni. Az ilyen memóriákat **RAM**-nak (**Random Access Memory, véletlen elérésű memória**) hívjuk, amely rossz elnevezés, mert a memórialapkák mindegyike véletlenszerűen érhető el, de ez a fogalom már nagyon megszokott, nehéz megszabadulnunk tőle. Két változata van a RAM-oknak, a statikus és a dinamikus. A **statikus RAM**-ok (**SRAM**) belső áramkörei hasonló felépítésűek, mint a mi alap D-flip-flopunk. Ezeknek a memóriáknak az a tulajdonsága, hogy tartalmuk addig marad meg, amíg a memória áramellátása biztosított: másodpercekre, percekre, órákra, sőt napokra. A statikus RAM-ok nagyon gyorsak. A tipikus elérési idő néhány ns. Emiatt a statikus RAM-ok nagyon népszerűk második szintű gyorsítótárak (memóriák) építésére.

A **dinamikus RAM**-ok (**DRAM**) az előzőkkel ellentétben, nem flip-flopot használnak. Ehelyett a dinamikus RAM a cellák egy tömbje, mindegyik cella egy tranzisztort és egy pici kondenzátort tartalmaz. A kondenzátor feltöltött vagy kisült, annak megfelelően, hogy 0-t vagy 1-et tárol. Mivel az elektromos töltés hajlamos a szivárgásra, a dinamikus RAM-ban minden bitet **frissíteni kell (refresh; reload)** néhány ezredmásodpercenként, hogy az adat ne „szivárogjon el". Mivel külső logikának kell vigyáznia a frissítésre, a dinamikus RAM-ok összetettebb kapcsolódást kívánnak, mint a statikusak, bár nagyon sok alkalmazásnál ezt a hátrányt ellensúlyozza a nagyobb tárolókapacitásuk.

Mivel a dinamikus RAM-okhoz csak egy tranzisztor és egy kondenzátor kell bitenként (szemben a hat tranzisztorral, amelyek a legjobb statikus RAM-hoz bitenként szükségesek), a dinamikus RAM-oknak nagyon nagy lehet a sűrűsége (sok bit van lapkánként). Emiatt a főmemóriák majdnem mindig dinamikus RAM-okból épülnek föl. Azonban ennek a nagy kapacitásnak megvan az ára: a dinamikus RAM-ok lassabbak (néhány tíz ns). Így a statikus RAM gyorsítótárak és a dinamikus RAM főmemóriák kombinációja megpróbálja kombinálni mindegyikük jó tulajdonságait.

A dinamikus RAM lapkáknak számos típusa van. A legöregebb típus, amelyet még használnak az **FPM (Fast Page Mode, gyors lapkezelésű)** DRAM. Belseje egy bitmátrix, és úgy dolgozik, hogy a hardverben megjelenik egy sorcím, a következő lépésben pedig az oszlopcím, ahogy azt a 3.31. ábránál a RAS és CAS jelekkel kapcsolatban leírtuk. Külön jel szólítja fel a memóriát, amikor dolgoznia kell, így a memória a fő rendszerórától eltérő szinkronizációval működik.

Az FPM DRAM-ot azonban fokozatosan lecserélik az **EDO (Extended Data Output, kiterjesztett adatkimenetű)** DRAM-mal, amely megengedi egy második memóriahivatkozás megkezdését, mielőtt az előző befejeződött volna. Ennek az egyszerű szállítószalagelvnek az alkalmazása az egyedi memóriahasználatot nem teszi gyorsabbá, de javítja a memória sávszélességét azzal, hogy több szót ad másodpercenként.

Az FPM és EDO típusok elfogadhatóan jól működnek, ha a memórialapka ciklusideje 12 nsec vagy ennél alacsonyabb. Amikor a processzorok olyan gyorsak, hogy gyorsabb memória is szükséges, FPM-et és EDO-t lecserélik **SDRAM**-ra (**Synehronous DRAM**), amely a statikus és a dinamikus RAM-ok keveréke, és a fő rendszeróra vezérli. Az SDRAM nagy előnye az, hogy az óra kiküszöböli azokat a vezérlőjeleket, amelyek megmondják a memórialapkának, hogy mikor válaszoljon. Ehelyett a CPU megmondja a memóriának, hogy hány ciklusban kell működnie, azután elindítja. Minden egymást követő ciklusra a memória 4, 8 vagy 16 bitet bocsát ki attól függően, hogy hány kimeneti vonala van. A vezérlőjelek elhagyása növeli az adatátviteli sebességet a CPU és a memória között.

Az SDRAM következő javítása a **DDR (Double Data Rate)** SDRAM. Az ilyen típusú memóriával a memórialapka adatot bocsát ki mind az órajel felfutó élénél, mind a lefutó élénél, megduplázva az adatsebességet. Így egy 8 bit széles DDR lapka 200 MHz-es környezetben másodpercenként 200 milliószor két alkalommal 8 bites adatot bocsát ki (egy rövid intervallumban, természetesen), teljesítve az elméleti 3,2 Gbps-os csoportos (burst) átviteli sebességet.

Nem felejtő memórialapkák

A RAM nem az egyetlen fajtája a memórialapkáknak. Sok alkalmazásnál – például a játékoknál, berendezéseknél, autóknál – a program és az adatok egy részének meg kell maradnia, még akkor is, ha az áramot kikapcsoljuk. Hasonlóan az egyszer telepített alkalmazásokhoz, ahol sem a program, sem az adat soha nem fog változni. Ezek a kívánalmak vezettek a **ROM (Read-Only Memories, csak olvasható memória)** kifejlesztéséhez, amely nem változtatható meg, nem törölhető sem belsőleg, sem más módon. A ROM-ban tárolt adatok betöltése a gyártás során történik, lényegében egy fényérzékeny anyag maszkon át történő megvilágításával kezdődik, a maszk tartalmazza a kívánt bitmintát, azután a megvilágított (vagy nem megvilágított) felület kimaratása következik. Csak egyetlen módon lehet megváltoztatni a ROM-ban tárolt programot, mégpedig úgy, hogy a teljes lapkát kicseréljük.

A ROM-ok jóval olcsóbbak, mit a RAM-ok; ha nagy mennyiségben rendeljük, arányosan csökken a maszk készítésének költsége. Másrészt nem rugalmas, mert nem lehet gyártás után módosítani, a megrendelés és a ROM-ok megérkezése közötti fordulási idő heteket vehet igénybe. Hogy megkönnyítsék a cégek az új ROM-alapú produktumok előállítását, kifejlesztették a **PROM**-ot **(Programmable ROM)**, a programozható ROM-ot. A PROM hasonlít a ROM-hoz, de (egyszer) mező-programozható, és így megtakarítható a fordulási idő. Sok PROM egy apró biztosítékokból álló tömböt tartalmaz. A speciális olvadó biztosítékot úgy égetjük ki, hogy a megfelelő sorokat és oszlopokat kiválasztjuk, és ezután nagy feszültséget kapcsolunk a lapka megfelelő lábára.

A következő fejlesztés ebben az irányban az **EPROM (Erasable PROM, törölhető PROM)**, amely nemcsak mező-programozható, hanem mező-törölhető is. Azzal, hogy az EPROM-ot kvarcüveg ablakon át erős ultraviola fénnyel 15 percig megvilágítjuk, az összes bitet 1-re állítjuk. Ha sok változtatás szükséges a tervezési ciklus alatt, az EPROM-ok jóval gazdaságosabbak, mint a PROM-ok, mert előbbiek újra felhasználhatók. Az EPROM-ok általában ugyanazt a szervezést tartalmazzák, mint a statikus RAM-ok. A 4 Mbites 27C040 EPROM például ugyanazt a szervezést használja, mint a 3.32. (a) ábrán bemutatott tipikus statikus RAM.

Az EPROM-oknál még jobbak az **EEPROM**-ok, amelyek impulzusokkal törölhetők ahelyett, hogy egy speciális kamrában ultraviola fénnyel világítanánk meg őket. Ráadásul az EEPROM a helyén programozható újra, míg az EPROM-ot ehhez be kell tenni egy speciális EPROM-programozó berendezésbe. A negatívum, hogy a legnagyobb EEPROM-ok kapaciása tipikusan csak 1/64 része a közönséges EPROM-okénak, és csak fele olyan gyorsak. Az EEPROM-ok nem tudják felvenni a versenyt a DRAM-okkal vagy az SDRAM-okkal, mert tízszer lassabbak, százszor kisebb a kapacitásuk és jóval drágábbak. Ezeket csak azokban a helyzetekben alkalmazzák, amikor a „nem felejtő" képességük a fő szempont.

A legújabb EEPROM-típus a **flash memória (flash memory)**. Ellentétben az EPROM-mal, amely ultraviola fénnyel való megvilágítással törölhető, és az EEPROM-mal, amely pedig bájtonként, a flash memória blokkonkénti törlést és újraírást tesz lehetővé. Akárcsak az EEPROM, a flash memória is törölhető anélkül, hogy az áramkörből elmozdítanánk. Különböző gyártók apró nyomtatott áramköri kártyát készítenek több tíz Mb-os flash memóriával, ezek digitális kamerákban képek tárolására „filmként" használhatók, továbbá sok más célra is. Valamikor majd flash memóriákat használhatunk a diszkek helyett, ami óriási előrelépés lenne az 50 ns-os elérési idejük miatt. A legnehezebb mérnöki probléma jelenleg az, hogy 100 000 törlés után elhasználódnak, amíg a diszkek évekig működhetnek, függetlenül attól, hogy milyen gyakran írjuk újra. A különböző memóriákról a 3.33. ábra ad összegzést.

Típus	Kategória	Törlés	Bájt-változtatás	Felejtés	Tipikus használat
SRAM	Olvasás/írás	Elektromos	Igen	Igen	2-es szintű gyorsítótár
DRAM	Olvasás/írás	Elektromos	Igen	Igen	Fő memória (régi)
SDRAM	Olvasás/írás	Elektromos	Igen	Igen	Fő memória (új)
ROM	Csak olvasás	Nem lehetséges	Nem	Nem	Nagy berendezések
PROM	Csak olvasás	Nem lehetséges	Nem	Nem	Apró berendezések
EPROM	Főleg olvasás	UV-fény	Nem	Nem	Prototípusok
EEPROM	Főleg olvasás	Elektromos	Igen	Nem	Prototípusok
Flash	Olvasás/írás	Elektromos	Nem	Nem	Filmként digitális kamerához

3.33. ábra. *A különböző típusú memóriák összehasonlítása*

3.4. CPU lapkák és sínek

Az SSI, MSI és memórialapkákról szerzett információval felfegyverkezve, most már megkezdhetjük összerakni az egyes részeket, hogy egy komplett rendszert kapjunk. Ebben az alfejezetben először a központi egység (CPU) néhány általános sajátosságát nézzük meg a digitális logika szintje felől, beleértve a **lábkiosztást (pinout)** is (hogy milyen jelek vannak a különböző lábakon). A központi egységek nagyon szorosan összefonódnak az általuk használt sínek tervezésével, ezért egy bevezető is található a sínek tervezéséhez. A következő szakaszokban részletes példát adunk a CPU-kra, sínjeikre és összekapcsolódásukra is.

3.4.1. CPU lapkák

Minden modern CPU egyetlen lapkán helyezkedik el. A kapcsolatuk a rendszer többi részével jól definiált. Minden egyes CPU lapkának van lábkészlete, ezen keresztül tartja a kapcsolatot a külvilággal. Néhány láb kimeneti jeleket kap a CPU-tól; mások jeleket fogadnak a külvilágból; néhány mindkettőre képes. Az összes láb funkciójának megértésével megtanulhatjuk, hogyan kapcsolódik a CPU a memóriához, a bemeneti/kimeneti (B/K, input/output, I/O) berendezésekhez a digitális logika szintjén.

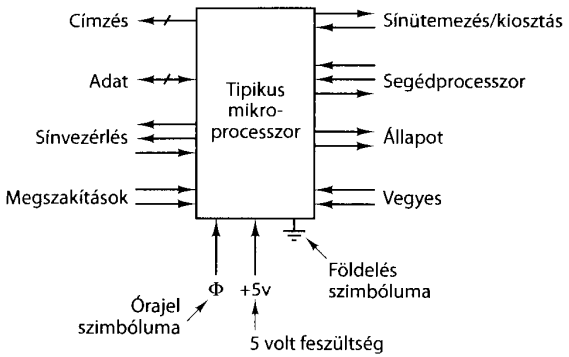
A CPU lapka lábait három csoportra oszthatjuk: cím, adat és vezérlés. Ezek a lábak párhuzamos huzalokon keresztül kapcsolatban vannak a memória és a B/K lapkák hasonló lábaival, ezeket a párhuzamos huzalokat sínnek hívjuk. Egy utasítás betöltéséhez a CPU először beállítja az utasítás memóriacímét a címlábain. Ezután beállít egy vagy több vezérlővonalat, informálva a memóriát, hogy olvasni szeretne (például) egy szót. A memória a kért szónak a CPU adatlábaira helyezésével válaszol, és beállít egy jelet, amely azt mondja, hogy ez megtörtént. Amikor a CPU látja ezt a jelet, elfogadja a szót és végrehajtja az utasítást.

Egy utasításnak szüksége lehet adatszavak olvasására és írására, ebben az esetben a teljes folyamat minden további szó esetében megismétlődik. Az olvasás és írás működésének részleteit később nézzük meg. Jelenleg egyet kell megértenünk: a CPU a memóriával és a B/K berendezésekkel úgy tartja a kapcsolatot, hogy jeleket küld és fogad a lábain. Másféle kapcsolat nem lehetséges.

Két lényeges paraméter határozza meg a CPU teljesítményét: az egyik a címlábak, a másik az adatlábak száma. Egy lapka m címlábbal 2^m memóriahelyet tud megcímezni. m szokásos értéke 16, 20, 32 és 64. Hasonlóan egy lapka n adatlábbal n bites szavakat tud írni és olvasni egyetlen művelet során. Az általános értékek itt 8, 16, 32, 36 és 64. Egy 8 adatlábbal rendelkező CPU négy művelettel olvas 32 bites szavakat, míg a 32 adatlábbal rendelkező ugyanezt a feladatot egy művelettel végzi el, ezért a 32 adatlábas lapka sokkal gyorsabb, de természetesen drágább is.

A cím- és adatlábakon kívül minden CPU-nak van néhány vezérlőlába is. A vezérlőlábak szabályozzák a folyamatot, a CPU-ból és a CPU-ba mozgó adatok időzítését, és további, vegyes funkcióik is vannak. Minden CPU-nak vannak lábai az áramellátásra (általában +3,3 vagy 5 volt), a földelésre és az órajelre (pontosan meghatározott frekvenciájú négyszögjel), de a többi láb lapkáról lapkára erősen változik. Mindemellett, a vezérlőlábakat durván a következő főbb kategóriákba csoportosíthatjuk:

1. Sínvezérlés;
2. Megszakítások;
3. Sínütemezés/kiosztás;
4. Segédprocesszor jelei;
5. Állapot;
6. Vegyes.



3.34. ábra. *Általános CPU logikai lábkiosztása. A nyilak jelzik a bemeneti és a kimeneti jeleket. A rövid átlós vonalak jelzik a többszörös lábak használatát, egy-egy konkrét CPU-nál egy szám jelzi a többszörözés számát*

A továbbiakban röviden ezeket a kategóriákat tárgyaljuk. Később, amikor példaként megnézzük a Pentium 4, UltraSPARC III és 8051 lapkákat, további részletekkel is szolgálunk. Egy általános CPU lapka a 3.34. ábrán látható jelcsoportokat használja.

A sínvezérlő lábak főként a CPU-ból a sínre történő kimenetként szolgálnak (ezek egyben bemenetek a memória- és a B/K lapkák számára), megmondják, hogy a CPU olvasni vagy írni akar, vagy valami mást akar tenni. Ezeket a lábakat a CPU a rendszer többi részének vezérlésére használja, és megmondja, hogy mit szeretne tenni.

A megszakítási lábak a B/K berendezésekről a CPU-ba tartó bemenetek. A legtöbb rendszerben a CPU megmondja egy B/K berendezésnek (eszköznek), hogyan kezdje el a műveletet, és azután magára hagyja, hogy valami hasznosat csináljon azalatt, amíg a lassú B/K eszköz elvégzi feladatát. Amikor a B/K eszköz befejezte, a B/K vezérlőlapka e lábak egyikén jelet küld, hogy megszakítsa a CPU-t, és az kiszolgálja a B/K berendezést, például ellenőrizze, hogy volt-e B/K hiba. Néhány CPU-nak kimeneti lába is van; ezen nyugtázza a megszakító jelet.

A sínütemezéslábak a sínen a forgalom irányításához szükségesek: megakadályozzák, hogy két berendezés egyszerre használja a sínt. A sínütemezés tekintetében a CPU egy berendezésnek számít a sínen, és akárcsak a többi berendezésnek, kérnie kell a sín használatát.

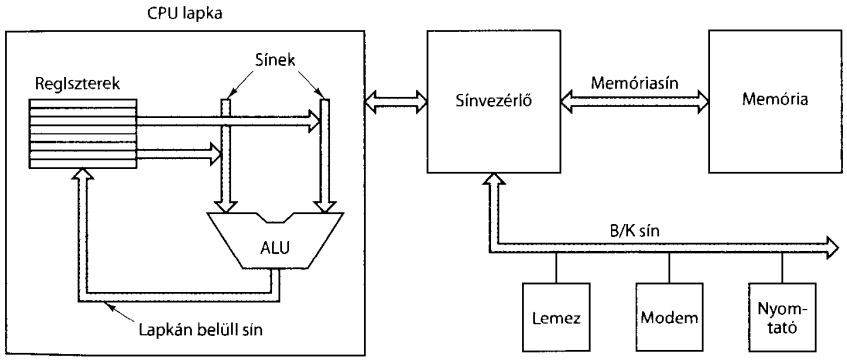
Néhány CPU lapkát úgy terveztek meg, hogy együttműködjön a segédprocesszorral (coprocessor), például a lebegőpontos, grafikus vagy más lapkával. A CPU és a segédprocesszor közötti kommunikáció biztosítására speciális lábak szolgálnak, amelyek küldik és fogadják a különböző kéréseket.

Ezeken kívül néhány CPU-nak más célú lábai is vannak. Ezek némelyike az állapotra vonatkozó információt szolgáltat vagy fogad, míg mások a számítógép alaphelyzetbe állításánál használatosak, és megint mások biztosítják a kompatibilitást a korábbi B/K lapkákkal.

3.4.2. Számítógépes sínek

A **sín (bus)** különböző eszközök közötti közös elektronikus pálya. A sineket funkcióik szerint osztályozhatjuk. Használhatjuk őket a CPU-n belül, hogy adatokat vigyünk be az ALU-ba vagy onnan ki, illetve a CPU-n kívül, hogy összekössük a CPU-t a memóriával vagy a B/K berendezésekkel. A sínek minden egyes típusának sajátos követelményei és tulajdonságai vannak. Ebben és a következő szakaszokban azokra a sínekre koncentrálunk, amelyek a CPU-t a memóriával és a B/K berendezésekkel kötik össze. A következő alfejezetben a CPU-n belüli sineket vizsgáljuk meg közelebbről.

A korai személyi számítógépeknek egyetlen külső sínje vagy **rendszersín**je **(system bus)** volt. Ez a sín 50–100 párhuzamos rézvezetékből állt, amelyet beépítettek az alaplapba (motherboard), szabályos távolságokban csatlakozókat helyeztek el, hogy memóriát és B/K kártyákat lehessen csatlakoztatni. A modern személyi számítógépekben általában egy speciális célú sín áll rendelkezésre a CPU és a memória között, és (legalább) egy másik sín a B/K berendezések számára. A 3.35. ábrán egy minimális rendszer látható egy memória- és egy B/K sínnel.



3.35. ábra. *Számítógépes rendszer több sínnel*

Az irodalomban a sineket gyakran vastag nyíllal ábrázolják, amint az ábrán is látszik. A vastag nyíl és a sima vonal közötti különbség (az utóbbi egy kisebb átlós vonallal áthúzott és egy bitszám is van rajta) kényes dolog. Amikor az összes bit azonos típusú, mondjuk, mind cím- vagy mind adatbit, akkor általában ez a rövid átlós vonalas jelölés használatos. Amikor cím-, adat- és vezérlési vonalak vannak együtt, a vastag vonal a szokásos.

Míg a lapka belsejében a CPU-tervezők szabadon használhatnak bármilyen sínt, addig a külső sínek működésére vonatkozóan pontosan meghatározott szabályokra van szükség, ezzel lehetővé téve, hogy harmadik fél tervezte kártyákat is a rendszerhez lehessen csatlakoztatni, és e szabályokat a csatlakoztatni kívánt összes berendezésnek be kell tartania. Ezeket a szabályokat hívjuk **sínprotokoll**nak **(bus protocol)**. Szükség van mechanikai és elektronikus előírásokra is, így a más gyártók által készített áramköri lapok meg fognak egyezni méretben, és olyan csatlakozásokkal fognak rendelkezni, amelyek mechanikusan jól illeszkednek az alaplap csatlakozásaihoz, valamint a feszültség, időzítés stb. tekintetében is megfelelnek.

Számos sínt széles körben használnak a számítástechnika világában. Néhány ma is használatos és történetileg is érdekes példa az ismertebbek közül: Omnibus (PDP–8), Unibus (PDP–11), Multibus (8086), VME sín (fizikai laboratóriumi berendezések), IBM PC sín (PC/XT), ISA sín (PC/AT), EISA sín (80386), Microchannel (PS/2), Nubus (Macintosh), PCI sín (sok PC), SCSI (sok PC és munkaállomás), Universal Serial Bus (modern PC-k) és FireWire (szórakoztatóelektronika). Valószínűleg jobb lenne a világ, ha egy kivételével hirtelen minden sín eltűnne a föld színéről (rendben, maradhat kettő). Sajnos a szabványosítás ezen a területen elég valószínűtlen, mert már túl sokat fektettek be ezekbe a nem kompatibilis rendszerekbe.

Térjünk át a sínek működésének a tanulmányozására. Néhány sínhez csatlakozó berendezés aktív, és átvitelt tud kezdeményezni, míg mások passzívak és kérésekre várnak. Az aktívakat **mesterek**nek **(masters)** hívjuk, a passzívakat pedig **szolgák**nak **(slaves)**. Amikor a CPU a lemezvezérlőtől egy blokk olvasását vagy írását kéri, a CPU mesterként viselkedik, a lemezvezérlő pedig szolga. Később azonban a lemezvezérlő mesterként is viselkedhet, amikor a memóriának ad ki parancsot, hogy fogadja el azt a szót, amelyet a lemezmeghajtóból olvasott. A 3.36. ábra néhány tipikus mester és szolga kombinációt mutat be. A memória sohasem lehet mester.

Mester	Szolga	Példa
CPU	Memória	Utasítások és adatok betöltése.
CPU	B/K eszközök	Adatátvitel kezdeményezése.
CPU	Segédprocesszor	CPU felkínálja az utasítást a segédprocesszornak.
B/K	Memória	DMA (Direct Memory Access, direkt memóriaelérés).
Segédprocesszor	CPU	A segédprocesszor átveszi az operandusokat a CPU-tól.

3.36. ábra. *Példák a sínmesterekre és -szolgákra*

A számítógép berendezései által kibocsátott bináris jelek, gyakran nem elég erősek, hogy elegendő áramot adjanak a sínnek, főleg akkor, ha a sín meglehetősen hosszú, és sok berendezés van rajta. Ezért a legtöbb sínmester egy **sínvezérlő**nek **(bus driver)** nevezett lapkával kapcsolódik a sínhez, mely lényegében egy digitális erősítő. Hasonlóan, a legtöbb szolga egy **sínvevő**vel **(bus receiver)** kapcsolódik a sínhez. Azok a berendezések, amelyek mesterként és szolgaként is működhetnek egy kombinált lapkát használnak, a **sínadóvevő**t **(bus transceiver)**. Ezek a sínhez kapcsolódó (bus interface) lapkák nagyon gyakran háromállapotú (tri-state) eszközök, hogy lekapcsolódhassanak a sínről, amikor nem szükséges a kapcsolat, máskor pedig **nyílt gyűjtők (open collector)**, amelyek némiképp eltérő módon kapcsolódnak a sínhez, de ugyanezt a hatást érik el. Amikor két vagy több berendezés egy ilyen nyílt gyűjtő vonalon egyidejűleg beállít egy jelet, az összes jel VAGY kapcsolata szolgáltatja az eredményt. Ezt az elrendezést gyakran **huzalo-**

zott VAGY-nak (**wired-OR**) hívják. A legtöbb sínen néhány vonal háromállapotú, a többi pedig, amelyek huzalozott VAGY tulajdonságot igényel, nyílt gyűjtő.

A CPU-hoz hasonlóan, egy sínnek is vannak cím-, adat- és vezérlővonalai. Azonban nem szükségképpen egy-egy értelmű a megfeleltetés a CPU lábai és a sín jelei között. Például néhány CPU három lábon kódolja, hogy a memóriából olvas, a memóriába ír, B/K-ról olvas, B/K-ra ír vagy mást csinál. Egy tipikus sínnek van egy vonala a memóriából olvasásra, egy másik a memóriába írásra, egy harmadik a B/K-ról olvasásra, egy negyedik a B/K-ra írásra és így tovább. Egy dekódoló lapkára van szükség a CPU és az ilyen típusú sín közé, hogy megfeleltesse a két oldalt, azaz, hogy a 3 bites kódolt jelet különböző jelekre válassza szét, amelyek a sín vonalait vezérlik.

A sín tervezése és működése meglehetősen bonyolult téma, ezért több, kizárólag e témának szentelt könyv jelent meg (Anderson és társai, 2004; Solari és Willse, 2004). A sín tervezésének alapvető kérdései: a sín szélessége, időzítése és ütemezése, valamint a sínműveletek. E területek mindegyike lényeges hatással van a sebességre és a sávszélességre. Most vizsgáljuk meg ezeket egyenként a következő négy szakaszban.

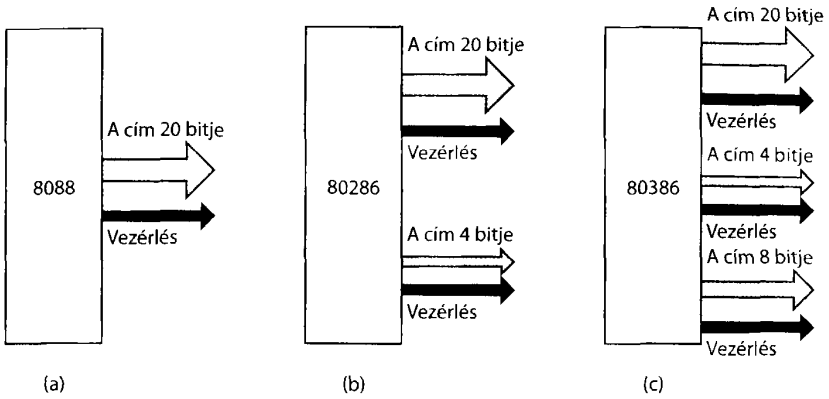
3.4.3. Sínszélesség

A sín szélessége az egyik legnyilvánvalóbb tervezési paraméter. Minél több címvezetéke van egy sínnek, annál nagyobb memóriát tud a központi egység közvetlenül megcímezni. Ha egy sínnek n címvezetéke van, akkor azt a CPU 2^n különböző memóriarekesz megcímzésére tudja használni. Nagy memória használatához a sínek sok címvezetéket igényelnek. Ez elég egyszerűen hangzik.

A szélesebb sínek több vezetéket igényelnek, mint a keskenyek. Ezenkívül több fizikai helyet is foglalnak (például az alaplapon), valamint nagyobb csatlakozókra van szükségük. Mindezek a tényezők egyre költségesebbé teszik a síneket. Szemmel láthatóan összefüggés van a maximális rendszermemória és a rendszer költsége között. Egy 64 címvezetékes sínnel és 2^{32} bájt memóriával rendelkező rendszer többe fog kerülni, mint az, amelyiknek csak 32 címvezetéke van és ugyanakkora, 2^{32} bájt memóriája. A későbbi bővítés lehetősége sincs ingyen.

Ennek a felismerésnek a következménye, hogy sok rendszertervező hajlamos a rövidlátásra, aminek később szerencsétlen következményei lehetnek. Az eredeti IBM PC egy 8088 típusú CPU-t tartalmazott és egy 20 bites címsínt, amint az a 3.37. (a) ábrán látható. Ez a 20 bit 1 MB memória megcímzését tette lehetővé a PC számára.

Amikor a következő CPU lapka (a 80286) megjelent, az Intel elhatározta, hogy megnöveli a címtartományt 16 MB-ra, így négy újabb címvezetéket kellett hozzáadni (anélkül, hogy az eredeti 20 vezetéket bántották volna, hogy a kompatibilitást megtarthassák), ez látható a 3.37. (b) ábrán. Az új címvezetékek miatt azonban több vezérlőjelre is szükség volt. Amikor a 80386 megjelent, további 8 címvezetékkel bővítették, több új vezérlővezetékkel együtt, amint az a 3.37. (c) ábrán látható. Az eredményül kapott kialakítás (az EISA sín) sokkal rendezetlenebb, mintha a sínnek már kezdetben 32 címvezetéke lett volna.



3.37. ábra. *A címsín szélességének növekedése az idők folyamán*

Nemcsak a címvezetékek száma hajlamos idővel a növekedésre, ugyanez történik az adatvezetékek számával is, bár ez utóbbinak valamennyire más az oka. Két módon tudjuk egy sín sávszélességét megnövelni: vagy csökkenteni kell a sínciklus idejét (több adatátvitel/s), vagy növelni kell a szélességet (több bitet kell átvinni egyszerre). Ugyan egy sínt fel is lehet gyorsítani, de ez nehéz, mivel az egyes vezetékeken a jelek kissé eltérő sebességgel haladnak – ez a probléma **sínaszimmetria**ként (**bus skew**) ismert. Minél gyorsabb egy sín, annál nagyobb az aszimmetria.

A másik probléma a felgyorsítással, hogy a felgyorsított sín visszafelé már nem lesz kompatibilis. Az alacsonyabb sebességű sínhez tervezett régi kártyák nem fognak működni az újjal. A régi kártyák alkalmatlanná nyilvánításának nem fognak örülni sem a régi kártyák tulajdonosai, sem előállítói. Ezért a teljesítmény növelésének az a szokásos módja, hogy az adatbitek számát növelik a 3.37. ábrán bemutatotthoz hasonlóan. Azonban, ahogyan az várható, ez a fokozatos növekedés nem vezet végül egy tiszta koncepcióhoz. Az IBM PC és utódai például 8 bites adatsínről 16 bitesre, majd 32 bitesre fejlődtek lényegében változatlan sín mellett.

A tervezők, hogy megkerüljék a túl széles sínek problémáját, időnként a **multiplexelt sínt (multiplexed bus)** részesítik előnyben. Ebben a konstrukcióban, ahelyett hogy külön cím- és külön adatvezetékek lennének, mondjuk, 32 vezeték van a címeknek és adatoknak együtt. A sín működésének kezdetén a vezetékeket a címzéshez használják, a későbbiekben pedig adattovábbításra. Memóriába írás esetén ez például azt jelenti, hogy a címvezetékeket kell először beállítani, és a címet a memóriához juttatni, mielőtt az adatok a sínre kerülhetnének. Külön vezetékek esetén a címet és az adatot egyszerre lehetne feltenni a sínre. A vezetékek multiplexelése csökkenti a sín szélességét (és a költségeket), ugyanakkor lassabb rendszert eredményez. A síntervezőknek gondosan mérlegelniük kell mindezeket a lehetőségeket, amikor kiválasztják a megfelelőt.

3.4.4. Sínek időzítése

A sínek két diszjunkt kategóriába csoportosíthatók, az időzítésüktől függően. A **szinkron sínnek (synchronous bus)** van egy vezetéke, amelyre egy kristályoszcillátor van kapcsolva. Ezen a vezetéken egy általában 5 és 100 MHz közé eső frekvenciájú négyszög hullámokból álló jel halad. Minden síntevékenység ezeknek a négyszög hullámoknak, a **sínciklusok**nak **(bus cycles)** az egész számú többszörösére tart. A másikfajta sínnek, az **aszinkron sín**nek **(asynchronous bus)** nincs ilyen fő órajel-generátora. A sínciklusok hossza igény szerint bármekkora lehet, és nem kell, hogy minden eszközpár között azonosak legyenek. Az alábbiakban megvizsgáljuk mindkét típusú sínt.

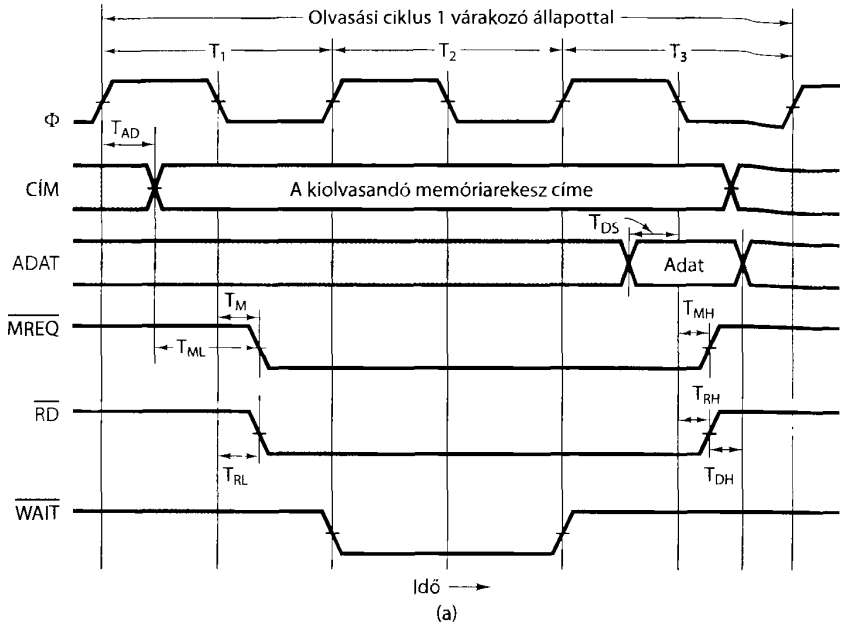
Szinkron sín

A szinkron sín működését vizsgáljuk meg egy példán, nézzük a 3.38. (a) ábra időzítését. Ebben a példában 100 MHz-es órajelet használunk, ami 10 ns-os sínciklust jelent. Ez ugyan kissé lassúnak tűnik a CPU 3 GHz-es vagy még annál is nagyobb sebességéhez, valójában kevés ennél lényegesen gyorsabb PC sín létezik. Például a népszerű PCI sín általában 33 vagy 66 MHz-en dolgozik. A jelenlegi sínek lassúságának okait már fentebb ismertettük: technikai tervezési problémák, mint például az aszimmetria és a visszafelé kompatibilitás kérdése.

Példánkban feltesszük továbbá, hogy a memóriából történő olvasás 15 ns időt igényel, attól számítva, hogy a cím a sínen van és stabil. Amint azt hamarosan láthatjuk, ezekkel a paraméterekkel egy szó beolvasása három órajelet igényel. Az első ciklus a T_1 felfutó élénél kezdődik el és a harmadik a T_4 felfutó élénél ér véget, ahogy ez a 3.38. ábrán látható. Figyeljük meg, hogy egyetlen felfutó vagy lefutó él sincs függőlegesen rajzolva, mivel semmilyen elektromos jel nem tudja a nagyságát 0 s idő alatt megváltoztatni. Ebben a példában feltesszük, hogy 1 ns szükséges egy jel számára, hogy megváltozzon. Az órajel, a CÍM, az ADAT, az $\overline{\text{MREQ}}$, az $\overline{\text{RD}}$ és a $\overline{\text{WAIT}}$ jelek ugyanabban az időskálában vannak ábrázolva.

A T_1 kezdetét az órajel felfutó éle definiálja. A T_1 periódus közben a CPU felteszi a kívánt memóriaszó címét a címvezetékekre. A cím nem egyetlen jel, mint mondjuk az órajel; ezért nem tudjuk egyetlen vonallal ábrázolni, hanem csak kettővel, amelyek akkor kereszteződnek, ha a cím megváltozik. Továbbá, az árnyékolás a keresztezés előtt azt jelenti, hogy az árnyékolt érték nem fontos. Ugyanezt az árnyékolási konvenciót használva láthatjuk, hogy az adatvezetékek tartalma sem lényeges, csak amikor már jól benn vagyunk a T_3-ban.

Amikor a címvezetékek már abban az állapotban vannak, hogy éppen felveszik az új értéküket, az $\overline{\text{MREQ}}$ és az $\overline{\text{RD}}$ jeleket beállítják. Az előbbi azt jelzi, hogy a memória (nem pedig egy B/K eszköz) elérése van folyamatban, az utóbbi jel olvasásnál alacsony, írás esetén negált (magas). Mivel a memória számára 15 ns szükséges, miután a cím stabil (valahol az első ciklus belsejében), ezért nem tudja T_2 alatt szolgáltatni a kívánt adatot. Hogy közölje a CPU-val, hogy ne várjon az adatra, a memória jelet küld a $\overline{\text{WAIT}}$ vezetéken T_2 kezdetekor. Ez **várakozó állapotok**at



Jelölés	Paraméter	Minimum	Maximum	Mértékegység
T_{AD}	Cím megérkezése a sínre		4	ns
T_{ML}	A cím a sínen van $\overline{\text{MREQ}}$ jel előtt T_{ML}-lel	2		ns
T_M	$\overline{\text{MREQ}}$ megjelenése Φ lefutó éle után T_1-ben		3	ns
T_{RL}	$\overline{\text{RD}}$ megjelenése Φ lefutó éle után T_1-ben		3	ns
T_{DS}	Az adat megjelenése Φ lefutó éle előtt	2		ns
T_{MH}	$\overline{\text{MREQ}}$ megszűnése Φ lefutó éle után T_3-ban		3	ns
T_{RH}	$\overline{\text{RD}}$ megszűnése Φ lefutó éle után T_3-ban		3	ns
T_{DH}	Az adatok tartási ideje az $\overline{\text{RD}}$ jel negációja után	0		ns

(b)

3.38. ábra. *(a) Az olvasás időzítése egy szinkron sínen. (b) Néhány kritikus időtartam specifikációja*

(wait states) szúr be (további sínciklusokat), amíg a memória be nem fejezi a feladatát, és nem negálja a $\overline{\text{WAIT}}$ jelet. A példánkban 1 várakozó állapot került be (a T_2), mivel a memória túl lassú. A T_3 kezdetekor, amikor már biztos, hogy ebben a ciklusban meglesz az adat, a memória negálja a $\overline{\text{WAIT}}$ jelet.

A T_3 első felében a memória felteszi az adatokat az adatvezetékekre. A T_3 lefutó élének hatására a CPU leolvassa az adatvonalakat, és feljegyzi az értéket egy belső regiszterben. Miután a CPU beolvasta az adatot, negálja az MREQ és $\overline{\text{RD}}$ jeleket. Ha szükséges, az órajel következő felfutó élénél újabb memóriaciklus kezdődhet. Ez a tevékenység ismétlődhet vég nélkül.

A 3.38. (b) időzítési specifikáció az időzítési diagramon látható nyolc jelölés jelentését magyarázza. A T_{AD} például az az időintervallum, amely a T_1 felfutó éle és a címvezetékek beállása között telik el. A specifikáció szerint $T_{AD} \leq 4$ ns. Ez azt jelenti, hogy a CPU-gyártó cég garantálja, hogy bármely olvasási ciklusban a CPU kiírja a beolvasni kívánt memória címét 4 ns-on belül a T_1 felfutó élének középpontjától számítva.

Az időzítési specifikációk azt is megkövetelik, hogy az adat hozzáférhető legyen az adatsínen legalább T_{DS} (2 ns) idővel T_3 lefutó éle előtt, hogy elég idő maradjon a stabilizálódásra, mielőtt a CPU leolvassa. A T_{AD} és T_{DS} feltételek kombinációjából adódik, hogy a legrosszabb esetben a memóriának csupán 25 – 4 – 2 = 19 ns idő áll a rendelkezésére az adat szolgáltatására attól számítva, hogy a cím megjelenik a vezetékeken. Egy 15 ns-os memória még a legrosszabb esetben is mindig tud válaszolni a T_3 periódus alatt, mivel 15 ns elegendő. Egy 20 ns-os memóriának már éppen nem elegendő ez az idő, és egy második várakozó állapotot is be kell szúrnia, és csak a T_4 alatt tud válaszolni.

Az időzítési specifikáció garantálja továbbá, hogy a cím 2 ns-mal azelőtt felkerül a sínre, mielőtt az $\overline{\text{MREQ}}$ jelet beállítanák. Ez az idő fontos lehet, ha $\overline{\text{MREQ}}$ vezérli a memórialapka kiválasztó jelét ($\overline{\text{RD}}$), mivel bizonyos memóriák elvárnak egy bizonyos címbeállási időt a lapka kiválasztása előtt. Nyilvánvalóan, egy rendszertervezőnek nem szabad olyan memórialapkát választani, amelynek 3 ns cím beállási időre van szüksége.

A T_M és T_{RL}-re vonatkozó feltételek azt jelentik, hogy az $\overline{\text{MREQ}}$ és az $\overline{\text{RD}}$ jelek mindegyike beáll 3 ns-on belül T_1 lefutó élétől számítva. A legrosszabb esetben a memórialapkának csak 10 + 10 – 3 – 2 = 15 ns ideje marad az $\overline{\text{MREQ}}$ és az $\overline{\text{RD}}$ beállítása után, hogy az adatot a sínre tegye. Ez egy további megszorítás azon kívül (és független attól), hogy 15 ns szükséges azután, hogy a cím stabil.

T_{MH} és T_{RH} azt mondja meg, hogy mennyi időre van szükség ahhoz, hogy $\overline{\text{MREQ}}$ és $\overline{\text{RD}}$ negáltra változzanak azután, hogy az adatokat beolvasták. Végül, T_{DH} azt mondja meg, hogy a memóriának mennyi ideig kell még tartania az adatokat a sínen azután, hogy az RD jelet negálták. Ami a példaként tekintett CPU-t illeti, a memória eltávolíthatja az adatokat a sínről, amint az $\overline{\text{RD}}$ jelet negálták; néhány konkrét CPU esetén az adatokat ennél egy kissé tovább kell stabilan tartani.

Szeretnénk felhívni a figyelmet arra, hogy a 3.38. ábra nagymértékben leegyszerűsített változata a valódi időzítési feltételeknek. A valóságban minden esetben sokkal több kritikus időtartamot adnak meg. Mindazonáltal jól érzékelteti a szinkron sín működését.

Utolsó megjegyzésünk, hogy a vezérlőjelek aktív állapota lehet alacsony vagy magas. A sín tervezőjére van bízva, hogy meghatározza, melyik a kényelmesebb, de a választás alapvetően tetszőleges. Úgy tekinthető ez, mint annak a hardvermegfelelője, hogy egy programozó megválaszthatja, hogy a szabad mágneslemez blokkokat egy bittérképben 0-kkal vagy 1-ekkel ábrázolja. A felülvonással jelölt vezérlőjelek (például $\overline{\text{MREQ}}$) aktív állapota a logikai alacsony szint.

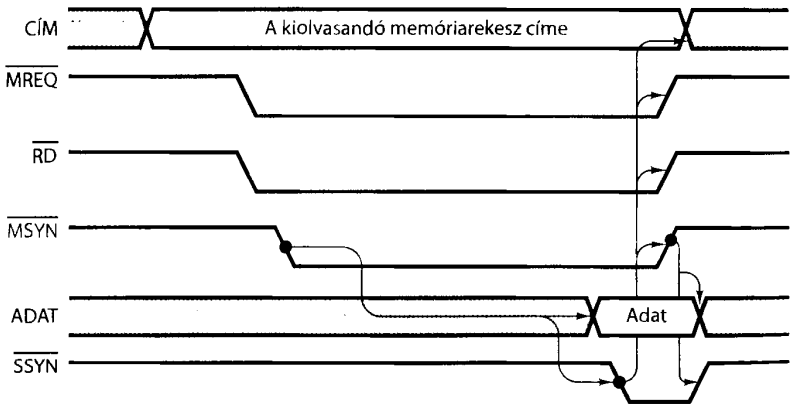
Az aszinkron sín

Bár a szinkron sínekkel diszkrét időintervallumaik miatt egyszerűbben dolgozhatunk, mégis van néhány problémájuk. Például minden a sínciklus többszörösével működik. Ha mondjuk a CPU végre tudna hajtani egy átvitelt 3,1 órajelciklus alatt, azt meg kell nyújtani 4,0 ciklusra, mert a tört ciklusok nem megengedettek.

Ami még rosszabb, ha kiválasztották a sín frekvenciáját, majd elkészítették a memória- és a B/K kártyákat, már nehéz a jövőben bekövetkező technológiai változások előnyeit kihasználni. Tegyük fel, hogy néhány évvel azután, hogy a 3.38. ábrán látható rendszer elkészült, új memórialapkák jelennek meg, 8 ns elérési idővel 15 ns helyett. Ez megszabadíthatna a várakozási állapotoktól, ami meggyorsíthatná a gépet. Azután mondjuk 4 ns-os memóriák jelennének meg. Ez azonban már nem hozna további nyereséget a teljesítményben, mert a memóriából olvasás minimális ideje ebben a tervben 2 órajelciklus.

Kissé másképpen megfogalmazva ugyanezt, ha egy sínen az eszközök egy heterogén halmaza található, egyesek gyorsak, míg mások lassúk. A gyors eszközök nem tudják hasznosítani a képességeiket, mivel a sín sebességét a leglassabbhoz kell igazítani.

A heterogén technológiával készült eszközöket kezelni tudjuk, ha áttérünk az aszinkron sínre, amelyben nincs órajel, ahogyan az a 3.39. ábrán látható. Ahelyett, hogy mindent az órajelhez kötnénk, a következőképpen járunk el: miután a sínmester beállította a címet, az $\overline{\text{MREQ}}$, az $\overline{\text{RD}}$ jeleket és minden mást, amire még szükség van, beállít egy speciális vezérlőjelet, amelyet $\overline{\text{MSYN}}$ (Master SYNchronization)



3.39. ábra. *Aszinkron sín működése*

mesterszinkronizációnak nevezünk. Amint a szolga ezt meglátja, elvégzi a munkát, amilyen gyorsan csak tudja. Amikor elkészült, beállítja az SSYN (Slave SYNchronization) szolgaszinkronizáció jelet. Amint a mester észleli hogy beállították az SSYN jelet, megtudja, hogy a rendelkezésére állnak az adatok, ezért tárolja azokat, majd negálja a címvezetékeket, az MREQ, az RD és a MSYN jelekkel együtt. Amint a szolga észleli MSYN negálását, megtudja, hogy a ciklus véget ért, ezért ő is negálja az SSYN jelet, ezzel visszaérkeztünk a kiindulási állapothoz, amikor minden jel negált, és az eszközök várakoznak a következő sínmesterre.

Az aszinkron sínek időzítési diagramján (de gyakran a szinkron sínekén is) nyilakat használnak, hogy az előidéző eseményeket és a hatásaikat jelöljék, mint ahogyan a 3.39. ábrán is tettük. Az MSYN jel hatására megjelennek az adatbitek a sínen, valamint a szolga beállítja az SSYN jelet. Az SSYN jel a címvezetékek, MREQ, RD, és MSYN jelek negálását eredményezi. Végül az MSYN jel negálása az SSYN negálását okozza, amellyel végződik a beolvasás és a rendszer visszatér eredeti állapotába.

Az ilyen módon egymáshoz kapcsolódó jelek halmazát hívják **teljes kézfogás**nak (**full handshake**). A lényeg négy eseményt takar:

1. MSYN beállítása.
2. SSYN beállítása válaszul MSYN beállítására.
3. MSYN negálása válaszul SSYN beállítására.
4. SSYN negálása válaszul MSYN negálására.

Világosnak kell lennie mindenki számára, hogy a teljes kézfogás független az időzítéstől. Minden eseményt egy megelőző esemény vált ki, nem egy újabb órajel. Ha egy adott mester-szolga pár lassú, az semmilyen befolyással sem bír a következő mester-szolga párra, amely sokkal gyorsabb lehet.

Reméljük, világos, hogy milyen előnyei vannak az aszinkron sínnek, azonban tény, hogy a legtöbb sín szinkron. Ennek az az oka, hogy egy szinkron rendszert könnyebb megépíteni. A CPU csak beállítja a szükséges jeleket és a memória csak válaszol. Nincs visszacsatolás (kiváltó esemény és hatás), de ha a komponenseket megfelelően válogatták össze, minden működni fog kézfogás nélkül is. A szinkron sín technológiájába sok befektetés is történt.

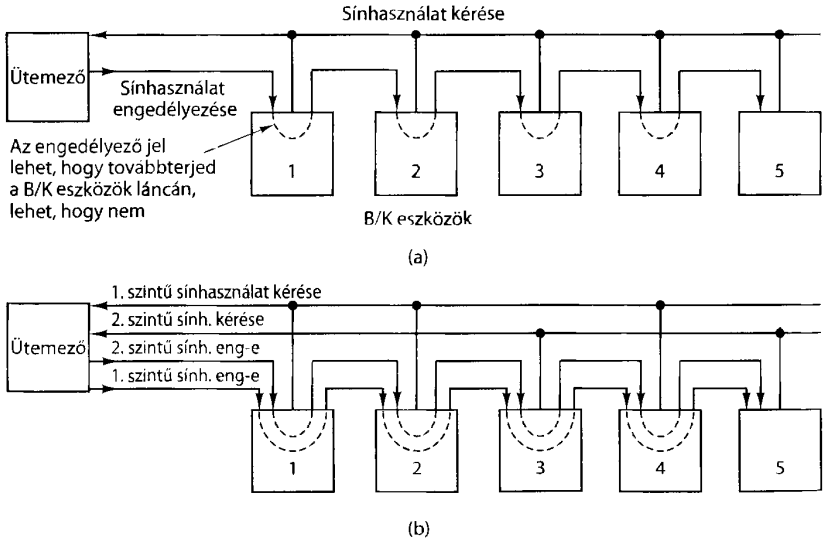
3.4.5. Sínütemezés

Eddig hallgatólagosan feltételeztük, hogy egyetlen sínmester van, mégpedig a CPU. A valóságban, a B/K lapkáknak is sínmesterekké kell válniuk, hogy olvashassák vagy írhassák a memóriát, és hogy megszakításokat kérhessenek. A koprocesszoroknak (társ processzoroknak) is szükségük lehet rá, hogy sínmesterekké váljanak. Felmerül a következő kérdés: „Mi történne, ha két vagy több eszköz egyszerre szeretne sínmester lenni?" Az a válasz, hogy **sínütemező (bus arbitration)** mechanizmusra van szükség, hogy elkerülhessük a káoszt.

Az ütemező (kiosztási) mechanizmus lehet centralizált vagy decentralizált. Először tekintsük át a centralizált mechanizmust. Egy különlegesen egyszerű ütemező mechanizmus látható a 3.40. (a) ábrán. Ezen a rajzon egyetlen sínütemező dönti el, ki következik. Sok CPU-ba be van építve a sínütemező, de időnként egy külön lapkára van ehhez szükség. A sín egyetlen huzalozott-VAGY kérés vezetéket tartalmaz, amelyen egyszerre több eszköz is adhat jelet. A sínütemező számára nincs semmilyen mód arra, hogy megtudja hány eszköz kéri a sínt. Az egyetlen dolog, amit meg tud különböztetni, hogy van-e kérés, vagy nincs.

Amikor a sínütemező egy sínkérést észlel, beállítja a használati engedély jelet a sínhasználatot engedélyező vonalon. Ez a vezeték sorban végigfut az összes B/K eszközön, mint egy olcsó karácsonyfaégő-füzéren. Amikor a sínütemezőhöz fizikailag legközelebbi eszköz meglátja az engedélyezést, ellenőrzi, hogy adott-e ki kérést. Ha igen, átveszi a sín irányítását, és nem továbbítja az engedélyt. Ha nem adott ki kérést, továbbítja az engedélyt a sorban következő eszköznek, amely ugyanígy viselkedik, és így tovább, amíg valamelyik eszköz nem fogadja az engedélyt, és át nem veszi a sínmester szerepét. Ezt a módszert **láncolás**nak (**daisy chaining**) nevezik. Megvan az a tulajdonsága, hogy lényegileg minden eszközhöz prioritást rendel, ami attól függ, hogy milyen közel van az eszköz a sínütemezőhöz. A legközelebbi eszköz nyer.

Azért, hogy el lehessen kerülni az ütemezőtől mért távolságon alapuló implicit prioritásokat, sok sínnek több prioritási szintje van. Minden szintnek van vezetéke a sínhasználat kérésére és sínhasználat engedélyezésére. A 3.40. (b) ábrán láthatónak két szintje van, 1 és 2 (az igazi síneknek gyakran 4, 8 vagy 16 szintje van). Minden eszköz valamelyik szinthez csatlakozik úgy, hogy a leginkább időkritikus eszközök



3.40. ábra. *(a) Centralizált egyszintű sínütemező láncolt engedélyezéssel. (b) Ugyanaz a sínütemező két prioritási szinttel*

a nagyobb prioritásúhoz kapcsolódnak. A 3.40. (b) ábrán az 1, 2 és 4 eszközök az 1 prioritási szinthez, míg a 3 és 5 eszközök a 2 prioritási szinthez kapcsolódnak.

Ha több prioritási szinten is érkezik kérés, az ütemező a sínhasználat engedélyét csak a legnagyobb prioritásúnak fogja megadni. Az azonos prioritású eszközök között pedig a láncolás működik. A 3.40. (b) ábrán konfliktus esetén a 2 eszköz megelőzi a 4 eszközt, amely viszont megelőzi a 3 eszközt. Az 5 eszköz prioritása a legkisebb, mivel a legkisebb prioritású lánc legvégén található.

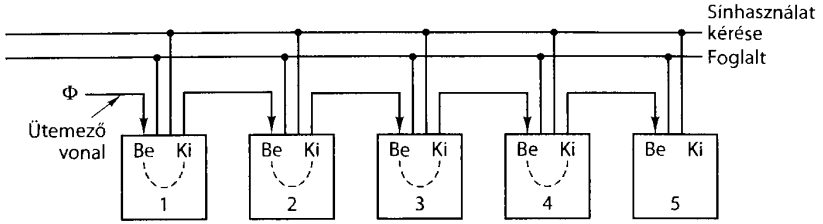
Mellékesen megjegyezzük, hogy technikailag nem szükséges a 2 prioritási szinthez tartozó engedélyező vezetéket végigvezetni az 1 és 2 eszközökön, mivel ezek nem tudnak ilyen kérést küldeni. Azonban a megvalósítás szempontjából kényelmesebb átvezetni az összes engedélyező vezetéket az összes eszközön, mint speciális huzalozást tervezni, amely attól függ, hogy melyik eszköznek milyen prioritása van.

Bizonyos sínütemezőknek van egy harmadik vezetéke is, amelyet akkor állít be egy eszköz, ha fogadta az engedélyt, és átvette a sín irányítását. Amint ezt a nyugtázó vezetéket beállították, a kérés és engedélyezés vezetékeket már lehet negálni. Ennek következtében további eszközök kérhetik a sínt, mialatt az első eszköz még használja. Mire az aktuális adatátvitel befejeződik, a következő sínmester már ki van választva. Azonnal el is kezdhet dolgozni, amint a nyugtázó vezetéket negálták, ezzel kezdődhet a következő sínütemezési forduló. Ez az elrendezés egy újabb vezetéket igényel, és több logikát is kell beépíteni az egyes eszközökbe, azonban a sínciklusok jobb kihasználását eredményezi.

Azokban a rendszerekben, ahol a memória a rendszersínen található, a CPU-nak szinte minden órajelciklusban versenyeznie kell a sínen lévő összes B/K eszközzel. Az egyik megoldás az, hogy a CPU a legkisebb prioritást kapja, tehát csak akkor kapja meg a sínt, ha senki más nem kéri. Az elv az, hogy a CPU mindig tud várni, viszont egy B/K eszköznek gyakran gyorsan meg kell kapnia a sínt, egyébként elveszíti a bejövő adatokat. A nagy sebességgel forgó mágneslemezek nem tudnak várakozni. A legtöbb modern számítógéprendszerben ezt a problémát úgy kerülik el, hogy a memóriát egy B/K eszközöktől elkülönített sínre helyezik, így nem kell versenyezniük a sín tulajdonlásáért.

A decentralizált sínütemezés szintén megvalósítható. Például, egy számítógépnek 16 prioritással rendelkező vezetéke lehet a sínhasználat kérésére. Amikor egy eszköz szeretné a sínt használni, beállítja a vezetékét a használat kérésére. Minden eszköz az összes ilyen vezetéket figyeli, így minden sínciklus végén, minden eszköz tudja, hogy ő volt-e a legnagyobb prioritású kérő eszköz, és hogy megkapja-e az engedélyt a sínhasználatra a következő ciklusban. Összehasonlítva a centralizált ütemezéssel, ez az ütemezési módszer több sínvezetéket igényel, viszont megtakarítható az ütemező várható költsége. Ezenkívül az eszközök száma nem lehet több a használati kérés vezetékek számánál.

Egy másikfajta decentralizált sínütemezés látható a 3.41. ábrán, amely mindössze három vezetéket használ, függetlenül attól, hogy hány eszköz van. Az első huzalozott-VAGY vonal a sínhasználat kérésére szolgál. A második vonalat FOGLALT-nak (BUSY) nevezzük, és az aktuális mester állítja be. A harmadik vonal szolgál magára az ütemezésre. Ez végig van láncolva az összes eszközön. A lánc feje mindig be van állítva úgy, hogy a tápegység az óra vezetékére van kötve.



3.41. ábra. *Decentralizált sínütemezés*

Amikor egyetlen eszköz sem kéri a sínt, a beállított (magas) ütemezési jel végighalad a láncon minden eszközön keresztül. Ahhoz, hogy a sínt lefoglalja egy eszköz, először megnézi, hogy a sín szabad-e, azaz a kapott ütemező jel, az IN magas-e (be van állítva). Ha az IN negált, az eszköz nem lehet sínmester, és ezért negálja az OUT jelet. Azonban ha az IN be van állítva, az eszköz negálja az OUT kimenetet; ami azt eredményezi, hogy az utána következő szomszédja negált IN bemenetet lát, és negálja az OUT kimenetét. Ugyanezért az utána következő összes eszköz negált IN bemenetet lát, és negálja az OUT kimenetét. Amikor azután leülepszik a por, egyetlen eszköz lesz, amelynél az IN magas és az OUT negált. Ez az eszköz lesz a sínmester, aki beállítja a FOGLALT és az OUT jeleket, majd elkezdi az adatátvitelét.

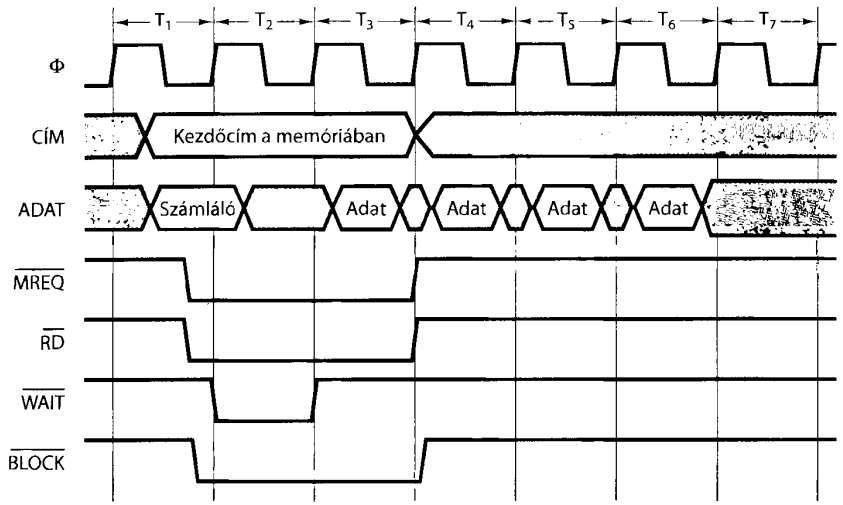
Némi gondolkodás után kiderül, hogy az a balról legelső eszköz, amely kéri a sínt, megkapja. Ezért ez a séma hasonló az eredeti láncoláshoz, de itt nincs ütemező, ezért ez olcsóbb, gyorsabb, és nem kell számolni az ütemező meghibásodásával.

3.4.6. Sínműveletek

Eddig közönséges sínciklusokkal foglalkoztunk, amelyekben egy mester (általában a CPU) adatokat olvas be a szolgától (tipikusan a memóriából), vagy adatokat ír ki számára. Valójában többféle más sínciklus is létezik. Most ezek közül veszünk szemügyre néhányat.

Rendszerint egyszerre egy szó kerül továbbításra. Azonban, ha gyorsítótárat használnak, egyszerre egy egész gyorsítósor (16 egymást követő 32 bites szó) átvitelére van szükség. A blokkátvitel gyakran hatékonyabb lehet, mint egyes szavak átvitele. Amikor egy blokkátvitel megkezdődik, a sínmester megadja a szolgának az átvinni kívánt szavak számát, például úgy, hogy a szavak számát felírja az adatvezetékekre a T_1 periódus alatt. Ezután ahelyett, hogy a szolga egyetlen szót adna vissza, a továbbiakban minden egyes órajelperiódusban vissza fog adni egy-egy szót, amíg csak a számláló ki nem merül. A 3.42. ábra a 3.38. (a) ábra egy módosítását mutatja, ebben egy új vezérlőjel, a BLOCK beállítása jelzi, hogy blokkátvitelre van szükség. A bemutatott példában egy négy szóból álló blokk beolvasása történik, amely a korábbi 12 helyett csak 6 órajelciklust igényel.

Másfajta sínciklusok is vannak. Például, egy többprocesszoros rendszerben, amelyben két vagy több CPU kapcsolódik rá ugyanarra a sínre, gyakran szükséges



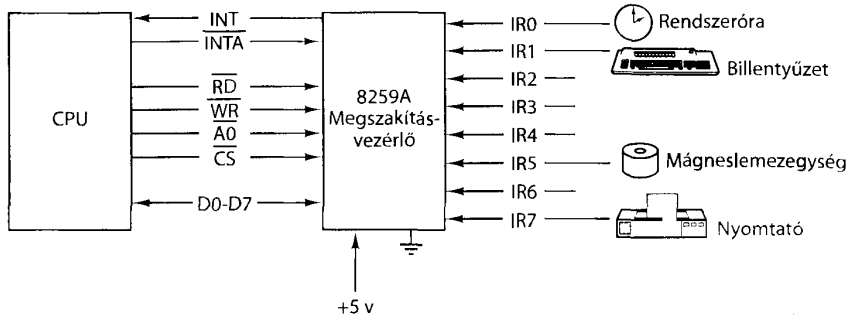
3.42. ábra. *A blokkátvitel időzítése*

annak biztosítása, hogy egyszerre csak egy CPU használhasson valamilyen kritikus adatstruktúrát a memóriából. Erre a célra egy tipikus megoldás, ha van a memóriában egy olyan változó, amely 0, ha egyik CPU sem használja az adatstruktúrát, és 1, ha használatban van. Ha egy CPU szeretne hozzáférni az adatstruktúrához, ki kell olvasnia a változót, és ha az értéke 0, be kell állítania 1-re. Baj akkor történik, ha egy szerencsétlen véletlen következtében két CPU is kiolvassa a változót két egymást követő sínciklusban. Ha mind a kettő azt látja, hogy a kontrollváltozó értéke 0, akkor mindegyik átállítja 1-re, és azt gondolja, hogy ő az egyetlen CPU, amely használja. Egy ilyen eseménysorozat káoszhoz vezet.

Ezt megelőzendő, a többprocesszoros rendszerekben gyakran van egy olvasás-módosítás-visszaírás sínművelet, amely bármelyik CPU-nak lehetővé teszi, hogy beolvasson egy szót, megvizsgálja, és visszaírja a memóriába anélkül, hogy közben elengedné a sínt. Ez a fajta sínművelet kizárja, hogy a CPU egy versenytársa használni tudja a sínt, és így befolyásolja az első CPU működését.

A sínciklusok egy másik fontos típusa a megszakítások kezelésére szolgál. Amikor a CPU utasít egy B/K eszközt, hogy tegyen meg valamit, általában megszakítást vár visszajelzésként, amikor a feladat elkészült. A megszakításkérés jelzéséhez szükség van a sínre.

Mivel több eszköz kérhet megszakítást egyszerre, hasonló ütemezési problémák merülnek fel itt is, mint a közönséges sínciklusoknál. A szokásos megoldás, hogy az egyes eszközökhöz prioritásokat rendelünk, egy központi ütemező használatával pedig a leginkább időkritikus eszközöknek magas prioritást adunk. Szabványos megszakításvezérlő lapkák vannak már, és ezeket széles körben használják is. Az IBM PC és minden követője a 3.43. ábrán látható Intel 8259A lapkát használja.



3.43. ábra. *A 8259A megszakításvezérlő használata*

Maximum nyolc B/K vezérlőlapka kapcsolható közvetlenül a 8259A nyolc IRx megszakításkérés (Interrupt Request) bemenetére. Amikor valamelyik eszköz egy megszakítást szeretne kérni, egy jelet küld saját IRx vonalára. Amikor a 8259A egy vagy több bemenetére jel érkezik, akkor a 8259A beállítja az INT, megszakítás (INTerrupt) kimenetét, amely közvetlenül vezérli a CPU megszakítás kivezetését. Ha a CPU képes fogadni a megszakítást, impulzust küld vissza a 8259A számára az INTA megszakításkérés fogadása (INTerrupt Acknowledge) vezetéken. Ezen a ponton a 8259A felteszi az adatsínre a bemenet sorszámát, hogy megmondja, melyik bemenete váltotta ki a megszakítást. Ez a művelet egy speciális sínciklust igényel. A CPU-hardver ezután ezt a számot indexként használva egy pointer táblázatból, amelyet **megszakításvektor táblának (interrupt vectors)** neveznek, a megadott indexű elemet kiválasztja, ez lesz annak az eljárásnak a kezdőcíme, amelyet futtatni kell a megszakítás kiszolgálásához.

A 8259A lapkának van néhány belső regisztere, amelyet a CPU tud olvasni és írni normál sínciklusok felhasználásával, valamint RD olvasás (ReaD), WR írás (WRite), CS lapkakiválasztás (Chip Select) és A0 kivezetései. Amikor a szoftver kezelte a megszakítást és készen áll a következő fogadására, egy speciális értéket ír a 8259A egyik regiszterébe, amely negálja az INT kimenetet, hacsak nincs másik várakozó megszakítás. Ezekbe a regiszterekbe lehet írni akkor is, ha a 8259A üzemmódjai közül szeretnénk kiválasztani valamelyiket, le akarunk tiltani egyes megszakításokat vagy engedélyezni szeretnénk más funkciókat.

Ha több mint nyolc B/K eszköz van jelen, a 8259A-kat sorba lehet kapcsolni. Szélsőséges esetben mind a nyolc input lába nyolc további 8259A kimenetére van kapcsolva, ez lehetővé teszi legfeljebb 64 B/K eszköz összekapcsolását egy kétállapotú megszakításhálózatban. A 8259A-nak van néhány kivezetése, amelyek a sorbakapcsolást kezelik, ezeket most mellőztük az egyszerűség kedvéért.

Bár ezzel még egyáltalán nem merítettük ki a síntervezés témáját, a fenti ismeretek elegendő hátteret adnak, hogy megértsük a sínek működésének lényegét, azt, hogy miként működnek együtt a központi egységek és a sínek. Most haladjunk az általánostól a speciális felé, és vizsgáljunk meg néhány valódi központi egységet és a hozzájuk kapcsolódó sínеket.

3.5. Példák CPU lapkákra

Ebben a fejezetben a Pentium 4, az UltraSPARC III és a 8051 CPU lapkákat vizsgáljuk meg bizonyos részletességgel, hardverszinten.

3.5.1. Pentium 4

A Pentium 4 az eredeti IBM PC-ben használt 8088 CPU közvetlen leszármazottja. Az első Pentium 4-et 2000 novemberében mutatták be, mint egy 42 millió tranzisztort tartalmazó CPU-t, amely 1,5 GHz frekvencián működött és a vezetékeinek szélessége 0,18 μm (mikron). A vezetékek szélessége mutatja, hogy a tranzisztorok között milyen szélesek a vezetékek (és maguknak, a tranzisztoroknak a mérete is ekkora). Minél keskenyebbek a vezetékek, annál több tranzisztor fér a lapkára. Moore törvénye alapvetően arról szól, hogy a mérnökök képesek folyamatosan csökkenteni a vezeték szélességet. A kisebb szélesség nagyobb órajelfrekvenciát is megenged. Összehasonlításul az emberi haj átmérője 20 és 100 mikron között van, a szőke haj finomabb szálú, mint a fekete.

A következő három év folyamán az Intel nagy gyakorlatra tett szert a gyártási folyamatban, és továbbfejlesztette az áramkört, amelynek már 55 millió tranzisztora volt, 3,2 GHz frekvencián működött, a vezeték szélessége pedig 0,09 mikron volt. Bár a Pentium 4 nagyon messze esik a 29 000 tranzisztoros 8088-tól, mégis teljesen kompatibilis vele, és módosítás nélkül tud futtatni 8088-as bináris programokat (a többi közbeeső processzorról nem is beszélve).

Szoftverszempontból a Pentium 4 egy tisztán 32 bites gép. Ugyanolyan felhasználói szintű ISA-szolgáltatásokkal, mint a 80386, 80486, Pentium, Pentium II, Pentium Pro és a Pentium III központi egységek, ugyanazokat a regisztereket tartalmazza, ugyanazokat a gépi utasításokat hajtja végre, ezenkívül tartalmazza a teljes IEEE 754 lebegőpontos aritmetikai szabvány lapkára integrált megvalósítását.

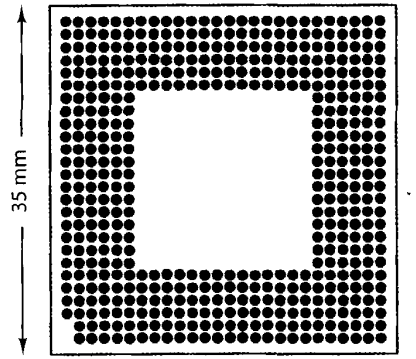
Hardverszempontból azonban a Pentium 4 egy részben 64 bites számítógép, mivel 64 bites egységekben tudja az adatokat mozgatni a memóriába, illetve a memóriából. Bár a programozó nem figyelheti meg közvetlenül ezeket a 64 bites átvitelcket, mégis gyorsabbá teszi a számítógépet, mintha az egy tisztán 32 bites számítógép lenne.

Belsőleg, a mikroarchitektúra szintjén, a Pentium 4 azonban alapvetően különbözik minden elődjétől. Közvetlen elődei – a Pentium II, a Pentium Pro és a Pentium III – mind ugyanazt a belső mikroarchitektúrát (P6) használták, amely csak sebességben és néhány apró dologban tért el. Ezzel szemben a Pentium 4 új mikroarchitektúrát használ, amelynek a neve NetBurst, és amely jelentősen eltér a P6-tól. Csővezetéke több állapottal rendelkezik, két ALU (Aritmetikai és Logikai Egység) található benne (mindegyik a CPU órajel-frekvenciájának kétszeresével működik, így egy ciklus alatt két műveletet végezhet), továbbá támogatja a hiperszálak használatát. Ez utóbbi tulajdonság két regiszterkészletet és bizonyos más belső erőforrásokat biztosít, amelyek lehetővé teszik, hogy a Pentium 4 nagyon gyorsan átváltson egyik programról egy másikra, mintha csak a számítógép két fizikai CPU-t tartalmazna. A mikroarchitektúrát a 4. fejezetben fogjuk megvizsgálni. Mint ahogyan az elődei, a Pentium 4 is végrehajthat több utasítást egyszerre, ezért szuperskaláris számítógép.

Bizonyos Pentium 4 modellek kétszintű, mások háromszintű gyorsítótárral rendelkeznek. Minden modellben van egy 8 KB méretű lapkára integrált SRAM elsőszintű (L1) gyorsítótár. A Pentium III-mal ellentétben, amelyben az L1 gyorsítótár csak nyers bájtokat tartalmaz, a Pentium 4 továbblép. Amikor gépi utasításokat tölt be a memóriából, azok mikroutasításokra konvertálódnak, hogy végrehajthatók lehessenek a Pentium 4 RISC-magjával. A Pentium 4 L1 gyorsítótára akár 12 000 dekódolt mikroutasítást is képes tárolni, így nem szükséges azokat ismételten dekódolni. A második szintű gyorsítótár legfeljebb 256 KB méretű lehet a régebbi, és 1 MB lehet az újabb modellek esetében. Semmit sem dekódol, nyers bájtokat tárol a memóriából az L2 gyorsítótárban. Utasítások és adatok keverékét tartalmazhatja. A Pentium 4 Extreme Edition tartalmaz még egy 2 MB méretű harmadik szintű gyorsítótárat a hatékonyság további növelésére.

Mivel minden Pentium 4 lapkának legalább két gyorsítótár szintje van, egy multiprocesszoros rendszernél probléma merül fel, ha az egyik processzor módosít egy memóriaszót a gyorsítótárában. Ha a másik CPU megpróbálja ezt a szót beolvasni a memóriából, egy már érvénytelen értéket fog kapni, mivel a módosított gyorsítótár szavak nem íródtak vissza azonnal a memóriába. Hogy a memória konzisztens állapotát fenntartsák, a multiprocesszoros rendszer mindegyik CPU-ja olyan memóriaszavak címe után **szimatol** a sínen, amelyek a gyorsítótárában vannak. Ha meglát egy ilyen hivatkozást, közbelép, és a gyorsítótárából megadja a kért adatot, mielőtt még a memóriának erre esélye lenne. A szimatolást a 8. fejezetben fogjuk tanulmányozni.

Két elsődleges külső sínt használnak a Pentium 4 rendszerekben, mindkettő szinkron. A memóriasínt a fő (S)DRAM, a PCI sínt pedig a B/K eszközök elérésére használják. Időnként egy **hagyományos (legacy bus)**, korábbi fajta sín is kapcsolódik a PCI sínhez, ami megengedi a régi fajta perifériákhoz történő csatlakozást.



3.44. ábra. *A Pentium 4 fizikai lábkiosztása*

Van egy lényeges különbség a Pentium 4 és az elődei között: a tokozása. Minden modern lapka problémája a felvett energia és a fejlesztett hő. A Pentium 4 a frekvenciától függően 63 W és 82 W közötti teljesítményt használ. Következésképpen az Intel folyamatosan keresi azokat a módszereket, amelyekkel kezelni tudja a CPU által fejlesztett hőt. A Pentium 4 egy 35 mm oldalú, négyzet alakú tokban kerül forgalomba. A tok 478 lábat tartalmaz alul, amelyből 85 az energiaellátást biztosítja, 180-at pedig földeltek a nagyfrekvenciás zaj mérséklése érdekében. A lábakat 26 × 26-os négyzetrácsban rendezték el, a középső 14 × 14 láb hiányzik. Két láb az egyik sarokban szintén hiányzik; ezzel megelőzhető, hogy a lapkát helytelenül dugják be a foglalatába. A fizikai lábkiosztás a 3.44. ábrán látható.

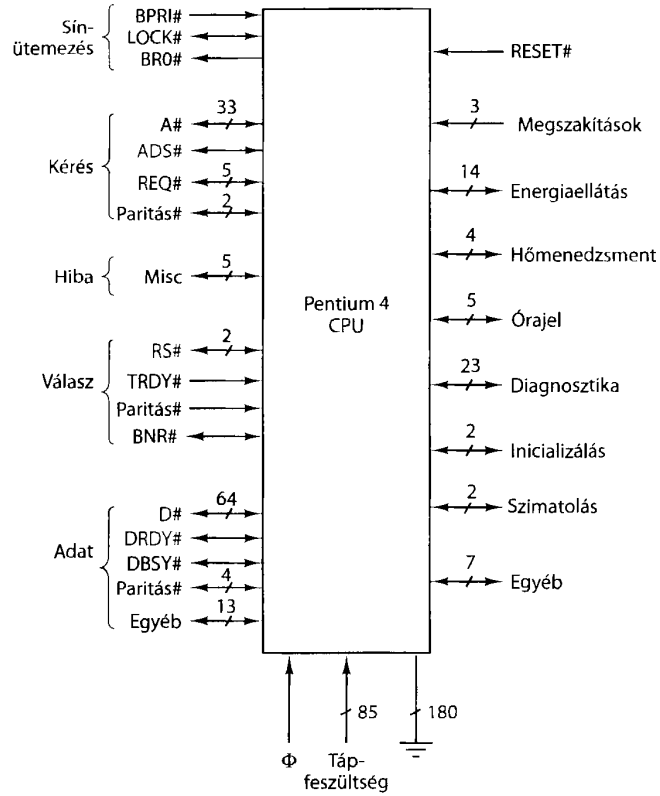
A lapkára egy tartókeretet szereltek a hűtőborda számára, amely elvezeti a hőt, és egy ventilátort a hűtésre. Hogy képet kapjunk arról, hogy mi is a probléma, kapcsoljunk be egy 60 W-os izzólámpát, és tegyük a kezünket közel hozzá (de ne érintsük meg). Ekkora mennyiségű hőt kell folyamatosan szétsugározni (disszipálni). Következésképpen, ha egy Pentium 4 megéri, hogy már mint CPU nem hasznosítható, még mindig jól beválhat kempingfőzőnek.

A fizika törvényei szerint, ami sok hőt sugároz szét, annak sok energiát kell felvennie. Egy hordozható számítógépben, behatárolt energiamennyiséget tároló akkumulátorokkal, a nagy energiafelhasználás nem kívánatos, mivel gyorsan kimeríti az akkumulátorokat. Ennek a kérdésnek a megoldására az Intel kifejlesztett egy módszert, amellyel a CPU alvás módba kapcsolható, amikor nincs végrehajtandó utasítás, vagy mély alvás módba, amikor ez valószínűleg hosszabb ideig így marad. Öt állapota van a teljesen aktívtól a mély alvásig. A közbenső állapotokban bizonyos funkcionalitások (mint például a gyorsítótár-szimatolás és a megszakítások kiszolgálása) engedélyezettek, a többi azonban tiltott. Mély alvásban a gyorsítótárak és a regiszterek tartalma megőrződik, viszont az óra és más belső eszközök leállnak. Mély alvás esetén a felébresztéshez külső áramköri jel szükséges. Az nem ismeretes, hogy egy Pentium 4 tud-e álmodni, amikor mély alvás állapotban van.

A Pentium 4 logikai lábkiosztása

A Pentium 4 478 érintkezőjéből 198 jel, 85 tápfeszültség (néhány különböző feszültségre), 180 földelés és 15 tartalék kivezetés a jövőbeli fejlesztések céljaira. Néhány logikai jel két vagy több érintkezőt is lefoglal (mint például a kért memóriacím), így végül csak 56 különböző jel van. Bizonyos mértékig egyszerűsített lábkiosztás látható a 3.45. ábrán. Az ábra bal oldalán látható a memóriasínjelek öt legfontosabb csoportja; a jobb oldalon pedig különböző egyéb jelek. A csupa nagybetűvel írt nevek az aktuális Intel-elnevezések. A kis- és nagybetűvel írt nevek pedig gyűjtőnevek, több egymással kapcsolatban álló jel számára.

Az Intel olyan elnevezéskonvenciót alkalmaz, amelyet fontos megértenünk. Mivel manapság minden lapkát számítógéppel terveznek, emiatt szükséges, hogy minden jel elnevezése egy ASCII szöveggel reprezentálható legyen. Annak jelzésére, hogy egy jelet alacsony szintre kell beállítani, felülvonást használni meglehetősen nehéz, ehelyett az Intel egy # szimbólumot tesz a név után. $\overline{\text{BPRI}}$ helyett BPRI# elnevezést használnak. Amint az ábrából látszik, a legtöbb Pentium 4 jel alacsony feszültségértéke az aktív.



3.45. ábra. *A Pentium 4 logikai lábkiosztása. A tisztán nagybetűs elnevezések az Intel hivatalos elnevezései, a kis- és nagybetűket is tartalmazó nevek az ezekhez kapcsolódó jelek, illetve elnevezések*

Vizsgáljuk meg a jeleket, kezdve a sín jeleivel. A jelek első csoportja a sínhasználat jogának megszerzésére (azaz sínütemezésre) szolgál. A BRI# a normál sínkérés jel. A BPRI# lehetővé teszi, hogy egy eszköz magas prioritású igényt támasszon, amely megelőzi a normál kéréseket. LOCK# beállításával egy CPU több ciklusra lefoglalhatja a sínt, és más eszköz addig nem kaphatja meg, amíg nem negálja LOCK#-ot.

Amint a CPU vagy egy másik sínmester megszerezte a sínhasználat jogát, egy újabb sínkérést továbbíthat a következő jelcsoport segítségével. A címek 36 bitesek, de az alsó 3 bitnek mindig 0-nak kell lennie, ezért ezekhez nem tartoznak kivezetések, tehát A# 33 érintkezőt jelent. Minden adatátvitel 8 bájtos és 8 bájt határra igazított (nyolccal osztható címen kezdődik). 36 címbit segítségével 2^{36} memóriabájt címezhető meg, ami 64 GB.

Amikor a cím kikerül a sínre, az ADS# jel beáll, ez tudatja a céleszközzel (például a memóriával), hogy a cím érvényes. A sínciklus típusa (például egy szó olvasása vagy egy blokk írása) a REQ# vonalakon továbbítódik. A két paritásjel védi az A# és a REQ# jeleket.

Az öt hibavonalat használják a lebegőpontos, a belső, az eszközellenőrző (hardver-) és más hibák jelzésére.

A válaszcsoport olyan jeleket tartalmaz, amelyekkel egy szolga vissza tud jelezni a mesternek. Az RS# a státuskódot tartalmazza. A TRDY# jelzi, ha a szolga (céleszköz) készen áll adatok fogadására a mestertől. Ezekhez a jelekhez is tartozik paritás-ellenőrzés.

Az utolsó sínjelcsoport az aktuális adatátvitel megvalósítására szolgál. D#-n egyszerre 8 bájtos adat továbbítása lehetséges. Amikor a sínen vannak, a DRDY# jel beállítása jelzi az adatok jelenlétét. A DBSY# jel jelzi, hogy a sín foglalt. Paritást használnak az adatok ellenőrzésére is. A további adatátviteli jelek az értékek tárolásával és más hasonló dolgokkal foglalkoznak.

A RESET# jel a CPU alapállapotba helyezésére szolgál katasztrofális esemény bekövetkezésekor, vagy ha a felhasználó megnyomja a reset gombot a PC elején.

A Pentium 4 konfigurálható úgy, hogy a megszakításokat ugyanúgy használja, mint a 8088 (a visszafelé kompatibilitás érdekében), vagy használhat egy új megszakításrendszert is, amely egy **APIC (Advanced Programmable Interrupt Controller, fejlett programozható megszakításvezérlő)** eszközre épül.

A Pentium 4 több különböző tápfeszültségről is tud működni, de tudnia kell, hogy melyikről. Az energiaellátás-jeleket automatikus tápfeszültség-választásra – hogy tudassák a CPU-val, hogy a tápfeszültség stabil – és más tápellátással kapcsolatos dolgokra használják. A különböző alvásállapotokat is itt kezelik, mivel az alvás főként energiatakarékossági okokból valósul meg.

A bonyolult energiamenedzsment ellenére a Pentium 4 nagyon felforrósodhat. A hőmenedzsmentcsoport foglalkozik a hőmérséklet szabályozásával, és lehetővé teszi, hogy a CPU jelezze környezetének, ha fennáll a túlmelegedés veszélye. A kivezetések egyikét a CPU állítja be, ha a belső hőmérséklet eléri a 130 °C-ot (266 °F). Ha a CPU valaha is eléri ezt a hőmérsékletet, talán arról álmodik, hogy visszavonul, és kempingfőző lesz.

Az órajelcsoportnak a rendszersín frekvenciájának meghatározásával kell foglalkoznia. A diagnosztikai csoport a rendszer tesztelésére és nyomkövetésre szolgáló jeleket tartalmaz, amelyek megfelelnek az IEEE 1149.1 JTAG tesztszabványnak. Az inicializálási csoport a rendszer betöltésével (indításával) foglalkozik.

Végül, az egyéb csoport jelek kusza összevisszasága – tartalmaz például olyan jelet, amely a CPU-foglalat foglaltságát jelzi, egy másik a 8088 emulációval kapcsolatos, továbbá mindenféle más célra szolgáló jeleket.

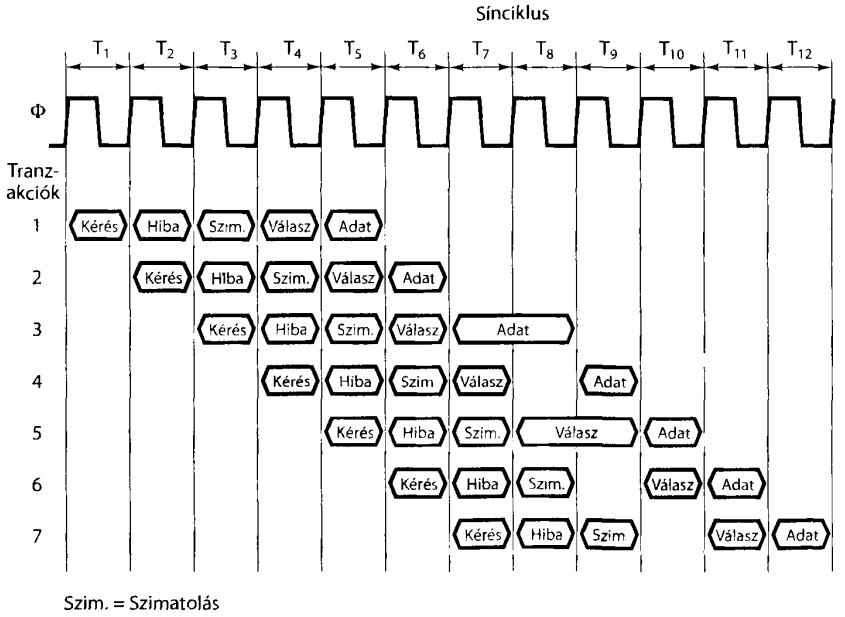
A Pentium 4 memóriasín csővezeték-architektúrája

A modern CPU-k, mint például egy Pentium 4, sokkal gyorsabbak, mint a modern DRAM memóriák. Hogy a CPU ne szenvedjen állandó adathiányban, lényeges, hogy a memóriából a lehetséges legnagyobb teljesítményt hozzuk ki. Ezért a Pentium 4 memóriasínje nagymértékben csővezetékszerű, olyannyira hogy egyszerre nyolc síntranzakció is folyhat. A csővezeték koncepcióját már megismertük a 2. fejezetben a csővezeték-architektúrájú CPU-kal kapcsolatban, azonban a memóriakezelés is kialakítható csővezeték-architektúrájúra.

Ahhoz, hogy a Pentium 4 memóriakezelése is csővezeték-architektúrájú lehessen, a memóriaigényeknek, az ún. **tranzakciók**nak **(transactions)** hat állapotát különböztetik meg:

1. sínütemezési fázis;
2. kérési fázis;
3. hibajelzési fázis;
4. szimatolási fázis;
5. válaszfázis;
6. adatfázis.

Nincs mindegyikre szükség minden egyes tranzakcióhoz. A sínütemezési fázis meghatározza, hogy a potenciális sínmesterek közül ki fog következni. A kérési fázisban lehet a címet a sínre feltenni, és kérni az adatot. A hibajelzési fázis lehetővé teszi, hogy egy sínszolga jelezze, ha paritáshibát talált a címben, vagy valami más baj volt. A szimatolási fázisban az egyik CPU figyelheti a többit; erre csak



3.46. ábra. *A kérések áthaladása Pentium 4 memóriasín csővezetékén*

többprocesszoros rendszerekben van szükség. A válaszfázisban a sínmester megtudhatja, hogy meg fogja-e kapni a kívánt adatokat. Végül, az adatfázisban megérkeznek a CPU által kért adatok.

A Pentium 4 csővezeték-architektúrájú memóriasínjének az a titka, hogy minden fázis különböző sínvezérlő jeleket használ, azaz minden egyes fázis teljesen független a többitől. A szükséges jelek 6 csoportja látható a 3.45. ábra bal oldalán. Például, egy CPU megpróbálhatja megszerezni a memóriasín-mesteri jogot a sínütemezési jelek használatával. Amint a használat jogát megkapta, ezeket a sínvezérlő jeleket elengedi, és a kérési fázishoz tartozó jelek csoportját kezdi el használni. Ezalatt a másik CPU vagy valamelyik B/K eszköz belépheti a sínütemezési fázisba és így tovább. A 3.46. ábra azt mutatja, hogy hogyan lehet folyamatban több tranzakció egyszerre.

A 3.46. ábrán a sínütemezési fázis nem látható, mert erre nem minden esetben van szükség. Például, ha az aktuális sínmester (gyakran a CPU) egy másik tranzakciót szeretne elindítani, nincs szüksége arra, hogy ismét elkérje a sínt. Csak akkor kell kérnie, ha közben a sínmesteri jog átkerült egy másik kérő eszközhöz. Az 1-es és 2-es tranzakció nyilvánvaló: öt fázis öt sínciklusban. A 3-as tranzakció egy hosszabb adatfázist használ, például mert az egy blokkátvitel, vagy mert a megcímzett memória egy várakozó állapotot iktatott be. Ennek következtében a 4-es tranzakció nem kezdheti el az adatfázisát akkor, amikor szeretné. Azt tapasztalja, hogy a DBSY# jel folyamatosan be van állítva, ezért arra vár, hogy a jel szintje az ellenkezőjére változzon. Az 5-ös tranzakcióban azt látjuk, hogy a válaszfázis is több sínciklusen keresztül tarthat, ami késlelteti a 6-os tranzakciót. Végül a 7-es tranzakcióban azt láthatjuk, hogy ha egy buborék kerül a csővezetékbe, az ott is marad, még akkor is, ha közben folyamatosan indulnak új tranzakciók. A napi gyakorlatban azonban valószínűtlen, hogy a CPU minden órajelben egy új tranzakciót kísérel meg indítani, ezért a buborékok nem tartanak olyan sokáig.

3.5.2. UltraSPARC III

Mivel ez a második példának választott CPU lapkánk, most a Sun UltraSPARC áramkörcsaládot fogjuk megvizsgálni. Az UltraSPARC család a Sun 64 bites SPARC CPU sorozata. Ezek a CPU-k teljesen megfelelnek a SPARC Version 9 architektúrának, amelyet a 64 bites központi egységek számára fejlesztettek ki. Sun-munkaállomásokban és -szerverekben használják, de más alkalmazásaik is ismertek. Ez a család tartalmazza az UltraSPARC I, UltraSPARC II és az UltraSPARC III központi egységeket, amelyek architekturálisan nagyon hasonlítanak egymáshoz, és elsődlegesen a kibocsátási időben, az órajel-frekvenciában és néhány új utasításban különböznek egymástól, amelyet minden újabb modellbe beépítettek. A korrektség érdekében az alábbiakban az UltraSPARC III-mal foglalkozunk, de a legtöbb architekturális (azaz gyártási technológiától független) megállapításunk fennáll a többi UltraSPARC-ra is.

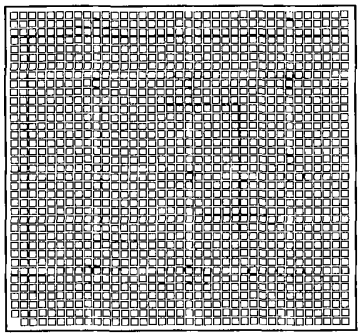
Az UltraSPARC III egy hagyományos RISC gép, amely binárisan teljesen kompatibilis a 32 bites SPARC Version 8 (V8) architektúrával. 32 bites SPARC V8

bináris programokat változtatás nélkül tud futtatni, mivel a SPARC V9-architektúra visszafelé kompatibilis a SPARC V8 architektúrával. Az egyetlen, amiben az UltraSPARC III eltér a SPARC V9 architektúrától, az az új VIS 2.0 utasításkészlet, amelyet 3D grafikus alkalmazásokhoz, valós idejű MPEG-dekódolásra, adattömörítésre, jelfeldolgozásra, Java-programok futtatására és hálózatkezelésre fejlesztettek ki.

Bár az UltraSPARC III-at munkaállomásokban is használják, valójában arra tervezték, hogy a Sun fő üzletágában, a nagy osztott memóriát használó, többprocesszoros szerverekben alkalmazzák, amelyeket az interneten és vállalati intranetekben használnak. Ennek megfelelően, a többprocesszoros rendszerekhez szükséges „ragasztó" legnagyobb része megtalálható minden UltraSPARC III lapka belsejében, amely megkönnyíti nagyszámú CPU összekapcsolását egymással.

Az első UltraSPARC III-at 2000-ben bocsátották ki, 600 MHz-en működött, és 0,18 mikron szélességű alumíniumvezetékeket használt. A lapka 29 millió tranzisztort tartalmaz. Mivel a Sun túl kicsi ahhoz, hogy a legkorszerűbb lapkákat előállító üzemet létesítsen, inkább a lapka tervezésére és a szoftverre koncentrál, a CPU lapkák előállítására pedig külső lapkagyártókkal köt szerződést. Az UltraSPARC III esetében a lapkákat a Texas Instrument állította elő. 2001-ben a TI (Texas Instrument), miután technológiai fejlesztéseket hajtott végre, megkezdte a 900 MHz-es 0,15 mikronos rézvezetékes lapkák készítését az alumíniumvezetékes helyett. 2002-ben a vezetékszélesség 0,13 mikronra csökkent, az órajel-frekvencia pedig 1,2 GHz-re emelkedett. Ezeknek a lapkáknak a teljesítménye már 50 W, így ugyanolyan hőelvezetési problémák támadtak, mint a a Pentium 4 esetében.

Nagyon nehéz összehasonlítani egy CISC (mint amilyen a Pentium 4) és egy RISC lapkát (mint amilyen az UltraSPARC III) kizárólag az órajel-frekvencia alapján. Például, az UltraSPARC III órajelciklusonként folyamatosan 4 utasítást tud elvégezni, amely majdnem akkora végrehajtási sebességet jelent, mintha egy egyutasításos gép 4,8 GHz-en működne. Az UltraSPARC-nak hat belső csővezetéke van, két 14 állapotú csővezeték az egész számokon végzett műveletekhez, kettő a lebegőpontos művelethez, egy a betöltés/tárolás műveletekhez és egy az elágazások végrehajtásához. Ugyancsak más-más megközelítést alkalmaz a gyorsítótárakkal, a szé-



3.47. ábra. *Az UltraSPARC III központi egység áramkör*

lesebb sínekkel és más tényezőkkel kapcsolatban, amelyek javítják a teljesítményt. A Pentium 4-nek is megvan a maga erőssége. A lényeg, hogy két nagyon különböző lapkának pusztán az órajel-frekvenciája alapján történő összehasonlítása nagyon keveset árul el egy bizonyos feladat elvégzése során mutatott teljesítményükről.

Az UltraSPARC III egy 1368 lábú LGA (Land Grid Array) tokban kerül forgalomba, amint az a 3.47. ábrán látható. A tok alján, négyzetes elrendezésben 37 × 37 = 1369 kivezetést tartalmaz, a bal alsó kivezetés hiányzik. A foglalat pontosan illik a lapkához, ez meggátolja, hogy a lapkát helytelenül dugják be a foglalatba.

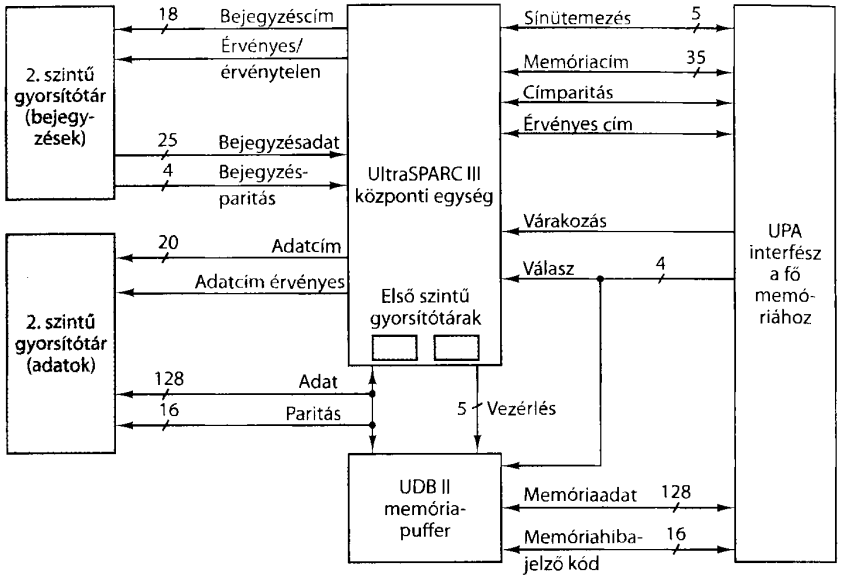
Az UltraSPARC III-nak két belső elsődleges (L1) gyorsítótára van: 32 KB az utasításoknak és 64 KB az adatoknak. Ugyancsak van egy-egy 2 KB-os előolvasási és írási gyorsítótára, amelyekkel összegyűjti a 2. szintű gyorsítótárba történő írásokat, így azokat nagy adagokban végezheti el, ami növeli a sávszélességet. Mint a Pentium 4, ez is használ a lapkán kívüli 2. szintű (L2) gyorsítótárat, viszont a Pentium 4-től eltérően az UltraSPARC III-nak nincs a tokba épített 2. szintű gyorsítótára. A gyorsítótár-vezérlő és a gyorsítótárblokkok címzéséhez szükséges logika a lapkán megtalálható, de az SRAM memória már nem. Ehelyett, a rendszer tervezői ha 2. szintű gyorsítótárat szeretnének, szabadon választhatnak bármilyen kereskedelmi forgalomban lévő gyorsítótárlapka közül.

Az a döntés, hogy a 2. szintű gyorsítótárat a Pentium 4-gyel integrálják, illetve, hogy az UltraSPARC III esetében attól különítsék el, részben technológiai okokra, részben az Intel és a Sun eltérő üzleti filozófiájára vezethető vissza. Technológiai szempontból egy külső gyorsítótár nagyobb és rugalmasabb (az UltraSPARC III 2. szintű gyorsítótárai 1 MB-tól 8 MB-ig terjedhetnek; míg a Pentium 4 gyorsítótárainak mérete rögzített: 512 KB). Másfelől viszont lehet, hogy lassabb, mivel nagyobb távolságra van a CPU-tól. Több kivezetett vezérlőjel is kell a külső gyorsítótár eléréséhez. Jelen esetben, az UltraSPARC III és az L2 gyorsítótár közötti kapcsolat 256 bit széles, ami biztosítja, hogy egy 32 bájtos gyorsítósort egyetlen órajelciklus alatt lehessen átvinni.

Üzleti szempontból nézve az Intel félvezetőgyártó, így képes arra, hogy saját 2. szintű gyorsítótárlapkákat tervezzen, gyártson és azokat egy nagy teljesítményű saját interfészen keresztül összekapcsolja a központi egységgel. A Sun ellenben számítógépeket, és nem lapkákat gyárt. Ugyan néhány lapkáját (mint az UltraSPARC-ok) maga tervezi, de az elkészítéssel más félvezetőgyártókat bíz meg. A Sun, amikor csak megteheti, előnyben részesíti a kereskedelmi forgalomban már kapható lapkák használatát, amelyek a piaci versenyben már kellően kifinomultak. A 2. szintű gyorsítótárakban használatos SRAM-ok számos lapkagyártótól beszerezhetők, így a Sunnak nem volt különösebben szüksége arra, hogy saját lapkát tervezzen. Ez a döntés azt jelenti, hogy a 2. szintű gyorsítótár függetlenné vált a központi egységtől.

Az UltraSPARC III 43 bit széles címsínt használ, amellyel legfeljebb 8 TB memóriát tud megcímezni. Az adatsín 128 bit széles, egyszerre 16 bájt átvitelét teszi lehetővé a CPU és a memória között. A sín sebessége 150 MHz, így 2,4 GB/s sávszélesség érhető el, ami sokkal gyorsabb, mint az 528 MB/s sebességű PCI sín.

Annak érdekében, hogy az UltraSPARC CPU-k (esetleg több) és a memóriák (esetleg több) kommunikálni tudjanak egymással, a Sun kifejlesztette az **UPA**-t



3.48. ábra. *Az UltraSPARC III rendszer magjának fontosabb jellemzői*

(**Ultra Port Architecture**). Az UPA megvalósítható úgy is, mint sín, hálózati csomópont (switch) vagy a kettő kombinációja. A különböző munkaállomások és szerverek különböző UPA-megvalósításokat alkalmaznak. Az UPA különböző megvalósításai nem befolyásolják a központi egységet, mivel az UPA interfész precízen definiált, és ez az az interfész, amelyet a központi egységnek támogatnia kell (és támogat is).

A 3.48. ábrán egy UltraSPARC III rendszer magja látható, tartalmazza a CPU lapkát, az UPA interfészt és a 2. szintű gyorsítótárat (két közönséges SRAM-ot). Az ábra egy **UDB II lapkát (UltraSPARC Data Buffer II)** is tartalmaz; ennek feladatát a későbbiekben ismertetjük. Amikor a CPU-nak egy memóriaszóra van szüksége, azt először az egyik (belső) 1. szintű gyorsítótárában keresi. Ha ott van a szó, folytatja a futtatást teljes sebességgel. Ha nincs a szó az 1. szintű gyorsítótárban, a 2. szintűvel próbálkozik.

A gyorsítótárakról majd a 4. fejezetben szólunk részletesen, néhány gondolatot azonban hasznos itt is előrebocsátani. A teljes fő memória 64 bájtos gyorsítótársorokra (blokkokra) van felosztva. A 256 leggyakrabban használt utasítás- és adatsor az 1. szintű gyorsítótárban van. A gyakran használt gyorsítósorok közül azok, amelyek nem férnek már be az 1. szintű gyorsítótárba, a 2. szintű gyorsítótárba kerülnek. Ez a gyorsítótár már véletlenszerűen, vegyesen tartalmaz utasítás- és adatsorokat. Ezek tárolódnak az ábrán a 2. szintű gyorsítótár (adatok) feliratú téglalapban. A rendszer nyilvántartja, hogy mely gyorsítósorok vannak a 2. szintű gyorsítótárban. Ezt az információt a 2. szintű gyorsítótár (bejegyzések) felirattal jelölt SRAM-ban tárolják.

Amikor egy gyorsítósor nem található az 1. szintű gyorsítótárban, a CPU a sor azonosítóját (bejegyzéscím) elküldi a 2. szintű gyorsítótárba. A válasz (bejegyzésadat) megadja a CPU számára, hogy a keresett sor bent van-e a 2. szintű gyorsítótárban, és ha igen, milyen állapotban (érvényes/érvénytelen). Ha a gyorsítósor ott van, a CPU elkéri, és meg is kapja. Az adatátvitel szélessége 16 bájt, így négy órajelciklus szükséges a teljes 64 bájtos gyorsítósor átvitelére az első szintű gyorsítótárba.

Ha a keresett sor a második szintű gyorsítótárban sincs, a fő memóriából kell behozni az UPA interfészen keresztül. Az UltraSPARC III UPA-t egy központi vezérlő segítségével valósították meg. A CPU-ból minden cím és címvezérlő jel (ha több CPU is van, akkor mindegyiké) oda fut be. A CPU a memóriasín megszerzéséhez először a sínütemezési jelek segítségével engedélyt kér, hogy ő lehessen a következő sínmester. Amint az engedélyt megkapta, a CPU felteszi a kért címet a memóriasínre (adatcím), megadja a kérés típusát, végül beállítja az adatcím érvényes jelet. (Ezek a jelek kétirányúak, mivel az UltraSPARC III többprocesszoros rendszerben más központi egységeknek szükségük lehet a távoli gyorsítótárak elérésére is, hogy az összes gyorsítótár tartalmát mindenkor konzisztens állapotban tarthassák.) A cím és a sínciklus típusa két órajel alatt íródik ki a címvezetékekre oly módon, hogy az első ciklusban a sorcím, a másodikban pedig az oszlopcím íródik ki, amint az a 3.31. ábrán látható.

Amíg az eredményre vár, a CPU folytathatja a munkát esetleg valami mással. Például egy gépi utasítás előre betöltésekor fellépő gyorsítótárhiány nem gátolja meg más, már betöltött gépi utasítások végrehajtását, amelyek mindegyike hivatkozhat olyan adatra, amely nincs bent egyik gyorsítótárban sem. Ily módon az UPA-val egyszerre több tranzakció is folyhat. Az UPA két független tranzakciófolyamot tud kezelni (nevezetesen olvasásokat és írásokat), mindegyikben több megkezdett tranzakcióval. A központi vezérlő feladata, hogy nyilvántartsa mindezt, és hogy az aktuális memóriaigényeket a leghatékonyabb sorrendben kielégítse.

Amikor végül megérkezik az adat a memóriából, 8 bájtnyi érkezhet egyszerre egy 16 bites hibajavító kóddal együtt, a nagyobb megbízhatóság érdekében. Egy tranzakció kérhet egy egész gyorsítósort, egy 8 bájtos szót, vagy akár kevesebb bájtot. Minden bejövő adat az UDB-be kerül, amely pufferezi azokat. Az UDB rendeltetése, hogy még inkább leválassza a központi egységet a memóriarendszerről, így azok aszinkron módon tudnak működni. Például, ha a CPU egy szót vagy egy gyorsítósort szeretne kiírni a memóriába, akkor ahelyett, hogy várna az UPA elérésére, azonnal beírhatja az adatokat az UDB-be, és az UDB-re hagyja, hogy azokat később a memóriába juttassa. Az UDB generálja és ellenőrzi a hibajavító kódokat is.

Csak megjegyzésképpen, az UltraSPARC III-nak a fenti leírása, ugyanúgy, mint az előtte található Pentium 4 bemutatása, nagymértékben leegyszerűsített, de a működésük lényegét bemutatta.

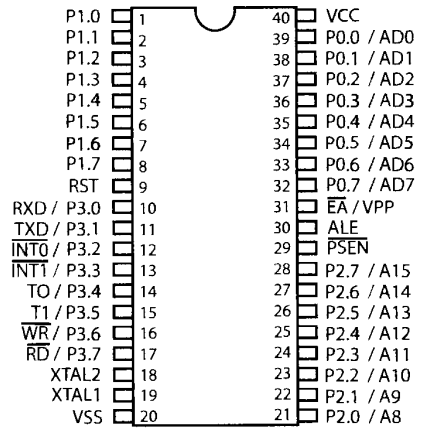
3.5.3. 8051

Mind a Pentium 4, mind az UltraSPARC III központi egység olyan nagy teljesítményű központi egységek képviselői, amelyeket különlegesen gyors személyi számítógépekbe vagy szerverekbe terveztek. Sok ember, amikor számítógépre gondol, ilyenfajta rendszert képzel el. De létezik a számítógépeknek egy egészen más világa, amely ráadásul még jóval nagyobb is: a beágyazott rendszerek világa. Az alábbiakban ezt a világot fogjuk röviden áttekinteni.

Valószínűleg csak kis túlzás, ha azt állítjuk, hogy minden olyan elektronikus készülék számítógépet tartalmaz, amely 100 dollárnál többe kerül. Természetesen a televíziókat, mobiltelefonokat, elektronikus menedzserkalkulátorokat, mikrohullámú sütőket, videokamerákat, videomagnókat, lézernyomtatókat, riasztókészülékeket, nagyothalló-készülékeket, elektronikus játékokat és más eszközöket, amelyek túl sokan vannak ahhoz, hogy fel lehessen sorolni, napjainkban mind számítógép vezérli. Az ezekben a készülékekben levő számítógépeket – sokkal inkább az alacsony ár érdekében, mintsem a nagy teljesítmény alapján – optimalizálják, ami másfajta kompromisszumokhoz vezet, mint az olyan nagy teljesítményű központi egységek esetében, amelyeket eddig tanulmányoztunk.

Amint azt az 1. fejezetben említettük a 8051 valószínűleg a legnépszerűbb, széles körben használt mikrovezérlő, ami leginkább nagyon alacsony árának köszönhető. Rövidesen meglátjuk, hogy egy egyszerű lapkáról van szó, amely egyszerűen alkalmazható és olcsó. Vizsgáljuk meg most a 8051 lapkát, amelynek a fizikai lábkiosztása a 3.49. ábrán látható.

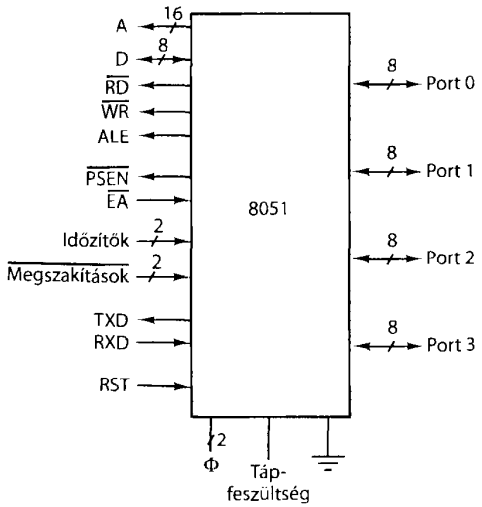
Ahogyan az ábrán is látható, a 8051 általában egy 40 lábú standard tokban kerül forgalomba (bár speciális alkalmazásokhoz más tokozás is rendelkezésre áll). 16 címvezetéke van, tehát legfeljebb 64 KB memóriát tud megcímezni. Az adatsín 8 bit széles, ezért a CPU és a memória közötti adatátvitel 1 bájtonként történik



3.49. ábra. *A 8051 fizikai lábkiosztása*

(ezzel szemben a Pentium 4-nél 8, az UltraSPARC III-nál pedig 16 bájtonként). Különböző vezérlőbemenetei vannak, amelyeket később ismertetünk. A legnagyobb különbség a Pentium 4-gyel és az UltraSPARC III-mal szemben, amelyek igazi CPU-k, a 32 B/K vonal, amelyet 4 db 8 bites csoportba rendeztek. Mindegyik B/K vonal hozzáköthető egy nyomógombhoz, kapcsolóhoz, LED-hez (Light Emitting Diode, fénykibocsátó/világító dióda) vagy a külvilág más eszközéhez, amellyel bemenetet biztosítunk a 8051-nek vagy a 8051 kimenetét vezethetjük el. Egy rádiós óra esetén minden nyomógombot és kapcsolót különböző B/K vonalhoz köthetünk, míg más B/K vonalakkal a megjelenítőt vezérelhetjük. Így a rádiós óra legtöbb funkcióját, ha nem mindet, a szoftverrel vezérelhetjük, kiküszöbölve ezzel a diszkrét logika drága áramköreit.

A 8051 logikai lábkiosztása a 3.50. ábrán látható. A 8051 egy 4 KB-os belső ROM-mal kerül forgalomba (a 8052-esnél ez 8 KB). Ha ez nem lenne elegendő az adott alkalmazáshoz, legfeljebb 64 KB külső memória csatlakoztatható a 8051-hez egy sínen keresztül. A 3.50. ábra bal oldalán az első hét jel a külső memóriához történő kapcsolódást vezérli, ha van ilyen. Az első jel az A, 16 címvezetéket tartalmaz a külső memóriában lévő beolvasandó vagy kiírandó bájt megcímzésére. A nyolc D vezeték adatátvitelre szolgál. Az alacsony helyértékű nyolc címvezeték ugyanezekre az adatvezetékekre van multiplexelve, hogy csökkentsék a kivezetések számát. Egy síntranzakció esetén ezeken a lábakon jelenik meg a cím az első órajelciklusban, és ezek továbbítják az adatokat az ezt követő ciklusokban.



3.50. ábra. *A 8051 logikai lábkiosztása*

Külső memória használatánál a 8051 az RD vagy a WR beállításával jelzi, hogy olvassa vagy írja a memóriát. Az ALE (Address Latch Enable) jelet akkor használja, ha van külső memória. A CPU állítja be ezt a jelet, hogy jelezze, a sínen lévő

cím érvényes. A külső memóriák tipikusan arra használják fel ezt a jelet, hogy ekkor tárolják a címvezetékek állapotát, mivel azokat nem sokkal ezután elengedi a CPU, hogy a következőkben az adatok átvitelére lehessen használni.

A PSEN és az EA jelek ugyancsak a külső memóriával kapcsolatosak. A PSEN (Program Store Enable) jel beállításával a 8051 jelzi, hogy a programot tároló memóriából kíván olvasni. Általában ezt a memória OE jelével kapcsolják össze, ahogyan az a 3.29. ábrán látható.

EA (External Access) vagy állandóan magas, vagy állandóan alacsony szintre van kötve, így mindig ugyanaz az értéke. Ha állandóan magas értéke van, a belső 4 KB-os (8052 esetén 8 KB-os) memóriát használja az ebbe a tartományába eső címek esetén és a külső memóriát 4 KB (8052-nél 8 KB) feletti címek esetén. Ha állandóan alacsony szintre van kötve, a külső memóriát használja minden cím esetén, és így a lapkán lévő memóriát lényegében mellőzi. 8031 és 8032 esetében az EA jelet alacsony szintre kell kötni, mivel nincs ROM a lapkán.

A két időzítő áramkör bemenet lehetővé teszi, hogy a CPU külső időzítő áramköröktől kapjon jeleket. A megszakításvonalak két külső eszköz számára biztosítják a megszakításkérés lehetőségét. A TXD (transmitted data, továbbított adat) és az RXD (received data, érkezett adat) vezetékek egy terminállal vagy egy modemmel soros B/K műveleteket tesznek lehetővé. Végül RST teszi lehetővé, hogy a felhasználó vagy egy külső eszköz alapállapotba állítsa (reset) a 8051-et. Ezt a jelet akkor állítják be, amikor valami elromlik, és a rendszert újra kell indítani.

Eddig a 8051 hasonlít a legtöbb 8 bites CPU-hoz azt kivéve, hogy soros B/K vonalai vannak. Ami megkülönbözteti a 8051-et az a 32 B/K vonal, amelyet 4 különálló portba szerveztek; ez látható a 3.50. ábra jobb oldalán. Mindegyik vonal kétirányú, és program segítségével írható vagy olvasható. Ez az elsődleges módja annak, ahogyan a 8051 kapcsolatot tart a külvilággal, és ami annyira értékessé teszi: egyetlen lapka, amely minden szükségest tartalmaz, CPU-t, memóriát és B/K lehetőségeket.

3.6. Példák sínekre

A sínek azok a ragasztók, amelyek összetartják a számítógéprendszereket. Ebben a fejezetben néhány népszerű sínt vizsgálunk meg közelebbről: az ISA sínt, a PCI sínt és az univerzális soros sínt. Az ISA sín az eredeti IBM PC sínjének egy kismértékű bővítése. A visszafelé való kompatibilitás miatt minden Intel-alapú PC-ben volt ilyen sín az utóbbi néhány évvel bezárólag, amikor is az Intel és a Microsoft megegyezett arról, hogy kiveszik. Azonban ezek a gépek változatlanul rendelkeznek egy második, gyorsabb sínnel is: a PCI sínnel. A PCI sín szélesebb, mint az ISA sín, és magasabb órajel-frekvencián működik. Ennek következtében több adatot képes továbbítani másodpercenként, mint az ISA sín. Az univerzális soros sín (USB) növekvő népszerűségű B/K sín az olyan alacsony sebességű perifériák számára, mint az egerek vagy a billentyűzet. Az USB második verziója még nagyobb sebességgel működik. A következő részekben ezeket a síneket egyenként megvizsgáljuk, hogy lássuk, hogyan működnek.

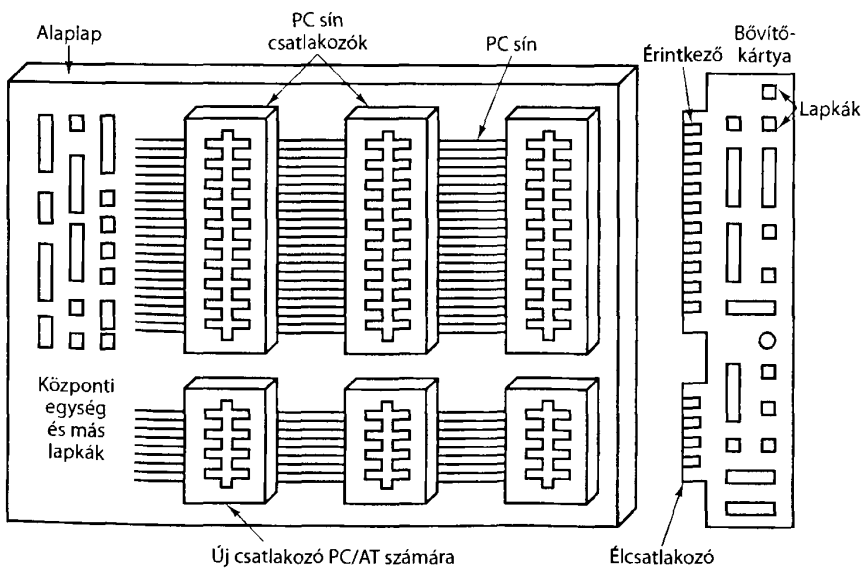
3.6.1. ISA sín

Az IBM PC sín de facto szabvány volt a 8088 alapú rendszerekben, mivel majdnem minden PC-hasonmás szállító ezt másolta le, hogy lehetővé tegye a sok létező, harmadik féltől származó B/K kártya felhasználását a számítógépekben. A sínnek 62 jelvezetéke volt, beleértve 20 memóriacím és 8 adatvezetéket, valamint egy-egy vezetéket a memóriából olvasás, memóriába írás, B/K olvasás, B/K írás beállításának céljaira. Ugyancsak voltak jelek megszakításkérésre, megszakításengedélyezésre, DMA használatára. Tehát egy nagyon egyszerű sín volt.

Fizikailag a sín a PC alaplapjára volt maratva, mintegy fél tucat, egymástól 2 cm távolságban lévő csatlakozóval, ahová bővítőkártyákat lehetett bedugni. Mindegyik kártyának volt egy füle, amely a csatlakozóba illett. A fülön 31 aranyozott vezetéksáv volt mindkét oldalon, amely az elektronikus csatlakozást biztosította.

Amikor az IBM megjelentette a 80286 alapú PC/AT-t, komoly problémával kellett szembenéznie. Ha teljesen elölről kezdték volna, és egy vadonatúj 16 bites sínt terveznek, sok vásárló tétovázott volna, hogy megvegye-e, mivel a harmadik féltől származó PC bővítőkártyák egyike sem működött volna az új számítógépben. De ha ragaszkodnak a PC sínhez, a 20 cím- és 8 adatvezetékhez, akkor nem lehetett volna kihasználni a 80286-osnak azt a képességét, hogy meg tud címezni 16 MB memóriát, és 16 bites szavakat tud továbbítani.

Végül a PC sín kibővítését választották. A PC bővítőkártyáknak 62 érintkezős élcsatlakozója van, ez azonban nem fut végig a kártya teljes hosszában. A PC/AT esetében alkalmazott megoldás az volt, hogy egy második élcsatlakozót alakítanak



3.51. ábra. *A PC/AT sín két komponense, az eredeti PC és az új rész*

ki a kártya alján, amely szomszédos a fő csatlakozóval, továbbá úgy alakítják ki az AT áramköreit, hogy mind a kétfajta kártyával működhessenek. Ezt az általános elképzelést mutatja be a 3.51. ábra.

A második csatlakozónak a PC/AT sínen 36 vezetéke van. Ezek közül 31 a több cím- és a adatvezeték, a több megszakításkérés, a több DMA csatorna, valamint a tápfeszültség és a föld miatt volt szükséges. A maradék öt vezeték a 8 és a 16 bites átvitel különbözőségét kezeli.

Amikor az IBM a PC és a PC/AT utódjaként kihozta a PS/2 sorozatot, úgy döntött, hogy itt az ideje újrakezdeni. A döntést részben technikai meggondolások indokolták (a PC sín akkorra már tényleg elavult), de kétségtelenül az az elhatározás is közrejátszott, hogy akadályt gördítsenek a PC-hasonmásokat előállító vállalatok elé, amelyek ez időre már átvették a piac kényelmetlenül nagy részét. Így a közepes és a nagy teljesítményű PS/2 gépeket olyan sínnel látták el, a Microchannellel, amely teljesen új volt, szabadalmakkal volt körülbástyázva és ügyvédek hada támogatta.

A személyi számítógépek iparának fennmaradó része úgy válaszolt erre a fejleményre, hogy saját szabványt fogadott el, az **ISA (Industrial Standard Architecture, ipari szabvány architektúra)** sínt, amely alapvetően egy 8,33 MHz frekvencián működő PC/AT sín volt. Ennek a megközelítésnek az volt az előnye, hogy megtartotta a kompatibilitást a létező számítógépekkel és kártyákkal. Ezenkívül az új szabvány olyan sínen alapult, amelynek a felhasználását az IBM nagyvonalúan sok vállalat számára engedélyezte, hogy ezzel az eredeti PC számára a lehető legtöbb harmadik féltől származó bővítőkártya előállítását biztosítsa – ez azóta is kísérti az IBM-et, és végül ez vezette ki a PC-üzletből is. Néhány évvel ezelőttig a legtöbb Intel-alapú PC-ben megtalálható volt ez a sín, bár egy vagy több más sínnel együtt.

Később az ISA sínt 32 bitesre bővítették ki, és néhány új szolgáltatással is kiegészítették (például többprocesszoros architektúra támogatása). Ezt az új sínt **EISA (Extended ISA, bővített ISA)** sínnek nevezték.

3.6.2. PCI sín

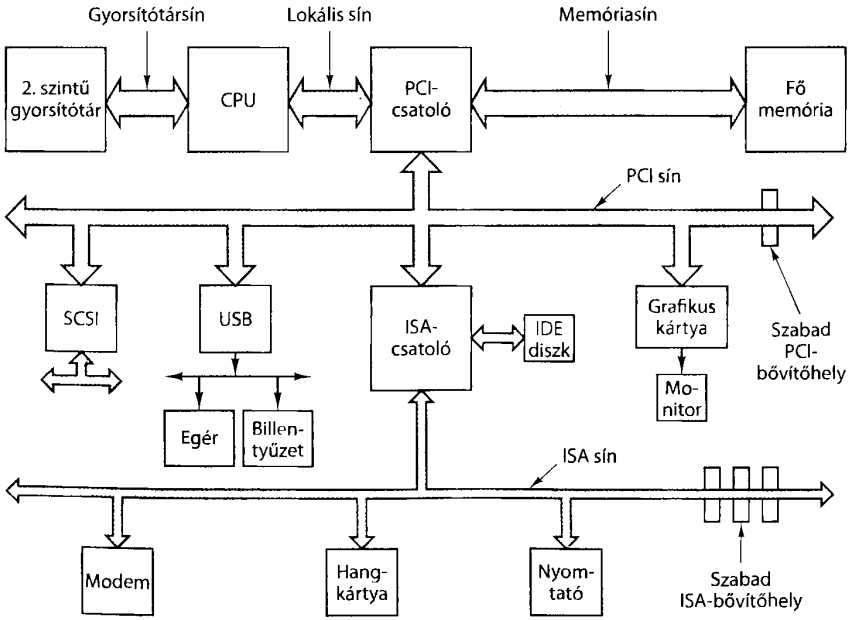
Az eredeti IBM PC-n a legtöbb alkalmazói program még szöveges képernyőt használt. A Windows bevezetésével fokozatosan használatba kerültek a grafikus felhasználói felületek. Egyik ilyen alkalmazás sem jelentett nagy megerőltetést az ISA sín számára. Azonban, ahogy az idő telt, egyre több program kezdte intenzíven használni, különösen a multimédiás játékok, a számítógépek képernyőjét teljes képernyőt igénylő grafika, mozgókép megjelenítésére. A helyzet ezzel gyökeresen megváltozott.

Végezzük el a következő egyszerű számítást. Vegyünk egy 1024 × 768-as képernyőt, amelyet valódi színekkel (3 bájt/képpont) szeretnénk használni mozgó képek megjelenítésére. Egyetlen képernyő 2,25 MB adatot tartalmaz. A folyamatos mozgáshoz 30 kép szükséges másodpercenként; ez 67,5 MB/s sebességnek felel meg. Valójában a helyzet még rosszabb: a kép megjelenítéséhez az adatoknak a mágneslemezről, a CD-ROM-ról vagy a DVD-ről a rendszersínen keresztül el

kell jutniuk a memóriába. Majd a megjelenítéshez az adatoknak ismét át kell haladniuk a rendszersínen a grafikus kártyához. Emiatt 2 × 67,5 MB/s, azaz 135 MB/s sínsávszélesség szükséges csupán a video számára, nem számítva azt a sávszélességet, amelyre a központi egységnek és más eszközöknek van szükségük.

Az ISA sín legfeljebb 8,33 MHz sebességgel működhet, és egy ciklusban 2 bájtot képes átvinni; ezzel a maximális sávszélesség 16,7 MB/s. Az EISA sín 4 bájtot tud átvinni egy ciklusban, így 33,3 MB/s sávszélességet tud elérni. Nyilvánvaló, hogy bármelyiket is nézzük, egyik sincs még a közelében sem annak a sebességnek, amire a teljes képernyő méretű videónak szüksége van.

Az Intel látva, hogy ez hamarosan bekövetkezik, 1990-ben új sínt tervezett, amelynek a sávszélessége még az EISA sínénél is nagyobb volt. Az új sínt **PCI (Peripheral Component Interconnect bus)** sínnek nevezték el. Az Intel, hogy használatukat elősegítse, szabadalmaztatta a PCI sínt, majd az összes szabadalmat mindenki számára megnyitotta, így bármelyik gyár készíthet PCI sínt használó perifériákat anélkül, hogy a szabadalom használatáért jogdíjat kellene fizetnie. Az Intel létrehozott egy ipari konzorciumot is – a *PCI iránt különösen érdeklődő szakemberek csoportját* (PCI Special Interest Group) –, amely a PCI sín jövőjét egyengeti. Mindezeknek köszönhetően a PCI sín különösen népszerű lett. A Pentiumtól kezdve gyakorlatilag minden Intel-alapú számítógép és sok más gép is rendelkezik



3.52. ábra. *Egy korai Pentium rendszer architektúrája. A vastagabb sínek nagyobb sávszélességgel rendelkeznek, mint a vékonyabbak, az ábra azonban az arányokat nem tükrözi*

PCI sínnel. Még a Sun UltraSPARC központi egységnek is van PCI sínt használó verziója, az UltraSPARC IIIi. A PCI sínnel kapcsolatban – a legapróbb részletekig – lásd (Shanley és Anderson, 1999; Solari és Willse, 2004).

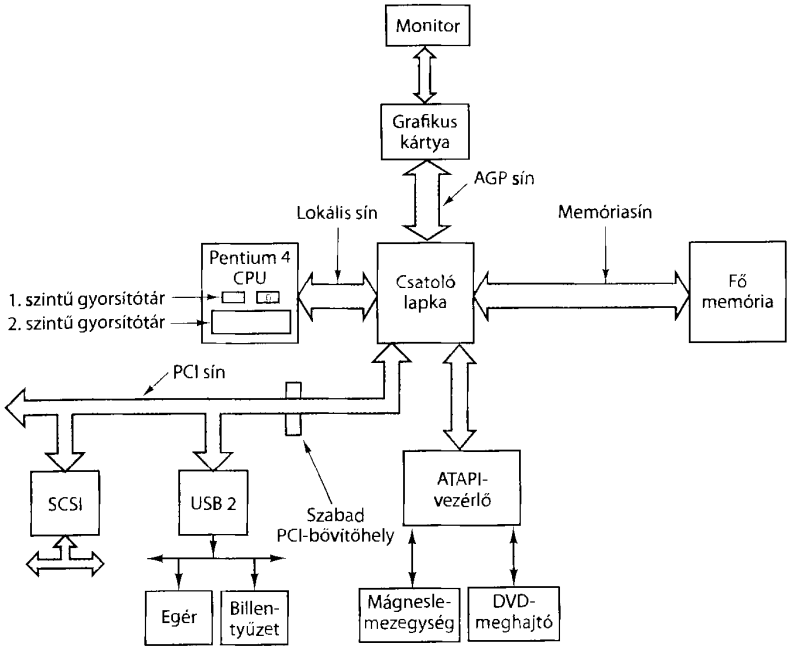
Az eredeti PCI sín 32 bitet tudott átvinni egy ciklus alatt 33 MHz sebességgel (30 ns ciklusidővel), a teljes sávszélesség 133 MB/s volt. 1993-ban bevezették a PCI 2.0-t, majd 1995-ben a PCI 2.1-t. A PCI 2.2 már a hordozható személyi számítógépek számára is nyújtott szolgáltatásokat (leginkább a telepek energiájával való takarékoskodáshoz). Az újabb PCI sín már 66 MHz frekvencián működik, és 64 bites átvitelt is képes kezelni, így teljes sávszélessége 528 MB/s. Ezzel a kapacitással már megvalósítható a teljes képernyős, folyamatos mozgókép (feltéve, hogy a mágneslemez és a rendszer többi része alkalmas a feladatra). Mindenesetre nem a PCI sín lesz a szűk keresztmetszet.

Bár az 528 MB/s meglehetősen gyorsnak hangzik, mégis van két probléma. Először is nem elég jó egy memóriasín számára. Másodszor pedig nem kompatibilis a régi ISA bővítőkártyákkal. Az Intel azon a megoldáson gondolkodott, hogy olyan számítógépeket kellene tervezni, amelyeknek három vagy több sínje van, amint az a 3.52. ábrán látható. Ezen a diagramon azt látjuk, hogy a központi egység a memóriával egy speciális memóriasínen keresztül kommunikálhat, és egy ISA sín kapcsolódhat a PCI sínre. Ez az elrendezés kielégít minden elvárást, és ennek következtében széles körben használták az 1990-es években.

A két kulcselem ebben az architektúrában a két csatoló lapka, más néven híd (amelyet az Intel gyárt – ebből fakad az érdeklődése az egész projekt iránt). A PCI csatoló áramkör a központi egységet, a memóriát és a PCI sínt köti össze. Az ISA csatoló lapka pedig a PCI sínt köti össze az ISA sínnel, valamint egy vagy két IDE mágneslemezegységgel. Majdnem minden Pentium 4 rendszer rendelkezik egy vagy több szabad PCI-csatlakozóval amelyek lehetővé teszik újabb nagy sebességű perifériák csatlakoztatását, továbbá egy vagy több ISA-csatlakozóval, amelyen keresztül kis sebességű perifériák csatlakoztathatók.

A 3.52. ábrán látható architektúra nagy előnye, hogy a központi egységnek a memória felé – egy saját memóriasín alkalmazásának köszönhetően – különlegesen nagy sávszélessége van. A PCI sín nagy sávszélességet biztosít az olyan gyors perifériák számára, mint a SCSI-diszkek vagy a grafikus kártyák stb., és a régi ISA kártyák is használhatók. Az USB-vel jelölt téglalap az univerzális soros sínt (Universal Serial Bus) jelenti, amelyről később esik szó ugyanebben a fejezetben.

Milyen szép is lenne, ha csak egyetlen fajta PCI kártya létezne! Sajnos, nem ez a helyzet. Többféle változat is létezik tápfeszültségben, sínszélességben és frekvenciában. A régebbi számítógépek gyakran 5 V tápfeszültséget használtak, az újabbak egyre inkább csak 3,3 V feszültséget. A PCI sín mindkettőt támogatja. A csatlakozók azonosak, kivéve azt a két kis műanyag lapocskát, amely meggátolja, hogy egy 5 V-os PCI kártyát bedugjanak egy 3,3 V-os sínbe, vagy fordítva. Szerencsére, olyan univerzális kártyák is léteznek, amelyek mindkét feszültséggel működhetnek, és bedughatók bármelyik csatlakozóba. A tápfeszültségben különböző változatokon kívül a kártyák lehetnek 32 vagy 64 bitesek. A 32 bites kártyáknak 120 kivezetése van, a 64 bites kártyáknak is megvan ugyanez a 120 kivezetése és még 64, hasonlóan ahhoz, ahogyan az IBM PC sínt kibővítették 16 bitesre (lásd



3.53. ábra. *Egy modern Pentium 4 rendszer sín struktúrája*

3.51. ábra). A 64 bites kártyákat fogadni képes rendszerbe behelyezhetők a 32 bites kártyák is, ennek a fordítottja azonban már nem igaz. Végül, a PCI sínek működhetnek 33 MHz vagy 66 MHz frekvencián. Választani a kettő között úgy lehet, hogy egy kivezetést a tápfeszültségre vagy a földre kötnek. A csatlakozók mindkét sebesség esetében azonosak.

Az 1990-es évek végére majdnem mindenki egyetértett azzal, hogy az ISA sín halott, ezért az új tervek már nem is tartalmazták. Ekkorra azonban megnőtt a monitorok felbontóképessége (néhány esetben elérte az 1600 × 1200-as felbontást) és a teljes képernyős folyamatos video iránti igény, különösen a magas szintű interaktív játékprogramok esetében, így az Intel egy másik sínt is hozzáadott, amely a grafikus kártyát vezérli. Ennek a sínnek a neve AGP (Accelerated Graphics Port) sín. A kezdeti verzió, az AGP 1.0 264 MB/s sávszélességű volt, amit 1× (egyszeres) sebességnek definiáltak. Bár lassabb volt, mint a PCI sín, a grafikus kártya vezérlésére szánták. Évek múltán új verziók jelentek meg, és az AGP 3.0-val már 2,1 GB/s sebességet (8×) értek el. Egy modern Pentium 4 rendszert mutat be a 3.53. ábra.

A tervezés szempontjából a csatoló lapka most központi szerepet tölt be. A rendszer öt fő alkotórészét kapcsolja össze: a CPU-t, a memóriát, a grafikus kártyát, az ATAPI-vezérlőt és a PCI sínt. Egyes változatokban még a hálózati kártyát és más nagy sebességű eszközöket is támogat. A kisebb sebességű eszközöket a PCI sínhez csatlakoztatják.

Belső felépítését tekintve a csatoló lapka két részre osztható: a memóriacsatolóra és a B/K csatolóra. A memóriacsatoló a CPU-t kapcsolja a memóriához és a grafikus kártyához. A B/K csatoló az ATAPI-vezérlőt, a PCI sínt és (esetleg) egyéb gyors B/K eszközt kapcsol össze egymással közvetlenül. A memóriacsatoló és a B/K csatoló áramkör nagyon nagy sebességű belső összeköttetéssel kapcsolódik egymáshoz.

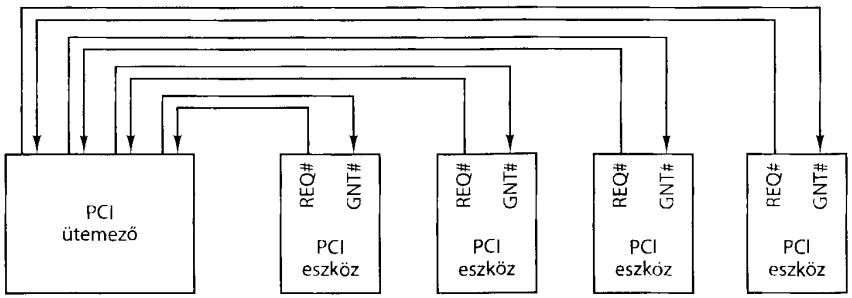
A PCI sín egy szinkron sín, mint minden PC sín, egészen az eredeti IBM PC-ig visszamenően. A PCI sínen minden tranzakció egy mester, amelyet hivatalosan **kezdeményezőnek** (**initiator**) neveznek, valamint egy szolga, hivatalosan **céleszköz** (**target**), között történik. A kivezetések számának alacsonyan tartása érdekében, a PCI sínen a cím- és adatvezetékek multiplexeltek. Így egy PCI kártyán elég mindössze 64 érintkező a cím- és az adatjelek számára, annak ellenére, hogy a kártya 64 bites címeket és 64 bites adatokat is megenged.

A multiplexelt cím- és adatkivezetések a következőképpen működnek. Egy olvasási művelet során, az első sínciklusban a mester kiírja a címet a sínre, a második ciklusban pedig eltávolítja, a sín iránya megfordul, így a szolga használni tudja. A harmadik ciklusban a szolga felteszi a sínre a kívánt adatot. Egy írási művelet során a sín irányának nem kell megfordulnia, mert a mester teszi ki a címet és az adatot is a sínre. Mindazonáltal a legrövidebb tranzakció így is 3 ciklus. Ha a szolga nem tud három cikluson belül válaszolni, várakozó állapotokat iktathat be. Tetszőleges méretű blokkműveletek is megengedettek, mint ahogyan egyéb sínciklusok is.

A PCI ütemezés

Ahhoz, hogy egy eszköz használhassa a PCI sínt, le kell foglalnia. A PCI sín centralizált sínütemezőt (kiosztót) használ, amint az a 3.54. ábrán látható. A legtöbb megvalósításban a sínütemezőt beépítik a csatoló áramkörök egyikébe. Minden PCI eszköztől két szál vezet a központi ütemezőbe. Az egyik a REQ#, amely a sín használati jogának kérésére szolgál. A másik vezeték a GNT#, amely a használati jog elnyerését jelzi.

A sín megszerzéséhez a PCI eszköz (beleértve a CPU-t is) beállítja a REQ# jelet, és addig várakozik, amíg nem látja, hogy a GNT# jelet beállította az ütemező. Amikor ez bekövetkezik, a következő órajelciklusban ez az eszköz használhatja a



3.54. ábra. *A PCI sín centralizált sínütemezőt használ*

PCI sínt. Az ütemező által használt algoritmus nincs definiálva a PCI specifikációjában. A körkörös ütemezés, a prioritásvezérelt ütemezés és más megoldások is mind megengedettek. Nyilvánvaló, hogy egy jó ütemezőnek igazságosnek kell lennie, és nem engedheti meg, hogy valamelyik eszköz a végtelenségig várakozzon.

A sín használatának engedélyezése egy tranzakcióra érvényes, azonban a tranzakció hossza elméletileg nincs korlátozva. Ha az eszköz szeretne egy második tranzakciót is, és semelyik más eszköz sem kéri a sínt, akkor ismét következhet, bár gyakran egy üres ciklust kell a tranzakciók közé iktatni. Azonban speciális feltételek mellett, ha nincs verseny a sín megszerzéséért, egy eszköz szorosan egymás után következő tranzakciókat is végrehajthat anélkül, hogy üres ciklust iktatna a tranzakciók közé. Ha a sínmester egy nagyon hosszú adatátvitelbe kezdett, és közben egy másik eszköz kéri a sínt, az ütemező negálhatja a GNT# vonalat. Az aktuális sínmesternek figyelnie kell a GNT# vonalat, és amint észleli a negálást, fel kell szabadítania a sínt a következő ciklusban. Ez a módszer nagyon hosszú átviteleket is megenged (amelyek hatékonyak), amikor csak egyetlen sínmesterjelölt van, de gyorsan kiszolgálja a többi versenyző eszközt is.

A PCI sín jelei

A PCI sín számos kötelező jelet alkalmaz, ezek a 3.55. (a) ábrán láthatók, valamint sok nem kötelezőt, ezek pedig a 3.55. (b) ábrán. A 120 vagy a 184 érintkezőből fennmaradó kivezetéseket tápfeszültség, föld és ehhez kapcsolódó egyéb olyan funkciók céljaira használják, amelyeket itt nem részletezünk. A *Mester* (kezdeményező) és a *Szolga* (cél eszköz) oszlopok azt mutatják, hogy melyikük állítja be a jelet normál tranzakció esetén. Ha a jel egy harmadik eszköztől származik (például a CLK), mindkét oszlop üresen marad.

Nézzük most meg az egyes PCI jeleket részletesen. A kötelező 32 bites jelekkel kezdünk, majd az opcionális 64 bitesekkel folytatjuk. A CLK jel időzíti a sínt. A legtöbb más jel szinkronban van vele. Ellentétben az ISA sínnel a PCI sín tranzakciók a CLK jel lefutó élére kezdődnek, ami valahol a ciklus közepén van, és nem az elején.

32 bites tranzakciókban a 32 bites AD jel a cím vagy az adat továbbítására szolgál. Általában, az első sínciklusban a cím kerül fel a sínre, a harmadik ciklusban pedig az adat. A PAR jel AD paritásbitje. A C/BE# jelet két különböző dologra használják. Az első ciklusban a sín parancsot tartalmazza (1 szó olvasása, blokk olvasása stb.). A második ciklusban egy bittérképet, amely azt mutatja meg, hogy a 32 bites szó mely bájtjai érvényesek. A C/BE# jel felhasználásával tetszőlegesen 1, 2, 3 bájtot vagy akár egy egész 32 bites szót tudunk olvasni vagy írni.

A FRAME# jelet a sínmester állítja be, amikor megkezdi a tranzakciót. Azt közli a szolgával, hogy a sín parancs és a cím már érvényes. Olvasáskor az IRDY# jelet a FRAME# jellel egyszerre állítja be. Ez azt mondja meg, hogy a sínmester kész a bejövő adatok fogadására. Írás esetén az IRDY# jel később állítódik be, amikor az adat már a sínen van.

Az IDSEL jel azzal kapcsolatos, hogy minden PCI eszköz egy 256 bájtos konfigurációs területtel rendelkezik, amelyet más eszközök tudnak olvasni (az IDSEL jel

Jel	Vezeték-szám	Mester	Szolga	Megnevezés
CLK	1			Órajel (33 vagy 66 MHz).
AD	32	x	x	Multiplexelt cím és adatvezetékek.
PAR	1	x		Cím-, illetve adatparitás bit.
C/BE	4	x		Sín parancs/engedélyezett bájtok bittérképe.
FRAME#	1	x		Jelzi, hogy AD és C/BE érvényes.
IRDY#	1	x		Olvasásnál: a mester fogadja az adatokat; írásnál: adat kész.
IDSEL	1	x		A konfigurációs terület választása a memória helyett.
DEVSEL#	1		x	A szolga felismerte saját címét, és várja az utasításokat.
TRDY#	1		x	Olvasásnál: adat kész; írásnál: a szolga várja az adatokat.
STOP#	1		x	A szolga a tranzakció azonnali leállítását kéri.
PERR#	1			A fogadó által észlelt adat paritása hibás.
SERR#	1			A cím paritása hibás vagy rendszerhiba.
REQ#	1			Sínütemezés: sín lefoglalásának kérése.
GNT#	1			Sínütemezés: lefoglalás engedélyezése.
RST#	1			A rendszer és minden eszköz alapállapotba helyezése.

(a)

Jel	Vezeték-szám	Mester	Szolga	Megnevezés
REQ64#	1	x		64 bites tranzakció kérése.
ACK64#	1		x	64 bites tranzakció engedélyezése.
AD	32	x		További 32 bit cím vagy adat.
PAR64	1	x		Az extra 32 bit cím, illetve adat paritása.
C/BE#	4	x		A további 4 bájt bittérképe.
LOCK#	1	x		A sín lefoglalása több egymás utáni tranzakcióhoz.
SBO#	1			Találat egy másik processzor gyorsítótárában (multiprocesszor számára).
SDONE#	1			Kész a szimatolás (multiprocesszor számára).
INTx	4			Megszakításkérés.
JTAG	5			Az IEEE 1149.1 JTAG teszt jelei.
M66EN	1			A tápfeszültségre vagy a földre kötve (66 vagy 33 MHz).

(b)

3.55. ábra. *(a) Kötelező PCI sín jelek (b) Opcionális PCI sín jelek*

beállításával). Ez a konfigurációs terület az adott eszköz tulajdonságait tartalmazza. Némely operációs rendszer „csatlakoztasd és működik" (Plug-and-Play) szolgáltatása ezt a konfigurációs területet használja annak meghatározására, hogy milyen eszközök vannak a sínen.

Most elérkeztünk azokhoz a jelekhez, amelyeket a szolga állít be. Ezek közül az első a DEVSEL#, amely azt jelenti, hogy a szolga felismerte saját címét az AD vezetékeken, és felkészült arra, hogy végrehajtson egy tranzakciót. Ha a DEVSEL# jelet

egy bizonyos idő elteltével sem állítják be, a mester nem vár tovább, és azt feltételezi, hogy a megszólított eszköz nincs jelen vagy elromlott.

A második jel, amelyet a szolga ad, a TRDY#; ez olvasáskor azt jelzi, hogy az adatokat már az AD vezetékekre tette, íráskor pedig, hogy készen áll az adatok fogadására.

A következő három jel a hibajelzéssel kapcsolatos. Az első ezek közül a STOP#, amellyel a szolga jelzi, hogy valami végzetes dolog történt, és meg akarja szakítani az aktuális tranzakciót. A következő a PERR#, amely arra szolgál, hogy paritáshibát jelezzen az előző ciklusban továbbított adattal kapcsolatban. Olvasásnál a mester állítja be, írásnál pedig a szolga. A fogadó eszközön múlik a szükséges lépések végrehajtása. Végül, az SERR# jel cím- vagy rendszerhibát jelez.

A REQ# és a GNT# jelek az ütemezésben vesznek részt. Ezeket nem az aktuális sínmester adja ki, hanem egy olyan eszköz, amely sínmester szeretne lenni. Az utolsó kötelező jel az RST#, amely a rendszert alapállapotba állítja; ez akkor állítódik be, ha a felhasználó megnyomja a RESET nyomógombot, vagy valamilyen rendszereszköz végzetes hibát észlelt. Ennek a jelnek a hatására minden eszköz alapállapotba kerül, és a számítógép újraindul.

Most elérkeztünk az opcionális jelekhez, amelyek legtöbbje a 32 bitről 64 bitre történő kibővítéssel kapcsolatos. A REQ64# jellel kérhet a sínmester 64 bites tranzakció elindítására engedélyt, az ACK64# jellel pedig a szolga jelezheti, hogy 64 bites tranzakciókat fogad. Az AD, PAR64 és a C/BE# jelek csak a megfelelő 32 bites jelek kiterjesztései.

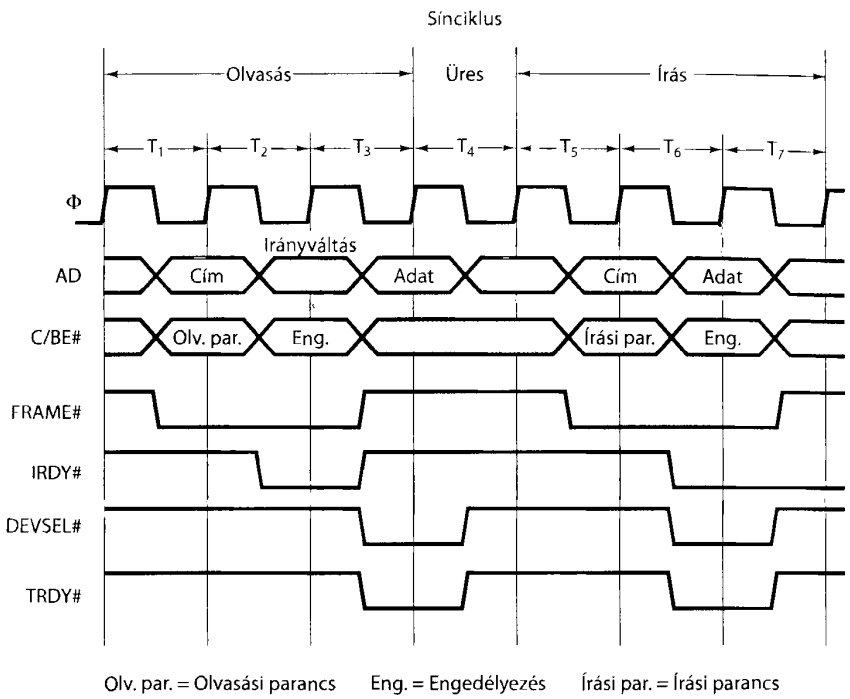
A következő három jel nem a 32 bitről 64 bitre történő bővítéssel, hanem a többprocesszoros rendszerekkel kapcsolatos, olyasvalamivel, amit a PCI kártyáknak nem lenne szükséges támogatniuk. A LOCK jellel foglalható le a sín több, egymást következő tranzakció számára. A másik két jel a sínszimatolással kapcsolatos, a gyorsítótárak koherenciáját biztosítja.

Az INTX jelek megszakítás kérésére szolgálnak. Egy PCI kártya maximum négy különálló logikai eszközt tartalmazhat, és mindegyiknek lehet saját megszakítást kérő vonala. A JTAG jelek az IEEE 1149.1 JTAG tesztelési eljárás számára vannak fenntartva. Végül az M66EN jel vagy a logikai magas, vagy az alacsony szintre van kötve; ezzel az órajel-frekvenciát állítják be. Ez a rendszer működése közben nem változhat meg.

PCI síntranzakciók

A PCI sín igazán egyszerű (ahogyan a sínek általában). Hogy jobban elképzelhessük, tekintsük a 3.56. ábrán látható időzítési diagramot. Ezen egy olvasási tranzakciót láthatunk, amelyet egy üres ciklus, majd pedig egy írási tranzakció követ ugyanazzal a sínmesterrel.

A T_1 ciklusban az órajel lefutó élénél a mester felteszi a memóriacímet az AD vonalakra és a sínparancsot a C/BE# vonalakra. Ezután beállítja a FRAME# jelet, hogy elindítsa a tranzakciót.



3.56. ábra. *Példák 32 bites PCI sín tranzakciókra. Az első három órajelciklusban olvasás zajlik, egy ciklus kimarad, majd a következő három ciklusban írás történik*

A T_2 periódus alatt a mester elengedi a címsínt, ezzel lehetővé teszi, hogy a sín irányt váltson, és a szolga átvehesse a vezérlést a T_3 periódusban. A mester beállítja a C/BE# biteket is, amellyel jelzi, hogy mely bájtokat akarja engedélyezni (azaz beolvasni) a megcímzett szóból.

A T_3 periódusban a szolga beállítja a DEVSEL# jelet, így a mester tudja, hogy megkapta a címet, és válaszolni fog. Felteszi a kért adatot az AD vonalakra, majd beállítja a TRDY# jelet, amely tudatja a mesterrel, hogy a készen van. Ha a szolga nem lenne képes ilyen gyorsan válaszolni, akkor is beállítaná a DEVSEL# jelet, amellyel jelzi a jelenlétét, de a TRDY# jelet negálná mindaddig, amíg nem tudja szolgáltatni az adatot. Ez az eljárás egy vagy több várakozó állapot beiktatását eredményezné.

Ebben a példában (és gyakran a valóságban is) a következő ciklus üres. A T_5 ciklustól kezdve ugyanezt a mestert látjuk, amint egy írás tranzakciót kezdeményez. A cím és a sín parancs beállításával kezdi, mint általában, csak most a második ciklusban a mester állítja be az adatot. Mivel ugyanaz az eszköz vezérli az AD vonalakat, nincs szükség irányváltoztatási ciklusra. A T_7 ciklusban a memória fogadja az adatot.

3.6.3. PCI Express

Bár a PCI sín megfelelően működik a legtöbb napi alkalmazásban, a nagyobb B/K sávszélesség iránti igény beleszemetelt az egykor szép, tiszta belső PC-architektúrába. A 3.53. ábrából világosan látszik, hogy már nem a PCI sín az a központi elem, amely a PC részeit összekapcsolja. A csatoló lapka vette át ezt a szerepet.

A probléma lényege, hogy egyre nagyobb számú B/K eszköz van, amely túl gyors a PCI sínhez. A sín órajel-frekvenciájának megbütykölése nem jó megoldás, mivel akkor megjelennek a sínaszimmetria problémái, az áthallás a vezetékek között és a kapacitáshatások, és csak még rosszabb lesz a helyzet. Minden olyan esetben, amikor egy B/K eszköz túl gyors kezd lenni a PCI sínhez (mint például a grafikus kártyák, beépített mágneslemezegységek, hálózati kártyák stb.), az Intel egy újabb speciális portot ad a csatoló lapkához, lehetővé téve, hogy ez az eszköz kikerülje a PCI sínt. Nyilvánvaló azonban, hogy ez nem egy hosszú távra szóló megoldás.

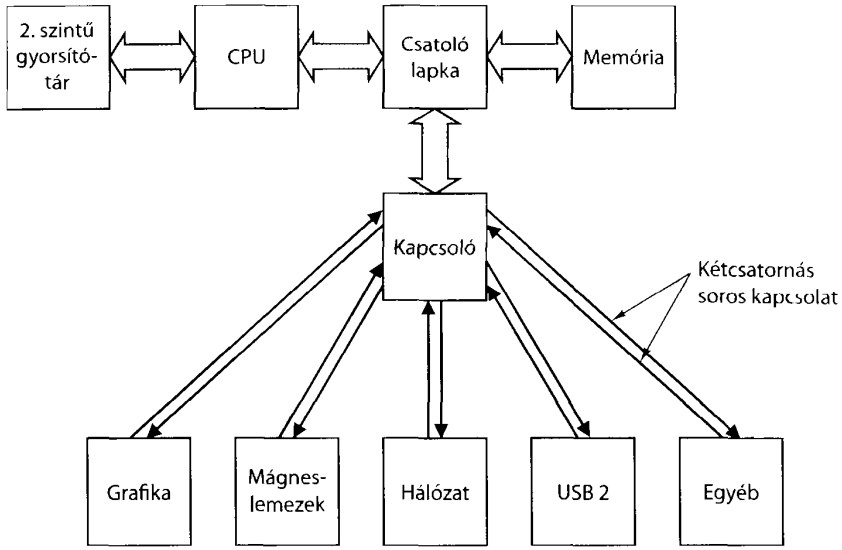
A másik probléma a PCI sínnel az, hogy a bővítőkártyák mérete elég nagy. Nem férnek be egy hordozható asztali számítógépbe (laptopba) vagy egy tenyér számítógépbe (palmtopba), a gyártók pedig még kisebb eszközöket szeretnének előállítani. Ezenkívül, néhány gyártó újra szeretné strukturálni a PC-t: a CPU-t és a memóriát egy lezárt dobozba tennék, a mágneslemezegységet pedig a monitor belsejébe. PCI kártyákkal ez lehetetlen.

Több megoldást is bejelentettek már, a legvalószínűbb nyertes azonban (nem kis részben azért, mert az Intel áll mögötte) a PCI Express. Nem sok köze van a PCI sínhez, valójában egyáltalán nem is sín, de a piaci szakemberek nem szeretnék a jól ismert PCI nevet elveszíteni. Már egy ideje kaphatók olyan PC-k, amelyek ilyet tartalmaznak. Most nézzük meg, hogyan működik.

A PCI Express architektúra

A PCI Express megoldás lényege, hogy szabaduljunk meg a párhuzamos síntől, annak sok sínmesterétől és szolgájától, és térjünk át jól megtervezett, nagy sebességű közvetlen soros kapcsolatokra. Ez a megoldás gyökeresen szakít az ISA/EISA/PCI síntradícióval. Sok ötletet a lokális hálózatok világából, különösen a kapcsolt Ethernet-hálózatoktól kölcsönöztek. Az alapötlet a következő: a PC legbelül CPU, memória- és B/K vezérlőlapkákból áll, amelyeket össze kell kapcsolni. A PCI Express egy általános célú kapcsolót biztosít a lapkák összekötéséhez soros kapcsolat segítségével. Egy tipikus ilyen konfigurációt mutat be a 3.57. ábra.

Ahogyan az a 3.57. ábrán is látszik a CPU, a memória és a gyorsítótár hagyományos módon kapcsolódik a csatoló lapkához. Ami új az a kapcsoló, amely a csatoló lapkához kapcsolódik (esetleg lehet magának a csatoló lapkának a része is). Mindegyik B/K lapkának saját egyedi összeköttetése van a kapcsolóhoz. Mindegyik kapcsolat két egyirányú csatornából álló pár; az egyik csatorna a kapcsolóhoz, a másik pedig a kapcsolótól szállít adatokat. Mindegyik csatorna két vezetéket jelent, egyik a jel, a másik a föld számára, amely így nagy zajtűréssel rendelkezik a



3.57. ábra. *Egy tipikus PCI Express rendszer vázlata*

nagy sebességű átvitel során. Ez az architektúra a jelenlegit egy sokkal egységesebb modellel váltaná fel, amelyben minden eszközt azonos módon kezelnek.

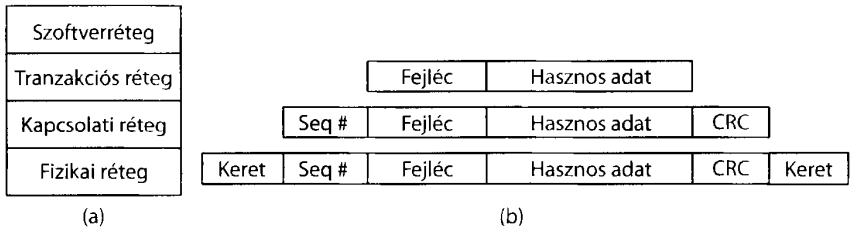
A PCI Express a régi PCI sín architektúrától három fontos dologban tér el. Kettőt már láttunk: a központosított kapcsoló a több leágazású sín helyett, valamint a keskeny közvetlen soros kapcsolat a széles, párhuzamos sín helyett. A harmadik ennél kifinomultabb. A PCI sín mögött rejlő elképzelés az, hogy a sínmester parancsot ad a szolgának, ha egy szót vagy egy adatblokkot be akar olvasni. A PCI Express modellje ezzel szemben az, hogy egyik eszköz egy adatcsomagot küld a másik eszköznek. A csomagfogalom, amely egy fejlécből és a hasznos adatból áll, a hálózatok világából származik. A fejléc vezérlési információt tartalmaz, így nincs szükség arra a sok vezérlőjelre, mint a PCI sínnél. A hasznos adat tartalmazza az átvinni kívánt adatot. Valójában egy PC a PCI Express-szel egy kicsi csomagkapcsolt hálózat.

E három fő területen szakítottak a múlttal, ezeken kívül azonban vannak még kisebb különbségek is. A negyedik különbség az, hogy hibajelző kódot használnak a csomagoknál, amellyel a PCI sínnél nagyobb fokú megbízhatóságot biztosítanak. Az ötödik, hogy a kapcsolat egy lapka és a soros kapcsoló között hosszabb lehet, elérheti az 50 cm-t; ez megengedi a rendszer áttervezését. A hatodik, hogy a rendszer bővíthető, mivel egy eszköz lehet akár egy másik soros kapcsoló is, amellyel kapcsolók fastruktúráját hozhatják létre. A hetedik különbség, hogy az eszközök melegen csatlakoztathatók, azaz csatlakoztathatók és eltávolíthatók a rendszerből, mialatt az tovább működik. Végül, a soros csatlakozók sokkal kisebbek, mint a régi PCI-csatlakozók, így az eszközök és a számítógépek sokkal kisebbek lehetnek. Mindent egybevéve, mindez jelentős eltávolodást jelent a PCI síntől.

A PCI Express protokollrendszer

Maradva a csomagkapcsolt hálózati modellnél, a PCI Express réteges kapcsolati protokollrendszerrel rendelkezik. A protokoll a két fél közötti párbeszédet irányító szabályok egy halmaza. A protokollrendszer protokollok hierarchiája, amelyben az egyes problémákkal külön rétegekben foglalkoznak. Példának vegyünk egy üzleti levelet. Vannak megállapodások a levél fejrészének tartalmáról és elhelyezkedéséről, a címzett adatairól, a dátumról, az üdvözlésről, a levél szövegéről, az aláírásról és így tovább. Vehetjük úgy, hogy ez a levél egy protokoll. Vannak további megállapodások a borítékról, például a méretéről, hogy hová kerül a feladó címe, és mi a formája, hová kerül a címzett és milyen formában, hová kerül a bélyeg és így tovább. Ez a két réteg és a hozzájuk tartozó két protokoll független. Például teljesen újraformázhatjuk a levelet, de ugyanazt a borítékot használjuk, vagy fordítva. A réteges protokollok a hatékony, moduláris tervezést szolgálják, és évtizedek óta széles körben elterjedtek a hálózati szoftverek világában. Ami újdonság, hogy most a sínhardverbe is beépítésre kerültek.

A PCI Express protokollrendszer a 3.58. (a) ábrán látható.



3.58. ábra. *(a) A PCI Express protokollrendszer. (b) A csomagok formátuma*

Vizsgáljuk meg az egyes rétegeket alulról felfelé. A legalsó réteg a **fizikai réteg**. Azzal foglalkozik, hogy biteket továbbít a küldőtől a fogadónak egy közvetlen kapcsolaton keresztül. Minden egyes kapcsolat egy vagy több szimplex (azaz egyirányú) csatornapárból áll. A legegyszerűbb esetben egy pár van, egy-egy csatorna mindkét irányban, de 2, 4, 8, 16 vagy 32 pár is megengedett. Egy-egy csatornát **sávnak** nevezzük. A sávok számának mindkét irányban azonosnak kell lennie. Az első generációs termékeknek bármely irányban legalább 2,5 Gbps (gigabit/s) sebességet kell biztosítaniuk, de azt remélik, hogy a sebesség hamarosan mindkét irányban eléri a 10 Gbps-t.

Az ISA/EISA/PCI sínekkel ellentétben a PCI Expressnek nincs fő órajel-generátora. Az eszközök azonnal elkezdhetnek adni, amint van elküldeni való adatuk. Ez a szabadság gyorsabbá teszi a rendszert, de egyúttal problémához is vezet. Tegyük fel, hogy az 1-es bit +3 V-tal kódolódik, a 0-s bit pedig 0 V-tal. Ha az első néhány bájt mind 0 bitekből áll, honnan tudja a fogadó, hogy adatok érkeznek. Elvégre egy 0 bit sorozat pontosan úgy néz ki, mint egy megszakadt kapcsolat. Ezt a problémát az úgynevezett **8b/10b kódolás**sal oldották meg. A megoldás az, hogy

10 bitet használnak egyetlen hasznos bájt kódolására egy 10 bites szimbólumban. Az 1024 lehetséges 10 bites értékből kiválasztották azokat a megengedhető szimbólumokat, amelyek elegendő jelváltást tartalmaztak ahhoz, hogy bithatárra szinkronizálják a küldőt és a fogadót fő órajel-generátor nélkül is. A 8b/10b kódolás következményeként a kapcsolat teljes 2,5 Gbps kapacitása csupán 2 Gbps hasznos (nettó) adatmennyiség átvitelére elegendő.

Amíg a fizikai réteg a bitek átvitelével foglalkozik, addig a **kapcsolati réteg** a csomagok átvitelével. Fogja a tranzakciós rétegtől kapott fejlécet és hasznos adatot, egy sorszámot és egy **CRC**-nek **(Cyclic Redundancy Check, ciklikus redundanciakód)** nevezett hibajelző kódot ad hozzá. A CRC egy bizonyos algoritmusnak a fejlécre és a hasznos adatokra történő futtatásával számítható ki. Amikor a csomag megérkezik, a fogadó is elvégzi ugyanezt a számítást a fejlécre és a hasznos adatokra, és összehasonlítja az eredményt a csomaghoz csatolt CRC-vel. Ha megegyeznek, egy rövid **nyugtázó csomag**ot küld vissza, ezzel jelzi a csomag hibátlan megérkezését. Ha a két eredmény nem egyezik, a fogadó kéri a csomag újraküldését. Ezen a módon az adatintegritás jelentősen javult a PCI sínhez képest, amely nem biztosított lehetőséget a sínen átküldött adatok ellenőrzésére és újraküldésére.

Annak elkerülésére, hogy egy gyors küldő eszköz olyan mennyiségű csomaggal árasszon el egy lassú fogadó eszközt, amelyet az nem tud kezelni, egy **folyamatvezérlő** mechanizmust használnak. A mechanizmus lényege, hogy a fogadó eszköz elküld a küldőnek egy bizonyos kredit (hitel) értéket, a beérkező csomagok tárolására rendelkezésre álló puffer méretét. Amikor a kredit elfogy, a küldőnek be kell fejeznie a küldést, amíg új kreditet nem kap. Ezt a sémát elterjedten használják mindenféle hálózatban, hogy megakadályozzák a küldő és a fogadó eltérő sebességéből eredő adatvesztést.

A **tranzakciós réteg** kezeli a síntevékenységeket. Egy szó olvasása a memóriából két tranzakciót kíván: az egyiket a CPU vagy egy DMA csatorna kezdeményezi, és kér bizonyos adatokat, a másikat az adatokat szolgáltató céleszköz. Azonban a tranzakciós réteg több annál, mint egyszerű olvasások és írások kezelése. A kapcsolati réteg által biztosított nyers csomagküldést további szolgáltatásokkal egészíti ki. Először is, minden egyes sávot feloszthat nyolc **virtuális áramkör**re, amelyek mindegyike különböző jellegű forgalmat bonyolít le. A tranzakciós réteg meg tudja címkézni a csomagokat, aszerint, hogy a nyolc forgalmi osztály melyikéhez tartoznak, a címkében lehetnek olyan attribútumok, mint a magas prioritás, alacsony prioritás, szimatolás tiltása, soron kívüli kézbesíthetőség és még sok egyéb. A kapcsolólapka használhatja ezeket a címkéket, amikor arról dönt, melyik lesz a legközelebb továbbítandó csomag.

Minden tranzakció az alábbi négy címtartomány egyikét használja:

1. Memóriaterület (közönséges olvasások és írások esetén);
2. B/K terület (a csatlakoztatott eszközök regisztereinek megcímzésekor);
3. Konfigurációs terület (a rendszer kezdeti beállításakor stb.);
4. Üzenetterület (jelzések küldésekor, megszakítások esetén stb.).

A memória- és a B/K terület hasonló a jelenlegi rendszerekben lévőkhöz. A konfigurációs terület olyan szolgáltatások megvalósítására szolgál, mint a plug-and-play („csatlakoztasd-és-működik"). Az üzenetterület a rengeteg létező hardvervezérlőjel szerepét veszi át. Valami ilyesmire szükség van, mivel a PCI sín vezérlőjelei közül egy sincs meg a PCI Expressnél.

A PCI Express rendszer a **szoftverrétegen** keresztül csatlakozik az operációs rendszerhez. Képes a PCI sín emulálására, így a létező operációs rendszerek változtatás nélkül tudnak működni a PCI Express rendszert alkalmazva. Természetesen ez a módszer nem hasznosítja a PCI Express minden erejét, azonban a visszafelé kompatibilitás szükséges rossz, amit nem hagyhatunk figyelmen kívül, amíg az operációs rendszereket nem módosítják úgy, hogy teljesen használatba vegyék a PCI Expresst. A tapasztalatok szerint ez eltarthat egy darabig.

Az információáramlást a 3.58. (b) ábra mutatja. Amikor a szoftverréteg parancsot kap, azt továbbadja a tranzakciós rétegnek, amely átalakítja a parancsot fejléc és hasznos adat alakúra. Ezt a két rész azután a kapcsolati réteghez kerül, amely egy sorszámot kapcsol a csomag elejéhez, és hibajelző kódot a végéhez. Az így bővített csomag kerül azután a fizikai réteghez, amely kerettel zárja le a csomag mindkét végét, hogy egy fizikai csomagot hozzon létre, amelyet aztán ténylegesen elküld.

Évtizedek óta nagy sikerrel használják a hálózatok világában azt az elképzelést, hogy minden réteg a protokoll rétegeiben egyre mélyebbre haladva további kiegészítő információt ad az adatokhoz. A hálózatok és a PCI Express között az a nagy különbség, hogy a hálózatok esetében a különböző rétegekben lévő kód majdnem mindig szoftver, amely az operációs rendszer része. A PCI Expressnél ez mind az eszközben lévő hardver része.

A PCI Express bonyolult dolog. További információért lásd (Mayhew és Krishnan, 2003; Solari és Congdon, 2005).

3.6.4. Univerzális soros sín

A PCI sín és a PCI Express kiválóan alkalmazható nagy sebességű perifériák számítógéphez kapcsolásához, azonban túlságosan drágák az alacsony sebességű perifériák, például az egér vagy a billentyűzet számára. Történelmileg minden szabványos B/K eszköz egyedi módon kapcsolódott a számítógéphez, amelyben az új eszközök számára volt néhány szabad ISA- vagy PCI-csatlakozó. Ez a megoldás már a kezdetektől problémás volt.

Például, az új perifériák gyakran saját ISA vagy PCI kártyával kaphatók. Általában a felhasználó felelős a kártyán lévő kapcsolók és vezetékáthidalások (jumper) helyes konfigurálásáért, valamint azért, hogy e beállítások ne kerüljenek konfliktusba más kártyákkal. Azután a felhasználónak fel kell nyitnia a számítógép házát, gondosan behelyezni az új kártyát, visszazárni a házat, majd újraindítani a számítógépet. Sok ember számára ez az eljárás nehézkes, és hibalehetőségeket rejt magában. Ráadásul az ISA- és PCI-csatlakozások száma meglehetősen korlátozott (általában kettő vagy három). A „csatlakoztasd és működik" kártyák

ugyan kiküszöbölik a vezetékáthidalásokkal való vesződést, azonban továbbra is a felhasználónak kell felnyitnia a számítógépet, és betennie a kártyát, valamint a sínCsatlakozások száma is korlátozott.

1993-ban hét vállalat – a Compaq, DEC, IBM, Intel, Microsoft, NEC és a Northern Telecom – képviselői összeültek, hogy megvitassák a problémát, és a számítógépek és az alacsony sebességű perifériák összekapcsolásának jobb módját dolgozzák ki. Azóta már cégek százai csatlakoztak hozzájuk. Az eredményül kapott szabványt, az **USB-t (Universal Serial Bus, univerzális soros sín)** hivatalosan 1998-ban jelentették be, és már széles körben alkalmazzák személyi számítógépekben. További információért lásd (Anderson, 1997; Tan, 1997).

Néhány cél azok közül, amelyeket az USB-t eredetileg kidolgozó és a munkát elindító cégek fogalmaztak meg:

1. A felhasználóknak ne kelljen minikapcsolókat (switch), vezetékátkötéseket (jumper) beállítaniuk sem a bővítőkártyákon, sem a készülékeken.
2. A felhasználóknak ne kelljen felnyitniuk a számítógép házát, hogy egy új perifériát csatlakoztathassanak.
3. Egyetlenfajta kábel legyen, amely az összes készülék számára alkalmas.
4. A perifériák az elektromos energiát is ezen a kábelen keresztül kapják.
5. Egyetlen számítógéphez legalább 127 eszközt lehessen csatlakoztatni.
6. A rendszer ki tudjon szolgálni valós idejű perifériákat is (például hang, telefon).
7. A készülékeket úgy is lehessen telepíteni, hogy a számítógép közben működik.
8. Ne kelljen újraindítani a számítógépet a készülék csatlakoztatása után.
9. Az új sín és a rajta lévő eszközök előállítási költsége ne legyen magas.

Az USB-szabvány kielégíti mindezeket az elvárásokat. Olyan alacsony sebességű eszközök számára tervezték, mint a billentyűzet, az egerek, állóképkamera, fényképszkenner, digitális telefon stb. Az USB 1.0-ás verziójának sávszélessége mintegy 1,5 MB/s, amely elegendő a billentyűzet és az egerek számára, az 1.1-es verzió pedig 12 MB/s sebességgel működik, amely elegendő a nyomtatók, digitális fényképezőgépek és sok más eszköz számára. Ezeket a viszonylag alacsony határértékeket a költségek alacsony szinten tartása miatt választották.

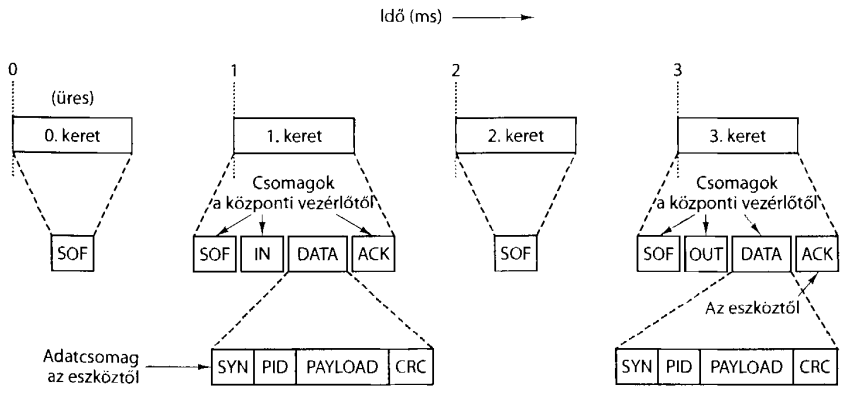
Egy USB-rendszer egy központi **csomópont**ból **(root hub)** áll, amely a rendszersínre csatlakozik (lásd 3.52. ábra). Ez a csomópont több csatlakozót tartalmaz a külső B/K eszközök vagy újabb csomópontok számára, amelyek további csatlakozási lehetőségeket biztosíthatnak, így az USB-rendszer topológiája egy olyan fa, amelynek a gyökere a számítógép belsejében lévő központi csomópont. A kábelek végén két különböző csatlakozó van, egyik csak a B/K eszközbe, a másik csak az elosztóba dugható be, ezzel meggátolható, hogy véletlenül két csomópont két csatlakozója legyen összekötve a kábellel.

A kábel négy vezetékszálat tartalmaz: kettőt az adatok, egyet a tápfeszültség (+5 V) és egyet a föld számára. A jeltovábbító rendszer a 0-kat feszültségátmenetként, az 1-eket pedig a feszültségátmenet hiányaként továbbítja, így egy hosszú 0 sorozat szabályos impulzusfolyamot generál.

Egy új eszköz csatlakoztatásakor a központi csomópont érzékeli az eseményt, és megszakítást kezdeményez. Ezután az operációs rendszer lekérdezi, milyen eszközről van szó, valamint hogy mekkora USB-sávszélességre van szüksége. Ha az operációs rendszer úgy dönt, hogy van elegendő sávszélesség az eszköz számára, akkor az eszköznek címet, egyedi azonosító számot (1–127) ad, majd ezt a címet és más paramétereket tölt a B/K eszközben lévő konfigurációs regiszterekbe. Ilyen módon az új eszközök menet közben csatlakoztathatók a számítógéphez anélkül, hogy a felhasználónak bármit is konfigurálnia kellene, vagy új ISA vagy PCI kártyákat kellene a számítógépbe helyeznie. A még konfigurálatlan eszköz címe kezdetben 0, tehát megcímezhető. A kábelezés megkönnyítése érdekében sok USB-eszköz beépített csomópontot tartalmaz, amelybe további USB-eszközök dughatók. Például egy USB-monitor rendelkezhet két csatlakozóval a bal, illetve a jobb oldali hangszóró számára.

Logikailag az USB-rendszer úgy tekinthető, mint bitcsatornák halmaza a központi csomóponttól az eszközökig. Minden eszköz legfeljebb 16 alcsatornára oszthatja fel a csatornáját a különböző jellegű adatok számára (például hang, video). Minden egyes csatornán vagy alcsatornán az adatok a központi csomópontból az eszköz felé vagy fordított irányban áramolnak. Két B/K eszköz között nincs adatforgalom.

Pontosan 1,00 ± 0,05 ms-ként a központi csomópont egy új üzenetváltási keretet (frame) küld szét, amelynek segítségével minden B/K eszközt szinkronizál. Egy üzenetváltási keret mindig egy bitcsatornához kapcsolódik, és csomagokból áll, amelyekből az elsőt minden esetben a központi csomópont küldi az eszköznek. A keretben lévő többi csomag iránya lehet ugyanilyen vagy az eszköztől a központi csomópont felé haladó. Egy négy keretből álló sorozat látható a 3.59. ábrán.



PAYLOAD = Hasznos adat

3.59. ábra. *Az USB központi csomópontja minden 1,00 milliszekundumban új üzenetváltási keretet küld*

A 3.59. ábrán a 0. és 2. keretben nincs feladat, ezért az egyetlen, amire szükség van egy SOF (Start of Frame) csomag elküldése. Ezt a csomagot a központi csomópont minden esetben minden eszköz számára továbbítja. Az 1-es keret egy lekérdezés, például kérés egy szkennerhez, hogy küldje el egy kép szkennelése során kapott bitsorozatot. A 3-as keret adatok küldése valamilyen eszköz, mondjuk, egy nyomtató számára.

Az USB négyféle keretet különböztet meg: vezérlési, izoszinkron, tömeges adat és megszakítási kereteket. A vezérlési keret az eszközök konfigurálására szolgál, parancsok küldésére vagy az eszközök állapotának lekérdezésére. Az izoszinkron kereteket a valós idejű eszközök használják, például a mikrofonok, hangszórók és telefonok, amelyeknek pontosan meghatározott időnként feltétlenül adatokat kell küldeniük vagy fogadniuk, bár a jeleknek jól kiszámítható késleltetése van. Hiba esetén az adatokat nem kell megismételni. A tömeges adat keretek nagy tömegű adat átvitelére szolgálnak a számítógéptől az eszközhöz, például a nyomtatókhoz, vagy fordítva, de valós idejű továbbításra nincs szükség. Végül, a megszakítási keretekre azért van szükség, mert az USB nem támogatja a megszakításokat. Például ahelyett, hogy a billentyűzet minden billentyű lenyomásakor megszakítást okozna, az operációs rendszer le tudja kérdezni 50 ms-ként, és össze tudja gyűjteni a billentyűlenyomásokat.

Egy keret egy vagy több csomagot tartalmaz, valószínűleg mindkét irányúból néhányat. Négy különböző csomag van: token, adat, kézfogás és speciális. A token csomagok a központi csomópont felől érkeznek az eszközökhöz, és a feladatuk a rendszer irányítása. A 3.59. ábrán látható SOF, IN és OUT ilyen token csomagok. A keret kezdete csomag (SOF) az első minden keretben; ez jelzi a keret kezdetét. Ha nincs teendő, a SOF lehet az egyetlen csomag a keretben. Az IN token csomag egy lekérdezés, az eszköztől kér bizonyos adatokat. Az IN csomagban lévő mezők azonosítják a bitalcsatornát, így az eszköz tudja, milyen adatokat kell visszaküldenie (ha több adatfolyama is van). Az OUT token csomag azt jelzi, hogy utána adatok következnek az eszköz számára. A negyedik token csomagtípus a SETUP (nem szerepel az ábrán), és az eszközök konfigurálására használják.

A token csomagon kívül még háromfajta csomag van. Ezek a DATA (legfeljebb 64 bájt adat küldésére használható tetszőleges irányban), a kézfogás és a speciális csomag. A DATA csomag formátuma szintén a 3.59. ábrán látható. Egy 8 bites szinkronizációs mezőből, egy 8 bites csomagtípusból (PID) és a hasznos adatból, valamint egy 16 bites **CRC (Cyclic Redundancy Check, ciklikus redundanciakód)** kódból áll, a hibás átvitel detektálására. Háromfajta kézfogás csomag van: az ACK (acknowledgement, a megelőző adatcsomag rendben megérkezett), a NAK (az átvitel során CRC hiba jelentkezett) és a STALL (kérem várjon – most éppen el vagyok foglalva).

Most vessünk ismét egy pillantást a 3.59. ábrára. Minden 1,00 milliszekundumban egy keretet kell küldenie a központi csomópontnak még akkor is, ha nincs semmilyen teendő. A 0. és 2. keretek csak egy SOF csomagot tartalmaznak, amely jelzi, hogy nincs teendő. Az 1. keret egy lekérdezés, amely tehát egy SOF-fal és egy IN-nel kezdődik, amelyeket a számítógép küld az eszközhöz, ezt követi egy DATA csomag az eszköztől a számítógépnek. Az ACK csomag jelzi az eszköznek, hogy a

csomag rendben megérkezett a számítógéphez. Hiba esetén a számítógép egy NAK csomagot küldene vissza az eszközhöz, ilyenkor a tömeges adatküldéshez tartozó csomagot újraküldi az eszköz (de izoszinkron keret esetén nem). A 3-as keret ugyanolyan szerkezetű, mint az 1-es, kivéve, hogy most az adatok ellenkező irányban, a számítógéptől az eszköz felé áramlanak.

Miután az USB-szabványt 1998-ban véglegesítették, az USB tervezőinek további feladatuk már nem volt, így az USB egy új, nagy sebességű, USB 2.0-nak nevezett verzióján kezdtek el dolgozni. A szabvány hasonlít a régebbi 1.1-es verzióhoz, és visszafelé kompatibilis is vele, azonban egy új harmadik sebességet (480 Mbps) vezet be a két meglévő mellé. Vannak még további kisebb különbségek is, mint például az interfész a központi csomópont és a vezérlő között. Az USB 1.1-ben kétféle interfész volt. Az elsőt, az UHCI-t (Universal Host Controller Interface), az Intel tervezte és a terhek zömét a szoftverkészítőkre (értsd: Microsoft) hárította. A másikat, az OHCI-t (Open Host Controller Interface) a Microsoft tervezte, és a terhek zömét a hardverfejlesztőkre (értsd: az Intel) hárította. Az USB 2.0-nál megegyeztek egyetlen új interfészben, amelynek a neve EHCI (Enhanced Host Controller Interface).

Most, hogy az USB 480 Mbps sebességgel működik, nyilvánvalóan versenytársa az IEEE 1394 szabványú sínnek, amelyet közismerten FireWire-nek neveznek, és ami 400 Mbps sebességgel működik. Bár majdnem minden új Pentium rendszer az USB 2.0-val kapható, mégsem valószínű, hogy az IEEE 1394 eltűnik, mert a szórakoztatóelektronikai ipar támogatja. A kézi kamerákat, DVD-lejátszókat és más hasonló eszközöket a belátható jövőben ezután is ellátják majd 1394-es interfésszel, mivel az eszközök gyártói nem akarják vállalni egy olyan szabványra történő áttérés költségeit, amely alig valamivel jobb, mint amijük jelenleg van. A vásárlók szintén nem szeretnek áttérni másik szabványra.

3.7. Kapcsolat a perifériákkal, interfészek

Egy tipikus kis vagy közepes számítógéprendszer CPU lapká(k)ból, memórialapkákból és néhány B/K vezérlőből áll, amit mind egy sín köt össze. A memóriákat, a központi egységeket és a síneket már megismertük bizonyos részletességgel. Itt az ideje, hogy megvizsgáljuk a kirakós játék utolsó darabját, a B/K lapkákat. A számítógép ezeken a lapkákon keresztül tart kapcsolatot a külvilággal.

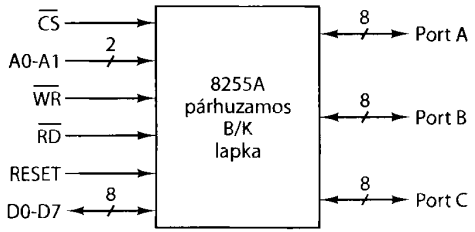
3.7.1. A B/K lapkák

Számos B/K lapka kapható, és állandóan egyre újabbakat hoznak forgalomba. A leggyakoribbak az UART, az USART, a képernyőmeghajtó, a diszkvezérlő és a PIO lapkák. Az **UART (Universal Asynchronous Receiver/Transmitter, univerzális aszinkron adó/vevő)** egy olyan lapka, amely egy bájtot tud olvasni az adatsínről, és azt bitenként továbbítja egy soros vonalon egy terminál felé, illetve soros

adatokat tud fogadni egy terminálról. Egy UART különböző sebességeket enged meg 50 bps-től 19 200 bps-ig; különböző karakterszélességekkel tud dolgozni 5–8 bit között; egy, másfél és két stop bittel; páros, páratlan vagy kikapcsolt paritást tud figyelni, mindezt program által vezérelt módon. Az **USART (Universal Synchronous Asynchronous Receiver/Transmitter, univerzális szinkron, aszinkron adó/vevő)** lapkák egyrészt különböző protokollok szerinti szinkron átvitelt tudnak kezelni, másrészt az UART lapkák minden funkcióját meg tudják valósítani. Mivel az UART lapkákat a 2. fejezetben már megismertük, most a párhuzamos interfész lapkát, a B/K lapkák egyik képviselőjét tanulmányozzuk.

PIO (Parallel Input/Output) lapkák

Egy tipikus **PIO (Parallel Input/Output, párhuzamos B/K)** lapka az Intel 8255A típusú, amely a 3.60. ábrán látható. E lapkának 24 B/K vonala van, amellyel bármilyen TTL kompatibilis eszközhöz tud kapcsolódni, például billentyűzethez, kapcsolókhoz, fényforrásokhoz vagy nyomtatókhoz. Dióhéjban, a központi egységben futó program 0-t vagy 1-et tud írni tetszőleges vonalára, vagy be tudja olvasni bármely vonal állapotát; ez nagyfokú rugalmasságot biztosít. Egy kis CPU-alapú rendszer PIO-val gyakran képes helyettesíteni egy teljes nyomtatott áramköri lapot tele SSI és MSI lapkákkal, különösen beágyazott rendszerekben.



3.60. ábra. *A 8255A PIO lapka*

Bár a központi egység számos módon tudja konfigurálni a 8255A lapkát a rajta lévő állapotregiszter feltöltésével, a következőkben néhány egyszerűbb működési módra fogunk szorítkozni. A legegyszerűbb módja a 8255A használatának, hogy három teljesen független 8 bites portra osztjuk, A-ra, B-re és C-re. Mindegyik port egy 8 bites tárolóregiszterrel van kapcsolatban. Ahhoz, hogy a port kimenő vonalait beállíthassuk, a központi egység beírja a megfelelő 8 bites számot a megfelelő regiszterbe, a 8 bites szám megjelenik a kimenő vonalakon, és ott meg is marad egészen a regiszter újraírásáig. Amikor a portot bemenetként használjuk, akkor a központi egység csupán a megfelelő tárolót olvassa ki.

Más működési módjai kézfogást biztosítanak a külső eszközökkel. Például, ha olyan eszköz számára küld adatokat, amely azokat nem mindig tudja fogadni, a 8255A megjeleníti az adatokat az egyik port kimenetén, majd impulzusra vár az

eszköztől, amely így jelez vissza, hogy fogadta az adatot és szeretne még továbbiakat. A 8255A-ba be van építve az a logikai áramkör, amely ahhoz szükséges, hogy ilyen impulzusokat tárolni tudjon, és azokat a központi egység számára hozzáférhetővé is tudja tenni.

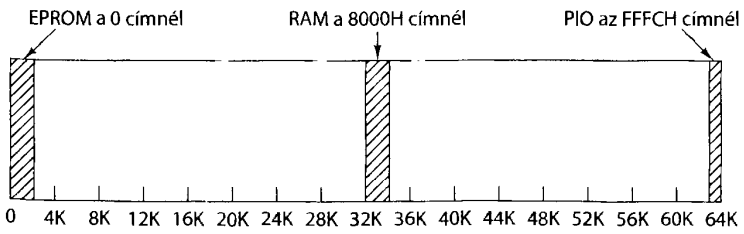
A 8255A funkcionális diagramjából látható, hogy a három porthoz kapcsolódó 24 érintkezőn kívül van még további 8 kivezetés, amelyek közvetlenül az adatsínre csatlakoznak, valamint egy lapkaválasztó, egy írás, egy olvasás, két cím és egy alapállapot kivezetése is. A két címvezetékkel lehet kiválasztani a négy belső regiszter közül az egyiket, az A, B, C portoknak és az állapotregiszternek megfelelőt. Az utóbbi bitjei határozzák meg, hogy mely portokat használjuk bemenetként, illetve kimenetként, és még további funkciókat. Általában a két címvezeték a címsín két legalsó vezetékéhez van kapcsolva.

3.7.2. Címdekódolás

Eddig meglehetősen homályos ismereteink voltak arról, hogyan állítódik be a lapkakiválasztás (chip select) a tárgyalt memória és a B/K lapkák esetében. Itt az ideje, hogy részletesebben is megvizsgáljuk, hogyan is történik ez. Tekintsünk egy egyszerű 16 bites számítógépet, amely egy központi egységből, a program számára egy 2 KB-os EPROM-ból, az adatok számára egy 2 KB-os RAM-ból és egy PIO lapkából áll. Ez a kis rendszer lehet például egy olcsó játék vagy egy háztartási gép agyának a prototípusa. Sorozatgyártásnál az EPROM-ot helyettesítheti majd egy ROM.

A PIO lapka kiválasztása a következő két módszer valamelyikével történhet: vagy mint valódi B/K eszköz, vagy mint a memória egy területe. Ha valódi B/K eszközként szeretnénk használni, egy külön sínvezetékre van szükség, amellyel jelezni tudjuk, hogy egy B/K eszközt szeretnénk használni, nem pedig a memóriát. Ha a másik módszert használjuk, a **memóriára leképezett B/K (memory-mapped I/O)** esetet, a memória 4 bájtját ki kell jelölnünk, hármat a három port és egyet a vezérlőregiszter számára. A hely kiválasztása majdnem tetszőleges. Mi most a memóriára leképezett B/K esetét fogjuk választani, mert segítségével bemutatható a B/K interfész kialakításának néhány érdekes kérdése.

Az EPROM 2 KB címterületet igényel, a RAM szintén 2 KB-t, a PIO pedig 4 bájtot. Mivel példánkban a teljes címtartomány 64 KB, ezért ki kell választanunk,



3.61. ábra. *Az EPROM, a RAM és a PIO elhelyezkedése a 64 KB címtartományban*

hová helyezzük el a három eszközt. Az egyik lehetséges megoldás a 3.61. ábrán látható. Az EPROM a 0-tól 2 KB-ig terjedő címterületet foglalja el, a RAM a 32 KB-tól 34 KB-ig terjedő területet, a PIO pedig a memória legfelső 4 bájtját 65 532-től 65 535-ig. A programozó szempontjából nincs különbség, hogy mely memóriacímeket használjuk, a csatlakozófelületre azonban már hatással van. Ha a PIO megcímzésére a másik módszert választottuk volna, a külön B/K címterületet, akkor az nem igényelt volna címtartományt a memóriában (viszont négy címet igényelne a B/K címtartományból).

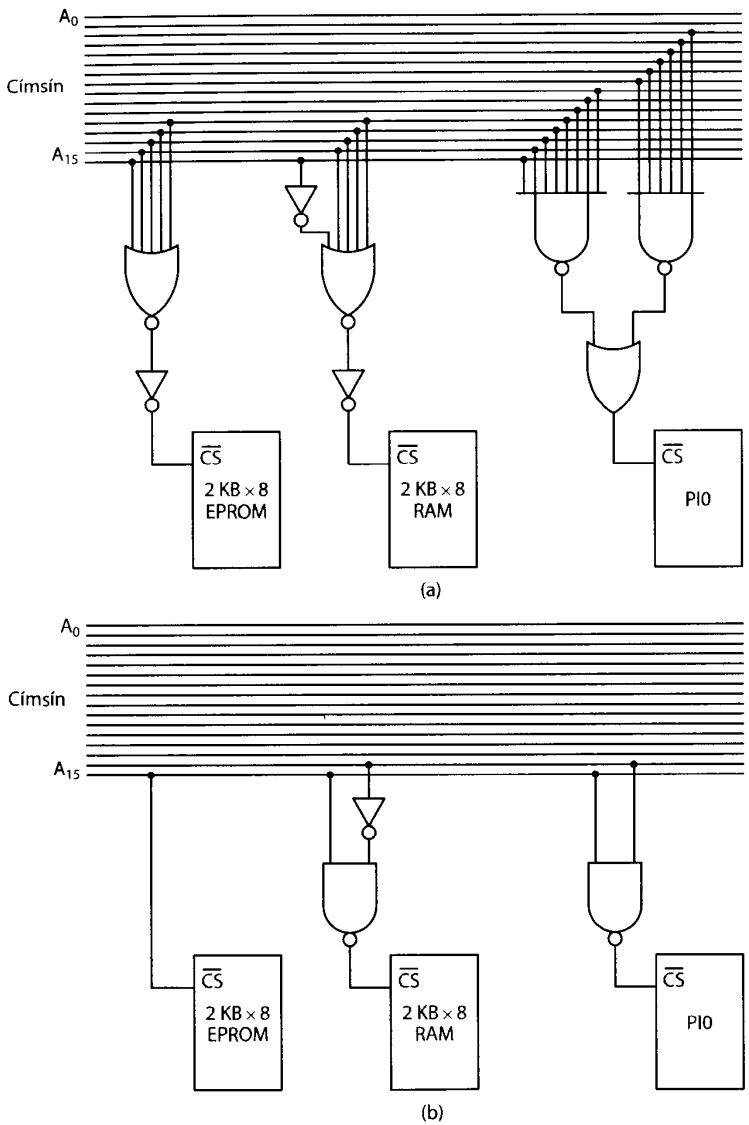
A 3.61. ábrán látható címkiosztás mellett az EPROM-ot kell kiválasztanunk minden olyan memóriacím esetén, amikor a 16 bites memóriacím 0000 0xxx xxxx xxxx (binárisan) formájú. Más szóval minden olyan cím, amelynek mind az 5 felső bitje 0, a memória alsó 2 KB területére, tehát az EPROM területére esik. Az EPROM lapka kiválasztó vezetékét egy 5 bites összehasonlító áramkör kimenetére kell kötni, az egyik bemenetét pedig a 00000-ra.

Ugyanennek az eredménynek az elérésére egy jobb megoldás, ha egy 5 bemenetű OR kaput használunk, amelynek az öt bemenetét az A11, ..., A15 címvezetékekre kapcsoljuk. Akkor és csak akkor lesz a kimenet 0, ha mind az öt bemenet 0, ezzel beállítja a $\overline{CS}$ jelet (a beállítását az alacsony feszültségszint jelzi). Sajnos, a szabványos SSI-áramkörök között nincs egyetlen ötbemenetű OR kapu sem. A legközelebbi, ami számításba jöhet, egy nyolcbemenetű NOR kapu. Három bemenetét a földre kötve, a kimenetét pedig invertálva megkaphatjuk a helyes jelet, ezt mutatja be a 3.62. (a) ábra. Az SSI-áramkörök olyan olcsók, hogy – bizonyos kivételes esetektől eltekintve – az sem számít, ha egyet kevésbé hatékonyan használnak fel. Megegyezés szerint az áramköri diagramokon nem tüntetjük fel a nem használt áramköri kivezetéseket.

Ugyanezt az elvet használhatjuk a RAM esetében is. Azonban a RAM-nak a 1000 0xxx xxxx xxxx alakú bináris címekre kell válaszolnia, ezért egy további inverterre is szükség van, ahogy az az ábrán látható. A PIO címdekódolás kissé bonyolultabb, mivel a PIO-t akkor kell választani, ha a cím 1111 1111 1111 11xx alakú. Egy lehetséges áramköri megoldás látható az ábrán, amely csak akkor állítja be a $\overline{CS}$ jelet, amikor a megfelelő cím megjelenik a címsínen. Két nyolcbemenetű NEM-ÉS kaput használ, amelyek egy OR kapu bemenetéhez vannak kapcsolva. Ahhoz, hogy a 3.62. (a) ábrán látható címdekódoló logikai áramkört SSI lapkákból kialakíthassuk, mindössze 6 lapkára van szükségünk, 4 nyolcbemenetű NEM-ÉS lapkára, egy VAGY kapura és egy olyan lapkára, amelyben három inverter van.

Ha azonban a számítógép valójában csak a központi egységből, két memórialapkából és a PIO-ból áll, a címdekódolás feladata egy trükkel nagymértékben egyszerűsíthető. A trükk azon alapul, hogy csak és kizárólag az EPROM címeinek esetében teljesül az, hogy a legfelső címvezeték, az A15 0. Ezért a $\overline{CS}$ lábat egyszerűen összeköthetjük az A15-tel, amint az a 3.62. (b) ábrán látható.

Ezen a ponton a RAM helyéül választott 8000H cím már kevésbé tűnik tetszőlegesnek. A RAM kiválasztása történhet csupán az alapján, hogy csak az 10xx xxxx xxxx xxxx alakú érvényes címek lehetnek a RAM-ban, ezért 2 bites dekódolás elegendő. Hasonlóan, minden cím, amely 11-gyel kezdődik, a PIO címe lesz. Az egész dekódoló logikai áramkör most csak két kétbemenetű NEM-ÉS kapuból és egy inverterből áll. Mivel az inverter kialakítható egy NEM-ÉS kapuból, a



3.62. ábra. *(a) Teljes címdekódolás. (b) Részleges címdekódolás*

két bemenő lábának összekötésével, egy egyszerű 4 NEM-ÉS kaput tartalmazó lapka bőven elegendő.

A 3.62. (b) ábrán megvalósított címdekódolást **részleges címdekódolás**nak nevezzük, mivel nem a teljes címet használja. Megvan az a tulajdonsága, hogy ha olvasunk a 0001 0000 0000 0000, 0001 1000 0000 0000 vagy a 0010 0000 0000 0000 címekről, ugyanazt az eredményt kapjuk. Valójában minden olyan cím esetén, amely a címtartomány alsó felére mutat, az EPROM lesz kiválasztva. Mivel az EPROM és RAM címein kívül más címeket nem fogunk használni, ez nem káros, ha azonban egy olyan számítógépet tervezünk, amelyet bővíteni szeretnénk a jövőben (ez egy játék esetén nem nagyon valószínű), a részleges címdekódolást el kell kerülni, mert túlságosan nagy címterületet köt le.

A másik lehetséges megoldás a címdekódolásra, ha olyan dekódoló áramkört használunk, mint ami a 3.13. ábrán látható. A három bemenő lábát a felső három címvezetékre kötve, nyolc kimenetet kapunk aszerint, hogy a cím az első 8 KB, a második 8 KB stb. területre esik. Egy olyan számítógép számára, amely 8 db 8 KB RAM-ot használ, egy ilyen dekódoló lapka a teljes címdekódolást megvalósítja. Egy olyan számítógép számára, amelyben 8 db 2 KB memória van, egyetlen ilyen dekóder szintén elegendő, feltéve hogy a lapkák mind a memóriatartomány különböző 8 KB méretű címterületén vannak. (Emlékezzünk vissza arra a korábbi megjegyzésünkre, hogy a memóriák és a B/K lapkák elhelyezkedése a címtartományban lényeges.)

3.8. Összefoglalás

A számítógépek integrált áramköri lapkákból épülnek fel, amelyek kicsiny kapcsolóelemeket, úgynevezett kapukat tartalmaznak. A leggyakoribb kapuk az ÉS, VAGY, NEM-ÉS, NEM-VAGY és NEM. A megfelelő kapuk közvetlen kombinálásával egyszerű áramkörök építhetők.

Bonyolultabb áramkörök a multiplexerek, demultiplexerek, kódolók, dekódolók, shift regiszterek és az aritmetikai-logikai egységek (ALU). Tetszőleges Boole-függvény megvalósítható programozható logikai mátrixszal (PLA). Ha sok Boole-függvényre van szükség, a PLA-áramkörök felhasználása még kifizetődőbb. A Boole-aritmetika törvényszerűségeinek segítségével transzformálhatjuk áramköreinket egyik formáról a másikra. Ezen a módon sok esetben gazdaságosabb áramköröket kaphatunk.

A számítógépes aritmetika eszközei az összeadó áramkörök. Egy 1 bites teljes összeadó kialakítható két félösszeadó áramkörből. Egy több bites összeadó felépíthető 1 bites teljes összeadókból úgy, hogy mindegyik összeadó átvitelkimenete hozzákapcsolódik a tőle balra álló szomszédja átvitelbemenetéhez.

A (statikus) memóriák fő építőelemei a tárolók és a billenőkörök (flip-flopok). Mindegyikük egyetlen bitinformációt tud tárolni. Kombinálhatók egyrészt lineárisan, így 8 bites tárolókat, illetve flip-flopokat kaphatunk, másrészt logaritmikusan, így pedig teljes, szószervezésű memóriákat kaphatunk. Sokféle memóriát ismerünk: RAM, ROM, PROM, EPROM, EEPROM és a flash memória. A statikus RAM-okat nem kell frissíteni; tartalmukat megőrzik egészen addig, amíg tápfeszültség alatt vannak. A dinamikus RAM-okat azonban állandóan frissíteni kell, hogy kompenzálni tudjuk a lapkán található kis kondenzátorokból a töltés elszivárgását.

A számítógépek alkotóelemeit sínek kötik össze. Egy tipikus CPU sok kivezetése, de azért nem mindegyik, vezérel közvetlenül egy sínvezetéket. A sín vezetékei feloszthatók cím-, adat- és vezérlővonalakra. A szinkron síneket egy fő órajel-generátor időzíti. Az aszinkron sínek teljes kézfogást használnak, hogy szinkronizálják a szolgát a mesterhez.

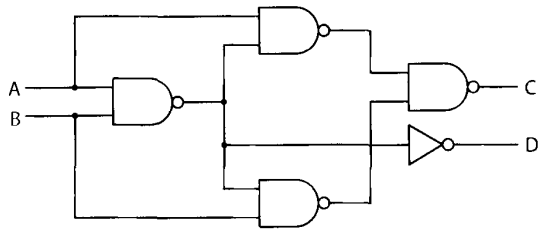
A Pentium 4 a modern központi egységekre példa. Az ilyet használó modern rendszereknek egy memóriasínje, egy PCI sínje, egy ISA sínje és egy USB sínje van. A PCI sín 64 bitet tud átvinni 66 MHz frekvenciával, amely elegendően gyors szinte minden periféria számára, azonban nem elég gyors a memória számára.

Kapcsolók, fényforrások, nyomtatók és sokféle más B/K eszköz kapcsolható össze a számítógépekkel olyan párhuzamos B/K lapkák segítségével, mint amilyen a 8255A. Ezek a lapkák konfigurálhatók úgy, hogy vagy a B/K címterület, vagy a memória címtartomány részei legyenek, a szükségleteknek megfelelően. Címzésük pedig lehet teljesen vagy részlegesen dekódolt, az alkalmazástól függően.

3.9. Feladatok

1. Egy matematikus behajtott egyszer egy autós étterembe, és ezt mondta: „Kérek egy hamburgert vagy egy hot dogot és sült krumplit." Sajnos, a szakács megbukott a hatodik osztályban, és nem tudta (vagy nem törődött azzal), hogy vajon az *és* precedenciája nagyobb-e, mint a *vagy*-é. Eddig meg volt győződve arról, hogy egyik interpretáció ugyanolyan jó, mint a másik. A következő esetek közül melyek a helyes interpretációi a rendelésnek? (Az angol *vagy* jelentése *kizáró-vagy*.)
   a) Csak egy hamburger.
   b) Csak egy hot dog.
   c) Csak sült krumpli.
   d) Egy hot dog és sült krumpli.
   e) Egy hamburger és sült krumpli.
   f) Egy hamburger és egy hot dog.
   g) Mind a három.
   h) Egyik sem – a matematikus éhesen tért haza, ha már ilyen nagyokos volt.
2. Egy misszionárius eltéved Dél-Kaliforniában, és megáll egy útelágazásnál. Tudja, hogy két motoros banda uralja a területet; az egyik mindig megmondja az igazat, a másik pedig mindig hazudik. Tudni szeretné, hogy melyik út vezet Disneylandbe. Milyen kérdést tegyen fel?
3. Igazságtáblázat segítségével bizonyítsa be, hogy $X = (X \text{ ÉS } Y) \text{ VAGY } (X \text{ ÉS NEM } Y)$.
4. Négy egyváltozós Boole-függvény létezik és 16 kétváltozós. Hány háromváltozós Boole-függvény létezik? Hány n változós létezik?
5. Mutassa be, hogyan lehet egy ÉS függvényt megvalósítani két NEM-ÉS kapuval.
6. A háromváltozós multiplexer lapka felhasználásával (lásd 3.12. ábra) készítsen egy függvényt, amelynek a kimenete az input paritása, azaz a kimenet akkor és csak akkor 1, ha páros számú input 1.

7. Most kösse jól fel a nadrágját. Egy háromváltozós multiplexer lapka (lásd 3.12. ábra) képes egy tetszőleges négyváltozós Boole-függvényt kiszámítani. Írja le hogyan, valamint példaként rajzolja le annak a függvénynek a logikai diagramját, amely 0, ha az igazságtábla sorában lévő érték angol számnévvel megnevezve páros számú betűből áll, és 1, ha páratlanból (pl. 0000 = zero = négy betű [páros] → 0; 0111 = seven = öt betű [páratlan] → 1; 1101 = thirteen = nyolc betű [páros] → 0). *Tipp:* ha a negyedik bemenő változót D-nek nevezzük, a nyolc input vonalat hozzákapcsolhatjuk D esetén a V_{cc}, esetén $\overline{D}$ a föld lábhoz.
8. Rajzolja le egy 2 bites kódoló áramkör logikai diagramját, amelynek négy bemenő vonala van, amelyből pontosan egy magas bármely időpillanatban, és két kimenő vezetéke, amelynek a 2 bites bináris értéke megmondja, melyik bemenő vonal a magas.
9. Rajzolja le egy 2 bites demultiplexer logikai diagramját, azt az áramkört, amelynek egyetlen input vonala a négy kimenő vonal egyikére irányítható a két vezérlőbemenet állapotától függően.
10. Rajzolja újra a 3.15. ábrán szereplő PLA (Programmable Logic Array, programozható logikai tömb) áramkört elég részletesen ahhoz, hogy bemutathassa hogy a 3.3. ábrán szereplő többségi függvény hogyan valósítható meg a felhasználásával. Különösen ügyeljen arra, hogy a megvalósított áramköri kapcsolatokat bemutassa mindkét mátrix esetében.
11. Mit valósít meg az ábrán látható áramkör?



12. Egy közönséges MSI lapka négybites összeadó. Négy ilyen lapkát összekapcsolva egy 16 bites összeadót kaphatunk. Ön szerint hány kivezetésének kell lennie egy 4 bites összeadó lapkának? Miért?
13. Egy n bites összeadó megalkotható n darab teljes összeadó sorba kapcsolásával úgy, hogy az i. helyen a C_i bemenő átvitel az $i - 1$. hely kimenő átvitele. A 0. szinten a C_0 bemenő átvitel értéke 0. Ha minden helynek T ns kell az eredmény és az átvitel kiszámításához, az i. szint átvitele nem érvényes iT ns-ig az összeadás kezdetétől számítva. Nagy n-re az átvitel végigvezetése az n. szintig elfogadhatatlanul nagy lehet. Tervezzen olyan összeadót, amely gyorsabban működik. *Tipp:* minden C_i kifejezhető az operandusainak, A_{i-1} és B_{i-1} bitjeivel, valamint a C_{i-1} átvitellel. Ennek a relációnak a felhasználásával kifejezhető a C_i a 0, ..., $i - 1$ szintek inputjainak függvényeként, azaz minden átvitelbit egyszerre generálható.

14. Ha a 3.19. ábrán látható minden kapunak 1 ns késleltetése van, és minden más késleltetés figyelmen kívül hagyható, mi az a legkorábbi időpont, amikor a diagram alapján működő áramkör Összeg kimenete biztosan érvényes lesz?
15. A 3.20. ábrán látható aritmetikai-logikai egység képes 8 bites 2-es komplemensű összeadásra. Vajon képes-e 2-es komplemensű kivonásra is? Ha igen, magyarázza el, hogyan. Ha nem, módosítsa úgy, hogy képes legyen a kivonás elvégzésére is.
16. Egy 16 bites ALU 16 db 1 bites ALU-ból épül fel, amelyek mindegyike 10 ns alatt végez el egy összeadást. Ha ezenkívül még 1 ns késleltetési idővel is kell számolnunk, amíg egyik ALU-tól a másikig terjed a jel, mennyi időbe telik, amíg a 16 bites eredmény a kimeneten megjelenik?
17. Sok esetben hasznos, ha egy 8 bites aritmetikai-logikai egység, mint amilyen a 3.20. ábrán látható, képes a –1 érték előállítására a kimenetén. Adjon meg két különböző módot is, ahogyan ez megtehető. Mindegyik megoldás esetén adja meg a 6 vezérlőjel értékét is.
18. Mi a nyugalmi állapota az *S* és *R* bemeneteknek egy SR tároló áramkör esetében, amely két NEM-ÉS kapuból épül fel?
19. A 3.26. ábrán látható áramkör egy flip-flop, amelyet az órajel felfutó éle vezérel. Módosítsa úgy az áramkört, hogy az órajel lefutó éle vezérelje.
20. A 3.29. ábrán látható 4 × 3 bites memória 22 ÉS kaput és 3 VAGY kaput használ. Ha az áramkört kibővítenénk 256 × 8-asra, hány kapura lenne szükségünk ezekből?
21. Hogy ki tudja fizetni az új PC-jének törlesztő részleteit, konzultációra kérték fel egy most induló SSI lapkakészítő üzemben. Egyik megrendelője egy olyan lapka kibocsátásán gondolkodik, amely 4 db D flip-flopot tartalmazna, mindegyiknek lenne Q és $\overline{Q}$ kimenete a potenciálisan fontos vevő kívánságára. A tervajánlat közösítette a négy órajelet, szintén kívánságra. Nincs viszont sem állapotbeállítás, sem törlés. Az Ön feladata a terv szakmai bírálata, kiértékelése.
22. Ahogyan egyre több és több memóriát préselnek rá egyetlen lapkára, úgy növekszik a címzéshez szükséges kontaktusok száma. Gyakran kényelmetlen, ha túl sok címkivezetés van egy lapkán. Találjon ki egy módszert arra, hogy 2^n memória szót n-nél kevesebb címvezetékkel címezhessünk meg.
23. Egy számítógép 32 bites adatsínnel rendelkezik és 1 M × 1 dinamikus RAM-memórialapkákat használ. Mekkora az a legkisebb memória (bájtokban), amelyet ez a számítógép tartalmazhat?
24. Tekintsük a 3.38. ábra időzítési diagramját. Tegyük fel, hogy lelassítjuk az órajelet, és egy órajel periódusideje 20 ns lesz az ábrán látható10 ns helyett, de a többi időzítési feltétel változatlan maradt. Mennyi ideje lenne legrosszabb esetben a memóriának arra, hogy a kért adatokat a sínen megjelenítse a T_3 periódusban, miután az $\overline{\text{MREQ}}$ beállt (0-ra)?
25. Továbbra is a 3.38. ábrára hivatkozunk, tegyük fel, hogy az órajel 100 MHz maradt, de T_{DS} megnövekedett 4 ns-ra. Használhatók-e továbbra is a 10 ns-os memórialapkák?

26. A 3.38. (b) ábra szerint T_{ML} legalább 3 ns. El tudna-e képzelni olyan lapkát, amelynél ez az érték negatív? Ebben az esetben, egy központi egység beállíthatja-e az $\overline{\text{MREQ}}$ jelet azelőtt, hogy a cím stabil lenne? Miért igen, vagy miért nem?
27. Tegyük fel, hogy a 3.42. ábra szerinti blokkátvitel a 3.38. ábrán látható sínen történik. Mennyivel nagyobb sávszélesség érhető el blokkátvitellel az egyedi átvitelhez képest hosszú blokkok esetén? Most tegyük fel, hogy az adatsín 32 bit széles 8 bit helyett. Válaszoljon ismét a kérdésre.
28. Jelölje a címvezetékek átviteli idejét a 3.39. ábrán T_{A1} és T_{A2}, az $\overline{\text{MREQ}}$ jel átviteli idejét T_{MREQ1} és T_{MREQ2} és így tovább. Írja le az összes, az átviteli időkre vonatkozó egyenlőtlenséget, amely teljes kézfogás esetén áll fenn.
29. A legtöbb 32 bites sín megenged 16 bites olvasást és írást is. Van-e bizonytalanság a címvezetékek értelmezésében, hogy hová kell az adatokat írni? Magyarázza meg.
30. Sok központi egységnek speciális sínciklusa van a megszakításkérések visszaigazolására. Miért?
31. Egy 64 bites számítógép 200 MHz sínfrekvenciával 4 órajelciklus alatt olvas be egy 64 bites szót a memóriából. A sín mekkora sávszélességét használja legfeljebb a központi egység?
32. Egy 32 bites központi egység az A2–A31 címvezetékekkel megköveteli, hogy a memóriahivatkozások szóhatárra legyenek igazítva. Azaz a memóriaszavak címe csak 4-gyel osztható, a félszavak címe pedig csak páros szám lehet. A bájtok lehetnek bármilyen címen. Hány különféle megengedett kombinációja van a memóriaeléréseknek, és hány lábra van szükség, hogy ezeket az eseteket megkülönböztethessük? Adjon két választ és egy-egy példát mindegyikre.
33. Miért lehetetlen egy Pentium 4 számára, hogy egy 32 bites PCI sínen dolgozzon anélkül, hogy elveszítené funkcionalitásának egy részét? A 64 bites adatsínnel rendelkező számítógépek mégiscsak tudnak 32 bites, 16 bites, sőt még 8 bites átvitelt is.
34. Tegyük fel, hogy egy központi egység 1. szintű gyorsítótára 1 ns, 2. szintű gyorsítótára 2 ns elérési idővel rendelkezik. A fő memória elérési ideje 10 ns. Ha az összes hivatkozás 20%-a az első szintű gyorsítótárban található, 60%-a a 2. szintű gyorsítótárban, mekkora az átlagos elérési idő?
35. Valószínű-e, hogy egy 8051 alapú beágyazott rendszerben legyen egy 8255A PIO lapka?
36. Számítsa ki a szükséges sávszélességet a sínen, hogy egy VGA (640 × 480), 24 bites színű (True color) mozifájlt megjelenítsünk 30 kép/s sebességgel. Tegyük fel, hogy az adatok kétszer mennek át a sínen, egyszer a CD-ROM felől a memóriába, majd a memóriából a képernyőt meghajtó kártyához.
37. Mit gondol, a Pentium 4 melyik kimenete vezérli a PCI sín FRAME# vonalát?
38. A 3.56. ábrán látható jelek közül, melyek nem feltétlenül szükségesek ahhoz, hogy a sínprotokoll működjön?
39. Egy PCI Express rendszernek 5 Mbps sebességű vonalai vannak (teljes sávszélesség). Hány vezetékre lenne szükség mindegyik irányban a nyolcszoros sebességű működéshez? Mi a teljes kapacitás az egyes irányokban? Mi a nettó sávszélesség az egyes irányokban?

40. Egy számítógép olyan gépi utasításokkal rendelkezik, amelyek mindegyike két órajelciklust igényel, egyet az utasítás, egyet pedig az adat betöltésére. Minden órajel 10 ns, és így minden utasítás 20 ns (a belső végrehajtási idő elhanyagolható). A számítógépnek van egy mágneslemeze, amelyen sávonként 2048 db 512 bájtos szektor van. A diszk forgási ideje 5 ms. A normál sebesség hány százalékára esik vissza a számítógép sebessége DMA-átvitel alatt, ha minden 32 bites DMA-átvitel 1 órajelciklust igényel?
41. A maximális hasznos adatméret egy izoszinkron adatcsomagban az USB sínen 1023 bájt. Tegyük fel, hogy egy eszköz csak egy adatcsomagot tud küldeni egy keretben, mi a maximális sávszélessége egy izoszinkron eszköznek?
42. Mi lenne a hatása annak, ha egy harmadik bemeneti vonalat kacsolnánk a 3.62. (b) ábrán látható áramkörben a PIO-t kiválasztó NEM-ÉS kapuhoz, és ezt a vezetéket az A13-hoz kapcsolnánk?
43. Írjon egy programot, amely szimulálni tudja egy $m \times n$-es két bemenetű NEM-ÉS kapukból álló mátrix viselkedését. Az áramkör, amely egyetlen lapkán foglal helyet, j bemenő és k kimenő vonallal rendelkezik. j és k, illetve m és n fordítási paraméterek legyenek. A program azzal kezdődjön, hogy beolvas egy „huzalozási listát". Minden vezeték bemeneteket és kimeneteket köt össze. Bemenetként a j bemenő vonal valamelyike vagy valamelyik NEM-ÉS kapu kimenete fordulhat elő. Kimenetként pedig a k kimenő vonal valamelyike vagy valamelyik NEM-ÉS kapu bemenete fordulhat elő. A nem használt bemenő vonalak jelszintje 1. Miután a program beolvasta a huzalozási listát, nyomtassa ki a k kimenő vonal állapotát az összes különböző 2^j bemenet esetén. A kapumátrix lapkákat, mint ez is, elterjedten használják arra, hogy a megrendelők igénye szerinti áramköröket egy lapkán megvalósítsák, mivel a munka nagyobbik része (kapumátrix kialakítása a lapkán) független magától az áramkörtől. Egyedül a huzalozás változik áramkörről áramkörre.
44. Írjon programot, amely beolvas két Boole-kifejezést, és megvizsgálja, hogy ugyanazt a függvényt valósítják-e meg. A bemenő nyelv betűkből, azaz Boole-változókból, az ÉS, VAGY, NEM logikai műveletekből és zárójelekből álljon. Minden kifejezésnek el kell férnie egyetlen sorban. A programnak ki kell számítania a két kifejezés igazságtáblázatát, és össze kell hasonlítania a kettőt.
45. Írjon egy programot, amely néhány Boole-kifejezést olvas be, majd kiszámítja azokat a 24 × 50-es vagy 50 × 6-os mátrixokat, amelyek szükségesek ahhoz, hogy a 3.15. ábrán látható PLA áramkörrel megvalósíthassuk azokat. A bemenő nyelv ugyanaz legyen, mint amit az előző, 44. probléma megoldásánál használt. A mátrixokat nyomtassa ki.

4. A mikroarchitektúra szintje

A digitális logika szintje feletti szint a mikroarchitektúra szintje. Feladata a felette lévő ISA-szint (utasításrendszer-architektúra szintje) megvalósítása, ahogy ezt az 1.2. ábrán bemutattuk. A mikroarchitektúra-szintjének tervezése függ az ISA-szint megvalósításától, valamint a számítógép ár- és teljesítmény-célkitűzéseitől. Sok modern ISA, kiváltképpen, ha RISC-elgondoláson alapuló, rendelkezik olyan egyszerű utasításokkal, amelyek általában egyetlen ciklusidő alatt végrehajthatók. Összetettebb ISA-k, mint a Pentium 4, egyetlenegy utasítás végrehajtásához több ciklust igényelhetnek. Egy utasítás végrehajtása igényelheti az operandusok helyének meghatározását a memóriában, azok kiolvasását és az eredmény visszatárolását a memóriába. Az egyetlen utasításon belüli műveletek sorba állítása gyakran vezet a vezérlés másfajta megközelítéséhez, mint az az egyszerű ISA-kban van.

4.1. Mikroarchitektúra-példa

Az lenne ideális, ha úgy vezethetnénk be ezt a témát, hogy elmagyarázzuk a mikroarchitektúra-tervezés általános alapelveit. Sajnos, azonban nincsenek ilyen alapelvek, minden eset speciális. Ezért részletes példákat fogunk helyette tárgyalni. ISA-példánknak a Java Virtual Machine (Java virtuális gép) egy részhalmazát választottuk, ahogy azt az 1. fejezetben megígértük. Ez a részhalmaz csak egész utasításokat tartalmaz, így elneveztük **IJVM**-nek.

Az a tervünk, hogy leírjuk a mikroarchitektúrát a legmagasabb szinten, melyen az IJVM-et meg fogjuk valósítani. Az IJVM-nek van néhány bonyolult utasítása. Sok hasonló architektúra gyakran a mikroprogramozás eszköztárával kerül implementálásra, ahogy azt az 1. fejezetben láttuk. Bár az IJVM kicsi, mégis jó kiindulási pont az utasítások vezérlésének és sorba állításának leírására.

Mikroarchitektúránk tartalmazni fog egy mikroprogramot (ROM-ban), melynek feladata az IJVM-utasítások betöltése, dekódolása és végrehajtása. Nem tudjuk használni a Sun JVM-interpretert a mikroprogramhoz, mert egy olyan kis mikroprogramra van szükségünk, amely hatékonyan vezérli az egyes kapukat az aktuális hardverben. Ezzel ellentétben a Sun JVM-interpreter C-ben készült a

hordozhatóság érdekében, és nem képes vezérelni a hardvert azon a részletezettségi szinten, amire szükségünk van. Mivel az általunk használt aktuális hardver elméletileg csak a 3. fejezetben leírt alapelemekből áll, a fejezet teljes megértése után az olvasónak képesnek kellene lennie arra, hogy elmenjen, és egy nagy szatyor tranzisztort véve felépítse a JVM gép egy részét. Azoknak a hallgatóknak, akik sikeresen teljesítik ezt a feladatot, extra kreditpont jár (és pszichiátriai kivizsgálás).

Megfelelő modell a mikroarchitektúra-tervezés számára, ha a tervezésre mint programozási problémára gondolunk, ahol az ISA-szint minden egyes utasítása egy függvény, melyet a főprogram hív meg. Ebben a modellben a főprogram egy egyszerű végtelen ciklus, amely meghatározza a meghívandó függvényt, meghívja azt, és kezdi újra az egészet, nagyon hasonlóan, mint a 2.3. ábrán.

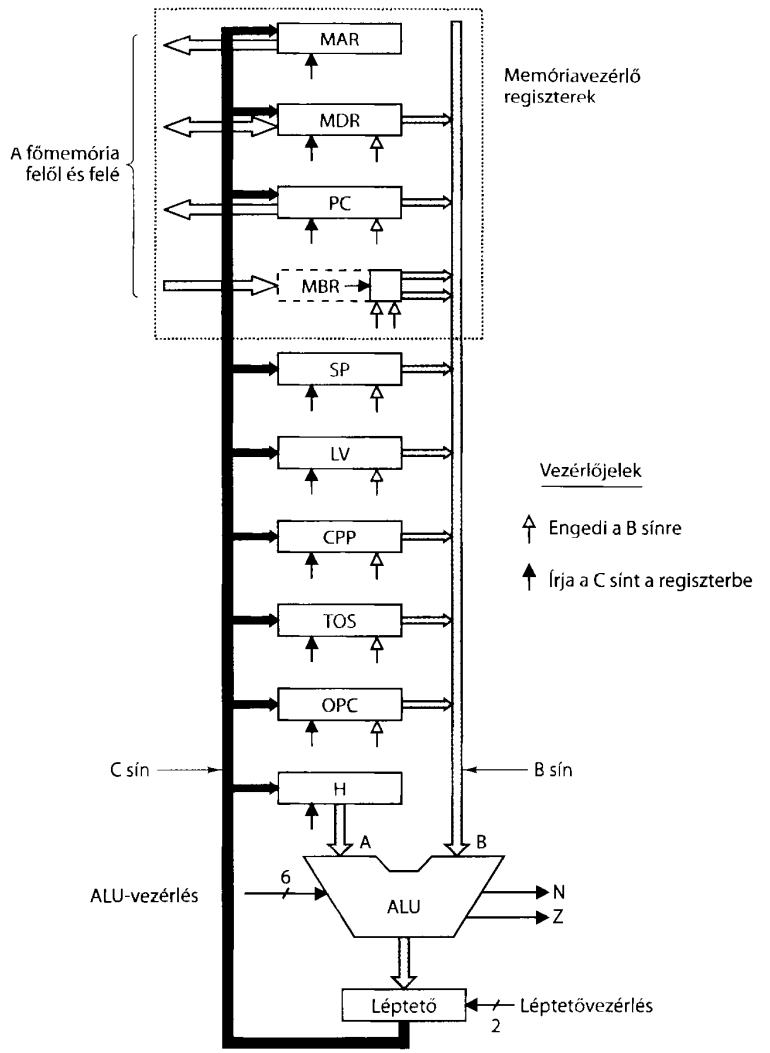
A mikroprogram változók egy halmazával rendelkezik, amit a számítógép **állapotának** nevezünk, és ezt minden függvény el tudja érni. Minden függvény megváltoztatja legalább az állapotot alkotó változók egy részét. Például az utasításszámláló (PC) az állapot része. Ez mutatja azt a memóriahelyet, amely tartalmazza a következő végrehajtandó függvényt (vagyis ISA-utasítást). Minden utasítás végrehajtása alatt a PC előrehalad, hogy a következő végrehajtandó utasításra mutasson.

Az IJVM utasításai rövidek és velősek. Minden utasítás tartalmaz néhány – általában egy vagy két – mezőt, és mindegyiknek van valami speciális szerepe. Minden utasítás első mezője a **művkód** (a **műveleti kód** rövidítése), amelyik azonosítja az utasítást, és megmondja, hogy ez ADD vagy BRANCH, vagy valami más. Sok utasításnak van egy további mezője is, amely az operandust határozza meg. Az olyan utasításoknak például, melyek lokális változót használnak, szükségük van egy mezőre, amelyik megmondja, hogy *melyik* változót használják.

A végrehajtásnak ez a **betölt-végrehajt ciklus**ként is nevezett modellje hasznos az elmélet szempontjából, és alapja lehet olyan ISA-k megvalósításának is, mint az IJVM, melynek bonyolult utasításai vannak. Az alábbiakban le fogjuk írni, hogy ez hogyan működik, hogy néz ki a mikroarchitektúra, ezt miként vezérlik a mikroutasítások, amelyek mindegyike az adatutat vezérli egy cikluson belül. A mikroutasítások listája együttesen mikroprogramot alkot, amelyet részletesen bemutatunk és megtárgyalunk.

4.1.1. Adatút

Az **adatút** a CPU-nak az a része, amelyik tartalmazza az ALU-t bemeneteivel és kimeneteivel együtt. Mikroarchitektúránk adatútja a 4.1. ábrán látható. Miközben gondosan optimizáltuk az IJVM-programok interpretálásához, meglehetősen hasonlóvá vált a legtöbb gépben használatos adatúthoz. Sok 32 bites regisztert tartalmaz, melyekhez szimbolikus neveket rendelünk, olyanokat mint PC, SP és MDR. Bár a nevek némelyike ismerős, fontos megértenünk, hogy ezek a regiszterek csak a mikroarchitektúra szintjén érhetők el (a mikroprogram által). Azért kapták ezeket a neveket, mert gyakran az ISA-szintű architektúra ugyanilyen nevű változójával megegyező értéket tartalmaznak. A legtöbb regiszter tartalmát a B sínre lehet irányítani. Az ALU kimenete a léptetőbe vezet, ezután a C sínre kerül, melynek értéke egy időben egy vagy több regiszterbe írható. Egyelőre nincs A sín, később majd hozzávesszük.



4.1. ábra. *Az ebben a fejezetben használt mikroarchitektúra-példa adatútja*

Az ALU megegyezik a 3.19. és 3.20. ábrán bemutatottal. Tevékenységét hat vezérlővonal határozza meg. A 4.1. ábrán a rövid vízszintes vonal „6" címkéje az

ALU hat vezérlővonalára utal. Közülük F_0 és F_1 határozza meg az ALU-műveletet, ENA és ENB egyedileg engedélyezi a bemeneteket, INVA invertálja a bal oldali bemenetet, és INC a legalacsonyabb helyértékű biten az „átvitel be" bemenet. INC beállítása gyakorlatilag 1-et hozzáad az eredményhez. Azonban az ALU-t vezérlő jelek nem mind a 64 kombinációja tesz valami hasznosat.

A 4.2. ábrán bemutatunk néhányat az érdekesebb kombinációkból. Az IJVM-hez nem szükséges ezen tevékenységek mindegyike, de a teljes JVM-hez sok közülük igen hasznos lehet. Sok esetben többféle lehetőségünk is van, hogy ugyanazt az eredményt elérjük. Ebben a táblázatban a + aritmetikai összeadásjelet, a – aritmetikai kivonásjelet jelöl, vagyis például $-A$ az A kettes komplemensét jelenti.

F0	F1	ENA	ENB	INVA	INC	Tevékenység
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	B
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

4.2. ábra. *Az ALU jeleinek hasznos kombinációi és az elvégzett tevékenység*

A 4.1. ábra ALU-jának két adatbemenetre van szüksége: a bal oldalira (A) és a jobb oldalira (B). A bal oldali bemenet a H (Holding) tartó regiszterhez kapcsolódik. A jobb oldali bemenet pedig a B sínhez, amelyik képes betölteni a szürke nyilak által megmutatott kilenc forrás bármelyikét. Egy alternatív terv két teljes sínnel a kompromisszumok egy másik halmazával rendelkezik, ezt ebben a fejezetben később tárgyaljuk.

A H úgy tölthető fel, hogy választunk egy olyan ALU-tevékenységet, amelyik a jobb oldali bemenetet (a B sínről) csupán átengedi az ALU kimenetére. Az egyik ilyen tevékenység az ALU bemeneteit összeadja, csak negált ENA-val, ami a bal oldali bemenetet nullává teszi. Nullát adva a B sín értékéhez, éppen a B sín értékét kapjuk. Ezt az eredményt azután módosítás nélkül küldjük tovább a léptetőn, és H-ban tároljuk.

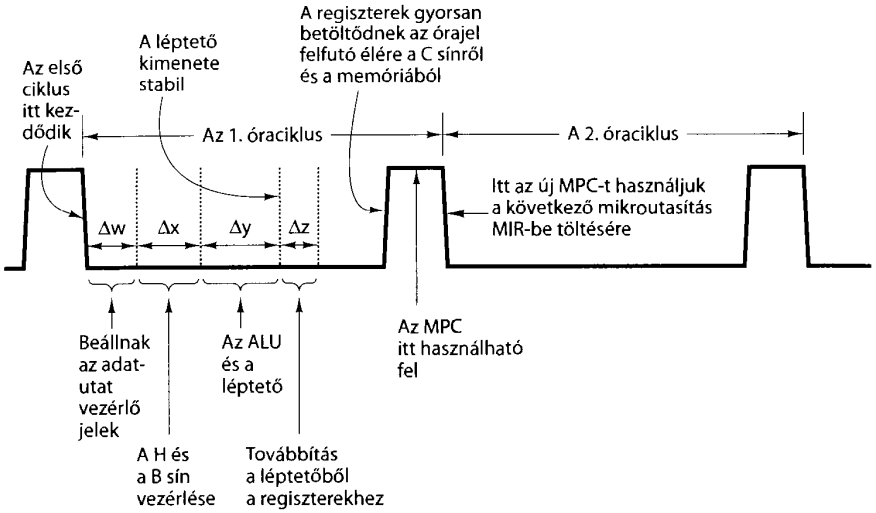
A fenti függvényeken kívül két másik vezérlővonal használható egymástól függetlenül az ALU kimenetének irányítására. Az SLL8 (Shift Left Logical, léptetés balra logikailag) a tartalmat 1 bájttal balra lépteti, feltöltve nullával a legalacsonyabb helyértékű 8 bitet. SRA1 (Shift Right Arithmetic, aritmetikai léptetés jobbra) a tartalmat 1 bittel jobbra lépteti, változatlanul hagyva a legmagasabb helyértékű bitet.

Hangsúlyozzuk, hogy lehetséges olvasni és írni ugyanazt a regisztert egy cikluson belül. Például megengedett, hogy az SP-t kitegyük a B sínre, tiltsuk az ALU bal oldali bemenetét, engedélyezzük az INC jelet, és tároljuk az eredményt SP-be, így megnövelve SP értékét 1-gyel (lásd 4.2. ábra 8. sor). Hogy miként olvasható és írható egy regiszter ugyanabban a ciklusban anélkül, hogy szemét keletkezne? A magyarázat az, hogy az olvasás és az írás valójában egy cikluson belül különböző időpillanatban hajtódik végre. Amikor kiválasztódik, hogy melyik regiszter legyen az ALU jobb oldali bemenete, akkor ennek az értéke a ciklus elején a B sínre kerül, és folyamatosan ott is marad csaknem a teljes ciklus alatt. Az ALU ekkor elvégzi a munkáját, az eredményt a léptetőn keresztül a C sínre küldi. A ciklus vége felé, amikor az ALU és a léptető kimenetéről tudjuk, hogy stabil, az órajel kiváltja a C sín tartalmának tárolását egy vagy több regiszterbe. Az egyik ilyen regiszter lehet az, amelyik bemenetként szolgált a B sín számára. Az adatút pontos időzítése lehetővé teszi ugyanannak a regiszternek egy cikluson belüli olvasását és írását, ahogy alább részletezzük.

Az adatút időzítése

Az események időzítését a 4.3. ábra mutatja. Itt egy rövid impulzus képződik minden óraciklus kezdetekor. Ez származhat a fő órától, mint az a 3.21. (c) ábrán látható. Az impulzus lefutó élén beállnak azok a bitek, amelyek az összes kaput irányítani fogják. Ez egy meghatározott és ismert Δw idő alatt bekövetkezik. Ezután kiválasztódik az a regiszter, amelyre a B sínnek szüksége van, és a B sínre kapcsolódik. Δx idő telik el, amíg az érték stabil lesz. Ekkor az ALU és a léptető elkezdi végrehajtani a műveletet az érvényes adaton. Egy másik Δy múlva az ALU és a léptető kimenetei stabilak lesznek. Egy további Δz után az eredmények továbbterjednek a C sín mentén a regiszterekhez, ahonnan betöltődhetnek a következő impulzus felfutó élénél. A betöltésnek élvezéreltnek és gyorsnak kell lennie, vagyis még akkor is, ha valamelyik input regiszter megváltozik, a következmények nem lesznek érzékelhetők a C sínen a regiszterek betöltése után még egy jó ideig. Ugyancsak az impulzus felfutó élénél a regiszter és a B sín kapcsolata megszakad, felkészülve a következő ciklusra. Az ábrán említett MPC, MIR és a memória szerepére rövidesen visszatérünk.

Fontos, hogy tisztában legyünk azzal, hogy bár nincsenek tárolóelemek az adatútban, van rajta egy meghatározott terjedési idő. A B sínen lévő érték változása (minden egyes lépés meghatározott késleltetése következtében) nem eredményezi a C sín változását, amíg egy meghatározott idő el nem telik. Tehát még akkor is, ha egy tárolás meg fogja változtatni az egyik bemeneti regiszter értékét, a korábbi



4.3. ábra. *Egy adatútciklus időzítési diagramja*

érték biztonságosan eljut az ALU-ba, jóval előbb, semhogy az új (most helytelen) érték rákerülve a B sínre(vagy H-ra), elérhetné az ALU-t.

Az ilyen tervezési munka megköveteli a szigorú időzítést, egy hosszú óraciklust, az ALU-n keresztüli minimális terjedési idő ismeretét, és a C sínről a regiszterbe történő gyors betöltést. Azonban gondos mérnöki munkával az adatút úgy megtervezhető, hogy mindig helyesen működjön. A tényleges gépek így működnek.

Kissé különböző módja az adatútciklus vizsgálatának, ha úgy tekintjük, hogy magától értetődő részciklusokra darabolódik. Az 1. részciklus kezdetét az órajel lefutó éle vezérli. Az alábbiakban a részciklusok alatti tevékenységeket mutatjuk be a részciklus hosszával (zárójelben) együtt.

1. A vezérlőjelek beállnak (Δw).
2. A regiszterek a B sínre töltődnek (Δx).
3. Az ALU és a léptető működik (Δy).
4. Az eredmények a C sín mentén visszakerülnek a regiszterekhez (Δz).

A következő óraciklus felfutó élénél az eredmények eltárolódnak a regiszterekbe.

Azt mondjuk, hogy a legjobb megközelítés az, hogy a részciklusok *értelemszerűek*. Ezen azt értjük, hogy nincs óra- vagy egyéb meghatározott jel, amelyik megmondja az ALU-nak, hogy mikor működjön, vagy megmondja az eredményeknek, hogy lépjenek a C sínre. A valóságban az ALU és a léptető mindvégig működik. Azonban a bemeneteik szemetek az órajel lefutó éle után $\Delta w + \Delta x$ ideig. Hasonlóan a kimeneteik szemetek, amíg el nem telik az órajel lefutó éle után $\Delta w + \Delta x + \Delta y$ idő. Az egyedüli meghatározott jelek, amelyek az adatutat vezérlik, az órajel lefutó éle,

amely indítja az adatútciklusát, valamint az órajel felfutó éle, amely feltölti a regisztereket a C sínről. A többi részciklus határait értelemszerűen határozzák meg a beépített áramkörökben rejlő terjedési idők. A tervezőmérnökök felelőssége annak biztosítása, hogy a $\Delta w + \Delta x + \Delta y + \Delta z$ idő eléggé az órajel felfutó éle előtt teljék le ahhoz, hogy a regiszterek betöltése idejében megtörténjék.

A memóriaművelet

Gépünknek két különböző módja van, hogy a memóriával kommunikáljon: egy 32 bites, szócímzésű memóriaport és egy 8 bites, bájtcímzésű memóriaport. A 32 bites portot két regiszter vezérli, a MAR (**Memory Address Register, memóriacímregiszter**) és az MDR (**Memory Data Register, memóriaadat-regiszter**), ahogy ez a 4.1. ábrán látható. A 8 bites portot egy regiszter, a PC vezérli, amely egy bájtot olvas be az MBR alsó 8 bitjébe. Ez a port csak adatot tud olvasni a memóriából; nem képes abba adatot írni.

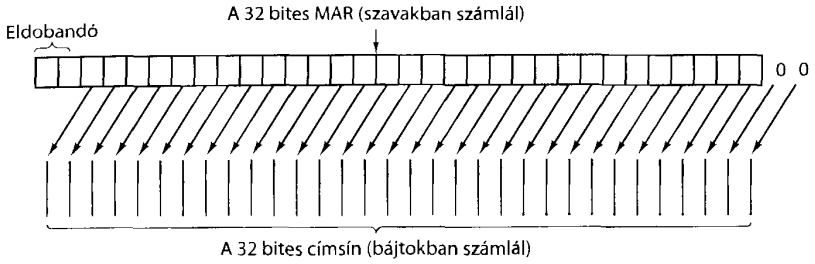
Ezeket a regisztereket (és a 4.1. ábra többi regiszterét is) egy vagy két **vezérlőjel** irányítja. A regiszter alatti nyitott nyíl azt a vezérlőjelet jelzi, amelyik engedi a regiszterek kimenetét a B sínre. Mivel a MAR nem kapcsolódik a B sínre, nincs engedélyező jele sem. A H-nak sincs egyikből sem, mert mindig engedélyezett, hiszen ez az ALU egyetlen lehetséges bal oldali bemenete.

A regiszter alatti, tömör fekete nyíl azt a vezérlőjelet jelzi, amelyik írja (vagyis betölti) a regisztert a C sínről. Mivel az MBR nem tölthető a C sínről, ezért nincs írójele se (bár van két másik engedélyező jele, amit alább részletezünk). Egy memóriaolvasás vagy -írás kezdeményezéséhez a megfelelő memóriaregisztereket fel kell tölteni, majd egy olvasó- vagy írójelet kell kibocsátani a memória felé (ez utóbbit nem mutatja a 4.1. ábra).

A MAR *szó*címeket tartalmaz úgy, hogy a 0, 1, 2 stb. értékek egymást követő szavakra hivatkoznak, a PC *bájt*címeket tartalmaz úgy, hogy a 0, 1, 2 stb. értékek egymást követő bájtokra hivatkoznak. Tehát 2-t téve a PC-be, és elindítva egy olvasást a memóriából, kiolvassa a memória 2. bájtját, és az MBR alsó 8 bitjére teszi. 2-t téve a MAR-ba, és elindítva egy olvasást, kiolvassa a memória 8–11. bájtjait (vagyis a 2. szót), és az MDR-be teszi.

Ez a működésbeli különbség szükséges, mert a MAR-t és a PC-t a memória két különböző részére való hivatkozásra fogjuk használni. A megkülönböztetés szükségessége később világosabbá válik. Pillanatnyilag elég annyi, hogy a MAR/MDR párt az ISA-szintű adatszavak olvasására és írására használjuk, a PC/MBR párt pedig a bájtfolyamból álló végrehajtható ISA-szintű program olvasására. Minden más regiszter, amely címet tartalmaz, szócímeket használ, mint a MAR.

A valódi fizikai megvalósításban csak egy tényleges memória van, az pedig bájtszervezésű. A MAR szavakban számlálhat (a JVM definíciója miatt erre van szükség), miközben a fizikai memória bájtokra van osztva. Ez egy egyszerű kis trükkel kezelhető. Amikor a MAR a címsínre kerül, a 32 bitje nem képződik le a 32 címvonalra közvetlenül 0–31-ig, hanem a MAR 0. bitje a címsín 2. vonalához kötődik, a MAR 1. bitje a címsín 3. vonalához kötődik és így tovább. A MAR legfelső 2 bitjét



4.4. ábra. *A MAR bitjeinek leképezése a címsínre*

eldobjuk, mivel ezek csak a 2^{32} feletti címekhez szükségesek, amelyek nem megengedettek 4 GB-os gépünkön. Ha ezt a leképezést használjuk, amikor a MAR 1, akkor a sínre kerülő cím 4; amikor a MAR 2, akkor a sínre kerülő cím 8 és így tovább. Ezt a trükköt a 4.4. ábra mutatja.

Mint fentebb említettük, a memóriából a 8 bites memóriaporton keresztüli olvasás az MBR-be történik, egy 8 bites regiszterbe. Az MBR két mód közül az egyikkel kapuzható (vagyis másolható) a B sínre: előjel nélkül vagy előjelesen. Amikor előjel nélküli értékre van szükségünk, akkor a B sínre kerülő 32 bites szó tartalmazza az MBR értékét az alsó 8 biten és nullákat a magasabb 24 biten. Az előjel nélküli értékek a táblázatok indexelésénél használatosak, vagy amikor egy 16 bites egészet kell összerakni az utasításfolyamban lévő 2 egymást követő (előjel nélküli) bájtból.

A másik lehetőség a 8 bites MBR 32 bites szóvá konvertálására, hogy az MBR-t egy –128 és +127 közötti előjeles értéknek tekintjük, és egy olyan 32 bites szót hozunk létre, amelynek ugyanez az értéke. Ez a konverzió az MBR előjelbitjének (bal szélső bit) a B sín felső 24 bitjére történő másolásával valósítható meg: a folyamat **előjel-kiterjesztés**ként ismert. Amikor ezt a lehetőséget választjuk, akkor a felső 24 bit vagy mind 0, vagy mind 1, attól függően, hogy a 8 bites MBR bal szélső bitje 0 vagy 1.

Azt, hogy vajon a 8 bites MBR előjel nélküli vagy előjeles számként kerüljön a B sínre, az határozza meg, hogy a két vezérlőjel (a 4.1. ábrán az MBR alatti nyitott nyilak) közül melyik van beállítva. Mindkét lehetőségre szükség van, ezért van két nyíl. Azt a lehetőséget, hogy a 8 bites MBR 32 bites forrásként viselkedhet a B sín számára, a 4.1. ábrán a szaggatott vonalú doboz jelzi az MBR bal oldalán.

4.1.2. Mikroutasítások

A 4.1. ábra adatútjának vezérléséhez 29 jelre van szükség. Ezek öt működési csoportba oszthatók, ahogy alább leírjuk.

9 jel vezérli az adatírást a C sínről a regiszterekbe.
9 jel vezérli a regiszterek engedélyezését a B sínre az ALU bemenete számára.
8 jel vezérli az ALU és a léptető tevékenységeit.

2 jel (nem látható) mutatja a memóriaolvasást/írást MAR/MDR-n keresztül.
1 jel (nem látható) mutatja a memóriabetöltést PC/MBR-n keresztül.

Ennek a 29 vezérlőjelnek az értéke határozza meg az adatút egy ciklusának műveleteit. A ciklus a regiszterek értékének B sínre másolásából, a jeleknek az ALU-n és a léptetőn keresztüli továbbításából, az eredmény C sínre viteléből, és végül az eredménynek a megfelelő regiszterbe vagy regiszterekbe írásából áll. Ezek után, ha a memóriaolvasás jel be van állítva, a memóriaművelet az adatútciklus végén kezdődik, miután a regiszterek (esetleg a MAR vagy PC) feltöltődtek. A memóriaadat a *következő* ciklus legvégén áll rendelkezésre az MBR vagy MDR-ben, és csak *az ezt követő* ciklusban használható fel.* Más szavakkal, a k. ciklus végén kezdeményezett bármelyik portra vonatkozó memóriaolvasás úgy szolgáltatja az adatot, hogy az nem használható a $k + 1$. ciklusban, csak a $k + 2$. ciklusban, vagy később.

Ezt a látszólag természetellenes viselkedést a 4.3. ábra magyarázza. A memóriát vezérlő jelek nem generálódnak az 1. óraciklusban addig, csak közvetlenül azután, hogy a MAR és PC feltöltődött az óra felfutó élére az 1. óraciklus vége felé. Feltételezzük, hogy a memória egy cikluson belül kiteszi a memóriasínre az eredményt, vagyis az MBR és/vagy MDR betölthető az óra következő felfutó élére, hasonlóan más regiszterekhez.

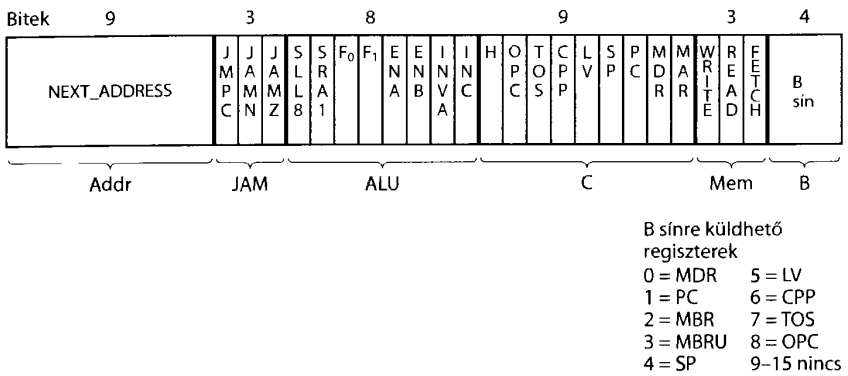
Más szóval, betöltjük a MAR-t az adatútciklus végén, és röviddel ezután indítjuk a memóriát. Következésképpen, valóban nem várhatjuk, hogy az olvasási művelet eredménye a következő ciklus elején az MDR-ben legyen, különösen, ha az óraimpulzus szélessége rövid. Nincs elegendő idő, ha a memória egy óraciklust igényel. Lennie kell egy adatútciklusnak a memóriaolvasás kezdete és az eredmény felhasználása között. Természetesen más művelet is végrehajtható ez alatt a ciklus alatt, de olyan nem, amelyik ezt a memóriaszót igényli.

Az a feltevés, hogy a memóriaművelet egy ciklust igényel, ekvivalens azzal, hogy feltesszük, hogy az 1. szintű gyorsítótár találati aránya 100%. Ez a feltevés sose igaz, de a változó hosszúságú memóriaciklus-idők bevezetésével kialakuló komplexitás több, mint amivel itt foglalkozni szeretnénk.

Az MBR és MDR más regiszterekhez hasonlóan az óra emelkedő élére töltődik fel, ezért olvashatók abban a ciklusban, amelyben az új memóriaolvasás megtörténik. Mindaddig a régi értékekkel térnek vissza, amíg az olvasás nem írja felül. Nincs itt ellentmondás; amíg az új értékek be nem töltődnek az MBR-be és MDR-be az óra emelkedő élénél, addig az előző érték ott található és használható. Megjegyezzük, hogy lehetséges szorosan egymás után olvasásokat végrehajtani két egymást követő ciklusban, mivel egy olvasás csak egy ciklusig tart. A kétfajta memóriaolvasás azonos időben is működhet. De ha megpróbáljuk párhuzamosan olvasni és írni ugyanazt a bájtot, definiálatlan eredményt kapunk.

Míg kívánatos lehet, hogy a C sínen lévő eredményt egynél több regiszterbe írjuk, addig sose kívánunk egyszerre egynél több regisztert a B sínre engedni. (Néhány igazi megvalósítás valóban fizikai kárt szenved, ha ez bekövetkezik.) Az áramkörök egy kis fejlesztésével redukálhatjuk azon bitek számát, melyek kivá-

* Ez csak az adatúton történő felhasználásra vonatkozik. *(A lektor)*



4.5. ábra. *A Mic-1 mikroutasítás formátuma*

lasztják a lehetséges források közül, hogy melyiket vezessük a B sínre. Csak kilenc lehetséges regiszter van, amit a B sínre vezethetünk (az MBR előjeles és előjel nélküli változatát külön számoljuk). Ennek következtében a B sín vezérlését 4 biten kódolhatjuk, és egy dekódert használhatunk a 16 vezérlőjel előállítására, melyből 7-re nincs szükségünk. A kereskedelmi tervezésben a tervezőmérnökök ellenállhatatlan kényszert éreznének, hogy valamelyik regisztert felszabadítsák, így 3 bit elég lenne a feladat végrehajtásához. Egyetemi oktatóként vállalható az egy bit elpazarlásának óriási luxusa a világosabb és egyszerűbb tervezés érdekében.

Ott tartunk, hogy az adatutat 9 + 4 + 8 + 2 + 1 = 24 jellel, azaz 24 bittel vezérelhetjük. Azonban ez a 24 bit csak egy ciklusra vezérli az adatutat. A vezérlés második része arra való, hogy meghatározza, mit szándékozunk tenni a következő ciklusban. Hogy ezt is bevonjuk a vezérlő tervezésébe, megalkotunk egy formátumot a végrehajtandó műveletek leírására, használni fogjuk a 24 vezérlőbitet, valamint további két mezőt: a NEXT_ADDRESS és a JAM mezőt. Mindkét mező tartalmát röviden vázoljuk. A 4.5. ábra egy lehetséges formátumot mutat, melyet hat csoportra osztottunk, és a következő 36 jelet tartalmazza:

Addr – A lehetséges következő mikroutasítás címét tartalmazza.
JAM – Meghatározza, hogy a következő mikroutasítás hogyan választódik ki.
ALU – Az ALU és a léptető tevékenységei.
C – Kiválasztja, hogy melyik regiszterbe írjunk a C sínről.
Mem – Memóriatevékenységek.
B – Kiválasztja a B sín forrását: kódolva van, ahogy bemutattuk.

A csoportok sorrendje elvben tetszőleges, bár valójában nagyon gondosan választottuk meg, hogy minimalizáljuk a vonalak keresztezéseit a 4.6. ábrán. A vonalkereszteződések a sematikus ábrán, amilyen a 4.6. ábra is, gyakran megfelelnek a huzalkereszteződéseknek az áramköri lapkán, amelyek hibát okoznak a kétdimenziós tervezésekben, és amelyeket a lehető legjobban minimalizálnak.

4.1.3. Mikroutasítás-vezérlés: Mic-1

Eddig leírtuk, hogy hogyan történik az adatút vezérlése, de még nem foglalkoztunk azzal, hogyan dől el, hogy a vezérlőjelek közül melyeket kell engedélyezni egy-egy cikluson belül. Ezt a **sorba állító** határozza meg, amely egy egyszerű ISA-utasítás végrehajtásához szükséges műveletek sorozatán való végiglépegetésért felelős.

A sorba állítónak minden ciklusban kétféle információt kell előállítania:

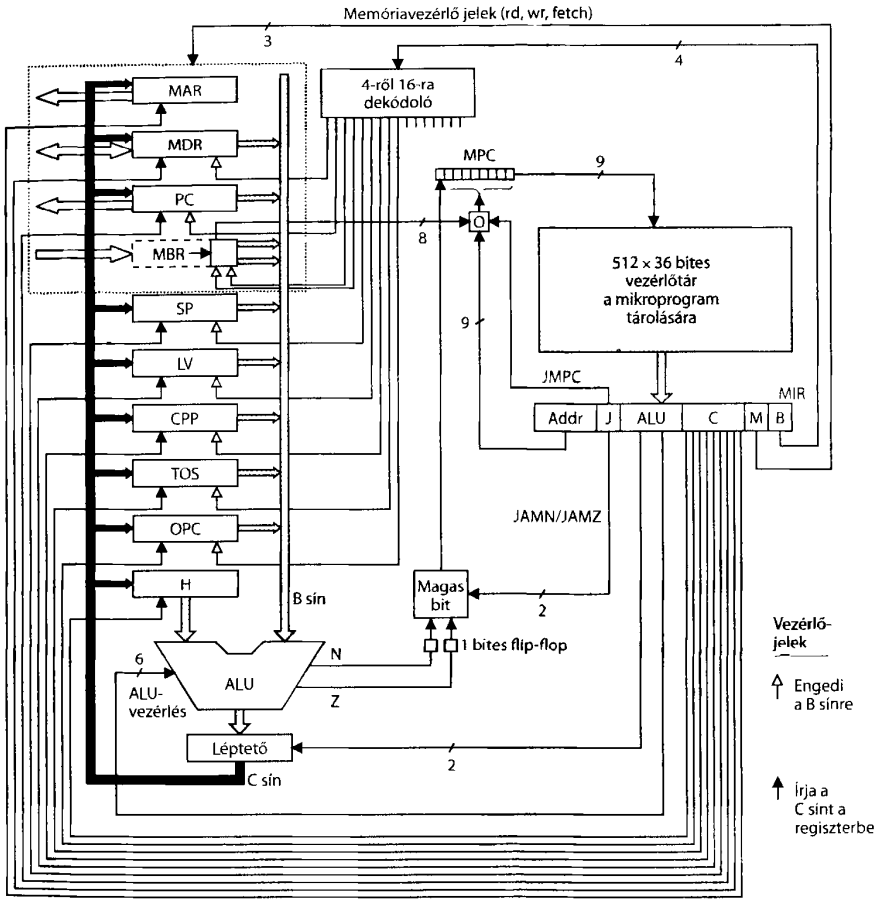
1. A rendszerben lévő összes vezérlőjel értékét.
2. A következő végrehajtandó mikroutasítás címét.

A 4.6. ábra a példagépünk, melyet **Mic–1**-nek nevezünk el, komplett mikroarchitektúrájának részletes blokkdiagramja. Kezdetben tiszteletet parancsolónak tűnhet, de megéri gondosan tanulmányozni. Ha valaki megérti az összes dobozt és vonalat ezen az ábrán, jó úton halad, hogy megértse a mikroarchitektúra-szintet. A blokkdiagramnak két része van: bal oldalon az adatút, amelyet már részletesen tárgyaltunk, a jobb oldalon pedig a vezérlőrész, amelyet a következőkben fogunk megvizsgálni.

A legnagyobb és a legfontosabb elem a gép vezérlőrészében egy memória, melyet **vezérlőtár**nak nevezünk. Kényelmes úgy tekinteni rá, mint egy komplett mikroprogramot tartalmazó memóriára, bár néha logikai kapuk halmazaként valósítják meg. Általában vezérlőtárként hivatkozunk rá, hogy össze ne tévesszük a főmemóriával, mely utóbbit az MBR-en és MDR-en keresztül érünk el. Működését tekintve azonban a vezérlőtár egy memória, amely ISA-utasítások helyett egyszerűen mikroutasításokat foglal magában. Példagépünknél 512 szót tartalmaz, minden szó egy 36 bites mikroutasításból áll, olyanból, melyet a 4.5. ábrán mutattunk be. Valójában nem minden szó szükséges, de – hogy röviden magyarázhassunk – 512 különböző szó számára van szükségünk címekre.

Egy fontos vonatkozásban a vezérlőtár eléggé különbözik a főmemóriától: a főmemóriában lévő utasítások végrehajtása mindig címsorrendben történik (kivéve az elágazásokat), a mikroutasításoké pedig nem. Az utasításszámláló növelésének megvalósítása azt a tényt fejezi ki, hogy az aktuális utasítás után végrehajtott utasítás alapértelmezésben az lesz, amelyik az aktuális utasítást a memóriában követi. A mikroprogramoknak rugalmasabbaknak kell lenniük, vagyis általában nem rendelkeznek ezzel a tulajdonsággal, mert a mikroutasítások sorozatánál rövidségre törekszünk. Ellenkezőleg, minden mikroutasítás világosan előírja, melyik fogja követni.

Mivel a vezérlőtár működését tekintve (csak olvasható) memória, szüksége van saját memóriacím-regiszterre és saját memóriaadat-regiszterre. Nincs szükség olvasó- és írójelekre, mert folyamatosan olvassuk. A vezérlőtár memóriacím-regiszterét **MPC**-nek **(MicroProgram Counter, mikroutasítás-számláló)** fogjuk nevezni. Ez a név ironikus, mivel a benne lévő helyek határozottan nem rendezettek, így a számlálás elve nem használható (de kik vagyunk mi, hogy vitatkozzunk a hagyománnyal?). A memóriaadat-regisztert **MIR**-nek **(MicroInstruction Register, mikroutasítás-regiszter)** nevezzük. Szerepe az, hogy tartalmazza az érvényes mikroutasítást, melynek bitjei meghajtják az adatutat működtető vezérlőjeleket.



4.6. ábra. *A mikroarchitektúra-példánk, a Mic–1 teljes blokkdiagramja*

A 4.6. ábrán a MIR regiszter ugyanazt a hat csoportot tartalmazza, mint a 4.5. ábrán. Az Addr és J (JAM) csoportok vezérlik a következő mikroutasítás kiválasztását; ezt röviden tárgyalni fogjuk. Az ALU csoport azt a 8 bitet tartalmazza, amelyek kiválasztják az ALU tevékenységét, és irányítják a léptetőt. A C bitek betöltetik az egyes regiszterekbe az ALU kimenetét a C sínről. Az M bitek vezérlik a memóriaműveleteket.

Végül az utolsó 4 bit irányítja a dekódert, amelyik meghatározza, mi kerüljön a B sínre. Ebben az esetben a standard 4-ről 16-ra dekódolót választjuk, annak ellenére, hogy csak kilenc lehetőségre van szükségünk. Egy finomabban hangolt tervezésnél 4-ről 9-re dekódolót használnánk. A kompromisszum az, hogy standard áramkört használunk az áramkörök könyvtárából ahelyett, hogy terveznénk egy egyedit. A standard áramkör használata egyszerűbb, és nem valószínű hibák telepítése. Saját tervezésnél kisebb lapkaterületet használunk, de a tervezés tovább tart, és el is ronthatjuk.

A 4.6. ábra működése a következő. Minden óraciklus kezdetén (az óra lefutó éle a 4.3. ábrán) a MIR feltöltődik az MPC által mutatott vezérlőtárbeli szóval. A MIR feltöltési idejét az ábrán Δw jelöli. Ha valaki részciklusokban gondolkodik, a MIR az első részciklus alatt feltöltődik.

Mihelyt a mikroutasítás bekerül a MIR-be, különböző jelek futnak az adatútra. Egy regiszter kikerül a B sínre, az ALU megtudja, melyik a végrehajtandó művelet, és sok tevékenység zajlik arrafelé. Ez a második részciklus. Az óra kezdetétől $\Delta w + \Delta x$ idő után az ALU bemenetei stabilak lesznek.

Egy másik Δy-nal később minden megállapodik, és az ALU, N, Z és a léptető kimenete stabil lesz. Az N és Z értékek ekkor eltárolódnak két 1 bites flip-flopban. Ezek a bitek az óra emelkedő élére tárolódnak, közel az adatútciklus végéhez, hasonlóan az összes regiszterhez, melyek a C sínről és a memóriából töltődnek fel. Az ALU kimenete nem tárolódik, hanem azonnal a léptetőbe töltődik. Az ALU és a léptető tevékenysége a 3. részciklusra esik.

Egy további Δz idő után, a léptető kimenete a C sínen keresztül eléri a regisztereket. Ekkor, a ciklus vége közelében (a 4.3. ábrán az órajel felfutó élére) a regiszterek feltölthetők. A 4. részciklus a regiszterek és az N és Z flip-flopok betöltéséből áll. Ez egy kicsivel az óra emelkedő éle után befejeződik, amikor minden eredmény el van már mentve, és az előző memóriaművelet eredményei rendelkezésre állnak, és az MPC is fel van töltve. Ez a folyamat folytatódik tovább, amíg valaki meg nem unja, és ki nem kapcsolja a gépet.

Párhuzamosan az adatút meghajtásával a mikroprogramnak meg kell határoznia, hogy melyik lesz a következő végrehajtandó mikroutasítás, hiszen nem abban a sorrendben futnak, ahogy a vezérlőtárban előfordulnak. A következő mikroutasítás címének kiszámítása azután kezdődik, hogy a MIR feltöltődik és stabilizálódik. Először a 9 bites NEXT_ADDRESS mező az MPC-be másolódik. Mialatt a másolás lezajlik, a JAM mező vizsgálat alá kerül. Ha az értéke 000, nincs semmi teendő; a NEXT_ADDRESS másolása után az MPC a következő mikroutasításra mutat.

Ha JAM egy vagy több bitje 1, több tevékenység szükséges. Ha a JAMN be van állítva, az 1 bites N flip-flop OR kapcsolattal kerül az MPC legmagasabb bitjére. Hasonlóan, ha a JAMZ be van állítva, az 1 bites Z flip-flop kerül OR kapcsolattal oda. Ha mindkettő be van állítva, mindkettő OR kapcsolattal kerül ugyanoda. Az N és Z flip-flop azért szükséges, mert az órajel emelkedő éle után (mialatt az óra magas) a B sín nincs már tovább vezérelve, így az ALU kimeneteiről a továbbiakban már nem tételezhetjük fel, hogy helyesek. Elmentve az ALU állapotjelzőit N-be és Z-be, a helyes értékeket elérhetővé és tartóssá tesszük az MPC kiszámításához, nem érdekes, hogy mi történik az ALU környékén.

A 4.6. ábrán ezt a számítást végző logikát „magas bit"-nek jelöljük. Ez a következő logikai függvényt valósítja meg:

F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDRESS[8]

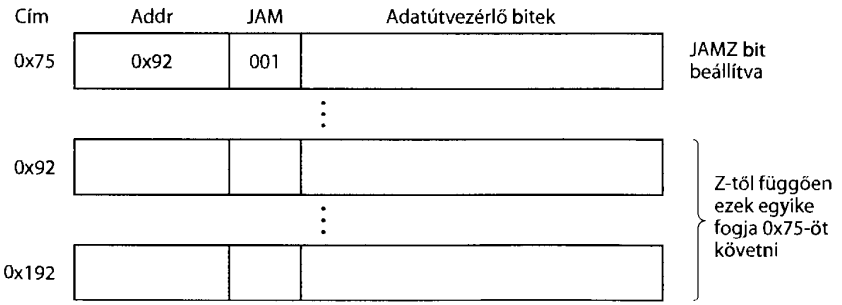
Megjegyezzük, hogy minden esetben az MPC csak a következő két lehetséges érték egyikét veheti fel:

1. A NEXT_ADDRESS értéke;
2. A NEXT_ADDRESS értéke, és a legmagasabb helyértékű biten OR 1 művelet.

Nincs más lehetőség. Ha a NEXT_ADDRESS legmagasabb bitje már 1, a JAMN vagy JAMZ használata értelmetlen.

Megjegyezzük, hogy amikor a JAM bitek mindegyike nulla, a következő végrehajtandó mikroutasítás címe egyszerűen a NEXT_ADDRESS mezőben lévő 9 bites szám. Amikor JAMN vagy JAMZ valamelyike 1, két lehetséges következő cím van: NEXT_ADDRESS vagy NEXT_ADDRESS OR 0x100 (feltéve, hogy NEXT_ADDRESS ≤ 0xFF, mert különben a két lehetőség egybeesik). (Megjegyezzük, hogy 0x jelöli, hogy a mögötte következő szám tizenhatos számrendszerben értendő.) A 4.7. ábrán ezt mutatjuk be. A 0x75 helyen lévő mikroutasításban NEXT_ADDRESS = 0x92, és a JAMZ értéke 1. Tehát a következő mikroutasítás címe az előző ALU-műveletnél tárolt Z bittől függ. Ha a Z bit 0, a következő mikroutasítás a 0x92-ről jön. Ha a Z bit 1, a következő mikroutasítás a 0x192-ről érkezik.

A harmadik bit a JAM mezőben a JMPC. Ha be van állítva, az MBR 8 bitje bitenkénti OR kapcsolatba kerül az aktuális mikroutasítás NEXT_ADDRESS mezőjének legalsó 8 bitjével. Az eredmény az MPC-be kerül. A 4.6. ábrán a O-val jelölt doboz végez OR műveletet az MBR és NEXT_ADDRESS között, ha a JMPC 1, de csupán átereszti NEXT_ADDRESS-t az MPC-hez, ha a JMPC 0. Amikor a JMPC 1, a NEXT_ADDRESS alsó 8 bitje általában nulla. A magas helyértékű bit lehet 0 vagy 1, vagyis a JMPC által használt NEXT_ADDRESS értéke általában 0x000 vagy 0x100. Azt, hogy miért használunk néha 0x000-t, máskor pedig 0x100-t, később fogjuk tárgyalni.



4.7. ábra. *Egy mikroutasításnak két lehetséges követője van, ha JAMZ értéke 1*

Azáltal, hogy OR műveletet végzünk az MBR és NEXT_ADDRESS között, és az eredményt az MPC-be tároljuk, lehetővé válik a többutas elágazás (ugrás) hatékony megvalósítása. Vegyük észre, hogy a 256 cím bármelyike előírható, és kizárólag az MBR-ben lévő bitek határozzák meg. Egy jellemző használatban az MBR egy műveleti kódot tartalmaz, így a JMPC használata egyértelmű választást eredményez

a következő mikroutasítás számára minden lehetséges műveleti kód esetén. Ez a módszer jól használható az éppen betöltött műveleti kódtól függő tevékenységre való gyors és közvetlen ugrásra.

A gép időzítésének megértése kritikus a következőkben, ezért talán nem haszontalan újra elismételni, és ezt a részciklusok terminológiájában fogjuk megtenni. Hiszen azt könnyű elképzelni, bár a valódi óraesemények csak a lefutó él, ami a ciklust indítja, és a felfutó él, amelyik betölti a regisztereket és az N és Z flip-flopot. Tekintsük még egyszer a 4.3. ábrát.

Az 1. részciklus alatt, melyet az óra lefutó éle indított, a MIR feltöltődik az MPC-ben jelenleg lévő címről. A 2. részciklus alatt a jelek szétterjednek MIR-ből, és a B sín feltöltődik a kiválasztott regiszterből. A 3. részciklus alatt az ALU és a léptető működik, és stabil eredményt állít elő. A 4. részciklus alatt a C sín, a memóriasínek és az ALU értékei stabillá válnak. Az óra emelkedő élére a regiszterek a C sínről feltöltődnek, az N és Z flip-flopok betöltődnek, és az MBR és MDR megkapják az előző adatútciklus végén indított memóriaműveletből (ha volt ilyen) származó eredményüket. Amint az MBR elérhető, az MPC betöltődik a következő mikroutasítás előkészítéséhez. Így az MPC megkapja értékét valamikor annak az intervallumnak a közepén, amikor az óra magas, de már az után, hogy MBR/MDR készen van.* Ez vagy szintvezérelt lehet (nem élvezérelt), vagy az élvezérelt egy meghatározott késleltetéssel az óra felfutó éle után. Egy a lényeg, az MPC addig nem töltődik be, amíg azok a regiszterek készen nincsenek, amelyektől függ (MBR, N és Z). Ahogy az órajel lefut, az MPC megcímezheti a vezérlőtárat, és egy új ciklus kezdődhet.

Megjegyezzük, hogy mindegyik ciklus önmagában zárt. Ez meghatározza, hogy mi menjen a B sínre, mit csináljon az ALU és a léptető, hova tároljon ki a C sínről, és végül, mi legyen a következő MPC érték.

Érdemes egy utolsó megjegyzést tenni a 4.6. ábrával kapcsolatban. Úgy tekintettük az MPC-t, mint egy valódi regisztert, amelynek 9 bit a tárolókapacitása, és akkor töltődik be, amikor az órajel magas. A valóságban nincs szükség arra, hogy itt legyen regiszter. Minden bemenete közvetlenül ráköthető a vezérlőtárra. Elegendő, ha az óra lefutó élénél megjelennek a jelek a vezérlőtárban, amikor a MIR értéket kap. Nem szükséges, hogy azokat ténylegesen kitároljuk az MPC-be. Ezért az MPC **virtuális regiszter**ként jól megvalósítható, ami csak gyűjtőhelye a jeleknek; inkább hasonlít egy elektonikus paneldarabra, mint egy igazi regiszterre. Egyszerűsödik az időzítés, ha az MPC-t virtuális regiszterként készítjük el; most az események csak az óra lefutó vagy felfutó élénél történnek, és sehol máshol. De ha valakinek könnyebb úgy gondolni az MPC-re, mint igazi regiszterre, az is elfogadható nézőpont.

* Tehát a következőként végrehajtandó mikroutasítás címének kiszámításához felhasználható az előző mikroutasításban kiadott fetch eredménye, de az adatúton még az MBR régi értéke érhető el. *(A lektor)*

4.2. ISA-példa: az IJVM

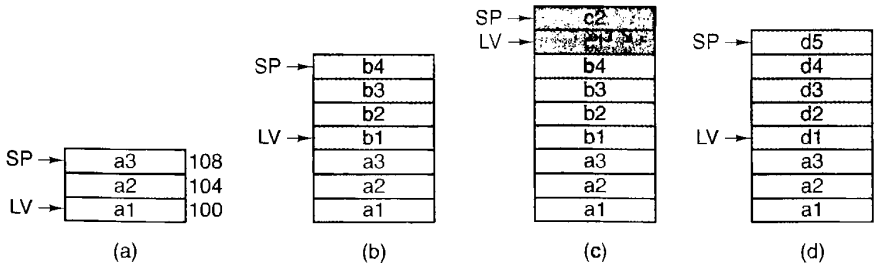
Folytassuk a példánkat a gép ISA-szintjének bevezetésével, amelyet a 4.6. ábra mikroarchitektúráján futó mikroprogrammal fogunk értelmezni (IJVM). Kényelemből néha az utasításrendszer-architektúra szintjére úgy hivatkozunk, mint **makroarchitektúrá**ra, hogy megkülönböztessük a mikroarchitektúrától. Mielőtt leírjuk az IJVM-et, teszünk mégis egy kis kitérőt, hogy megindokoljuk.

4.2.1. Vermek

Úgy tűnik, minden programozási nyelv támogatja az eljárásokat (metódusokat), amelyeknek lokális változóik vannak. Ezek a változók elérhetők az eljárás belsejéből, de megszűnik az elérhetőségük, amikor az eljárás visszatért. Így felmerül a kérdés: „Hol kell tartani ezeket a változókat a memóriában?"

Az a legegyszerűbb megoldás nem működik, hogy minden változónak adunk egy abszolút memóriacímet. A probléma az, hogy az eljárás saját magát is hívhatja. Ezeket a rekurzív eljárásokat az 5. fejezetben fogjuk tanulmányozni. Pillanatnyilag elég annyit mondanunk, hogy ha az eljárás kétszer aktív (meghívott), lehetetlen a változóit abszolút memóriahelyeken tárolni, mert a második hívás összeütközésbe kerül az elsővel.

Ehelyett egy másik stratégiát alkalmazunk. Egy **verem**nek hívott memóriaterületet tartunk fenn a változóknak, de ezen belül az egyedi változók nem kapnak abszolút címet. Ehelyett egy regisztert állítunk be, mondjuk az LV-t, hogy mutasson a lokális változók kiindulópontjára az aktuális eljárás számára. A 4.8. (a) ábrán egy *A* eljárást hívtunk meg, amelynek lokális változói *a1*, *a2* és *a3*, így memóriát foglalunk a lokális változói számára az LV által mutatott memóriahelytől kezdve. Egy másik regiszter, SP, az *A* eljárás lokális változói közül a legmagasabb szóra mutat. Ha LV 100, és a szavak 4 bájtosak, SP 108 lesz. Változókra úgy hivatkozunk, hogy megadjuk az eltolását (távolságát) LV-től. Az LV és SP közötti adatstruktúrát (beleértve mindkét mutatott szót), az *A* **lokális változó mezőjé**nek nevezzük.



4.8. ábra. *A verem használata lokális változók tárolására. (a) Miközben A aktív. (b) Miután A meghívta B-t. (c) Miután B meghívta C-t. (d) Miután C és B visszatért, és A meghívta D t*

Most vegyük fontolóra, mi történik, ha *A* meghív egy másik *B* eljárást. Hol lesz tárolva *B* négy lokális változója (*b1*, *b2*, *b3*, *b4*)? Válasz: a veremben, az *A* változói fölött, ahogy az a 4.8. (b) ábrán látható. Vegyük észre, hogy az eljáráshívás LV-t *B* lokális változóira állította be, *A* változói helyett. *B* lokális változóira úgy hivatkozhatunk, hogy megadjuk LV-től való távolságukat. Hasonlóan, ha *B* meghívja *C*-t, LV-t és SP-t ismét beállítjuk úgy, hogy *C* két lokális változójának helyet foglaljon, mint az a 4.8. (c) ábrán látható.

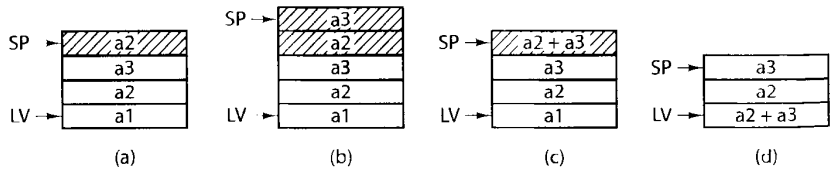
Amikor *C* visszatér, *B* válik ismét aktívvá, és a verem visszaáll a 4.8. (b) ábra szerint úgy, hogy LV most ismét *B* lokális változóira mutat. Hasonlóképpen, ha *B* visszatér, visszatérünk a 4.8. (a) ábrán látható helyzethez. Minden körülmények között LV az éppen aktív eljárás veremszerkezetének alapjára mutat, SP pedig a veremszerkezet tetejére.

Most tegyük fel, hogy *A* meghívja *D*-t, amelynek öt lokális változója van. A 4.8. (d) ábrán lévő helyzet áll elő, amelyben *D* lokális változói azt a memóriaterületet használják, amelyet előzőleg *B* változói, valamint *C* változóinak egy része használt. Ezzel a memóriaszervezéssel csak az éppen aktív eljárásoknak van foglalva memória. Ha egy eljárás visszatér, a lokális változói által használt memória felszabadul.

A vermeknek van más felhasználása is, azon kívül, hogy a lokális változókat tárolják. Használhatók arra is, hogy tárolják az operandusokat egy aritmetikai kifejezés kiszámolása alatt. Ha erre használjuk, a vermet **operandusverem**nek nevezzük. Tegyük fel például, hogy *B* meghívása előtt *A*-nak el kell végeznie a következő számolást:

```
a1 = a2 + a3
```

Az összeg elkészítésének egyik módja, hogy az *a2*-t betesszük a verembe, ahogy az a 4.9. (a) ábrán látható. Itt az SP megnőtt az egy szóban lévő bájtok számával, azaz 4-gyel, és az első eltárolt operandus most az SP által mutatott címen van. Ezután *a3*-at tesszük a verembe, amint azt a 4.9. (b) ábra mutatja. Mellesleg megjegyezzük, hogy minden programrészt HELVETICA-val szedünk, mint fent. Ugyanezt a betűtípust fogjuk használni assembly nyelvű műveleti kódokra és gépi regiszterekre, azonban futószövegekben a programváltozókat és eljárásokat *dőlt betűvel* adjuk meg. A különbség az, hogy a változókat és eljárásneveket a felhasználó választja, a műveleti kódok és regiszternevek viszont beépítettek.



4.9. ábra. *Operandusverem használata aritmetikai számítás elvégzéséhez*

Most már elvégezhető a folyamatban levő számolás egy utasítás végrehajtásával, amely a veremből kivesz két szót, összeadja őket és az eredményt visszateszi a verembe, mint az a 4.9. (c) ábrán látható. Végül a legfelső szót kivehetjük a veremből, és visszatárolhatjuk az *u1* lokális változóba, ahogy azt a 4.9. (d) ábrán ábrázoljuk.

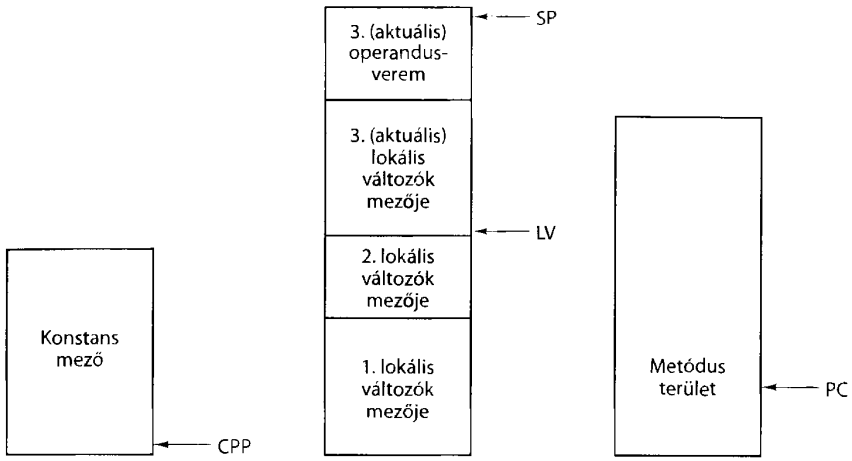
A lokális változó mezője és az operandusverem összevegyíthetők. Például, amikor egy olyan kifejezést számolunk, mint $x^2 + f(x)$, a kifejezés egy része (például x^2) bent lehet az operandusveremben, amikor az f függvényt meghívjuk. A függvény eredménye bent marad a veremben, x^2 felett, így a következő utasítás azokat összeadhatja.

Említésre érdemes, hogy míg minden gép használ vermet a lokális változók tárolására, nem mindegyik használ ehhez hasonló operandusvermet az aritmetika működéséhez. Tény, hogy a legtöbb nem használ, de a JVM és IJVM így dolgozik, ezért mutattuk be itt a veremműveleteket. Részletesebben majd az 5. fejezetben foglalkozunk velük.

4.2.2. Az IJVM memóriamodellje

Most már készek vagyunk az IJVM architektúrájának megismerésére. Alapvetően olyan memóriából áll, amely két módon látható: egy 4 294 967 296 bájtos (4 GB) vagy egy 1 073 741 824 szavas tömb, ahol minden szó 4 bájtból áll. Ellentétben a legtöbb ISA-val, a Java virtuális gép nem hoz létre az ISA-szinten közvetlenül látható abszolút memóriacímeket, de van számos implicit (közvetlenül nem látható) cím, amely egy mutató alapjául szolgál. Az IJVM-utasítások csak e mutatóktól való indexeléssel érhetik el a memóriát. Mindig a következő memóriaterületek vannak definiálva:

1. *Konstans mező* (konstansok tárterülete). Erre a területre IJVM-program nem írhat, konstansokból, stringekből és olyan mutatókból áll, amelyek a memória más, hivatkozható területeire mutatnak. Ez akkor töltődik be, amikor a program a memóriába kerül, és azután már nem változik. A CPP implicit regiszter a konstans mező első szavának címét tartalmazza.
2. *Lokális változók mezője.* Egy metódus minden hívásakor egy terület foglalódik le, hogy a hívás ideje alatt a változók itt tárolódjanak. Ezt a **lokális változók mezőjének** nevezzük. Ennek a területnek az elején vannak a paraméterek (argumentumoknak is hívjuk), amelyekkel a metódust meghívtuk. A lokális változók mezője nem foglalja magában az operandusvermet, amely teljesen különálló. Mindamellett, hatékonysági megfontolásból, megvalósításunk azt választja, hogy az operandusvermet közvetlenül a lokális változók mezője fölött valósítja meg. Van egy közvetlen regiszter, amely tartalmazza a lokális változók mezőjében az első hely címét. Ezt a regisztert LV-nek fogjuk nevezni. A metódus hívásakor átadott paraméterek a lokális változók mezőjének elején vannak tárolva.
3. *Operandusverem.* A verem mező garantáltan nem halad meg egy bizonyos méretet, amit a Java-fordítóprogram előre kiszámol. Az operandusverem helye közvetlenül a lokális változók mezője fölött van lefoglalva, ahogy azt a 4.10. ábra mutatja. Megvalósításunkban gondolhatunk úgy az operandusveremre, hogy az a lokális változók mezőjének a része. Mindenesetre van egy SP implicit regiszter, amely tartalmazza a verem legfelső szavának címét. Vegyük észre, hogy az CPP-vel és LV-vel ellentétben, ez a mutató változik a metódus végrehajtása során, amikor operandusokat teszünk a verembe, vagy veszünk ki onnan.
4. *Metódus mező.* Végül, a memóriának van egy olyan része, amely a programot tartalmazza, „szöveg" területként hivatkozunk rá UNIX processzusban. Van egy implicit regiszter, amely tartalmazza annak az utasításnak a címét, amelyet következőként be kell tölteni. Ez a mutató az utasításszámláló vagy PC. A memória más területeitől eltérően a metódus mezőt bájttömbként kezeljük.



4.10. ábra. *Az IJVM-memória különböző részei*

Egy megjegyzést kell tennünk a mutatókra vonatkozóan. A CPP, LV és SP regiszterek *szavakra* és nem *bájtokra* vonatkozó mutatók, szavakban kell megadni az eltolást. Az IJVM-ben a konstans mezőben, a lokális változók mezőjében és a veremben lévő elemre történő minden hivatkozás szóra vonatkozik, és minden eltolás, amelyet e mezők indexelésére használunk, szóeltolás. Például LV, LV + 1 és LV + 2 a lokális változók mezőjének első három szavára hivatkozik, ezzel ellentétben LV, LV + 4 és LV + 8 egymástól 4 szó (16 bájt) távolságra lévő szavakra.

Ezzel szemben a PC egy bájt címét tartalmazza, és a PC-hez való hozzáadás vagy kivonás a címet bájtok számával fogja megváltoztatni, és nem szavak számával. A PC címzése különbözik a többitől, és ez a tény szembetűnő a speciális memóriaporton, amely a PC-ről gondoskodik a Mic-1-nél. Emlékezzünk, hogy az csak 1 bájt széles. A PC növelése eggyel és egy olvasás kezdeményezése a következő *bájt* betöltését eredményezi. Az SP növelése eggyel és egy olvasás kezdeményezése a következő *szó* betöltését eredményezi.

4.2.3. Az IJVM utasításkészlete

Az IJVM utasításkészlete a 4.11. ábrán látható. Minden utasítás egy műveleti kódból és néha egy operandusból áll, amely például eltolás vagy konstans lehet. Az első oszlop az utasítás hexadecimális kódját, a második az assembly nyelvi alakját, a harmadik pedig a hatásának rövid leírását adja meg.

Hex	Mnemonic	Jelentés
0x10	BIPUSH *byte*	Beteszi a bájtot a verembe.
0x59	DUP	A verem legfelső szavát lemásolja, és beteszi a verembe.
0xA7	GOTO *offset*	Feltétel nélküli elágazás.
0x60	IADD	Kivesz a veremből két szót; az összegüket beteszi.
0x7E	IAND	Kivesz a veremből két szót; a logikai AND eredményét beteszi.
0x99	IFEQ *offset*	Kivesz a veremből egy szót; és elágazik, ha az nulla.
0x9B	IFLT *offset*	Kivesz a veremből egy szót; és elágazik, ha az kisebb, mint nulla.
0x9F	IF_ICMPEQ *offset*	Kivesz a veremből két szót; és elágazik, ha azok egyenlők.
0x84	IINC *varnum const*	Egy konstanst ad egy lokális változóhoz.
0x15	ILOAD *varnum*	Beteszi a lokális változót a verembe.
0xB6	INVOKEVIRTUAL *disp*	Meghív egy metódust.
0x80	IOR	Kivesz a veremből két szót; a logikai OR eredményét beteszi.
0xAC	IRETURN	Visszatér a metódusból egész értékkel.
0x36	ISTORE *varnum*	Kivesz egy szót a veremből és eltárolja a lokális változóba.
0x64	ISUB	Kivesz a veremből két szót; a különbségüket beteszi.
0x13	LDC_W *index*	A konstans mezőről egy konstanst betesz a verembe.
0x00	NOP	Nem csinál semmit.
0x57	POP	A verem legfelső szavát eldobja.
0x5F	SWAP	A verem két felső szavát megcseréli.
0xC4	WIDE	Prefix utasítás; a következő utasításnak 16 bites indexe van.

4.11. ábra. *Az IJVM utasításkészlete. A byte, const és varnum operandusok 1 bájtosak. A disp, index és offset operandusok 2 bájtosak*

Utasítások gondoskodnak arról, hogy különböző forrásokból egy szó bekerüljön a verembe. Ilyen forrás a konstans mező (LDC_W), a lokális változók mezője (ILOAD) és maga az utasítás (BIPUSH). Egy változó is kivehető a veremből, és tárolható a lokális változók mezőjébe (ISTORE). Két aritmetikai művelet (IADD és ISUB), valamint két logikai (Boolean) művelet (IAND és IOR) van, amelyek a verem két felső szavát használják operandusként. Mind az aritmetikai, mind a logikai műveletekben két szót veszünk ki a veremből, és az eredményt visszatesszük a verembe. Négy elágazási utasítás áll rendelkezésre, egy feltétel nélküli (GOTO) és három feltételes (IFEQ, IFLT és IF_ICMPEQ). Minden elágazási utasítás az utasításban a műveleti kódot követő (16 bites, előjeles) eltolás mértékével módosítja a PC értékét. Ez az eltolás hozzáadódik a műveleti kód címéhez. Vannak még IJVM-utasítások a verem két felső elemének megcserélésére (SWAP), a felső elem megkettőzésére (DUP) és eltávolítására (POP).

Néhány utasításnak több formája van, megengedve egy rövidebb formát is az általánosan használt változatra. Az IJVM-be két, JVM-ben használt mechanizmust is beépítettünk, hogy ezt teljesítsük. Egyik esetben átugorjuk a rövid formát az általánosabb kedvéért. A másik esetben megmutatjuk, hogyan használható a WIDE prefix utasítás a következő utasítás módosítására.

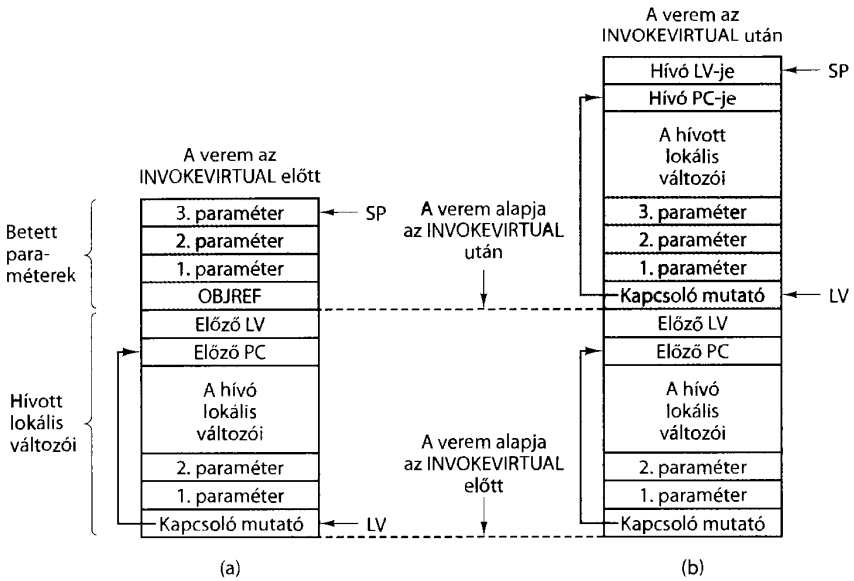
Végül, van egy utasítás (INVOKEVIRTUAL) egy másik metódus meghívására, és egy másik utasítás (IRETURN) arra, hogy kilépjünk a metódusból, és visszaadjuk a vezérlést annak a metódusnak, amelyik ezt hívta. A mechanizmus összetettsége miatt kicsit egyszerűsítjük a definíciót, lehetővé téve egy lényeggel törődő mechanizmus létrehozását hívás és visszatérés alkalmazására. A Javához képest az a megszorítás, hogy egy metódusnak csak saját objektumon belüli metódust engedünk meghívni. Ez a korlátozás egyszerűen tönkreteszi az objektumorientáltságot, de lehetővé teszi számunkra, hogy egy sokkal egyszerűbb mechanizmust mutassunk be, elkerülve azt a követelményt, hogy a metódust dinamikusan helyezzük el. (Ha nem jártas az objektumorientált programozásban, nyugodtan figyelmen kívül hagyhatja ezt a megjegyzést. Mindössze annyit tettünk, hogy a Javát visszabutítottuk nem objektumorientált nyelvvé, olyanná, mint a C vagy a Pascal.) Minden számítógépen, a JVM-et kivéve, a meghívott eljárás címét közvetlenül a CALL utasítás határozza meg, így megközelítésünk valójában a normális eset, nem kivétel.

Egy metódus meghívásának a mechanizmusa a következő. A hívó először betesz a verembe egy hivatkozást (mutatót), ami a meghívott objektumra utal. (Ez a hivatkozás az IJVM-nél nem szükséges, mivel másik objektum nem adható meg, de meghagyjuk, hogy konzisztensek maradjunk a JVM-mel.) A 4.12. (a) ábrán ezt a hivatkozást OBJREF mutatja. Ezután a hívó beteszi a metódus paramétereit a verembe, ebben a példában ez *1. paraméter*, *2. paraméter* és *3. paraméter*. Végül végrehajtódik az INVOKEVIRTUAL utasítás.

Az INVOKEVIRTUAL utasítás magában foglal egy eltolást, amely a konstans mezőn azt a helyet jelöli, amely tartalmazza a metódus mezőn belül a hívott metódus kezdőcímét. A metódus kódja az ezen mutatóval jelölt helyen van, de a metódus mező első 4 bájtja speciális adatokat tartalmaz. Az első két bájtot mint egy 16 bites egészet tekintjük, amely a metódus paramétereinek a számát jelöli (maguk a paraméterek már a veremben vannak). Ebből a szempontból OBJREF-et is paraméternek tekintjük: ez a 0. paraméter. Ez a 16 bites egész, az SP értékével együtt szolgáltatja az OBJREF helyét. Megjegyezzük, hogy LV az OBJREF-re mutat, nem az első valódi paraméterre. Az, hogy hova mutasson LV, némiképp önkényesen választható.

A metódus mező második 2 bájtját úgy értelmezzük, mint egy másik 16 bites egészet, amely a meghívott metódus számára a lokális változók mezőjének a méretét jelzi. Ez azért szükséges, mert új verem létesül a metódus számára, amely közvetlenül a lokális változók mezője fölött kezdődik. Végül a metódus mező ötödik bájtja tartalmazza az első végrehajtandó műveleti kódot.

Az INVOKEVIRTUAL-nál bekövetkező tevékenységsorozatot az alábbiakban ismertetjük, és a 4.12. ábrán ábrázoljuk. A két előjel nélküli bájt, amely a műveleti kódot követi, egy a konstans mezőn használandó index készítésére szolgál (az első a magasabb helyértékű bájt). Az utasítás kiszámítja az új lokális változók mezőjének

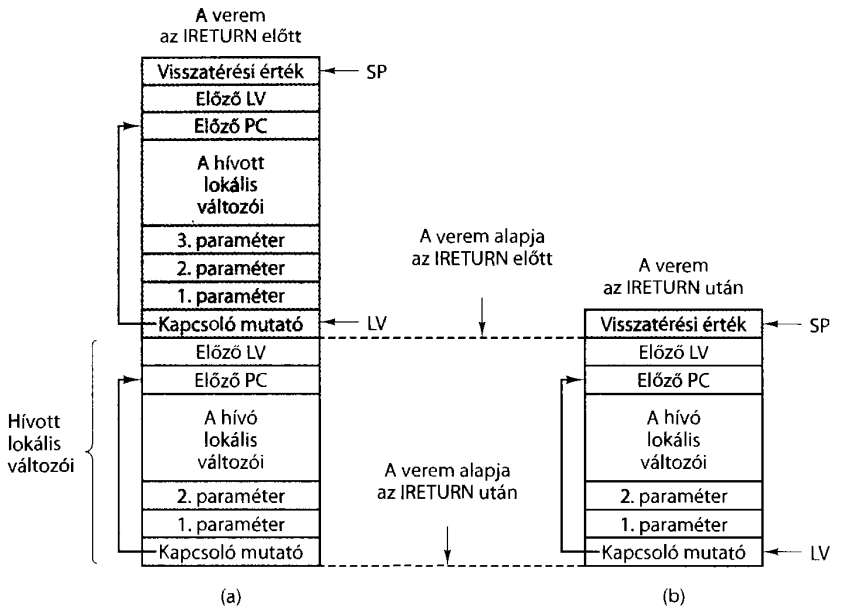


4.12. ábra. *(a) A verem az INVOKEVIRTUAL végrehajtása előtt. (b) A végrehajtása után*

kezdőcímét úgy, hogy a paraméterek számát kivonja a veremmutatóból, és LV-t úgy állítja be, hogy OBJREF-re mutasson. Ekkor OBJREF-et felülírva, eltárolja annak a helynek címét, ahová a régi PC-t tároljuk. Ezt a címet úgy számítjuk ki, hogy hozzáadjuk a lokális változók mezőjének méretét (paraméterek + lokális változók) az LV-ben tárolt címhez. Közvetlenül a régi PC-t tároló cím felett található az a cím, ahová a régi LV-t tároljuk. Közvetlenül e cím felett van az újonnan meghívott metódus számára a verem kezdete. SP-t úgy állítjuk be, hogy a régi LV-re mutasson, azaz arra címre, amely rögtön a verem első üres helye alatt van. Emlékezzünk arra, hogy SP mindig a verem legfelső szavára mutat. Ha a verem üres, a verem alja alatti legelső helyre mutat, mivel vermeink a magasabb címek felé emelkednek. Ábráinkon a vermek mindig a lap tetején lévő magasabb címek felé növekednek.

Az utolsó művelet, amely az INVOKEVIRTUAL végrehajtásához szükséges az, hogy a PC-t úgy állítsa be, hogy a metódus kódjában az ötödik bájtra mutasson.

Az IRETURN utasítás „megfordítja" az INVOKEVIRTUAL utasítás műveleteit, ahogy az a 4.13. ábrán látható. Felszabadítja a visszatérő metódus által használt területet. Visszaállítja a vermet is a korábbi állapotába, kivéve, hogy (1) a (most felülírt) OBJREF szót és minden paramétert eltávolít a veremből, és (2) a visszatérő értéket a verem tetejére teszi, ahol korábban OBJREF volt. A régi állapot visszaállításához az IRETURN-nek képesnek kell lenni a PC és LV mutatókat a régi értékükre visszaállítani. Ezt úgy hajtja végre, hogy hozzáfér a kapcsoló mutatóhoz (ez az a szó, ahova az aktuális LV mutat). Ne felejtsük el, hogy ide, ahol eredetileg OBJREF volt, az INVOKEVIRTUAL utasítás eltárolta a régi PC-t tartalmazó címet. Ezt a szót és a felet-



4.13. ábra. *(a) A verem az IRETURN végrehajtása előtt. (b) A végrehajtása után*

te lévő szót azért hozzuk vissza, hogy visszaállítsuk PC és LV régi értékét. A visszatérési érték, amely a befejeződő metódus vermének tetején helyezkedik el, bemásolódik oda, ahol eredetileg OBJREF volt tárolva, és az SP visszaállítódik úgy, hogy erre a helyre mutasson. A vezérlés így visszatér közvetlenül az INVOKEVIRTUAL utasítást követő utasításra.

Mindeddig még nem volt gépünknek bemenet/kimenet utasítása. Nem is áll szándékukban megadni ilyet. Egyáltalán nincs rá szüksége, akárcsak a Java virtuális gépnek, a hivatalos JVM-specifikációk még csak nem is említik a B/K-t. Az elv az, hogy egy olyan gép, amelynek nincs se bemenete, se kimenete, az „biztonságos." (Speciális B/K metódusok meghívásával tudunk olvasni és írni JVM-ben.)

4.2.4. Java fordítása IJVM-re

Most nézzük meg, hogy viszonyul egymáshoz a Java és az IJVM. A 4.14. (a) ábrán egy egyszerű Java-kódrészletet látunk. Ha betápláljuk egy Java-fordítóba, a fordító valószínűleg a 4.14. (b) ábrán látható IJVM assembly nyelvű programot készíti el. Az assembly nyelvű program bal oldalán lévő sorszámok 1-től 15-ig nem részei a fordító kimenetének. Ugyancsak nem részei a megjegyzések (melyek //-rel kezdődnek). Azért vannak ott, hogy segítsenek megmagyarázni az ábrát. A Java assembler ezután az assembly nyelvű programot lefordítja bináris programmá, ame-

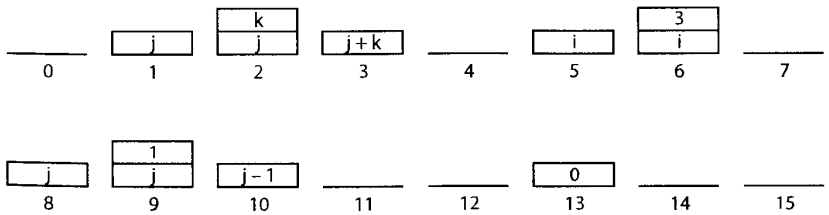
lyet a 4.14. (c) ábrán láthatunk. (Valójában a Java-fordító saját assemblyt csinál, és közvetlenül készíti el a bináris programot.) Ennél a példánál feltételezzük, hogy *i* az 1., *j* a 2. és *k* a 3. lokális változó.

I = j + k;	1		ILOAD j	// i = j + k	0x15 0x02
if (i == 3)	2		ILOAD k		0x15 0x03
k = 0;	3		IADD		0x60
else	4		ISTORE i		0x36 0x01
j = j - 1;	5		ILOAD i	// if (i == 3)	0x15 0x01
	6		BIPUSH 3		0x10 0x03
	7		IF_ICMPEQ L1		0x9f 0x00 0x0d
	8		ILOAD j	// j = j - 1	0x15 0x02
	9		BIPUSH 1		0x10 0x01
	10		ISUB		0x64
	11		ISTORE j		0x36 0x02
	12		GOTO L2		0xa7 0x00 0x07
	13	L1:	BIPUSH 0	// k = 0	0x10 0x00
	14		ISTORE k		0x36 0x03
	15	L2:			
(a)			(b)		(c)

4.14. ábra. *(a) Egy Java-programrészlet. (b) A megfelelő Java assembly nyelvű program. (c) Az IJVM-program hexadecimális formában*

A lefordított kód magáért beszél. Először *j*-t és *k*-t betesszük a verembe, összeadjuk azokat, és az eredményt *i*-be tároljuk. Ezután *i*-t és a 3 konstanst tesszük a verembe, és összehasonlítjuk. Ha egyenlők, elágazást hajtunk végre *L1*-hez, ahol *k*-t 0-ra állítjuk. Ha nem egyenlők, az összehasonlítás hamis eredményt ad, és az IF_ICMPEQ-t követő utasítást hajtjuk végre. Ha ez kész, elágazunk *L2*-höz, ahol a then és else rész egyesül.

A 4.14. (b) ábrán lévő IJVM-program operandusverme a 4.15. ábrán látható. Mielőtt a kód végrehajtását megkezdenénk, a verem üres, amit a 0 feletti vízszintes vonal jelöl. Az első ILOAD után *j* van a veremben, amit az 1 feletti bekeretezett *j* jelöl (ez azt jelenti, hogy az 1. utasítás végrehajtása megtörtént). A második ILOAD után két szó van a veremben, ahogy 2 felett látható. Az IADD után csak egy szó van



4.15. ábra. *A verem a 4.14. (b) ábra egyes utasításai után*

a veremben, és ez a $j + k$ összeget tartalmazza. Ha a felső szót kivesszük a veremből, és eltároljuk *i*-be, akkor a verem üres lesz, ahogy az 4 felett látszik.

Az 5. utasítás (ILOAD) indítja az if utasítást azzal, hogy beteszi *i*-t a verembe (5-ben). Ezután jön a 3 konstans (6-ban). Az összehasonlítás után a verem ismét üres (7). A 8-as utasítás indítja a Java-programrész else ágát. Az else rész a 12-es utasításig folytatódik, ott átugrik a then rész felett, és az *L2* címkénél folytatódik.

4.3. Példa a megvalósításra

Mind a mikroarchitektúrát, mind a makroarchitektúrát pontosan és részletesen meghatároztuk, a megmaradó téma a megvalósítás. Más szóval, milyen az a program, amely a mikroarchitektúrán fut, és értelmezi a makroarchitektúrát, és hogyan működik? Mielőtt válaszolhatnánk ezekre a kérdésekre, gondosan át kell tekintenünk azt a jelölésrendszert, amit a megvalósítás leírására használni fogunk.

4.3.1. Mikroutasítások és jelölésrendszer

Elvben leírhatnánk a vezérlőtárat binárisan, 36 bites szavakkal. De, mint a hagyományos programozási nyelvekben, itt is nagy haszna van egy jelölésrendszer bevezetésének, amelyik közvetíti a téma lényegét, amivel foglalkoznunk kell, és elfedjük a mellőzhető részleteket, vagy automatikusan jobban kezelhetjük. Fontos, hogy megértsük, a választott nyelvet arra terveztük, hogy leírja a fogalmakat, és nem arra, hogy megkönnyítse a hatékony tervezést. Ha az utóbbi lenne a célunk, másmilyen jelölést használnánk, hogy végsőkig fokozzuk a tervezők által igénybe vehető hajlékonyságot. Egy szempont, a címek kiválasztása. Mivel a memória nem logikailag rendezett, ezért nincs természetes módon megvalósítandó „következő utasítás", ahogy mi műveletek egy sorozatát meghatározzuk. Ennek a vezérlésszervezésnek a nagyobb ereje a tervezők (vagy az assembler) azon képességéből adódik, hogy hatékonyan választhatják ki a címeket. Ennek megfelelően egy egyszerű szimbolikus nyelv bevezetésével kezdünk, amely teljesen leír minden egyes műveletet anélkül, hogy teljesen elmagyaráznánk, hogy az összes címet hogyan lehet meghatározni.

A jelölésünk egyetlen sorban pontosan meghatároz minden olyan tevékenységet, ami egy egyszerű óraciklusban történik. Elvileg használhatnánk magas szintű nyelvet a műveletek leírására, de a ciklusról ciklusra történő vezérlés nagyon fontos, mert lehetőséget teremt arra, hogy végrehajtsunk összetett műveleteket egyidejűleg, és szükséges, hogy képesek legyünk elemezni minden egyes ciklust, hogy megértsük és ellenőrizzük a műveleteket. Ha egy gyors, hatékony megvalósítás a cél (amennyiben más dolgok egyenrangúak, a gyors és hatékony mindig jobb, mint a lassú és eredménytelen), minden ciklus számít. Egy valódi megvalósításban sok finom trükk rejtve marad a programban, homályos sorrendeket vagy műveleteket használ, hogy megtakarítson egyetlen ciklust. Nagy győzelem egy ciklus megta-

karítása: egy négyciklusos utasítás, amely 2 ciklussal csökkenthető, kétszer olyan gyorsan fut. És ez a gyorsítás mindig érvényesül, amikor az utasítást végrehajtjuk.

Egy lehetséges megközelítés, hogy minden egyes óraciklusban egyszerűen felsoroljuk azokat a jelzéket, amelyeket aktivizálni kell. Tegyük fel, hogy egy ciklus alatt szeretnénk növelni az SP értékét. Kezdeményezni akarunk egy olvasásműveletet is, és azt akarjuk, hogy a következő utasítás az legyen, amelyik a vezérlőtár 122-es helyén található. Írhatjuk, hogy:

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

ahol WSP azt jelenti, hogy „írjunk az SP regiszterbe." Ez a jelölés teljes, de nehéz megérteni. Helyette vegyíteni fogjuk a műveleteket természetesen és ösztönösen, hogy megragadjuk az összhatását annak, ami történik:

```
SP = SP + 1; rd
```

Nevezzük a magas szintű Mikro assembly nyelvünket „MAL"-nak (franciául „beteg," amivé válsz, ha túl sok kódot kell ezen írnod). A MAL úgy van kialakítva, hogy tükrözze a mikroarchitektúra jellegzetességeit. Minden egyes ciklus alatt bármelyik regiszter írható, de jellemzően csak egy. Csak egy regiszter kerülhet az ALU B oldalára. Az A oldalon a választék +1, 0, –1 és a H regiszter. Így használhatunk egy egyszerű értékadó utasítást, mint a Javában, hogy jelezzük a végrehajtandó műveletet. Például, átmásolni valamit SP-ből MDR-be, mondhatjuk, hogy:

```
MDR = SP
```

Hogy jelezzük az ALU tevékenységeinek használatát, ne csak a B sín átküldését, írhatjuk például, hogy:

```
MDR = H + SP
```

amelyik hozzáadja a H regiszter tartalmát SP-hez, és beírja az eredményt az MDR-be. A + művelet kommutatív (ami azt jelenti, hogy az operandusok sorrendje nem számít), tehát a fenti utasítást írhatjuk a következőképp is:

```
MDR = SP + H
```

és ugyanazt a 36 bites mikroutasítást hozza létre, annak ellenére, hogy szigorúan megmondtuk, hogy H-nak bal oldali ALU operandusnak kell lennie.

Ügyelnünk kell arra, hogy csak megengedett műveleteket használjunk. A legfontosabb megengedett műveleteket a 4.16. ábra mutatja, ahol SOURCE lehet az MDR, PC, MBR, MBRU, SP, LV, CPP, TOS vagy OPC bármelyike (MBRU az MBR előjel nélküli változatát jelenti). Ezen regiszterek mindegyike az ALU forrása lehet a B sínen. Hasonlóan a DEST lehet a MAR, MDR, PC, SP, LV, CPP, TOS, OPC vagy H bármelyike, mindegyik az ALU kimenetének lehetséges célja a C sínen. Ez a forma megtévesztő, mert sok látszólag értelmes utasítás illegális. Például:


```
MDR = SP + MDR
```

teljesen elfogadhatónak látszik, de nincs mód arra, hogy egy ciklus alatt végrehajtsuk a 4.6. ábra adatútján. Ez a megszorítás azért van, mert egy összeadás (más, mint az 1-gyel való növelés vagy csökkentés!) egyik operandusának H-nak kell lennie. Hasonlóan,

```
H = H - MDR
```

hasznos lehetne, de szintén lehetetlen, mert a kivonandó (a kivont érték) egyetlen lehetséges forrása a H regiszter. Az assembler képes visszautasítani azokat az utasításokat, amelyek érvényesnek látszanak, de valójában illegálisak.

Kiterjesztjük a jelölésrendszert úgy, hogy megengedjünk többszörös értékadást több egyenlőségjel használatával. Például, 1-et adjunk SP-hez, és tároljuk vissza SP-be, és tároljuk MDR-be is. Ezt a következő módon írhatjuk fel:

```
SP = MDR = SP + 1
```

Annak jelzésére, hogy 4 bájtos adatszavakat olvas és ír a memória, csak egy rd-t és wr-t fogunk kitenni a mikroutasításban. Egy bájt betöltését az 1 bájtos porton keresztül a fetch jelzi. Értékadások és memóriaműveletek ugyanazon cikluson belül előfordulhatnak. Ezt azzal jelezzük, hogy ugyanabba a sorba írjuk azokat.

A zűrzavar elkerülése érdekében emlékezzünk arra a tényre, hogy a Mic–1 kétféleképpen érheti el a memóriát. 4 bájtos adatszavak olvasására és írására a MAR/MDR-t használjuk, és a mikroutasításban rd-vel, illetve wr-rel jelöljük. Az utasításfolyamból egybájtos műveleti kódok olvasására a PC/MBR-t használjuk, és a mikroutasításban fetch-csel jelöljük. A memóriaműveletek mindkét fajtája egyidejűleg folyamatban lehet.

Azonban ugyanaz a regiszter nem fogadhat értéket a memóriából és az adatútról ugyanabban a ciklusban. Tekintsük a következő kódot:

```
MAR = SP; rd
MDR = H
```

Az első mikroutasítás hatása az, hogy átad egy értéket a memóriából az MDR-be a második mikroutasítás végén. Azonban a második mikroutasítás ugyanakkor szintén betesz egy értéket az MDR-be. Ez a két értékadás összeütközésben van, és nincs megengedve, mert az eredménye definiálatlan.

Ne felejtsük el, hogy minden egyes mikroutasításnak félreérthetetlenül szolgáltatnia kell a következő végrehajtandó mikroutasítás címét. Azonban rendszeresen előfordul, hogy egy mikroutasításra csak egyetlen másik mikroutasítás utal, nevezetesen az, amelyik a közvetlenül felette lévő sorban van. Hogy megkönnyítsük a

DEST = H
DEST = SOURCE
DEST = $\overline{H}$
DEST = $\overline{SOURCE}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE − H
DEST = SOURCE − 1
DEST = −H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = −1

4.16. ábra. *Az összes megengedett művelet. A fenti utasítások bármelyike kiterjeszthető „<< 8" hozzáadásával, hogy az eredményt 1 bájttal balra léptessük. Például gyakori művelet H = MBR << 8*

mikroprogramozó munkáját, a mikroassembler minden mikroutasításhoz természetszerűleg meghatároz egy címet (nem feltétlenül a soron következőt a vezérlőtárban), és kitölti a NEXT_ADDRESS mezőt, vagyis azok a mikroutasítások, amelyeket egymásra következő sorokba írtunk, egymást követően lesznek végrehajtva.

Azonban néha a mikroprogramozó elágazni akar feltétlenül vagy feltételes módon. A feltétlen elágazás jelölése könnyű:

```
goto label
```

és bármely mikroutasítás tartalmazhatja, amelyik határozottan megnevezi a következőjét. Például, a legtöbb mikroutasítás-sorozat visszatéréssel végződik a főciklus első utasítására, így minden ilyen sorozatban jellemzően az utolsó utasítás ez:

```
goto Main1
```

Megjegyezzük, hogy egy goto-t tartalmazó mikroutasítás alatt az adatút a közönséges műveletek számára egészen végig elérhető. Végül is minden egyes mikroutasítás tartalmaz egy NEXT_ADDRESS mezőt. Mindössze annyit csinál a goto, hogy utasítja a mikroassemblert, hogy tegyen egy meghatározott értéket a következő sorban lévő mikroutasítás címe helyett. Elvben minden sornak kellene lennie egy goto utasításának, csupán a mikroprogramozók kényelmét szolgálja, hogy amikor a célcím a következő sor, akkor ez elhagyható.

Feltételes elágazások számára egy másik jelölésre van szükségünk. Ne felejtsük el, hogy a JAMN és JAMZ az N és Z biteket használja, melyek az ALU kimenetétől

függnek. Néha szükséges megvizsgálni egy regisztert, hogy lássuk például, hogy nulla-e. Az egyik módja az lehet ennek, hogy átfuttatjuk az ALU-n és visszatároljuk saját magára. Azt írni, hogy:

```
TOS = TOS
```

furcsának tűnhet, bár megcsinálja a feladatot (beállítja a Z flip-flopot TOS alapján). Azonban, hogy a mikroprogramokat barátságosabb kinézetűvé tegyük, kiterjesztjük a MAL-t, hozzáveszünk két új képzeletbeli regisztert, N-t és Z-t, melyeknek értéket adhatunk. Például:

```
Z = TOS
```

átküldi TOS-t az ALU-n, és beállítja a Z (és N) flip-flopokat, de nem tárol egyik regiszterbe se. Z vagy N célként való használata valójában azt jelenti, hogy megmondja a mikroassemblernek, hogy a 4.5. ábra C mezőjének összes bitjét állítsa be 0-ra. Az adatút végrehajt egy közönséges ciklust minden szabályos műveletet megengedve, de egyik regiszterbe se ír. Megjegyezzük, hogy nem számít, hogy a cél N vagy Z; a mikroassembler által előállított mikroutasítás ugyanaz. A programozókat, akik szándékosan „rosszat" választanak, kényszeríteni kellene, hogy büntetésül 4,77 Mhz-es eredeti IBM PC-n dolgozzanak egy hétig.

A szintaxis, amelyik megmondja a mikroassemblernek, hogy állítsa be a JAMZ bitet:

```
if (Z) goto L1; else goto L2
```

Mivel a hardver megköveteli, hogy ez a két cím azonos legyen az alsó nyolc bitjében, a mikroassembler feladata, hogy kiosszon nekik ilyen címeket. Másrészről, mivel *L2* bárhol lehet a vezérlőtár alsó 256 szavában, a mikroassemblernek nagy szabadsága van a megfelelő pár megtalálására.

Természetesen, ez a két utasítás összekapcsolódhat, például:

```
Z = TOS; if (Z) goto L1; else goto L2
```

Ennek az utasításnak a hatása az, hogy a MAL előállít egy olyan mikroutasítást, amelyben a TOS átfut az ALU-n (de nem tárolódik sehova), így az értéke beállítja a Z bitet. Kicsivel azután, hogy Z feltöltődött az ALU feltételbitjéből, egy OR művelettel bekerül az MPC legmagasabb bitjére, hogy kialakítsa azt a címet, ahonnan a következő mikroutasítás betöltődik, ami vagy *L2* vagy *L1* (melynek pontosan 256-tal kell nagyobbnak lennie, mint *L2*). Az MPC stabil lesz, és kész arra, hogy a következő mikroutasítás betöltéséhez használjuk.

Végül szükségünk van jelölésre a JMPC bit használatához. Az egyetlen, amit használni fogunk:

```
goto (MBR OR value)
```

Ez a szintaxis azt mondja a mikroassemblernek, hogy a NEXT_ADDRESS számára használja a *value*-t, és állítsa be a JMPC bitet úgy, hogy az MBR OR művelettel kerüljön MPC-be, NEXT_ADRESS-szel együtt. Ha a *value* 0, ami a gyakori, elegendő éppen csak annyit írni, hogy:

```
goto (MBR)
```

Megjegyezzük, hogy csak az MBR alsó 8 bitjét vezetjük az MPC-be (lásd 4.6. ábra), így az előjel-kiterjesztés témája (vagyis az MBR vagy MBRU) itt nem merül fel. Azt is megjegyezzük, hogy csak az az MBR érték használható, ami az aktuális ciklus végén elérhető. Egy ebben a mikroutasításban kezdődött betöltés túl későn történik ahhoz, hogy a következő mikroutasítás választására hatással legyen.

4.3.2. IJVM megvalósítása Mic–1 felhasználásával

Végre elértünk ahhoz a ponthoz, amikor minden darabot összeilleszthetünk. A 4.17. ábra egy mikroprogram, amelyik Mic–1-en fut, és IJVM-et értelmezi. Ez egy meglepően rövid program – csak 112 mikroutasítás. Minden mikroutasításhoz három oszlop van megadva: a szimbolikus név, a hozzá tartozó mikrokód és a megjegyzés. Megjegyezzük, hogy az egymást követő mikroutasítások nem feltétlenül egymást követő címeken helyezkednek el a vezérlőtárban, ahogy erre már utaltunk.

Mostanra a 4.1. ábra legtöbb regisztere számára a nevek kiválasztása nyilvánvalóvá vált: CPP, LV és SP arra szolgálnak, hogy mutatókat tartalmaznak rendre a konstans mezőre, lokális változókra és a verem tetejére, míg a PC az utasításfolyam következőként betöltendő bájtjának címét tartalmazza. Az MBR egy 1 bájtos regiszter, amelyik az utasításfolyam bájtjait egymást követően tartalmazza, ahogyan azok a memóriából jönnek és értelmezésre kerülnek. A TOS és az OPC különleges regiszterek. Használatukat az alábbiakban írjuk le.

Bizonyos időnként e regiszterek mindegyike garantáltan egy bizonyos értéket tartalmaz, de ideiglenes regiszterként használhatók, ha szükséges. Minden egyes utasítás elején és végén TOS tartalmazza annak a memóriahelynek az értékét, melyre az SP mutat, vagyis a verem legfelső szavát. Ez az érték felesleges, mivel mindig kiolvasható a memóriából, de regiszterben tartva gyakran memóriahivatkozást takarítunk meg. Néhány olyan utasítás számára, amely a TOS-t karbantartja, ez *többlet* memóriaműveletet jelent. Ilyen például a POP utasítás, amelyik eldobja a legfelső szót, és ennek következtében be kell töltenie az új verem tetején lévő szót a memóriából a TOS-ba.

Az OPC regiszter ideiglenes (azaz firkáló) regiszter. Nincs előre meghatározott használata. Használjuk például annak a címnek a tárolására, ahol az elágazó utasítás műveleti kódja van, miközben a PC-t megnöveljük, hogy a paramétereket elérjük. Használjuk ideiglenes regiszterként is az IJVM feltételes elágazás utasításainál.

Mint minden értelmezőnek, a 4.17. ábra mikroprogramjának is van egy főciklusa, amelyik betölti, dekódolja és végrehajtja az értelmezendő program utasításait, jelen esetben IJVM-utasításokat. A főciklus a Main1-gyel címkézett sorban kezdődik. Mindig úgy indul, hogy a PC egy műveleti kódot tartalmazó memóriahely címét tartalmazza, továbbá ez a műveleti kód már be van töltve az MBR-be. Megjegyezzük, hogy ebbe beleértjük azt is, hogy gondoskodnunk kell arról, hogy amikor visszakerülünk erre a helyre, a PC úgy legyen frissítve, hogy a következő értelmezendő műveleti kódra mutasson, és maga a műveleti kód már be legyen töltve az MBR-be.

Ez a kezdő utasítássorozat minden utasítás elején végrehajtódik, vagyis nagyon fontos, hogy a lehető legrövidebb legyen. A Mic–1 hardverének és szoftverének nagyon gondos tervezése által sikerült a főciklust egyetlen mikroutasításra csökkenteni. Amint elindul a főciklus, minden alkalommal végrehajtódik ez a mikroutasítás, a végrehajtandó IJVM műveleti kód már MBR-ben van. A mikroutasítás elágazik az IJVM-utasítást értelmező mikrokódhoz, és kezdeményezi a műveleti kódot követő bájt betöltését is, amelyik vagy egy operandusbájt, vagy a következő műveleti kód lehet.

Most feltárhatjuk a valódi okát annak, hogy minden mikroutasítás határozottan megnevezi a követőjét ahelyett, hogy a vezérlőtárban egymást követő utasítások hajtódnának végre. Minden műveleti kóddal megegyező vezérlőtárcímet fenn kell tartani a megfelelő utasításértelmező első szava számára. Így a 4.11. ábrából látjuk, hogy az a kód, amelyik a POP-ot értelmezi 0x57-nél kezdődik, és az a kód, amelyik a DUP-ot értelmezi 0x59-nél kezdődik. (Az, hogy hogyan tudja a MAL a POP-ot 0x57-re tenni, egyike a világegyetem rejtélyeinek – feltehetőleg létezik valahol egy leírás, amely elmeséli.)

Sajnos, a POP értelmezéséhez három mikroutasítás szükséges, így ha egymást követő szavakban helyeznénk el, összeütközésbe kerülne a DUP kezdetével. Mivel így az összes műveleti kódhoz tartozó vezérlőtárcím ténylegesen foglalt, minden egyes sorozatban az elsőn kívüli mikroutasításoknak a foglalt címek közötti lyukakba kell kerülniük. Ezért a sok ide-oda ugrálás, így ha külön mikroelágazásokkal (olyan mikroutasítás, amelyik elágazik) kellene néhány lyukról újabb lyukra ugrani, az nagyon pazarló lenne.

Ahhoz, hogy lássuk, az értelmező hogyan dolgozik, tegyük fel például, hogy az MBR a 0x60 értéket tartalmazza, vagyis az IADD műveleti kódját (lásd 4.11. ábra). Az egy mikroutasításos főciklusban három dolgot hajtunk végre:

1. Növeljük a PC-t, ezzel elérjük, hogy a műveleti kódot követő első bájt címét tartalmazza.
2. Kezdeményezzük a következő bájt betöltését MBR-be. Ez a bájt mindig szükséges előbb vagy utóbb, vagy mint egy operandus a mostani IJVM-utasítás számára, vagy mint a következő műveleti kód (mint az IADD művelet esetében, amelynek nincs operandusbájtja).
3. Végrehajtunk egy többirányú elágazást arra a címre, amelyet a Main1 kezdetén az MBR tartalmaz. Ez a cím megegyezik a jelenleg végrehajtás alatt lévő műveleti kód numerikus értékével. Ezt még egy előző mikroutasítás tette ide. Gondosan figyeljük meg, hogy az az érték, ami ebben a mikroutasításban kerül betöltésre, nem játszik semmilyen szerepet ebben a többirányú elágazásban.

A következő bájt betöltése itt már elkezdődött, így a harmadik mikroutasítás kezdetére rendelkezésre fog állni. Akkor vagy szükség lesz rá, vagy nem, de végül is szükség lesz rá, vagyis a betöltés mostani elindítása semmi esetre sem tud bajt okozni.

Ha az MBR-ben lévő bájt történetesen csupa 0, ez a NOP műveleti kódja, a következő mikroutasítás a nop1-gyel címkézett 0. helyről betöltött lesz. Mivel az utasítás nem csinál semmit, ez egyszerűen egy visszaugrás a főciklus elejére, ahol a sorozat ismétlődik, de már az MBR-be betöltött új műveleti kóddal.

Ismét hangsúlyozzuk, hogy a mikroutasítások a 4.17. ábrában nem ebben a sorrendben vannak a memóriában, és hogy a Main1 nem a vezérlőtár 0. címén van (mert nop1-nek kell a 0. címen lennie). A mikroassembler feladata, hogy minden mikroutasítást valahova eltegyen, és rövid sorozatokká kapcsolja össze a NEXT_ADDRESS mezőt használva. Minden sorozat azon a címen kezdődik, amelyik megegyezik az értelmezendő IJVM műveleti kód numerikus értékével (például a POP a 0x57-en kezdődik), de a sorozat maradék része a vezérlőtárban bárhol lehet, és nem szükségszerűen egymást követő címeken.

Most tekintsük az IJVM IADD utasítását. A főciklus az iadd1-gyel címkézett mikroutasításra ágazik el. Ez az utasítás indítja az IADD esetén végzendő munkát:

1. A TOS már rendelkezésre áll, de a verem teteje alatti szót még be kell tölteni a memóriából.
2. A TOS-t hozzá kell adni a memóriából betöltött, a verem teteje alatti szóhoz.
3. Az eredményt, amelyet a verembe fogunk tenni, vissza kell tárolni a memóriába, és egyben tárolni kell a TOS regiszterbe is.

Ahhoz, hogy a memóriából betöltsük az operandust, szükséges, hogy a veremmutató értékét csökkentsük, és beírjuk a MAR-ba. Megjegyezzük, hogy alkalmasint ez a cím ugyanaz, mint amit a következő írásnál használni fogunk. Továbbá, mivel ez a hely a verem új teteje lesz, az SP-nek is ezt az értéket kell adnunk. Ennek következtében egyetlenegy művelet meg tudja határozni az SP és MAR új értékét, csökkenti az SP-t, és beírja mindkét regiszterbe.

Ezek a dolgok befejeződnek az első ciklusban, iadd1-ben, és az olvasásművelet elkezdődik. Ezenkívül az MPC értéket kap az iadd1 NEXT_ADDRESS mezőjéből, ez az iadd2 címe, bárhol is legyen az. Ezután az iadd2 kiolvasódik a vezérlőtárból. A második ciklusban, amíg várjuk, hogy az operandus beolvasódjon a memóriából, átmásoljuk a verem legfelső szavát a TOS-ból a H-ba, ahol majd rendelkezésre áll az összeadás számára, amikor az olvasás befejeződik.

A harmadik ciklus (iadd3) elején az MDR tartalmazza a memóriából betöltött összeadandót. Ebben a ciklusban ez a H tartalmához hozzáadódik, és az eredmény visszatárolódik az MDR-be és egyben a TOS-ba is. Egy írásművelet szintén elkezdődik, amely visszatárolja az új verem teteje szót a memóriába. Ebben a ciklusban a goto hatása az, hogy Main1 címét az MPC-be teszi, ezzel visszajuttat bennünket a következő utasítás végrehajtásának kezdőpontjára.

Ha a soron következő IJVM műveleti kód, amit most az MBR tartalmaz, 0x64 (ISUB), akkor szinte pontosan ugyanez az eseménysor történik meg ismét. Main1

Címke	Műveletek	Megjegyzések
Main1	PC = PC + 1; fetch; goto (MBR)	MBR-ben a műveleti kód; veszi a következő bájtot, elágazás.
nop1	goto Main1	Semmit sem csinál.
iadd1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
iadd2	H = TOS	H = verem teteje
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Összeadja a két felső szót; beírja a verem tetejére.
isub1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
isub2	H = TOS	H = verem teteje
isub3	MDR = TOS = MDR – H; wr; goto Main1	Végrehajtja a kivonást; beírja a verem tetejére.
iand1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
iand2	H = TOS	H = verem teteje
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Végrehajtja AND-et; beírja a verem új tetejére.
ior1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
ior2	H = TOS	H = verem teteje
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Végrehajtja OR-t; beírja a verem új tetejére.
dup1	MAR = SP = SP + 1	Növeli SP-t és MAR-ba másolja.
dup2	MDR = TOS; wr; goto Main1	Beírja az új verem szót.
pop1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
pop2		Várja, hogy az új TOS bekerüljön a memóriából.
pop3	TOS = MDR; goto Main1	Az új szót a TOS-ba másolja.
swap1	MAR = SP – 1; rd	Legyen MAR = SP – 1; beolvassa a verem 2. szavát.
swap2	MAR = SP	Legyen MAR a verem teteje.
swap3	H = MDR; wr	TOS-t H-ba menti; a 2. szót a verem tetejére írja.
swap4	MDR = TOS	A régi TOS-t MDR-be másolja.
swap5	MAR = SP – 1; wr	Legyen MAR SP – 1; 2. szóként a verembe írja.
swap6	TOS = H; goto Main1	Frissíti a TOS-t.
bipush1	SP = MAR = SP + 1	MBR = az a bájt, amit a verembe kell tenni
bipush2	PC = PC + 1; fetch	Noveli PC-t, betolti a kovetkező müveleti kódot.
bipush3	MDR = TOS = MBR; wr; goto Main1	A bájt előjel kiterjesztése, és betesszük a verembe.
iload1	H = LV	MBR tartalmazza az indexet; LV-t H-ba másoljuk.
iload2	MAR = MBRU + H; rd	MAR = a betöltendő lokális változó címe
iload3	MAR = SP = SP + 1	SP a verem új tetejére mutat; az írás előkészítése.
iload4	PC = PC + 1; fetch; wr	PC növelése; következő műveleti kód betöltése; verem tetejének írása.
iload5	TOS = MDR; goto Main1	Frissíti a TOS-t.
istore1	H = LV	MBR tartalmazza az indexet; LV-t H-ba másoljuk.
istore2	MAR = MBRU + H	MAR = a tárolandó lokális változó címe
istore3	MDR = TOS; wr	TOS másolása MDR-be; szó írása.
istore4	SP = MAR = SP – 1; rd	Beolvassa a verem teteje alatti szót.
istore5	PC = PC + 1; fetch	Noveli PC-t, betölti a következő műveleti kódot.
istore6	TOS = MDR; goto Main1	Frissíti a TOS-t.
wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)	Betölti az operandusbájtot vagy a műveleti kódot, többirányú elágazás a magas bit beállításával.
wide_iload1	PC = PC + 1; fetch	MBR tartalmazza az 1. indexbájtot; a 2. betöltése.
wide_iload2	H = MBRU << 8	H = az első indexbájt 8 bittel balra léptetve
wide_iload3	H = MBRU OR H	H = a lokális változó 16 bites indexe
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = a betoltendő lokális változó címe
wide_istore1	PC = PC + 1; fetch	MBR tartalmazza az 1. indexbájtot; a 2. betöltése.
wide_istore2	H = MBRU << 8	H = az első index bájt 8 bittel balra léptetve
wide_istore3	H = MBRU OR H	H = a lokális változó 16 bites indexe
wide_istore4	MAR = LV + H; goto istore3	MAR = a betoltendő lokális változó címe
idc_w1	PC = PC + 1; fetch	MBR tartalmazza az 1. indexbájtot; a 2. betöltése.
idc_w2	H = MBRU << 8	H = 1. indexbájt << 8

4.17. ábra. *A Mic–1 mikroprogramja*

Címke	Műveletek	Megjegyzések
idc_w3	H = MBRU OR H	H = 16 bites index a konstans mezőre
idc_w4	MAR = H + CPP; rd; goto iload3	MAR = a konstans címe a mezőn
iinc1	H = LV	MBR tartalmazza az indexet; LV-t H-ba másoljuk.
iinc2	MAR = MBRU + H; rd	LV + index másolása MAR-ba; változó olvasása.
iinc3	PC = PC + 1; fetch	Konstans betöltése.
iinc4	H = MDR	Változó másolása H-ba.
iinc5	PC = PC + 1; fetch	Következő műveleti kód betoltése.
iinc6	MDR = MBR + H; wr; goto Main1	Összeg MDR-be tárolása; változó frissítése.
goto1	OPC = PC - 1	A műveleti kód címének tárolása.
goto2	PC = PC + 1; fetch	MBR = az eltolás 1. bájtja; a második betöltése
goto3	H = MBR << 8	Léptetés és az előjeles első bájt H-ba tárolása.
goto4	H = MBRU OR H	H = 16 bites elágazáseltolás
goto5	PC = OPC + H; fetch; goto Main1	Az eltolás hozzáadása OPC-hez.
iflt1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót..
iflt2	OPC = TOS	TOS-t ideiglenesen OPC-be menti.
iflt3	TOS = MDR	A verem új tetejét TOS-ba teszi.
iflt4	N = OPC; if (N) goto T; else goto F	Elágazás N bit szerint.
ifeq1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
ifeq2	OPC = TOS	TOS-t ideiglenesen OPC-be menti.
ifeq3	TOS = MDR	A verem új tetejét TOS-ba teszi.
ifeq4	Z = OPC; if (Z) goto T; else goto F	Elágazás Z bit szerint.
if_icmpeq1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
if_icmpeq2	MAR = SP = SP - 1	Beállítja MAR-t a verem új tetejének olvasásához.
if_icmpeq3	H = MDR; rd	A verem második szavának másolása H-ba.
if_icmpeq4	OPC = TOS	TOS-t ideiglenesen OPC-be menti.
if_icmpeq5	TOS = MDR	A verem új tetejét TOS-ba teszi.
if_icmpeq6	Z = OPC - H; if (Z) goto T; else goto F	Ha a felső 2 szó egyenlő, T-re megy, kulonben F-re.
T	OPC = PC - 1; goto goto2	Ugyanaz, mint goto1; szükséges a célcímhez.
F	PC = PC + 1	Átugorja az első eltolásbájtot.
F2	PC = PC + 1; fetch; goto Main1	PC most a következő műveleti kódra mutat.
invokevirtual1	PC = PC + 1; fetch	MBR = indexbájt 1; PC növelése; 2. bájt betöltése.
invokevirtual2	H = MBRU << 8	Az első bájt léptetése és H-ba tárolása.
invokevirtual3	H = MBRU OR H	H = a metódus mutató eltolása CPP-től
invokevirtual4	MAR = CPP + H; rd	A metódus mutatójának kinyerése a konstans mezőből.
invokevirtual5	OPC = PC + 1	Visszatérési PC tárolása OPC-be ideiglenesen.
invokevirtual6	PC = MDR; fetch	PC az új metódusra mutat; paraméterszám betöltése.
invokevirtual7	PC = PC + 1; fetch	A paraméterszám 2. bájtjának betöltése.
invokevirtual8	H = MBRU << 8	Az első bájt léptetése, és H-ba tárolása.
invokevirtual9	H = MBRU OR H	H = a paraméterek száma
invokevirtual10	PC = PC + 1; fetch	A lokálisok száma első bájtjának betöltése.
invokevirtual11	TOS = SP - H	TOS = OBJREF címe - 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = OBJREF címe (új LV)
invokevirtual13	PC = PC + 1; fetch	A lokálisok száma második bájtjának betöltése.
invokevirtual14	H = MBRU << 8	Az első bájt léptetése, és H-ba tárolása.
invokevirtual15	H = MBRU OR H	H = lokálisok száma
invokevirtual16	MDR = SP + H + 1; wr	OBJREF felulírása a kapcsoló mutatóval.
invokevirtual17	MAR = SP = MDR;	Legyen SP, MAR az a hely, ahol a régi PC-t tartjuk.
invokevirtual18	MDR = OPC; wr	A régi PC tárolása a lokális változók felett.
invokevirtual19	MAR = SP = SP + 1	SP arra a helyre mutat, ahol az LV-t tartjuk.
invokevirtual20	MDR = LV, wr	A régi LV tárolása az eltárolt PC felett.
invokevirtual21	PC = PC + 1; fetch	Az új metódus első műveleti kódjának betoltése.

4.17. ábra. *A Mic-1 mikroprogramja (folytatás)*

Címke	Műveletek	Megjegyzések
invokevirtual22	LV = TOS	LV mutasson a lokális mező keretre.
invokevirtual23	TOS = MDR; goto Main1	TOS = a hívó LV-je
ireturn1	MAR = SP = LV; rd	SP, MAR visszaállítása a kapcsoló mutató olvasásához.
ireturn2		Várakozás olvasásra.
ireturn3	LV = MAR = MDR; rd	LV legyen a kapcsoló mutató; kivesszük a régi PC-t.
ireturn4	MAR = LV + 1	MAR beállítása a régi LV olvasására.
ireturn5	PC = MDR; rd; fetch	PC helyreállítása; új műveleti kód beolvasása.
ireturn6	MAR = SP	MAR beállítása TOS írására.
ireturn7	LV = MDR	LV helyreállítása.
ireturn8	MDR = TOS; wr; goto Main1	Visszatérési érték eltárolása a verem eredeti tetején.

4.17. ábra. *A Mic-1 mikroprogramja (folytatás)*

végrehajtása után a vezérlés áthelyeződik a 0x64-nél (isub1) lévő mikroutasításhoz. Ezt a mikroutasítást követi az isub2 és isub3, és azután ismét a Main1. Az egyetlen különbség a mostani és az előző sorozat között, hogy az isub3-ban H tartalma kivonódik az MDR-ből, és nem hozzáadódik.

Az IAND értelmezése majdnem azonos az IADD-dal és ISUB-bal, kivéve, hogy a verem tetején lévő két szón bitenkénti AND műveletet végzünk ahelyett, hogy összeadnánk vagy kivonnánk. Teljesen hasonló történik az IOR-nál.

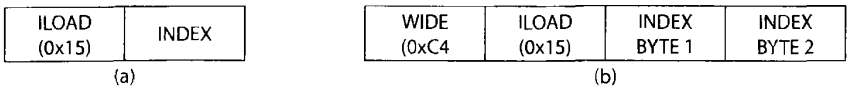
Ha az IJVM műveleti kód DUP, POP vagy SWAP, a vermet módosítani kell. A DUP művelet egyszerűen másolatot készít a verem tetején lévő szóról. Mivel ennek a szónak az értéke már a TOS-ba tárolva lett, a művelet egyszerűen növeli az SP-t, hogy egy új helyre mutasson, és tárolja a TOS-t erre a helyre. A POP művelet majdnem ilyen egyszerű, csak csökkenti az SP-t, hogy eldobhassa a verem tetején lévő szót. Viszont annak érdekében, hogy karbantartsuk a legfelső szót a TOS-ban, szükséges, hogy beolvassuk az új legfelső szót a memóriából és beírjuk a TOS-ba. Végül a SWAP utasításcserét bonyolít le két memóriahely értékein: a verem tetején lévő két szón. Ez valamivel könnyebb azáltal, hogy a TOS már tartalmazza a két érték közül az egyiket, így ezt nem kell beolvasni a memóriából. Ezt az utasítást egy kicsivel később fogjuk részletesebben tárgyalni.

A BIPUSH utasítás egy kicsivel bonyolultabb, mert a műveleti kódot egy bájt követi, ahogy az a 4.18. ábrán látszik. A bájtot előjeles egészként értelmezzük. Ezt a bájtot, amelyiket már betöltöttünk az MBR-be a Main1-ben, 32 bitessé kell tennünk előjel-kiterjesztéssel, és a verem tetejére kell tennünk. Ennek következtében a sorozatnak ki kell terjesztenie az MBR-ben lévő bájtot 32 bitessé, és be kell másolnia az MDR-be. Végül az SP-t megnöveljük és bemásoljuk MAR-ba, lehetővé téve az operandus kiírását a verem tetejére. Közben az operandust a TOS-ba is be kell másolni. Vegyük észre, hogy a PC-t növelni kell mielőtt visszatérünk a főprogramba, hogy a következő műveleti kód rendelkezésre álljon Main1-ben.

BIPUSH (0x10)	BYTE

4.18. ábra. *A BIPUSH utasítás alakja*

Következőnek tekintsük az ILOAD utasítást. Az ILOAD-nak is van egy műveleti kódot követő bájtja, mint az a 4.19. (a) ábrán látható, de ez a bájt egy (előjel nélküli) index a lokális változók mezőjén annak a szónak az azonosításához, amelyiket a verembe kell tenni. Mivel ez csak egy bájt, csak $2^8 = 256$ szót tudunk megkülönböztetni, nevezetesen az első 256 szót a lokális változók mezőjén. Az ILOAD művelet megkövetel egy olvasást (hogy megkapjuk a szót) és egy írást (hogy a verem tetejére tegyük). Annak érdekében, hogy meghatározzuk az olvasáshoz a címet, az MBR-ben lévő eltolást hozzá kell adni az LV tartalmához. Mivel mind az MBR, mind az LV csak a B sínen keresztül érhető el, először az LV-t másoljuk H-ba (iload1-ben), azután az MBR-t adjuk hozzá. Az összeadás eredményét a MAR-ba másoljuk, és egy olvasást kezdeményezünk (iload2-ben).



4.19. ábra. *(a) Az ILOAD az 1 bájtos indexszel. (b) WIDE ILOAD 2 bájtos indexszel*

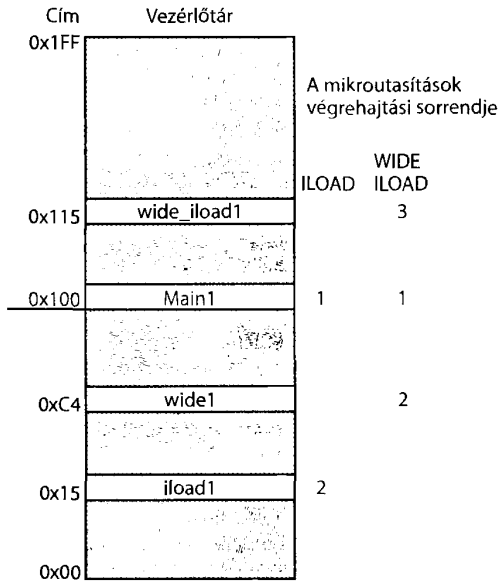
Az MBR indexként való használata alig különbözik attól, amit a BIPUSH-nál láttunk, ott előjel-kiterjesztést alkalmaztunk. Egy index esetében az eltolás mindig pozitív, így a bájt eltolást előjel nélküli egészként kell értelmeznünk, nem úgy, mint BIPUSH-nál, ahol 8 bites előjeles egészként volt értelmezve. Az MBR illesztése a B sínre körültekintően van megtervezve, hogy mindkét műveletet lehetővé tegye. A BIPUSH esetében (előjeles 8 bites egész) a megfelelő művelet az előjel-kiterjesztés, vagyis az 1 bájtos MBR bal szélső bitjét másoljuk a B sín felső 24 bitjére. Az ILOAD esetében (előjel nélküli 8 bites egész) a megfelelő művelet a nullával való kitöltés. Itt a B sín felső 24 bitjét egyszerűen kipótoljuk nullával. A két műveletet eltérő vezérlőjelek különböztetik meg, amelyek mutatják, hogy melyik műveletet kell végrehajtani (lásd 4.6. ábra). A mikrokódban ezt az MBR jelzi (előjel-kiterjesztett, mint a bipush3-ban) vagy a MBRU (előjel nélküli, mint iload2-ben).

Míg várjuk, hogy a memória az operandust szolgáltassa (iload3-ban), az SP-t növeljük, hogy tartalmazza azt az értéket, ahova az eredményt tároljuk, a verem új tetejét. Ezt az értéket a MAR-ba is átmásoljuk, hogy előkészítsük az operandus kiírását a verem tetejére. A PC-t ismét növelni kell, hogy a következő műveleti kódot betöltsük (iload4-ben). Végül MDR-t TOS-ba másoljuk, hogy tükrözzük a verem új tetejét (iload5-ben).

Az ISTORE az ILOAD fordított művelete, vagyis egy szót kiveszünk a verem tetejéről, és tároljuk arra a helyre, amit az LV és az utasításban lévő index összege meghatároz. Ugyanazt a formát használjuk, mint ILOAD-nál, ahogy a 4.19. (a) ábrán látszik, kivéve, hogy a műveleti kód 0x15 helyett 0x36. Ez a művelet némiképp különbözik attól, amit várnánk, mert a verem legfelső szava már ismert (TOS-ban), tehát azonnal tárolhatjuk. Azonban az új verem-teteje szót be kell tölteni. Így egy olvasás és egy írás szükséges, de ezek tetszőleges sorrendben végrehajthatók (akár párhuzamosan, ha ez lehetséges).

Mind az ILOAD, mind az ISTORE korlátozottak abban, hogy a lokális változókból csak az első 256-ot tudják elérni. Míg a legtöbb program számára elegendő lehet ez, de természetesen elengedhetetlen, hogy el tudjunk érni egy változót, bárhol is legyen a lokális változó mezőn. Hogy ezt elérjük, az IJVM ugyanazt a mechanizmust alkalmazza, mint a JVM: egy speciális műveleti kódot, a WIDE-ot, amit az ILOAD vagy ISTORE műveleti kód követ. A WIDE kód **prefix bájt**ként vagy prefixumként ismert. Az ILOAD és az ISTORE definíciója azzal módosul, amikor ez a sorozat előfordul, hogy 16 bites index követi a műveleti kódot 8 bites index helyett, ahogy a 4.19. (b) ábrán látható.

A WIDE-ot a szokásos módon dekódoljuk, ez a wide1-hez való elágazáshoz vezet, ami a WIDE műveleti kódot kezeli. Bár a kiszélesített műveleti kód már az MBR-ben elérhető, wide1 betölti a műveleti kód utáni első bájtot, mert a mikroprogram logikája mindig elvárja, hogy ez ott legyen. Ezután egy második többirányú elágazás történik wide2-ben, ez alkalommal a WIDE-ot követő bájtot használva kiindulásnak. De mivel a WIDE ILOAD más mikrokódot igényel, mint az ILOAD, és a WIDE ISTORE mást, mint az ISTORE stb., a második többirányú elágazás nem használhatja ugyanazt a műveleti kódot, mint célcímet, ahogy azt a Main1 teszi.



4.20. ábra. *A kezdeti mikroutasítás sorozat ILOAD és WIDE ILOAD számára. A címek példák*

Ehelyett a wide1 0x100-zal és a műveleti kóddal végzett OR művelet eredményét teszi MPC-be. Eredményként a WIDE ILOAD értelmezése 0x115-ön kezdődik (0x15 helyett), a WIDE ISTORE értelmezése a 0x136-ön kezdődik (0x36 helyett) és így tovább. Ilyen módon minden WIDE műveleti kód a vezérlőtárban egy 256 (vagy-

is 0x100) szóval magasabb címen kezdődik, mint a neki megfelelő közönséges műveleti kód. Az ILOAD és a WIDE ILOAD utasításokhoz tartozó mikroutasítások kezdő sorozata a 4.20. ábrán látható.

Végre eljutunk oda, hogy a WIDE ILOAD (0x115) megvalósuljon. A közönséges ILOAD-tól a kód csak annyiban különbözik, hogy az indexet két indexbájt összefűzésével kell elkészíteni, egyetlenegy bájt egyszerű kiterjesztése helyett. Az összefűzést és az azutáni összeadást lépésenként kell elvégezni, először az INDEX BYTE 1-t 8 bittel balra léptetve betesszük a H-ba. Mivel az index előjel nélküli egész, ezért az MBRU használatával az MBR nulla kiterjesztésű lesz. Most az index második bájtját hozzáadjuk (az összeadás művelet megegyezik az OR művelettel, mivel H alacsonyabb bájtja most nulla, ez biztosítja, hogy nem lesz átvitel a bájtok között), és az eredményt megint H-ba tároljuk. Innen kezdve a művelet pontosan úgy hajtható végre, mintha közönséges ILOAD lenne. Az ILOAD végső utasításainak (iload3 – iload5) másolása helyett egyszerűen a wide_iload4-ből iload3-ba ágazunk. Megjegyezzük azonban, hogy a PC-t kétszer kell növelni az utasítás végrehajtása alatt, hogy végül a következő műveleti kódra mutasson. Az ILOAD művelet egyszer megnöveli; a WIDE_ILOAD sorozat szintén megnöveli egyszer.

Hasonló helyzet fordul elő a WIDE_ISTORE-nál: az első négy mikroutasítás végrehajtása után (wide_istore1 – wide_istore4), a sorozat ugyanaz, mint ISTORE sorozata az első két utasítás után, tehát a wide_istore4-ből istore3-ba ágazunk.

Következő példának tekintsük az LDC_W utasítást. Ez a műveleti kód két dologban különbözik az ILOAD-tól. Az első, hogy 16 bites előjel nélküli eltolása van (mint az ILOAD bővített változatának). Másodszor, a CPP-t indexeljük az LV helyett, mivel az a feladata, hogy a konstans mezőről olvasson, és nem a lokális változó mezőről. (Valójában van egy rövid alakja az LDC_W-nek (LDC), de mi ezt nem vettük be az IJVM-be, mivel a hosszú forma magában foglalja a rövid forma összes lehetséges változatát, csak 3 bájtot igényel 2 helyett.)

Az IINC utasítás az egyetlen IJVM-utasítás az ISTORE-on kívül, amelyik módosítani tud egy lokális változót. Ez kétoperandusú művelet, mindegyik 1 bájt hosszúságú, mint ahogy a 4.21. ábrán látható.

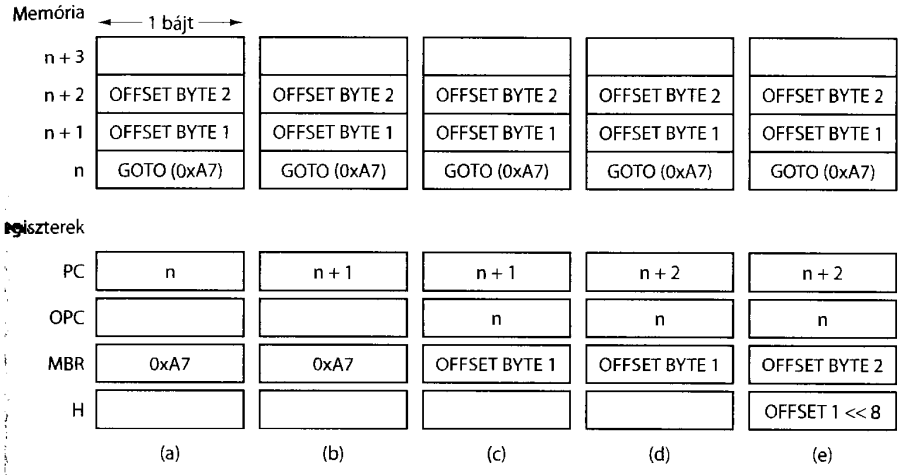
4.21. ábra. *Az IINC utasításnak két különböző operandusmezője van*

Az IINC utasítás az INDEX-et arra használja, hogy meghatározza az eltolást a lokális változó mező kezdetétől. Beolvassa a változót, megnöveli CONST-tal, egy olyan értékkel, amit az utasítás tartalmaz, és visszatárolja ugyanoda. Megjegyezzük, hogy ez az utasítás negatív mennyiséggel is növelhet, vagyis a CONST egy 8 bites előjeles konstans, a –128, +127 tartományban. A teljes JVM tartalmazza az IINC bővített változatát, ahol minden operandus 2 bájt hosszú.

Most elérkeztünk az IJVM első elágazó utasításához: a GOTO-hoz. Ennek az utasításnak egyetlen tevékenysége, hogy megváltoztatja a PC értékét, hogy a következő végrehajtandó IJVM-utasítás azon a címen legyen, melyet úgy számolunk

ki, hogy az (előjeles) 16 bites eltolást hozzáadjuk az elágazó utasítás címéhez. Az okoz bonyodamat, hogy az eltolás a PC azon értékére vonatkozik, ami az utasítás dekódolásának kezdetén volt, és nem arra az értékre, ami a 2 eltolás bájt betöltése után kialakult.

Hogy ezt a részletet világosabbá tegyük, a 4.22. (a) ábrán bemutatjuk a helyzetet a Main1 indulásakor. A műveleti kód már az MBR-ben van, de a PC még nincs megnövelve. A 4.22. (b) ábrán látjuk a helyzetet a goto1 indulásakor. Mostanra a PC már növelve van, de az első eltolásbájt még nem töltődött be MBR-be. Egy mikroutasítással később [4.22. (c) ábra] a műveleti kódra mutató régi PC már OPC-be került. Ez az érték azért szükséges, mert az IJVM eltolása a GOTO utasításra vonatkozik és nem a PC mostani értékére. Valójában ez az oka, hogy az OPC regiszterre szükségünk van.



4.22. ábra. *A helyzet különböző mikroutasítások indulásakor. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4*

A goto2 mikroutasítás elkezdi az eltolás második bájtjának betöltését, és a 4.22. (d) ábrához jutunk goto3 indulásakor. Az első eltolás bájtot balra léptetjük 8 bittel és bemásoljuk H-ba, és elérkezünk goto4-hez és a 4.22. (e) ábrához. Most a balra léptetett első eltolás bájt a H-ban van, a második eltolás bájt az MBR-ben, az alap pedig az OPC-ben. Elkészítjük a 16 bites eltolást a H-ban, majd hozzáadjuk ezt az alaphoz, megkapjuk az új címet, amit a PC-be teszünk goto5-nél. Óvatosan megjegyezzük, hogy az MBR helyett az MBRU-t használjuk goto4-ben, mert nem akarjuk a második bájt előjel-kiterjesztését. A 16 bites eltolást valójában úgy készítettük el, hogy a két felet OR művelettel összekapcsoltuk. Végül be kell tölteni a következő műveleti kódot mielőtt visszaugrunk Main1-re, mert ez az utasítás arra számít, hogy a következő MBR-ben van. goto5-ben ugorhatunk Main1-hez, mert az MPC új értékének kiszámításához idejében megérkezik (a ciklus vége felé) a következő IJVM-utasítás kódja.

A goto IJVM-utasításban használt eltolások előjeles 16 bites értékek, minimum –32 768 és maximum +32 767. Ez azt jelenti, hogy olyan elágazások nem lehetségesek, melyek olyan címkékhez vezetnek, melyek ennél nagyobb távolságra vannak. Ez a tulajdonság tekinthető akár hibának, akár sajátosságnak az IJVM-ben (és a JVM-ben is). A hibatábor azt mondaná, hogy a JVM-nek nem kellett volna korlátozni ezt a programozói stílust. A sajátosságtábor azt mondaná, hogy sok programozó munkája gyökeresen javulna, ha nem lenne lidércnyomásuk a fordítóprogram rettegett üzenetétől:

A program túl nagy és veszélyes. Át kell írnia. A fordításból semmi se lesz.

Sajnos (nézetünk szerint), ez az üzenet akkor fordul elő, amikor a then vagy else ág 32 KB-nál nagyobb, jellegzetesen legalább 50 oldal Javában.

Most tekintsük az IJVM három feltételes elágazó utasítását: IFLT-t, IFEQ-t és IF_ICMPEQ-t. Az első kettő kiveszi a verem tetején lévő szót, és elágazik, ha a szó kisebb, mint nulla, illetve egyenlő nullával. Az IF_ICMPEQ kiveszi a verem tetején lévő két szót, és akkor és csak akkor ágazik el, ha egyenlők. Mindhárom esetben szükséges az új verem-teteje szó beolvasása és TOS-ba tárolása.

A vezérlés hasonló a három utasítás számára: először az operandus vagy operandusok a regiszterekbe kerülnek, azután az új verem-teteje szó beolvasásra kerül TOS-ba, végül a vizsgálat és az elágazás megtörténik. Tekintsük először az IFLT-t. A vizsgálandó szó már TOS-ban van, de mivel IFLT kidob egy szót a veremből, az új verem tetejét be kell olvasni és tárolni TOS-ba. Ez az olvasás iflt1-nél kezdődik. iflt2-nél a vizsgálandó szót OPC-be mentjük átmenetileg, így az új érték rövidesen TOS-ba tehető anélkül, hogy a most érvényeset elvesztenénk. Az iflt3-nál az új verem-teteje szó MDR-ben rendelkezésre áll, kimásoljuk TOS-ba. Végül, iflt4-ben a vizsgálandó, most OPC-ben lévő szót átfuttatjuk az ALU-n anélkül, hogy tárolnánk, de az N bit tárolódik. Ez a mikroutasítás elágazást is tartalmaz T-t választva, ha az N bit igaz, különben F-t.

Ha T-t választotta, a művelet hátralévő része alapvetően ugyanaz, mint a GOTO utasítás kezdeténél, és a sorozatot egyszerűen a GOTO sorozatban folytatjuk goto2-nél. Amennyiben F-t, egy rövid sorozat (F és F2) szükséges, hogy az utasítás maradék részét (az eltolást) átlépjük, azután visszatérünk Main1-re, hogy folytassuk a következő utasítással.

Az ifeq2-n és ifeq3-n lévő kód ugyanazt a logikát követi, csak a Z bitet használjuk az N bit helyett. Mindkét esetben a MAL assemblerén múlik, hogy felismerje, hogy a T és F címek speciálisak, és hogy biztosítsa, hogy a címek olyan helyre kerüljenek a vezérlőtárban, hogy csak a bal szélső bitben különböznek.

Az IF_ICMPEQ logikája nagyjából megegyezik IFEQ-éval, kivéve, hogy itt a második operandust is be kell olvasni. A második operandust H-ba tároljuk if_icmpeq3-mal, ahol az új verem-teteje szó beolvasása kezdődik. Ismét OPC-be mentjük az aktuális verem-teteje szót, és az újat tesszük TOS-ba. Végül a vizsgálat if_icmpeq6-nál hasonló ifeq4-hez.

Most tekintsük az INVOKEVIRTUAL és IRETURN megvalósítását, azon utasításokét, melyek metódus hívásra és metódusból visszatérésre szolgálnak, ahogy a 4.2.3. részben leírtuk. Az INVOKEVIRTUAL egy 23 mikroutasításból álló sorozat, és a megvalósított IJVM-utasítások közül a legösszetettebb. Működését a 4.12. ábrán bemutattuk. Az utasítás arra használja a 16 bites eltolását, hogy meghatározza a meghívandó metódus címét. A mi megvalósításunkban az eltolás egyszerűen egy eltolás a konstans mezőn. Ez a konstans mezőn lévő hely a meghívandó metódusra mutat. Ne felejtsük el azonban, hogy minden metódus első 4 bájtja *nem* utasítás, hanem két 16 bites szám. Az első a paraméterszavak számát adja (beleértve OBJREF-et, lásd 4.12. ábra). A második a lokális változó mező méretét adja szavakban. Ezek a számok a 8 bites porton keresztül töltődnek be, és ugyanúgy állítjuk össze két 16 bites eltolássá, mintha az utasítás részei lettek volna.

A gép előző állapotának visszaállításához szükséges kapcsoló információt – a régi lokális változó mező kezdőcíme és a régi PC – közvetlenül az újonnan létrehozott lokális változó mező felett, és az új verem alatt tároljuk el. Végül a következő utasítás műveleti kódját betöltjük, és a PC-t megnöveljük, mielőtt visszatérünk Main1-hez, hogy elkezdjük a következő utasítást.

Az IRETURN egy egyszerű utasítás, amely nem tartalmaz operandusokat. Egyszerűen azt a címet használja, hogy a kapcsoló információhoz hozzájusson, amelyik a lokális változó mező első szavában van letárolva. Ekkor visszaállítja SP, LV és PC előző értékét, és átmásolja a visszatérési értéket az aktuális verem tetejéről az eredeti verem tetejére, ahogy a 4.13. ábrán mutattuk.

4.4. A mikroarchitektúra szintjének tervezése

Mint majdnem minden más a számítástudományban, a mikroarchitektúra-szint tervezése is tele van kompromisszumokkal. A számítógépeknek számos kívánatos jellemző tulajdonsága van, beleértve a sebességet, árat, megbízhatóságot, könnyű használatot, energiaszükségletet és a fizikai méretet. Azonban kompromisszum vezet a legfontosabb választáshoz, amelyet a CPU-tervezőnek meg kell tennie: sebesség vagy ár. Ebben a fejezetben részletesen megvizsgáljuk ezt a témát, megnézzük, hogy mi mi ellen van, hogyan érhető el a magas teljesítmény, és ennek mi az ára a hardverben és a bonyolultságban.

4.4.1. Sebesség vagy ár

Bár a gyorsabb technológia eredményezte a legnagyobb gyorsulást bármely időszakban, ez túlmutat ezen írás határán. A szervezésnek köszönhető sebességnövelés, bár kevésbé látványos, mint ami a gyorsabb áramköröknek köszönhető, ennek ellenére hatásos. A sebesség sokféleképpen mérhető, de megadva egy áramköri technológiát és egy ISA-t, három alapvető megközelítése van a végrehajtási sebesség növelésének:

1. Csökkenteni egy utasítás végrehajtásához szükséges óraciklusok számát.
2. Lecgyszerűsíteni a felépítést, így az óraciklus rövidebb lehet.
3. Átlapolni az utasítások végrehajtását.

Az első kettő nyilvánvaló, de a tervezési lehetőségek meglepően változatosak, ami drámaian befolyásolhatja akár az óraciklusok számát, akár az óraciklust vagy – leggyakrabban – mindkettőt. Ebben a részben példát adunk arra, hogy egy művelet kódolása és dekódolása hogyan befolyásolhatja az óraciklust.

Egy utasításhalmaz végrehajtásához szükséges óraciklusok számát úgy hívják, hogy **úthossz**. Néha az úthossz speciális hardver hozzáadásával rövidíthető. Például, ha a PC-t ellátjuk egy növelővel (lényegileg egy összeadóval, amelynek az egyik komponense mindig 1), akkor nem kell a továbbiakban használni az ALU-t a PC növelésére, és ezzel ciklusokat küszöbölünk ki. A megfizetett ár a több hardver. Azonban ez a képesség nem segít annyit, mint talán elvárható. A legtöbb utasításnál az elhagyható PC-t növelő ciklusok egyben azok a ciklusok, ahol egy olvasásművelet is végrehajtódik, és a következő utasítás semmiképp se hajtható végre előbb, mert az a memóriából érkező adattól függ.

Az utasításokhoz szükséges ciklusok számának csökkentése a betöltő utasítások esetén többet kíván, mint egy további áramkört, amelyik a PC-t növeli. Annak érdekében, hogy az utasításbetöltést jelentős mértékben gyorsítsuk, a harmadik módszert – az utasítások végrehajtásának átlapolását – kell használnunk. Az a leghatékonyabb, ha elválasztjuk az utasítás betöltéséhez használt áramköröket – a 8 bites memóriaportot és az MBR és PC regisztereket – hogy az egység függetlenül működjön a fő adatút elemeitől. Így a következő műveleti kódot vagy operandust a maga módján töltheti be, a CPU többi részéhez képest akár aszinkron módon előre betöltve egy vagy több utasítást.

Sok utasítás végrehajtásának egyik leginkább időigényes szakasza egy 2 bájtos eltolás betöltése, megfelelő kiterjesztése és összegyűjtése a H regiszterbe, hogy egy összeadásra előkészítsük, példa erre egy ugrás PC ± *n* bájtra. Az a lehetséges megoldás, hogy a memóriaportot 16 bitesre kiszélesítjük, nagyon bonyolulttá teszi a műveletet, mert a memória valójában 32 bit széles. A szükséges 16 bit átnyúlhat szóhatáron, és egyetlenegy 32 bites olvasás nem feltétlenül fogja betölteni mindkét szükséges bájtot.

Az utasítások végrehajtásának átlapolása messze a legérdekesebb, és a legkedvezőbb alkalmat kínálja a sebesség drámai növelésére. Az utasításbetöltés és végrehajtás egyszerű átlapolása meglepően hatékony. Kifinomultabb módszerek azonban sokkal tovább mennek, több utasítás átlapolásos végrehajtásához. Valóban ez az ötlet a modern számítógép-tervezés központi kérdése. Az alábbiakban az átlapolásos utasítás-végrehajtás alapvető módszerei közül fogunk néhányat tárgyalni, és megindokolunk néhány kifinomultabbat.

A sebesség a kép egyik fele: a másik fele az ár. Az ár szintén sokféleképpen mérhető, de az ár pontos definíciója problematikus. Egy mértékegység egyszerűen a komponensek árának összege. Ez különösen igaz volt akkor, amikor a processzorok diszkrét elemekből voltak felépítve, melyeket megvásároltak és összeraktak. Ma az egész processzor egyetlenegy lapkán van, de a nagyobb, összetettebb

lapka sokkal drágább, mint a kisebb, egyszerűbb. Egyedi komponensek – például tranzisztorok, kapuk vagy funkcionális egységek – megszámolhatók, de gyakran a darabszám nem olyan fontos, mint az integrált áramkörön igényelt terület mérete. Minél nagyobb a funkció által igényelt terület, annál nagyobb a lapka. És a lapka gyártási költségei gyorsabban nőnek, mint a területe. Ezért a tervezők gyakran „hasznos terület"-ben – az áramkör által igényelt területben – beszélnek a költségekről (feltételezhetően piko-acre-ben mérik).*

Az egyik legalaposabban tanulmányozott áramkör a történelemben a bináris összeadó. Tervek ezrei születtek, és a leggyorsabbak sokkalta gyorsabbak, mint a leglassabbak. És sokkal bonyolultabbak is. A rendszertervezőknek el kell dönteniük, vajon a nagyobb sebesség megéri-e a hasznos területet.

Az összeadók nem az egyedüli olyan komponensek, amelyeknek sok alternatívája van. Majdnem minden részegység gyorsabb vagy lassabb futásra tervezhető, különböző költségekkel. A tervezők számára az a kihívás, hogy megállapítsák azokat a részegységeket, amellyel a sebesség emelésével a rendszer a legtöbbet javul. Elég érdekes, hogy sok egyedi alkatrész kicserélhető egy gyorsabbra úgy, hogy ez csak kis vagy semmilyen hatással sincs a sebességre. A következő fejezetben néhány tervezési kérdést vizsgálunk majd meg, a hozzá kapcsolódó kompromisszumokkal együtt.

Az óra maximális sebességének meghatározásában az egyik kulcstényező annak a munkának a mennyisége, amelyet el kell végezni egy óraciklus alatt. Nyilvánvaló, hogy minél több az elvégzendő munka, annál hosszabb az óraciklus. Természetesen ez nem ilyen egyszerű, mert a hardver elég jó abban, hogy párhuzamosan végezzen dolgokat, így valójában az egyetlen óraciklus alatt *egymás után* végrehajtandó műveletek sorozata határozza meg, hogy milyen hosszúnak kell lennie egy óraciklusnak.

A befolyásolható dolgok egyike a végrehajtandó dekódolás mennyisége. Elevenítsük fel, például, amit a 4.6. ábrán láttunk, hogy kilenc regiszter bármelyike betölthető ALU-ba a B sínről, mégis csak 4 bitet igényeltünk a mikroutasításban annak meghatározására, hogy melyik regiszter van kiválasztva. Sajnos, ezek a megtakarítások sokba kerülnek. A dekódoló áramkör késleltetése növeli a kritikus utat. Ez azt jelenti, hogy akármelyik regiszter kap is engedélyt arra, hogy az adatait a B sínre tegye, ezt az utasítást kicsit később fogja megkapni, és kicsit később fogja az adatait a B sínre tenni. Ez a hatás tovább terjed, az ALU a bemeneteit kicsit később kapja meg, az eredményt pedig csak kicsit később állítja elő. Végül, a C sínen elérhető eredményt egy kicsit később írja a regiszterekbe. Mivel ez a késedelem gyakran az a tényező, ami meghatározza, hogy az óraciklusnak milyen hosszúnak kell lennie, ez azt jelentheti, hogy az óra nem tud elég gyorsan futni, és így az egész számítógépnek kicsit lassabban kell működnie. Ez kompromisszum a sebesség és az ár között. A vezérlőtár szavanként 5 bittel való csökkentéséért az óra lassulásával fizetünk. A tervezőmérnököknek a tervezési célokat figyelembe kell venniük, amikor eldöntik, hogy mi a helyes választás. Egy nagy teljesítményű gép megvalósításhoz a dekóder használata valószínűleg nem jó ötlet, egy alacsony költségű gépnél azonban jó lehet.

* 1 acre = 0,58 hektár. *(A ford.)*

4.4.2. A végrehajtási út hosszának csökkentése

A Mic-1-et úgy terveztük, hogy közepesen egyszerű és közepesen gyors legyen, bár kétségtelenül hatalmas feszültség van a két célkitűzés között. Röviden, az egyszerű gépek nem gyorsak, és a gyors gépek nem egyszerűek. A Mic-1 CPU is minimális mennyiségű hardvert használ: 10 regisztert, a 3.19. ábrán látható egyszerű ALU-t 32-szer ismételve, egy léptetőt, egy dekódert, egy vezérlőtárat és itt-ott néhány összekötő elemet. Az egész rendszer kevesebb mint 5000 tranzisztorból felépíthető, hozzáadva bármilyen vezérlőtárat (ROM) és főmemóriát (RAM).

Miután láttuk, az IJVM hogyan valósítható meg egyszerűen mikrokódban egy kevés hardverrel, ideje megvizsgálnunk más, gyorsabb megvalósításokat is. A következőkben olyan módszereket vizsgálunk, melyek csökkentik az ISA-utasításonkénti mikroutasítások számát (vagyis csökkentik a végrehajtási út hosszát). Ezután egyéb megközelítéseket is fontolóra veszünk.

Az értelmező ciklus és a mikrokód összefűzése

A Mic-1-nél a főciklus egy mikroutasításból áll, amit minden IJVM-utasítás kezdetén végre kell hajtani. Néhány esetben lehetőség van arra, hogy ez átfedésben legyen az előző utasítással. Valójában ez már részlegesen teljesült. Vegyük észre, hogy amikor a Main1 végrehajtódik, az értelmezendő műveleti kód már MBR-ben van. A műveleti kód azért van ott, mert vagy már az előző főciklus betöltötte (ha az előző utasításnak nem volt operandusa), vagy az előző utasítás végrehajtása alatt töltődött be.

Az utasításkezdet átfedésének ez az elve továbbvihető, és valójában a főciklus bizonyos esetekben teljesen eltüntethető. Ez a következőképpen történhet. Tekintsünk minden olyan mikroutasítás-sorozatot, amelyik a Main1-hez való elágazással fejeződik be. Minden ilyen helyen a főciklus mikroutasítás a sorozat végéhez fűzhető (a következő sorozat kezdete helyett), egy többirányú elágazást másolva most sok helyre (de mindig a célpontok ugyanazon beállításával). Bizonyos esetekben a Main1 mikroutasítás hozzáfűzhető az előző mikroutasításokhoz, mivel az utasítások nem mindig vannak teljesen kihasználva.

A 4.23. ábrán egy dinamikus utasítássorozatot mutatunk be a POP utasításhoz. A főciklus minden utasítás előtt és után szerepel, az ábrán csak a POP utasítás utáni előfordulást mutatjuk be. Vegyük észre, hogy ennek az utasításnak a végrehajtása négy óraciklust igényel: hármat a POP-ra jellemző mikroutasítások, és egyet a főciklus.

Címke	Műveletek	Megjegyzések
pop1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
pop2		Várja, hogy az új TOS bekerüljön a memóriából.
pop3	TOS = MDR; goto Main1	Az új szót TOS-ba másolja.
Main1	PC = PC + 1; fetch; goto (MBR)	MBR-ben a műveleti kód; veszi a következő bájtot; elágazás.

4.23. ábra. *Eredeti mikroprogramrészlet a POP végrehajtására*

A 4.24. ábrán a sorozatot három utasításra csökkentettük azáltal, hogy kihasználtuk azt az óraciklust, amikor az ALU-t nem használjuk, a pop2-t, hogy akkor hajtsuk végre a Main1 főciklus egy részét, hogy megtakarítsunk egy ciklust. Mindenképpen vegyük észre, hogy ezen utasítássorozat vége közvetlenül ágazik el a következő utasításra jellemző kódhoz, így összesen csak három ciklus szükséges. Ez a kis trükk egy ciklussal csökkenti a következő mikroutasítás végrehajtási idejét, így például, a következő IADD négy ciklusról háromra csökken. Ez így egyenértékű azzal, hogy ingyen növeljük a sebességet 250 MHz-ről (4 ns-os mikrontasítások) 333 MHz-re (3 ns-os mikroutasítások).

Címke	Műveletek	Megjegyzések
pop1	MAR = SP = SP – 1; rd	Beolvassa a verem teteje alatti szót.
Main1.pop	PC = PC + 1; fetch	MBR-ben a műveleti kód; betölti a következő bájtot.
pop3	TOS = MDR; goto (MBR)	Az új szó bemásolása TOS-ba; elágazás a művkód szerint.

4.24. ábra. *Javított mikroprogramrészlet a POP végrehajtására*

A POP utasítás különösen jól illeszkedik ehhez a bánásmódhoz, mert van egy meddő ciklusa a közepén, amely nem használja az ALU-t. A főciklus azonban használja az ALU-t. Így, hogy csökkenteni tudjuk eggyel az utasítás hosszát egy utasításon belül, szükséges, hogy találjunk egy ciklust az utasításban, ahol az ALU nincs használatban. Ilyen meddő ciklusok nem sokszor, de előfordulnak, így megéri hozzáfűzni a Main1-t mindegyik mikroutasítás sorozat végéhez. Ez mindössze egy kis vezérlőtárba kerül. Így megvan az első módszerünk az úthossz csökkentésére:

Fűzd az értelmező ciklust mindegyik mikrokód sorozat végéhez.

Egy háromsínes architektúra

Mi egyebet tehetünk a végrehajtási út hosszának csökkentéséért? Egy másik egyszerű javítás, hogy két teljes bemenő sínt biztosítunk az ALU-hoz, egy A sínt és egy B sínt, összesen három sínünk lesz. Minden (vagy legalább több) regiszternek el kellene érnie mindkét bemenő sínt. Ha van két bemenő sínünk, annak az az elő-

Címke	Műveletek	Megjegyzések
iload1	H = LV	MBR tartalmazza az indexet; LV-t H-ba másoljuk.
iload2	MAR = MBRU + H; rd	MAR = a betöltendő lokális változó címe
iload3	MAR = SP = SP + 1	SP a verem új tetejére mutat; az írás előkészítése.
iload4	PC = PC + 1; fetch; wr	PC növelése; következő műveleti kód betöltése; verem tetejének írása.
iload5	TOS = MDR; goto Main1	Frissíti TOS-t.
Main1	PC = PC + 1; fetch; goto (MBR)	MBR-ben a műveleti kód; veszi a következő bájtot, elágazás.

4.25. ábra. *Mic–1 kód ILOAD végrehajtására*

nye, hogy akkor bármely regisztert bármely másik regiszterrel egy cikluson belül összeadhatunk. Hogy lássuk ennek a tulajdonságnak az értékét, tekintsük az ILOAD megvalósítását Mic–1-en, melyet ismét megmutatunk a 4.25. ábrán.

Látjuk, hogy iload1-ben LV-t átmásoljuk H-ba. Az egyetlen ok, hogy H-ba másoljuk, hogy így hozzáadhatjuk MBRU-hoz iload2-ben. A mi eredeti kétsínes tervünkben nincs mód két tetszőleges regiszter összeadására, így a kettő közül az egyiket először H-ba kell másolni. Az új háromsínes tervünkben megtakaríthatunk egy ciklust, ahogy a 4.26. ábra mutatja. Hozzávettük az értelmező ciklust (főciklust) az ILOAD-hoz, de ezzel nem is növeltük, nem is csökkentettük a végrehajtási út hosszát. Az újabb sín mégis hat ciklusról ötre csökkenti az ILOAD teljes végrehajtási idejét. Most megvan a második módszerünk az úthossz csökkentésére:

Térj át a kétsínes tervről a háromsínes tervre.

Címke	Műveletek	Megjegyzések
iload1	MAR = MBRU + LV; rd	MAR = a betöltendő lokális változó címe
iload2	MAR = SP = SP + 1	SP a verem új tetejére mutat; az írás előkészítése.
iload3	PC = PC + 1; fetch; wr	PC növelése; következő műveleti kód betöltése; verem tetejének írása.
iload4	TOS = MDR	Frissíti TOS-t.
iload5	PC = PC + 1; fetch; goto (MBR)	MBR-ben a műveleti kód; veszi a következő bájtot, elágazás.

4.26. ábra. *Háromsínes kód ILOAD végrehajtására*

Egy utasításbetöltő egység

Mindkét módszert megéri használni, de hogy drámai javulást érjünk el, szükségünk van valami sokkal mélyrehatóbbra. Lépjünk vissza, és vizsgáljuk meg minden utasítás közös részét: az utasítás mezőinek betöltését és dekódolását. Vegyük észre, hogy a következő műveletek minden utasításban szerepelhetnek:

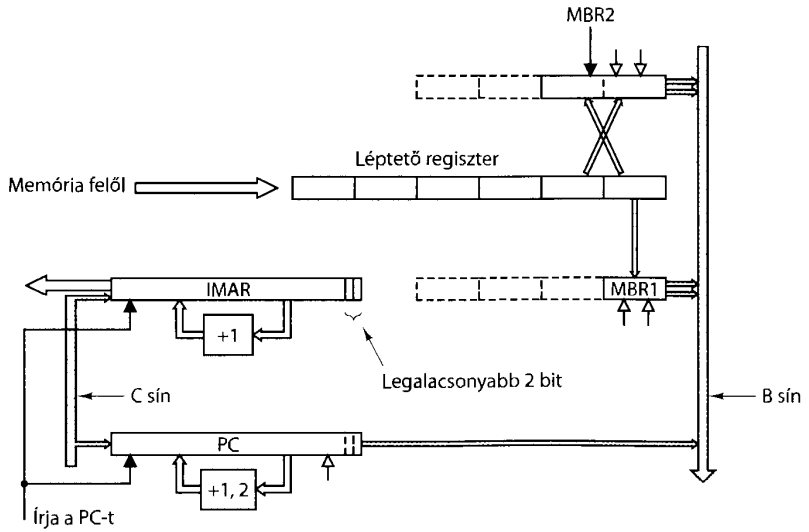
1. A PC-t átküldjük az ALU-n és növeljük.
2. A PC-t felhasználjuk az utasításfolyam következő bájtjának betöltésére.
3. Az operandusokat kiolvassuk a memóriából.
4. Az operandusokat a memóriába írjuk.
5. Az ALU elvégez egy műveletet, és az eredményt visszatároljuk.

Ha egy utasításnak további mezői vannak (operandusok számára), minden mezőt be kell tölteni, egyszerre csak egy bájtot, és össze kell rakni, mielőtt felhasználnánk. Egy mező betöltése és összerakása bájtonként legalább egy ciklusra leköti az ALU-t, hogy megnövelje a PC-t, és azután ismét egyre, hogy összerakja a következő indexet vagy eltolást. Az ALU-t szinte minden ciklusban sokféle műveletre használjuk az utasításbetöltéssel és az utasításon belüli mezők összerakásával kapcsolatban, az utasítás „valódi" munkáján felül.

Annak érdekében, hogy a főciklust átlapoljuk, szükséges, hogy az ALU-t felszabadítsuk a fenti feladatok némelyikétől. Ez megtehető egy második ALU bevezetésével, bár sok tevékenységhez nem szükséges egy teljes ALU. Vegyük észre, hogy néhány esetben az ALU-t egyszerűen egy útnak használjuk, hogy egy értéket átmásoljunk az egyik regiszterből a másikba. Ezek a ciklusok talán kiküszöbölhetők további adatutak bevezetésével, amelyek nem mennek keresztül az ALU-n. Némi haszon származhat például abból, hogy csinálunk egy utat a TOS-ból az MDR-be, vagy az MDR-ből a TOS-ba, hiszen a verem legfelső szavát gyakran másoljuk e két regiszter között.

A Mic–1-ben sok betöltés elválasztható az ALU-tól egy független egység megalkotásával, amely betölti és feldolgozza az utasításokat. Ez az egység, amit **IFU**-nak (**Instruction Fetch Unit, utasításbetöltő egység**) nevezünk, képes függetlenül növelni PC-t, és betölteni a bájtokat a bájtfolyamból, még mielőtt szükség lenne rájuk. Ez az egység csak egy növelőt igényel, egy áramkört, ami sokkal egyszerűbb, mint egy teljes összeadó. Továbbgondolva, az IFU összerakhat 8 és 16 bites operandusokat is, így azok készen állnak azonnali használatra, valahányszor szükséges. Legalább két mód van, ahogy ez végrehajtható:

1. Az IFU ténylegesen értelmezhet minden egyes műveleti kódot: meghatározhatja, hogy hány további mezőt kell betölteni, és összerakhatja azokat egy regiszterbe, hogy készen álljanak, amikor a fő végrehajtási egység használni akarja.
2. Az IFU kihasználhatja az utasítások folyam természetét, és mindig elérhetővé teheti a következő 8 és 16 bites részt, akár szükség van rá, akár nincs. A fő végrehajtási egység bármikor kérheti ezeket, ha szüksége van rá.



4.27. ábra. *Egy betöltő egység Mic–1-hez*

A második terv csíráit a 4.27. ábrán mutatjuk be. Az egyetlen 8 bites MBR helyett most 2 MBR-ünk van: egy 8 bites MBR1 és egy 16 bites MBR2. Az IFU figyeli a fő végrehajtási egység által felhasznált bájtot vagy bájtokat. Éppúgy elérhetővé teszi a következő bájtot az MBR1-ben, mint a Mic–1, az egyetlen eltérés, hogy önműködően érzékeli, amikor az MBR1-t kiolvassák, ekkor előre behozza a következő bájtot, és közvetlenül betölti az MBR1-be. Mint a Mic–1-ben, két kapcsolódási lehetősége van a B sínhez: az MBR1 és MBR1U. A korábbi 32 bites előjel kiterjesztésű, az utóbbi nulla kiterjesztésű.

Hasonlóképpen, az MBR2 működését tekintve ugyanazt nyújtja, de a következő két bájtot tartalmazza. Ennek is két kapcsolódási lehetősége van a B sínhez: az MBR2 a 32 bites előjel-kiterjesztésű, és az MBR2U a nulla kiterjesztésű érték kibocsátásához.

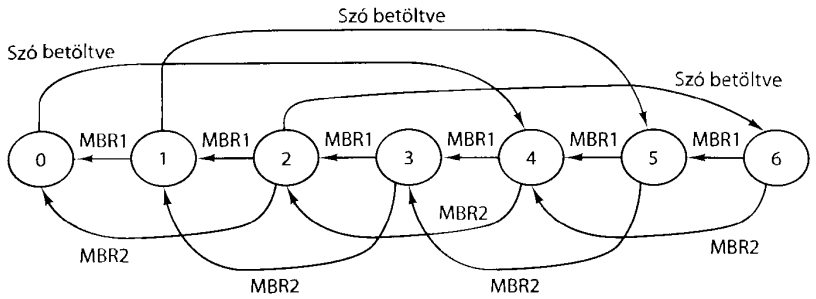
Az IFU végzi a bájtfolyam betöltését. Ezt azzal éri el, hogy a hagyományos 4 bájtos memóriaportot használva előre betölt teljes 4 bájtos szavakat, és az egymást követő bájtokat egy léptető regiszterbe tölti, ami egyesével vagy kettesével szolgáltatja azokat a betöltés sorrendjében. A léptető regiszternek az a szerepe, hogy fenntart egy bájt sort a memóriából, és továbbítja az MBR1-be és az MBR2-be.

Minden időpillanatban az MBR1 a léptető regiszter bájtját tartalmazza és az MBR2 a 2 legkorábbi bájtot (a legkorábbi bájt balra van), hogy egy 16 bites egészet alakítson ki [lásd 4.19. (b) ábra]. Az MBR2-ben lévő két bájt lehet, hogy különböző memóriaszavakból alakul ki, mert az IJVM-utasítások a memóriában nincsenek szóhatárra igazítva.

Valahányszor kiolvasás történik az MBR1-ből, a léptető regiszter jobbra léptet egy bájtot. Amikor pedig az MBR2-ből, akkor jobbra léptet 2 bájtot. Ezután az MBR1 és MBR2 újratöltődik a legkorábbi bájtból, illetve a 2 legkorábbi bájtból. Ha most elegendő hely marad a léptető regiszterben egy másik teljes szó számára, az IFU beolvasó memóriaciklust kezdeményez. Feltételezzük, hogy amikor bármelyik MBR regiszter kiolvasása megtörténik, a következő ciklus kezdetére újból feltöltődik, így egymást követő ciklusokban kiolvasható.

Az IFU tervét egy **FSM**-ről **(Finite State Machine, véges állapotú gép)** mintázhatjuk meg, ahogy azt a 4.28. ábra mutatja. Minden FSM-nek két része van: az **állapotok**, amit a körök jeleznek, és az **átmenetek**, amit az egyik állapotból a másikba menő élek mutatnak. Minden állapot egy olyan lehetséges helyzetet ábrázol, amiben az FSM lehet. Ennek a bizonyos FSM-nek hét állapota van, amelyik megfelel a 4.27. ábrán lévő léptető regiszter hét állapotának. A hét állapot annak felel meg, hogy a léptető regiszterben jelenleg hány bájt van, a szélső értékeket is beleértve, ez egy 0 és 6 közötti szám.

Minden él egy lehetséges eseményt ábrázol. Itt három különböző esemény következhet be. Az első esemény az, amikor egy bájtot olvasunk MBR1-ből. Ez az esemény működésbe hozza a léptető regisztert, és egy bájt kilép a jobb oldali végén, ekkor az állapotot eggyel csökkentve. A második esemény az, amikor két bájtot olvasunk az MBR2-ből, ez az állapotot kettővel csökkenti. Mindkét átmenet kiváltja az az MBR1 és az MBR2 újratöltését. Amikor az FSM a 0., 1. vagy 2. állapotba kerül, akkor egy új szót betöltő memóriaciklus kezdődik (feltéve, hogy a memória már nincs elfoglalva egy szó beolvasásával). A szó megérkezése 4-gyel növeli az állapotot.



Átmenetek
MBR1: Akkor fordul elő, amikor az MBR1-t kiolvassuk
MBR2: Akkor fordul elő, amikor az MBR2-t kiolvassuk
Szó betöltve: Akkor fordul elő, amikor egy memóriaszót olvasunk és 4 bájtot a léptető regiszterbe teszünk

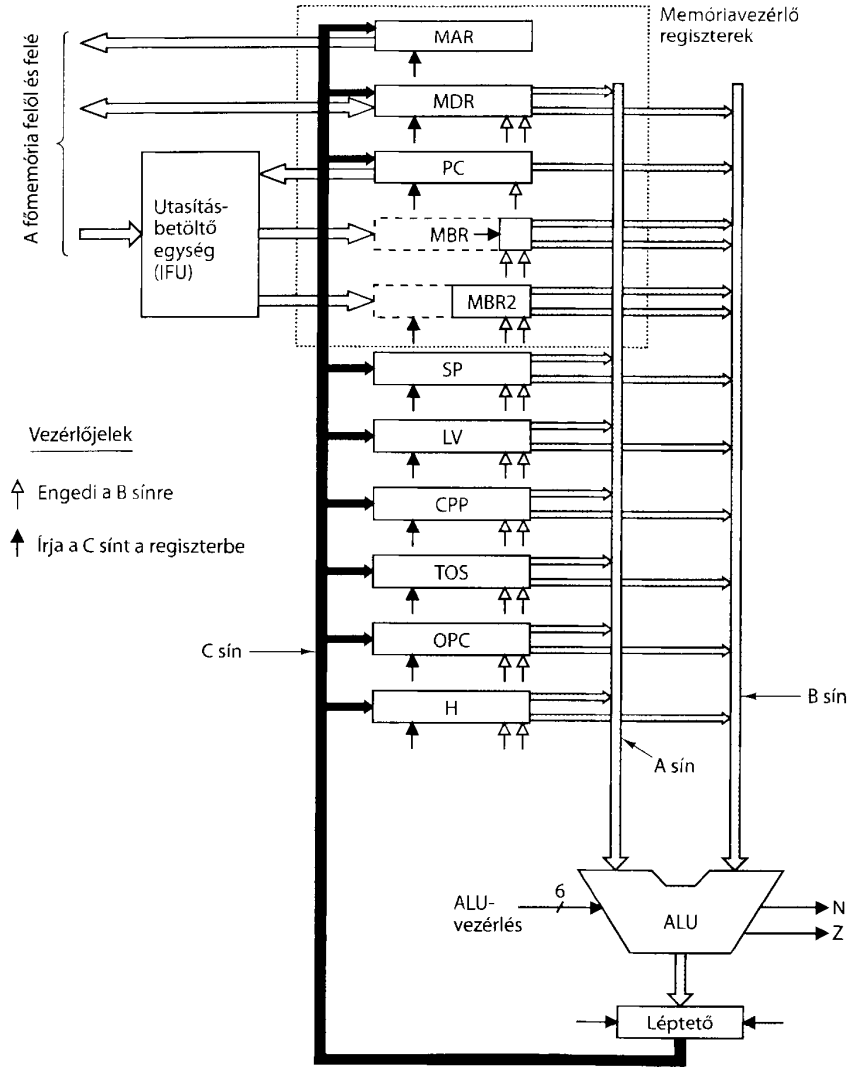
4.28. ábra. *Egy véges állapotú gép az IFU megvalósításához*

Ahhoz, hogy helyesen dolgozzunk, az IFU-nak le kell blokkolnia, amikor arra kérik, hogy tegyen meg valamit, amit nem tud, például szolgáltassa az MBR2 értékét, amikor csak 1 bájt van a léptető regiszterben, és a memória még el van foglalva egy új szó betöltésével. Ezenkívül egyszerre csak egy dolgot tud csinálni, így a beérkező eseményeket sorba kell állítania. Végül, valahányszor a PC megváltozik, IFU-t frissíteni kell. Ilyen részletek IFU-t még bonyolultabbá teszik, mint bemutattuk. Mégis, sok hardvereszközt FSM-ként készítünk el.

Az IFU-nak saját memóriacím-regisztere van, az IMAR, melyet a memória címzésére használunk, amikor egy új szót be kell tölteni. Ennek a regiszternek saját növelője van, így a fő ALU-ra nincsen szükség, amikor a következő szó eléréséhez növeljük IMAR-t. Az IFU-nak úgy kell folyamatosan figyelnie a C sínt, hogy valahányszor a PC feltöltődik, az új PC értéket az IMAR-ba is be kell másolni. Mivel a PC-ben lévő új érték lehet, hogy nincsen szóhatáron, az IFU-nak be kell tölteni a szükséges szót, és hozzá kell igazítania a léptető regisztert ennek megfelelően.

IFU használata esetén a fő végrehajtási egység csak akkor ír a PC-be, amikor meg kell változtatni az utasításfolyam soros jellegét, tehát ír egy sikeres elágazó utasításnál, az INVOKEVIRTUAL-nál és az IRETURN-nél.

Mivel a mikroprogram már nem közvetlenül növeli a PC-t a műveleti kódok betöltésénél, az IFU-nak kell a PC-t karbantartania. Ezt azzal éri el, hogy érzékeli, amikor az utasításfolyamból egy vagy két bájt felhasználásra kerül, vagyis amikor az MBR1-t vagy az MBR2-t (vagy az előjel nélküli verziókat) kiolvassák. A PC-hez kapcsolódik egy önálló növelő, amelyik attól függően képes 1 vagy 2 bájttal növelni, hogy hány bájtot használtunk fel. Így a PC mindig a még fel nem használt első bájt címét tartalmazza. Minden egyes utasítás kezdetén az MBR tartalmazza az ehhez az utasításhoz tartozó műveleti kód címét.



4.29. ábra. *A Mic–2 adatútja*

Megjegyezzük, hogy két független növelő van, és különböző tevékenységeket végeznek. A PC bájtokban számlál, és 1-gyel vagy 2-vel növelődik, az IMAR pedig szavakban, és csak 1-gyel növelődik (4 új bájthoz). A MAR-hoz hasonlóan, az IMAR is részsútosan kerül a címsínre, azaz az IMAR 0. bitje a 2. címvonalhoz kapcsolódik és így tovább, végrehajtva egy magától értetődő átalakítást szócímekről bájtcímekre.

Mint nemsokára részletesen látni fogjuk, nagy nyereség, hogy a főciklusban nem kell a PC-t növelni, mert azokban a mikroutasításokban, melyek a PC-t növelik, gyakran semmi egyéb nem történik a PC növelésén kívül. Ha ez a mikroutasítás kihagyható, csökkenthető a végrehajtási út. Itt a kompromisszum, hogy több hardver kell a gyorsabb géphez. Tehát a harmadik módszerünk az úthossz csökkentésére:

Egy erre specializált működési egységgel töltsd be az utasításokat a memóriából.

4.4.3. Terv előre betöltéssel: a Mic–2

Az IFU nagyon lecsökkentheti az átlagos utasítás úthosszát. Először is, teljesen eltávolítja a főciklust, mivel minden egyes utasítás vége egyszerűen közvetlenül ágazik el a következő utasításra. Másodszor, elkerüli, hogy az ALU-t a PC növelésére kössük le. Harmadszor, csökkenti az úthosszt valahányszor egy 16 bites indexet vagy eltolást számítunk ki, mert összerakja a 16 bites értéket, és ezt, mint 32 bites értéket, közvetlenül szolgáltatja az ALU-nak, elkerülve, hogy a H-ban kelljen összerakni. A 4.29. ábra a Mic–2-t, a Mic–1-nek a 4.29. ábrán látott IFU-val kibővített változatát mutatja. A kibővített gép mikrokódja a 4.30. ábrán látható.

Mic–2 működésének szemléltetéshez vizsgáljuk meg az IADD utasítást. Ez betölti a veremben lévő második szót, és elvégzi az összeadást, mint előbb, csak most nem kell a Main1-re ugrania, miután elkészült, hogy a PC-t növelje és elirányítsa a következő mikroutasításhoz. Amikor az IFU észleli, hogy az iadd3-ban hivatkozunk az MBR1-re, a belső léptető regisztere mindent jobbra tol, és újratölti az MBR1-t és az MBR2-t. Végrehajt egy átmenetet is a jelenlegi állapotából az eggyel alacsonyabba. Ha az új állapot 2, az IFU elkezd betölteni egy szót a memóriából. Mindez hardverben történik. A mikroprogramnak semmit se kell tennie. Ezért csökkenthető az IADD négy mikroutasításról háromra.

A Mic–2 néhány utasítást jobban felgyorsít, mint másokat. Az LDC_W kilenc mikroutasításról csupán háromra változik, harmadolva a végrehajtási időt. Másrészről, a SWAP csak nyolcról hat mikroutasításra változik. Átlagos teljesítménynél valójában a gyakoribb utasításoknál kapott nyereség számít. Ezek közé tartozik az ILOAD (6 volt, most 3), az IADD (4 volt, most 3) és az IF_ICMPEQ (sikeres esetben 13 volt, most 10; sikertelen esetben 10 volt, most 8). Ahhoz, hogy megmérjük a gyorsítást, választanunk és futtatnunk kellene egy tesztágyat, de világos, hogy itt nagy a nyereség.

Címke	Műveletek	Megjegyzések
nop1	goto (MBR)	Elágazás a következő utasításhoz.
iadd1	MAR = SP = SP + 1; rd	Beolvassa a verem teteje alatti szót.
iadd2	H = TOS	H = verem teteje
iadd3	MDR = TOS = MDR + H; wr; goto (MBR1)	Összeadja a felső két szót; beírja a verem új tetejére.
isub1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
isub2	H = TOS	H = verem teteje
isub3	MDR = TOS = MDR - H; wr; goto (MBR1)	Kivonja TOS-t a betöltött TOS-1-ből.
iand1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
iand2	H = TOS	H = verem teteje
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	TOS AND betöltött TOS-1.
ior1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
ior2	H = TOS	H = verem teteje
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	TOS OR betoltott TOS-1
dup1	MAR = SP = SP + 1	Növeli SP-t és MAR-ba másolja.
dup2	MDR = TOS; wr; goto (MBR1)	Beírja az új verem szót.
pop1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
pop2		Olvasásra vár.
pop3	TOS = MDR; goto (MBR1)	Az új szót a TOS-ba másolja.
swap1	MAR = SP - 1; rd	Legyen MAR SP-1; beolvassa a verem 2. szavát.
swap2	MAR = SP	Előkészíti írásra az új 2. szót.
swap3	H = MDR; wr	Az új TOS-t elmenti; a 2. szót a verembe írja.
swap4	MDR = TOS	A régi TOS-t MDR-be másolja.
swap5	MAR = SP - 1; wr	A régi TOS-t a verem 2. helyére írja.
swap6	TOS = H; goto (MBR1)	Frissíti TOS-t.
bipush1	SP = MAR = SP + 1	Beállítja MAR-t a verem új tetejének írására.
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Frissíti a vermet TOS-ban és a memóriában.
iload1	MAR = LV + MBR1U; rd	MAR legyen LV + index; operandus beolvasása.
iload2	MAR = SP = SP + 1	SP növelése; MAR legyen az új SP.
iload3	TOS = MDR; wr; goto (MBR1)	Frissíti a vermet TOS-ban és a memóriában.
istore1	MAR = LV + MBR1U	MAR legyen LV + index.
istore2	MDR = TOS; wr	TOS másolása; szó írása.
istore3	MAR = SP = SP - 1; rd	SP csökkentése; az új TOS beolvasása.
istore4		Olvasásra vár.
istore5	TOS = MDR; goto (MBR1)	Frissíti TOS-t.
wide1	goto (MBR1 OR 0x100)	A következő cím: művkód OR 0x100.
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Azonos iload1-gyel, de 2 bájtos indexet használ
wide_istore1	MAR = LV + MBR2U; goto istore2	Azonos istore1-gyel, de 2 bájtos indexet használ.
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Olyan, mint wide3_iload1, de CPP-t indexeli.
iinc1	MAR = LV + MBR1U; rd	MAR legyen LV + index az olvasáshoz.
iinc2	H = MBR1	H legyen a konstans.
iinc3	MDR = MDR + H; wr; goto (MBR1)	Noveli a konstanssal és frissíti.
goto1	H = PC - 1	PC-t H-ba másolja.
goto2	PC = H + MBR2	Eltolást hozzáadja és frissíti PC-t.
goto3		Várnia kell, hogy IFU frissüljon.
goto4	goto (MBR1)	Elágazás az új utasításra.
iflt1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
iflt2	OPC = TOS	TOS-t ideiglenesen OPC-be menti.

4.30. ábra. *A Mic–2 mikroprogramja*

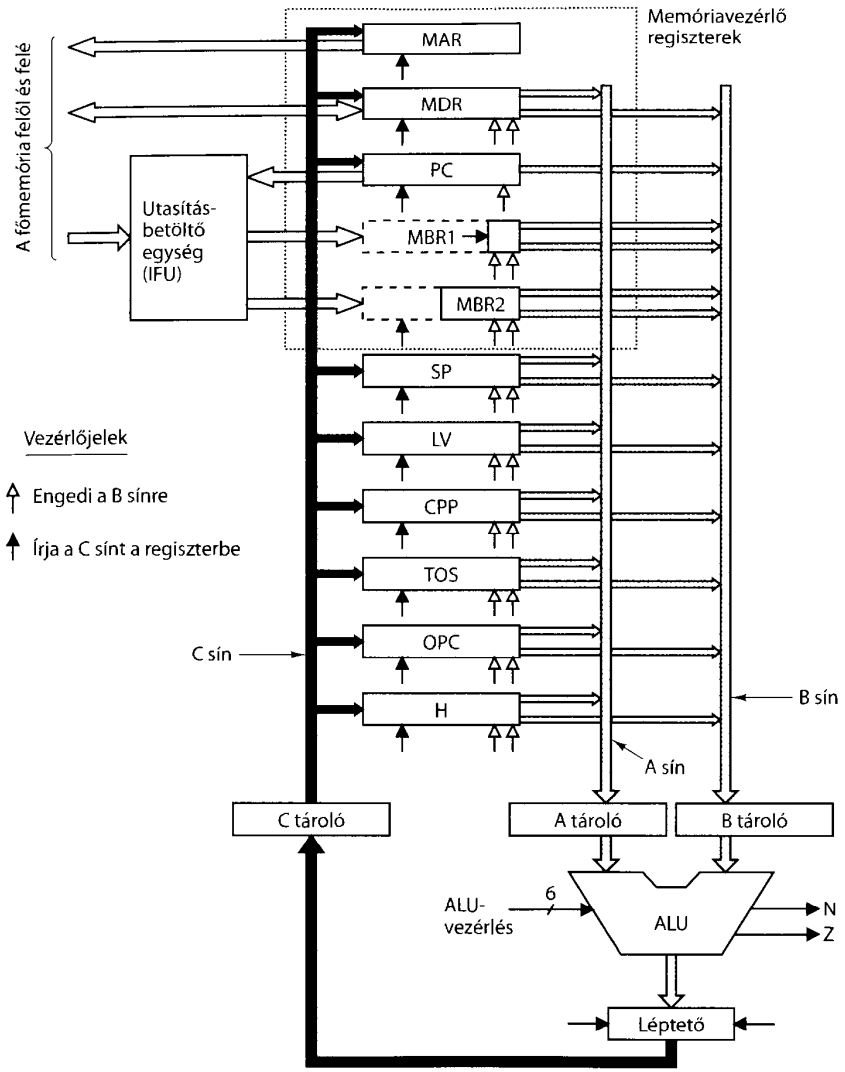
Címke	Műveletek	Megjegyzések
iflt3	TOS = MDR	A verem új tetejét TOS-ba teszi.
iflt4	N = OPC; if (N) goto T; else goto F	Elágazás N bit szerint.
ifeq1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
ifeq2	OPC = TOS	TOS-t ideiglenesen OPC-be menti.
ifeq3	TOS = MDR	A verem új tetejét TOS-ba teszi.
ifeq4	Z = OPC; if (Z) goto T; else goto F	Elágazás Z bit szerint.
if_icmpeq1	MAR = SP = SP - 1; rd	Beolvassa a verem teteje alatti szót.
if_icmpeq2	MAR = SP = SP - 1	Beállítja MAR-t a verem új tetejének olvasásához.
if_icmpeq3	H = MDR; rd	A verem második szavának másolása H-ba.
if_icmpeq4	OPC = TOS	TOS-t ideiglenesen OPC-be menti.
if_icmpeq5	TOS = MDR	A verem új tetejét TOS-ba teszi.
If_icmpeq6	Z = H - OPC; if (Z) goto T; else goto F	Ha a felső 2 szó egyenlő, T-re, különben F-re.
T	H = PC - 1; goto goto2	Ugyanaz, mint goto1.
F	H = MBR2	Beolvassa MBR2-t, és eldobja.
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	A metódus mutatójának címét MAR-ba teszi.
invokevirtual2	OPC = PC	Menti a visszatérő címet OPC-be.
invokevirtual3	PC = MDR	PC mutasson a metódus kódjának első bájtjára.
Invokevirtual4		Várnia kell, hogy IFU frissüljön
invokevirtual5	TOS = SP - MBR2U	TOS = OBJREF címe - 1
Invokevirtual6	TOS = MAR = H = TOS + 1	TOS = OBJREF címe
Invokevirtual7	MDR = SP + MBR2U + 1; wr	Felulírja OBJREF-t a kapcsoló mutatóval.
Invokevirtual8	MAR = SP = MDR	Legyen SP, MAR az a hely, ahol a régi PC-t tartjuk.
invokevirtual9	MDR = OPC; wr	Előkészíti a régi PC mentését.
Invokevirtual10	MAR = SP = SP + 1	A novelt SP oda mutat, ahol a régi LV-t tartjuk.
invokevirtual11	MDR = LV; wr	A régi LV tárolása.
invokevirtual12	LV = TOS;	LV mutasson a nulladik paraméterre.
invokevirtual13	TOS = MDR; goto (MBR1)	TOS = a hívó LV-je
ireturn1	MAR = SP = LV; rd	SP, MAR visszaállítása a kapcsoló mutató olvasásához.
ireturn2		Várakozás a kapcsoló mutatóra.
ireturn3	LV = MAR = MDR; rd	LV, MAR legyen a kapcsoló mutató; a régi PC olvasása.
ireturn4	MAR = LV + 1	MAR mutasson a régi LV re.
ireturn5	PC = MDR; rd	PC helyreállítása; régi LV olvasása.
ireturn6	MAR = SP	A verem új tetejere írás előkészítése.
ireturn7	LV = MDR	LV helyreállítása.
ireturn8	MDR = TOS; wr; goto (MBR1)	Visszatérési érték tárolása a verem eredeti tetején.

4.30. ábra. *A Mic–2 mikroprogramja (folytatás)*

4.4.4. Csővonalas terv: a Mic–3

Világos, hogy a Mic–2 a Mic–1 továbbfejlesztése. Gyorsabb, és kevesebb vezérlőtárat használ, bár az IFU költsége vitathatatlanul nagyobb, mint amit a kisebb vezérlőtárral nyerünk. Tehát jelentékeny mértékben gyorsabb gépet kaptunk valamivel magasabb áron. Nézzük meg, vajon tehetjük-e még gyorsabbá.

Mi lenne, ha megpróbálnánk csökkenteni a ciklusidőt? A ciklusidőt jelentős mértékben a lapkatechnológia határozza meg. Minél kisebbek a tranzisztorok, és minél kisebb a közöttük lévő fizikai távolság, annál gyorsabban tud az óra futni. Egy adott technológia esetén egy teljes adatútművelet végrehajtásához szükséges



4.31. ábra. *A háromsínes adatút, amelyet a Mic–3 használ*

idő állandó (legalábbis a mi szemszögünkből). Mindamellett van némi szabadságunk, és nemsokára ezt a legteljesebb mértékben ki is fogjuk használni.

Másik választásunk az, hogy több párhuzamosságot viszünk a gépbe. Jelen pillanatban a Mic–2 erősen szekvenciális. A regisztereket a sínekre teszi, vár az ALU-ra és a léptetőre, hogy feldolgozza, azután visszaírja az eredményeket a regiszterekbe. Az IFU kivételével nem sok párhuzamosság jelenik meg. A párhuzamosság alkalmazása komoly lehetőség.

Mint előbb említettük, az óraciklust behatárolja az az idő, ami a jel tovaterjedéséhez szükséges az adatúton. A 4.3. ábra a különböző komponensek miatt fellépő késleltetések láncolatát mutatja be az adatútciklus alatt. Három fő részből áll az adatútciklus ideje:

1. A kiválasztott regiszterek A és B sínekre vezetéséhez szükséges idő.
2. Az ALU és a léptető regiszter munkájához szükséges idő.
3. Az eredmények regiszterekhez vezetéséhez és tárolásához szükséges idő.

A 4.31. ábrán egy új, az IFU-t is tartalmazó háromsínes architektúrát mutatunk be, három további tárolóval (regiszterrel), melyeket egy-egy sín közepére illesztünk. A tárolókat minden egyes ciklusban írjuk. Végeredményben a regiszterek szétválasztják az adatutat különböző részekre, amelyek most egymástól függetlenül működhetnek. Erre **Mic–3**-ként vagy **csővonalas** modellként fogunk hivatkozni.

Hogyan segíthetnek ezek az új regiszterek? Most három óraciklus kell az adatút használatához: egy az A és B tárolók betöltéséhez, egy az ALU és léptető működéséhez és a C tároló feltöltéséhez, és egy a C tároló visszatöltéséhez a regiszterekbe. Őrültek vagyunk? (*Tipp:* nem.) A tárolók beillesztésének értelme kétszeres:

1. Gyorsíthatjuk az órát, mert a maximális késleltetés most rövidebb.
2. Minden ciklus alatt az adatút minden részét használhatjuk.

Az adatút három részre osztásával a maximális késleltetés csökken annak eredményeképpen, hogy az órafrekvencia növekedhet. Tegyük fel, hogy az adatút három részre osztásával mindegyik körülbelül harmad olyan hosszú, mint az eredeti, így az óra sebességét megháromszorozhatjuk. (Ez nem teljesen valószerű, mert két másik regisztert is hozzá kell venni az adatúthoz, de első közelítésnek működni fog.)

Mivel feltételeztük, hogy minden memóriaolvasás és -írás kielégíthető az 1. szintű gyorsítótárból, és ez a gyorsítótár ugyanabból az anyagból készül, mint a regiszterek, továbbra is azt tételezzük fel, hogy egy memóriaművelet egy ciklust vesz igénybe. Ezt a gyakorlatban, sajnos, lehet, hogy nem olyan könnyű elérni.

A második szempont többet foglalkozik az átmenő teljesítménnyel, mint egy egyedi utasítás sebességével. A Mic–2-ben minden óraciklus első és harmadik részében, az ALU üresen jár. Azzal, hogy az adatutat három részre osztottuk, képesek vagyunk az ALU-t minden ciklusban használni, háromszor annyi munkát kihozva a gépből.

Most nézzük, hogy a Mic–3 adatútja hogyan dolgozik. Mielőtt hozzákezdünk, egy jelölésre van szükségünk a tárolókkal kapcsolatban. Nyilvánvaló, hogy a tárolókat A, B és C betűkkel jelöljük, és úgy bánunk velük, mint a regiszterekkel, nem feledkezve meg az adatút megszorításairól. A 4.32. ábrán látható példa a SWAP utasítást megvalósító kódsorozatot mutatja Mie–2-re.

Címke	Műveletek	Megjegyzések
swap1	MAR = SP – 1; rd	Legyen MAR SP-1; beolvassa a verem 2. szavát.
swap2	MAR = SP	Előkészíti írásra az új 2. szót.
swap3	H = MDR; wr	Az új TOS-t elmenti; a 2. szót a verembe írja.
swap4	MDR = TOS	A régi TOS-t MDR-be másolja.
swap5	MAR = SP – 1; wr	A régi TOS-t a verem 2. helyére írja.
swap6	TOS = H; goto (MBR1)	Frissíti TOS-t.

4.32. ábra. *A SWAP Mic–2 kódja*

Most valósítsuk meg ezt a sorozatot a Mic–3-on is. Ne felejtsük el, hogy az adatút most három ciklust igényel működéséhez: egyet az A és B betöltéséhez, egyet a művelet elvégzéséhez és a C feltöltéséhez, és egyet az eredmény a regiszterekbe írására. Minden egyes ilyen darabot **mikrolépés**nek hívunk.

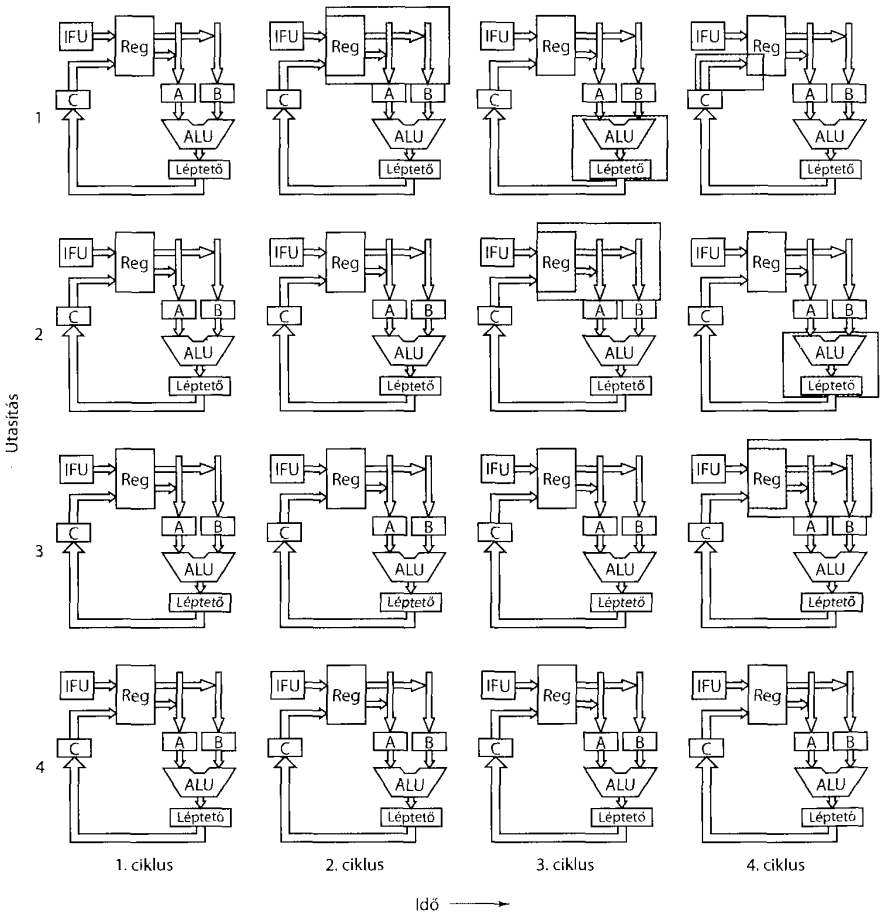
A 4.33. ábra a SWAP megvalósítását mutatja Mic–3-ra. Az 1. ciklusban, a swap1-nél azzal kezdünk, hogy átmásoljuk az SP-t a B-be. Nem érdekes, hogy mi kerül az A-ba, mert ahhoz, hogy a B-ből kivonjunk 1-t, az ENA negálva van (lásd 4.2. ábra). Az egyszerűség kedvéért, nem fogunk olyan értékadásokat mutatni, amelyeket nem használunk. A 2. ciklusban végrehajtjuk a kivonást. A 3. ciklusban az eredményt a MAR-ba tároljuk, és az olvasásművelet megkezdődik a 3. ciklus végén (a MAR-ba való tárolás után). Mivel a memóriaolvasás egy ciklust igényel, ezért ez most nem lesz kész csak a 4. ciklus végén, erre utalunk, amikor egy MDR-be történő értékadást mutatunk a 4. ciklusban. Az MDR-ből az érték nem olvasható ki az 5. ciklus előtt.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cy	**MAR = SP – 1; rd**	**MAR = SP**	**H = MDR; wr**	**MDR = TOS**	**MAR = SP – 1; wr**	**TOS = H; goto (MBR1)**
1	B = SP					
2	C = B – 1	B = SP				
3	MAR = C; rd	C = B				
4	MDR = Mem	MAR = C				
5			B = MDR			
6			C = B	B = TOS		
7			H = C; wr	C = B	B = SP	
8			Mem = MDR	MDR = C	C = B – 1	B = H
9					MAR = C; wr	C = B
10					Mem = MDR	TOS = C
11						goto (MBR1)

4.33. ábra. *A SWAP megvalósítása a Mic–3-on*

Most térjünk vissza a 2. ciklusra. A swap2-t most megkezdhetjük mikrolépésekre darabolni, és el is indíthatjuk azokat. A 2. ciklusban átmásolhatjuk az SP-t a B-be, azután átfuttathatjuk az ALU-n a 3. ciklusban, és végül tároljuk a MAR-ba a 4. ciklusban. Eddig rendben van. Érthető, ha tovább – minden ciklusban egy új mikroutasítást indítva – dolgozunk ezzel a sebességgel, akkor megháromszorozzuk a gép sebességét. Ez a növekedés abból a tényből fakad, hogy minden óraciklusban kiadhatunk egy új mikroutasítást, és a Mic–3-nak háromszor annyi óraciklusa van másodpercenként, mint amennyi a Mic–2-nek. Valójában egy csővezetékes CPU-t építettünk.

Sajnos, nehézségbe ütköztünk a 3. ciklusban. Szeretnénk elkezdeni a munkát a swap3-mal, de az első elvégzendő dolog az MDR átfuttatása az ALU-n, az MDR pedig nem töltődik fel a memóriából az 5. ciklus kezdetéig. Azt a helyzetet, ahol egy mikrolépés nem tud indulni, mert egy eredményre vár, amit egy előző mikrolépés még nem hozott létre, **valódi függőség**nek vagy **RAW függőség**nek nevezzük. A függőségekre gyakran **akadályok**ként hivatkozunk. RAW a Read After Write (írás után olvasás) rövidítése, és azt jelzi, hogy egy mikrolépés olyan értéket akar olvasni egy regiszterből, amit még nem írtunk bele. Itt csak egyetlen ésszerű dolgot



4.34. ábra. *Egy csővezeték munkájának grafikus illusztrációja*

tehetünk, késleltetjük a swap3 indulását, amíg az 5. ciklusban az MDR felhasználhatóvá válik. **Elakadás**nak nevezzük azt, amikor megállunk, hogy egy szükséges értéket megvárjunk. Ezek után folytathatjuk a mikroutasítások indítását minden ciklusban, hiszen több függőség nincs, bár a swap6 csak éppen hogy teljesíti ezt, mert olvassa a H-t az azt követő ciklusban, hogy a swap3-ban beleírtunk. Ha a swap5 próbálta volna olvasni a H-t, egy ciklusra elakadt volna.

Habár a Mic-3 program több ciklust igényel, mint a Mic–2, mégis gyorsabban fut. Ha a Mic–3 ciklusidő ΔT ns, akkor a Mic–3 11ΔT ns-ot igényel a SWAP végrehajtásához. Ellenben Mic–2 6 ciklust igényel, melyek mindegyike 3ΔT, összesen 18ΔT. A csővezeték gyorsabbá teszi a gépet, még akkor is, ha egyszer le kellett állni, hogy elkerüljünk egy függőséget.

A csővezeték kulcsmódszer minden korszerű CPU-ban, így nagyon fontos, hogy jól megértsük. A 4.34. ábrán a 4.31. ábra adatútját látjuk szemléletesen ábrázolva, mint egy csővezetéket. Az első oszlop azt fejezi ki, hogy mi történik az 1. ciklus alatt, a második oszlop a 2. ciklust mutatja és így tovább (feltételezve, hogy nincs elakadás). Az árnyékolt terület az 1. utasítás 1. ciklusában azt jelzi, hogy az IFU el van foglalva az 1. utasítás betöltésével. Egy órajellel később, a 2. ciklus alatt az 1. utasítás által igényelt regiszterek az A és B tárolókba töltődnek, és eközben, az IFU a 2. utasítás betöltésével foglalkozik, ezt jelzi a két árnyékolt téglalap a 2. ciklusban.

A 3. ciklus alatt az 1. utasítás az ALU-t és a léptetőt használja, hogy elvégezze saját műveletét, az A és B tároló a 2. utasítás számára töltődik fel, és a 3. utasítás betöltődik. Végül, a 4. ciklus alatt négy utasítás dolgozik egy időben. Az 1. utasítás eredményei tárolódnak, az ALU elvégzi a munkát a 2. utasítás számára, az A és B tároló feltöltődik a 3. utasítás számára, és a 4. utasítás betöltődik.

Ha az 5. és a következő ciklusokat megmutattuk volna, a séma ugyanaz lenne, mint a 4. ciklusban: az adatút mind a négy része egymástól függetlenül dolgozna. Ez a terv egy 4 szakaszos csővezetéket mutat, melynek szakaszai: utasításbetöltés, operanduselérés, ALU-műveletek és regiszterekbe írás. Ez hasonló a 2.4. (a) ábra csővezetékéhez, attól eltekintve, hogy nincs dekódoló szakasz. Egy fontos részletet itt kiemelünk: bár egy egyedülálló utasítás négy óraciklust igényel a végrehajtáshoz, minden egyes óraciklusban egy új utasítás kezdődik, egy régi pedig befejeződik.

A 4.34. ábrát úgy is nézhetjük, hogy minden egyes utasítást vízszintesen végigkövetünk az egész oldalon keresztül. Az 1. utasításnál az 1. ciklusban az IFU dolgozik rajta. A 2. ciklusban a regisztereit az A és B sínekre tesszük. A 3. ciklusban az ALU és a léptető dolgozik neki. Végül, a 4. ciklusban az eredményeit visszatároljuk a regiszterekbe. Egy dolgot jegyezzünk meg: négy hardverszelet érhető el, és minden egyes ciklus alatt egy adott utasítás csak az egyiküket használja, szabadon hagyva a többi szeletet más utasítások számára.

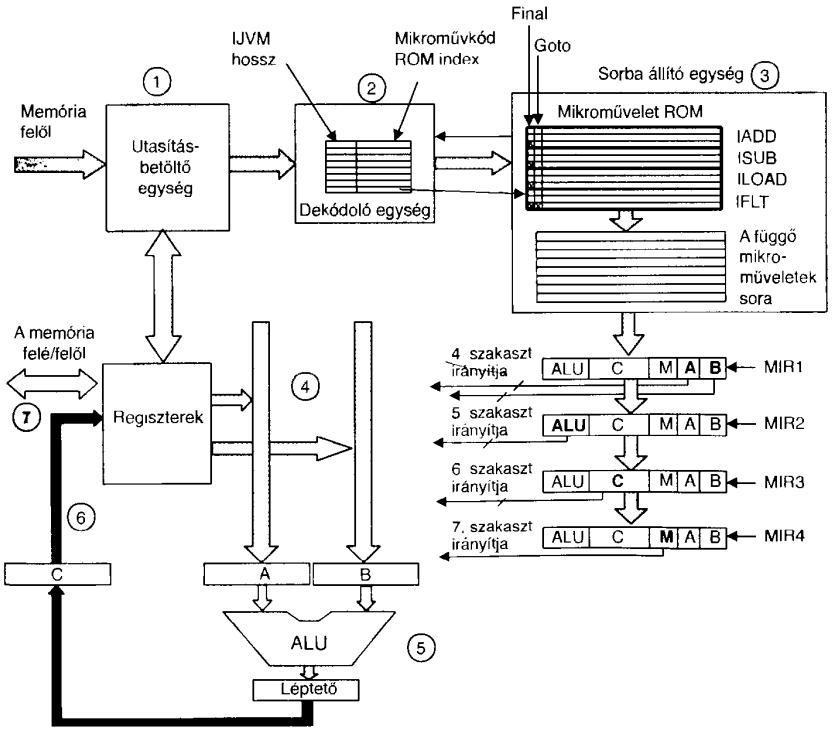
Hasznos analógia csővezetékes tervünkhöz egy gépkocsi-összeszerelő gyár szerelőszalagja. E modell lényegének megragadásához képzeljük el, hogy minden percben megütnek egy nagy gongot, és ekkor minden autó egy hellyel tovább mozog a szalagon. Minden egyes helyen a munkások elvégeznek néhány műveletet az előttük álló autón, például hozzáadják a kormánykereket vagy beszerelik a féket. Minden egyes gongütésre (1 ciklus) egy új autót helyeznek a szerelőszalag elejére, a végén pedig egy kész autó gördül le. Így, bár egy autó elkészítése ciklusok

százait igényelheti, minden egyes ciklusban elkészül egy autó. A gyár képes percenként egy autót gyártani függetlenül attól, hogy egy autó összeszerelése ténylegesen mennyi időt vesz igénybe. Ez a csővezeték ereje, és a CPU-kban ugyanolyan jól alkalmazzák, mint az autógyárakban.

4.4.5. Hétszakaszú csővezeték: a Mic–4

Azt a tényt, hogy minden egyes mikroutasítás kiválasztja a követőjét, már elmagyaráztuk. A legtöbbjük éppen a következőt választja ki a folyamatban lévő sorozatban, de az utolsó, mint amilyen a swap6, általában egy többutas elágazást csinál, ami eldugítja a csővezetéket, mert utána lehetetlen folytatni az előre betöltést. A kérdés kezelésének jobb módszerére van szükségünk.

A következő (és egyben utolsó) mikroarchitektúránk a Mic–4. Fő alkotóelemeit a 4.35. ábrán mutatjuk be, tekintélyes mennyiségű részletet azonban az érthetőség kedvéért elrejtettünk. Mic–3-hoz hasonlóan, van egy IFU, amelyik előre betölt szavakat a memóriából, és kezeli a különböző MBR-eket.



4.35. ábra. *A Mic–4 fő alkotóelemei*

Az IFU a bejövő bájtfolyamot egy új alkatrész, a **dekódoló egység** számára adagolja. Ennek az egységnek van egy belső ROM-ja, amit az IJVM műveleti kódjával indexelünk. Minden egyes bejegyzés (sor) két részből áll: ennek az IJVM-utasításnak a hossza és egy index egy másik ROM-ba, a mikroműveletek ROM-jába. Az IJVM-utasítás hossza azért kell, hogy a dekódoló egység felismerje a bejövő bájtfolyamban az utasításokat, így mindig tudja, melyek a műveleti kód bájtjai és melyek az operandusok. Ha egy utasítás hossza éppen 1 bájt (például POP), a dekódoló egység tudja, hogy a következő bájt egy műveleti kód. Ha azonban a jelenlegi utasítás hossza 2 bájt, a dekódoló egység tudni fogja, hogy a következő bájt egy operandus, melyet közvetlenül egy másik műveleti kód követ. Amikor a WIDE előtag feltűnik, a következő bájtot egy speciális széles műveleti kódra alakítja át, például WIDE + ILOAD átalakul WIDE_ILOAD-dá.

A dekódoló egység elküldi a mikroműveletek ROM-jához a saját táblázatában talált indexet a következő rész, a **sorba állító egység** számára. Ez az egység némi logikát és két belső táblázatot tartalmaz, egyet ROM-ban és egyet RAM-ban. A ROM azt a mikroprogramot tartalmazza, ahol minden IJVM-utasításnak néhány egymást követő **mikroműveletnek** nevezett bejegyzése van. A bejegyzéseknek sorrendben kell elhelyezkedniük, így olyan trükk nincs megengedve, mint a Mic–2-ben wide_iload2 ugrása iload2-re. Minden egyes IJVM-sorozatot pontosan és teljes terjedelmében el kell helyezni, bizonyos sorozatokat többször is.

A mikroműveletek hasonlítanak a 4.5. ábra mikroutasításaihoz, a NEXT_ADDRESS és JAM mezők azonban hiányoznak, és egy új kódoló mező szükséges az A sín bemenet előírásához. Két új bit is kell: a Final és a Goto. A Final bit be van állítva minden egyes IJVM-mikroművelet sorozatának utolsó mikroműveleténél, hogy jelezze a sorozat végét. A Goto bit a feltételes mikroelágazások megjelölésére szolgál. Ezeknek a közönséges mikroművelettől különböző formája van, amely tartalmazza a JAM biteket és a mikroművelet ROM-jába mutató indexet. Azokat a mikroutasításokat, amelyek előzőleg valamit csináltak az adatúttal, és végrehajtottak egy feltételes mikroelágazást is (például iflt4), most két mikroműveletre kell szétdarabolnunk.

A sorba állító egység a következőképpen működik. Átvesz egy indexet a mikroművelet ROM-hoz a dekódoló egységtől. Ez alapján kikeresi a mikroműveletet, és bemásolja egy belső sorba. Majd a következő mikroműveletet is bemásolja a sorba, és ezután a következőt is. Ezt addig folytatja, amíg talál egy olyat, ahol a Final bit 1. Ezt is átmásolja, és megáll. Ha nem talált olyan mikroműveletet, amelyben a Goto bit 1, és maradt még elegendő hely a sorban, a sorba állító egység egy nyugtázó jelet küld vissza a dekódoló egységnek. Miután a dekódoló egység fogadta a nyugtázást, a következő IJVM-utasítás indexét elküldi a sorba állító egységnek.

Ilyen módon a memóriában lévő IJVM-utasítások sorozata alapvetően átalakul mikroműveletek sorozatára egy sorban. Ezek a mikroműveletek táplálják a MIR-eket, amelyek jeleket küldenek az adatút vezérléséhez. De van itt egy másik tényező is, amit figyelembe kell vennünk: a mezők az egyes mikroműveletben nem ugyanabban az időben hatnak. Az A és B mezők az első, az ALU mező a második, a C mező pedig a harmadik ciklus alatt vezérel, és valamennyi memóriaművelet a negyedik ciklusban kerül sorra.

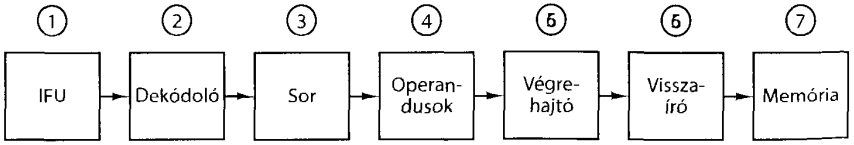
Négy külön MIR-t vezetünk be, hogy teljessé tegyük ezt a munkát (lásd 4.35. ábra). Minden egyes óraciklus kezdetekor (a 4.3. ábrán a Δw idő) a MIR3 átmásolódik MIR4-be, MIR2 MIR3-ba, MIR1 MIR2-be, és a MIR1 egy friss mikroműveletttel töltődik fel a mikroműveletek sorából. Ezután minden egyes MIR kiadja a saját vezérlőjeleit, de ezek közül csak néhányat használunk. A MIR1-ből az A és B mezőket arra használjuk, hogy kiválasszuk azokat a regisztereket, amelyeket az A és B tárolókhoz vezetünk, de az ALU mezőt a MIR1-ben nem használjuk, és nincs semmi máshoz hozzákapcsolva az adatútban.

Egy óraciklussal később ez a mikroművelet átmegy a MIR2-be, és azok a regiszterek, melyeket kiválasztott, most biztonságosan az A és B tárolókban vannak; várva a kalandok eljövetelét. Az ALU mezőjét most arra használjuk, hogy az ALU-t vezéreljük. A következő ciklusban a C mező fogja az eredmények regiszterekbe írását vezérelni. Azután átmegy a MIR4-be, és elindít egy memóriaműveletet, ha szükséges, felhasználva a most betöltött a MAR-t (és íráshoz az MDR-t).

A Mic–4 utolsó szempontja még némi megbeszélést igényel: ezek a mikroelágazások. Néhány IJVM-utasítás, mint az IFLT, megkívánja, hogy a feltételes elágazás, mondjuk, az N biten alapuljon. Amikor egy mikroelágazás sikeres (megtörténik), a csővezeték nem folytatódhat. Hogy ezt kezelni tudjuk, hozzávettük a Goto bitet a mikroműveletthez. Amikor a sorba állító egység a mikroműveletek átmásolása közben talál egy olyan mikroműveletet, amelynek ez a bitje be van állítva, ráébred, hogy nehézségek következnek, és elhalasztja a nyugtázás elküldését a dekódoló egységnek. Eredményként a gép elakad ezen a ponton, amíg a mikroelágazás meg nem oldódik.

Elképzelhető, hogy az elágazáson túli IJVM-utasítások némelyike már be van táplálva a dekódoló egységbe (de a sorba állító egységbe nem), mivel nem küldött vissza nyugtázó (vagyis folytasd) jelet, amikor megtalált egy Goto bittel beállított mikroműveletet. Speciális hardver és működés szükségesek, hogy az összevisszaságot tisztázzák, és visszatáncoljanak, de ezek már túlmutatnak ennek a könyvnek a témakörén. Amikor Edsger Dijkstra megírta híres levelét „GOTO Statement Considered Harmful" [A GOTO utasítást károsnak tartjuk. (Dijkstra, 1968a)], sejtelme sem volt arról, milyen igaza volt.

Hosszú utat jártunk be a Mic–1 óta. A Mic–1 nagyon egyszerű hardver, szinte teljes szoftvervezérléssel. A Mic–4 egy erősen csővezetékezett terv hét szakasszal, és sokkal bonyolultabb hardverrel. A csővezetéket vázlatosan a 4.36. ábrán mutatjuk be, a bekarikázott számok a 4.35. ábra alkotóelemeire hivatkoznak. A Mic–4 önműködően előre betölt egy bájtfolyamot a memóriából, dekódolja IJVM-utasításokká, egy ROM-ot használva átalakítja mikroműveletek sorozatává, és sorba



4.36. ábra. *A Mic–4 csővezeték*

állítja felhasználásra, ahogy kell. A csővezeték első három szakasza az adatút órájához köthető, de nincs mindig elvégzendő munka. Például, az IFU biztosan nem képes minden óraciklusnál egy új IJVM műveleti kódot táplálni a dekódoló egységbe, mert az IJVM-utasítások néhány ciklust igényelnek a végrehajtáshoz, a sor pedig gyorsan túlcsordulna.

Minden egyes óraciklusban a MIR-ek előre lépnek, és a sor alján lévő mikroművelet bemásolódik a MIR1-be, hogy elkezdje a végrehajtást. A négy MIR-ből jövő vezérlőjelek ekkor kiterjednek az adatút felé, tevékenységek bekövetkezését eredményezve. Az egyes MIR-ek az adatút különböző részeit vezérlik, és ennek megfelelően különböző mikrolépéseket.

Ebben a tervben egy mélyen csővezetékezett CPU-nk van, amely lehetővé teszi, hogy az egyes lépések nagyon rövidek legyenek, és ennélfogva az órafrekvencia magas legyen. Számos CPU alapvetően ilyen módon van tervezve, különösen azok, melyeknek egy régebbi (CISC) utasításhalmazt kell megvalósítaniuk. Például, a Pentium 4 megvalósítása elviekben bizonyos szempontból hasonló a Mic–4-hez, ahogy azt később látni fogjuk ebben a fejezetben.

4.5. A teljesítmény növelése

Minden számítógépgyártó azt szeretné, hogy rendszerei a lehető leggyorsabban fussanak. Ebben a szakaszban megvizsgálunk számos fejlett technikát, amelyet azért kutatnak, hogy növeljék a rendszer (elsősorban a CPU és a memória) teljesítményét. A számítógépipar rendkívül versenyképes természetének köszönhetően meglepően rövid a lemaradás a számítógépek gyorsítási lehetőségeit feltáró új kutatási elképzelések és azok termékekben való megtestesülése között. Következésképpen, a legtöbb itt tárgyalt elv már létező termékek széles választékában használatos.

A megtárgyalandó elméletek durván két fogalomkörbe esnek: a megvalósítás javítása és az architektúra tökéletesítése. A megvalósítás javítása módszer arra, hogy új CPU-t vagy memóriát építsünk a rendszer futásának gyorsítására az architektúra megváltoztatása nélkül. Ez azt is jelenti, hogy a régi programok futnak majd az új gépen, ami az egyik fő értékesítési szempont. A megvalósítás javításának egyik módja gyorsabb óra használata, de nem ez az egyetlen mód. Az elért teljesítmény a 80386-tól a 80486-on a Pentiumon és a Pentium Pro-n keresztül a Pentium 4-ig a jobb megvalósításnak köszönhető, mivel az architektúra alapvetően ugyanaz maradt mindegyiknél.

Bizonyosfajta javulás elérhető csupán az architektúra megváltoztatásával. Néha ezek a változások bővítések, mint például új utasítások vagy regiszterek hozzáadása úgy, hogy a régi programok továbbra is fussanak az új modelleken. Ebben az esetben a csúcsteljesítményt úgy érhetjük el, hogy megváltoztatjuk a szoftvert, vagy legalábbis újrafordítjuk egy olyan fordítóprogrammal, amely kihasználja az új tulajdonságok előnyeit.

Mindazonáltal, valamikor az elmúlt évtizedekben a tervezők rájöttek, hogy a régi architektúrákban rejlő összes lehetőséget kiaknázták már, és a fejlődés egyetlen útja, ha mindent elölről kezdenek. Ilyen áttörés volt az 1980-as években a RISC-forradalom; egy másik pedig most van a levegőben. Ennek egy példáját (az Intel IA–64-et) az 5. fejezetben nézzük meg.

A szakasz hátralévő részében a CPU teljesítményének javítására négy különböző módszert fogunk megvizsgálni. Három jól bevált megvalósításjavítással kezdünk, és azután áttérünk egy olyanra, amely kis architektúratámogatást is igényel, hogy jobban dolgozzon. Ezek a módszerek a gyorsítótár, az elágazásjövendölés, a sorrendtől eltérő végrehajtás regiszterátnevezéssel és a feltételezett végrehajtás.

4.5.1. Gyorsítótár

A történelem folyamán a számítógép-tervezés egyik legnagyobb kihívása egy olyan memóriarendszerről gondoskodni, amely azzal a sebességgel képes szolgáltatni az operandusokat a processzornak, ahogy az fel tudja azokat dolgozni. Az utóbbi időkben a processzor nagyarányú sebességnövekedését nem követte a memória megfelelő sebességnövekedése. Az utóbbi évtizedek alatt a CPU-hoz viszonyítva a memóriák lassabbá váltak. Mivel az elsődleges memóriának óriási a jelentősége, ez a helyzet nagyban korlátozta a nagy teljesítményű rendszerek kifejlesztését, és arra ösztönözte a kutatást, hogy a memória sebességének problémája felé forduljon, mely sebesség sokkal alacsonyabb, mint a CPU-é, a két sebesség aránya pedig minden évben egyre rosszabbá válik.

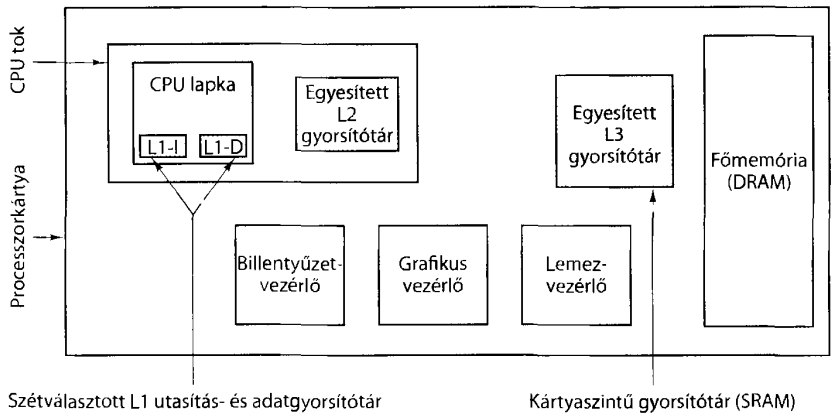
A modern processzorok nyomasztó követelést támasztanak a memóriarendszerekkel szemben mind várakozásban (egy operandus szolgáltatásának késése), mind sávszélességben (időegység alatt szolgáltatott adatmennyiség) kifejezve. Sajnos, a memóriarendszer e két szempontja hadilábon áll egymással. Sok sávszélességet növelő módszer csak a várakozás növelésének rovására működik. Például a Mic–3-ban használt csővezetékes módszert többszörös, átlapolt memóriakérések hatékony kezelésére alkalmazhatjuk egy memóriarendszernél. Sajnos, mint a Mic–3-nál, ez nagyobb várakozást eredményez az egyedi memóriaműveleteknél. Ahogy a processzor órasebessége nagyobbá válik, egyre nehezebb olyan memóriarendszerről gondoskodni, amely alkalmas egy vagy két óraciklus alatt szolgáltatni az operandusokat.

Az egyik mód ennek a problémának a legyőzésére a gyorsítótár alkalmazása. Mint azt a 2.2.5. részben láttuk, a gyorsítótár a legutóbbi időben használt memóriaszavakat tartalmazza egy kis, gyors memóriában, felgyorsítva azok elérését. Ha a szükséges memóriaszavak elég nagy százaléka van a gyorsítótárban, a tényleges memóriavárakozás óriási mértékben csökkenhet.

Az egyik leghatékonyabb módszer, mind a sávszélesség, mind a várakozás javítására a többszörös gyorsítótárak alkalmazása. Az alapmódszer, ami nagyon hatékonyan működik, az, hogy különálló gyorsítótárakat vezetünk be az utasítások és az adatok számára. Több előny származik abból, ha az utasítások és az adatok számára különálló gyorsítótár van, amit gyakran **szétválasztott gyorsítótár**nak neveznek. Először is, a memóriaműveletek függetlenül indulhatnak minden egyes gyorsítótárban, ténylegesen megkettőzve a memóriarendszer sávszélességét. Ez

az oka annak, hogy van értelme két külön memóriaportról gondoskodni, mint azt a Mic-1-nél tettük: minden portnak megvan a saját gyorsítótára. Megjegyezzük, hogy minden egyes gyorsítótárnak független hozzáférése van a főmemóriához.

Ma már sok memóriarendszer bonyolultabb ennél, és egy további, **2. szintű gyorsítótár**nak nevezett gyorsítótár lehet az utasítás- és adatgyorsítótár és a főmemória között. Valójában a gyorsítótárnak három vagy több szintje is lehetséges, ahogy egyre kifinomultabb memóriarendszerekre van igény. A 4.37. ábrán látunk egy rendszert, háromszintű gyorsítótárral. Maga a CPU lapka tartalmaz egy kis – jellemzően 16–64 KB-os – utasítás- és egy adatgyorsítótárat. Azután ott van a 2. szintű gyorsítótár, amelyik nincs rajta a CPU lapkán, de benne lehet a CPU tokban, közel a CPU lapkához, és azzal egy nagy sebességű útvonal köti össze. Ez a gyorsítótár általában egyesített: adatok és utasítások keverékét tartalmazza. Az L2 gyorsítótár jellemző mérete 512 KB és 1 MB között van. A harmadik szintű gyorsítótár a processzorkártyán van, és néhány megabájt SRAM-ot tartalmaz, ami sokkal gyorsabb a DRAM-memóriánál. A gyorsítótárak általában egyre bővülők, az 1. szintű gyorsítótár teljes tartalma benne van a 2. szintű gyorsítótárban, és a 2. szintű gyorsítótár teljes tartalma benne van a 3. szintű gyorsítótárban.



4.37. ábra. *Egy rendszer háromszintű gyorsítótárral*

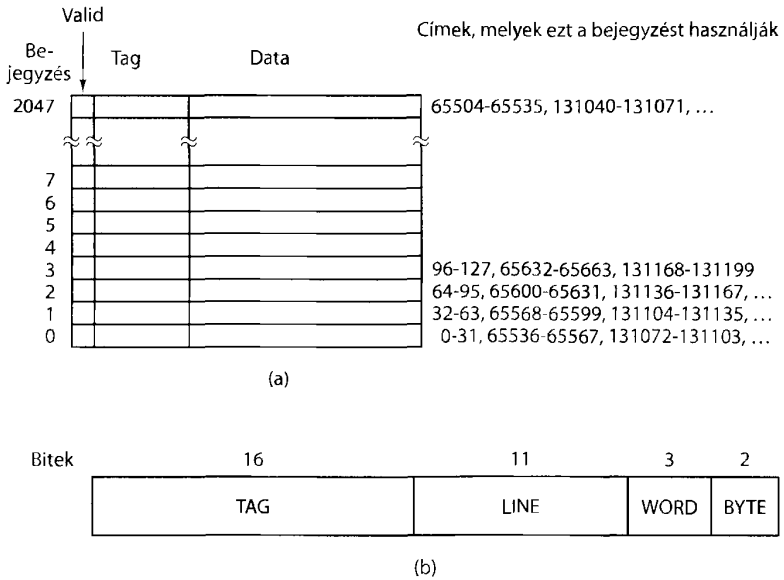
A gyorsítótárak céljuk elérésében kétféle címlokalitástól függnek. A **térbeli lokalitás** az a megfigyelés, hogy a közeli jövőben valószínűleg el kell érni azokat a memóriahelyeket, amelyek címe számszerűen közel van nemrégiben elért memóriahely címéhez. A gyorsítótárak azzal használják ki ezt a tulajdonságot, hogy a kértnél több adatot hoznak be, abban a reményben, hogy a következő kéréseknek elébe mehetnek. Az **időbeli lokalitás** azt jelenti, hogy a nemrégiben elért memóriahelyeket ismét el kell érni. Ez előfordulhat például a verem tetejéhez közeli memóriahelyekkel, vagy egy ciklus belsejében lévő utasításokkal. Az időbeli lokalitást a gyorsítótár tervezésénél elsősorban annak eldöntésénél lehet kihasználni, hogy gyorsítótárhiány esetén mit dobjunk el egy gyorsítótárból. Sok gyorsítótár-

cserélő algoritmus kihasználja az időbeli lokalitást, amikor azokat a beírásokat dobja el, amelyeket mostanában nem használtak.

Minden gyorsítótár a következő modellt használja. A főmemória fel van osztva rögzített méretű blokkokra, amiket **gyorsítósorok**nak nevezünk. Egy gyorsítósor jellemzően 4–64 egymást követő bájtból áll. A sorok 0-val kezdve meg vannak számozva, így 32 bájtos sorméretnél a 0. sor a 0–31. bájt, az 1. sor a 32–63. bájt és így tovább. Bármelyik pillanatban néhány sor a gyorsítótárban van. Amikor memóriahivatkozás történik, a gyorsítótár vezérlő áramköre megvizsgálja, hogy a hivatkozott szó éppen a gyorsítótárban van-e. Ha igen, akkor az ott lévő értéket lehet használni, és ezzel megtakarítunk egy utat a főmemóriához. Ha nincs ott a szó, akkor valamelyik sorbejegyzést kitesszük a gyorsítótárból, és helyette betöltjük a szükséges sort a memóriából vagy a magasabb szintű gyorsítótárból. Ennek a tervnek több változata is létezik, de mindegyikben az az ötlet, hogy a legsűrűbben használt sorokból annyit tartsunk a gyorsítótárban, amennyit csak lehet, és ezáltal maximalizáljuk a gyorsítótárból kielégített memóriahivatkozások számát.

Direkt leképezésű gyorsítótárak

A legegyszerűbb gyorsítótár a **direkt leképezésű gyorsítótár**. A 4.38. (a) ábrán az egyszintű direkt leképezésű gyorsítótárra láthatunk példát. Itt a gyorsítótárnak 2048 bejegyzése van. A gyorsítótárban minden egyes bejegyzés (sor) pontosan egy



4.38. ábra. *(a) Egy direkt leképezésű gyorsítótár. (b) Egy 32 bites virtuális cím*

gyorsítósort tartalmazhat a főmemóriából. 32 bájtos gyorsítósorméretttel (mint ebben a példában) a gyorsítótár 64 KB-ot tartalmazhat. Minden gyorsítótár-bejegyzés három részből áll:

1. A Valid bit jelzi, hogy van-e ebben a bejegyzésben érvényes adat, vagy nincs. Amikor a rendszer elindul (bootol), minden bejegyzést érvénytelennek jelöl meg.
2. A Tag mező egy 16 bites egyedi értékből áll, amely azt a memóriasort azonosítja, ahonnan az adat származik.
3. A Data mező a memóriában lévő adat másolatát tartalmazza. Ez a mező egy 32 bájtos gyorsítósort tartalmaz.

Egy direkt leképezésű gyorsítótárban egy adott memóriaszó pontosan egy helyen lehet tárolva. Ha adott egy memóriacím, azt pontosan egy helyen keressük a gyorsítótárban. Ha nincs ott, akkor nincs a gyorsítótárban. A gyorsítótárban az adatok tárolásához és kinyeréséhez négy alkotóelemre tagoljuk a címet, ahogyan az a 4.38. (b) ábrán látható:

1. A TAG (címke) mező a gyorsítótárban tárolt Tag bitek megfelelője.
2. A LINE mező jelzi, hogy melyik gyorsítótár-bejegyzés tartalmazza a megfelelő adatokat, ha azok a gyorsítótárban vannak.
3. A WORD mező mondja meg, hogy egy soron belül melyik szóra történik hivatkozás.
4. A BYTE mezőt általában nem használjuk, de ha csak egyetlen bájtra érkezik kérés, ez mondja meg, hogy a szón belül melyik bájtra van szükség. Egy olyan gyorsítótárnál, amelyik csak 32 bites szavakat szolgáltat, ez a mező mindig 0.

Amikor a CPU előállít egy memóriacímet, a hardver kiveszi a címből a 11 LINE bitet, és indexelésre használja a gyorsítótárban, hogy megtalálja a 2048 bejegyzés egyikét. Ha az a bejegyzés érvényes, a memóriacím TAG mezője és a gyorsítótár bejegyzésének Tag mezője összehasonlításra kerül. Ha megegyeznek, akkor a gyorsítótár-bejegyzés tartalmazza a keresett szót, ezt az esetet **gyorsítótár-találat**nak nevezzük. Egy találatnál a kiolvasandó szót a gyorsítótárból vehetjük ki, nem kell a memóriához fordulni. Csak az éppen szükséges szót vesszük ki a gyorsítótár-bejegyzéséből. A bejegyzés többi részét nem használjuk. Ha a gyorsítótár bejegyzése érvénytelen, vagy a címkék nem egyeznek meg, akkor a keresett bejegyzés nincs a gyorsítótárban, ezt az esetet **gyorsítótárhiány**nak nevezzük. Ebben az esetben a 32 bájtos gyorsítósor betöltődik a memóriából, és tárolódik a gyorsítótár-bejegyzésen, átírva azt, ami ott volt. Természetesen, ha a korábbi gyorsítótár-bejegyzés a betöltés óta módosult, az átírás előtt vissza kell írni a főmemóriába.

A döntés bonyolultsága ellenére a szükséges szó elérése figyelemre méltóan gyors lehet. Amint a cím ismert, a szó pontos helye is ismert, *ha benne van a gyorsítótárban*. Ez azt jelenti, hogy lehetőség van a szó kiolvasására a gyorsítótárból és továbbítására a processzorhoz, amikor a címkék (tag) összehasonlításával eldől, hogy ez a megfelelő szó. Így a processzor azzal egy időben, vagy akár még előbb megkap egy szót a gyorsítótárból, mielőtt megtudná, hogy az a kívánt szó.

Ez a leképezési séma az egymás után következő memóriasorokat egymás után következő gyorsítótár-bejegyzésekre teszi. Valójában ebben a gyorsítótárban legfeljebb 64 KB egymás mellett lévő adatot tárolhatunk. Viszont két olyan sor, amelynek a címe pontosan 64 KB-tal (65 536 bájttal) vagy ennek bármely egész számú többszörösével különbözik, nem lehet egyszerre a gyorsítótárban (mivel ugyanaz a LINE értékük). Ha egy program például az X helyen lévő adatot használja, és utána olyan utasítást hajt végre, amelynek az $X + 65\,536$ helyen (vagy ugyanezen a soron belül bárhol másutt) lévő adatra van szüksége, a második utasítás ki fogja kényszeríteni, hogy a gyorsítótár bejegyzése újratöltődjön, felülírva azt, ami ott volt. Ha ez elég gyakran megtörténik, az gyenge viselkedést eredményezhet. Valójában a gyorsítótár legrosszabb esetben tanúsított viselkedése rosszabb, mintha egyáltalán nem lenne gyorsítótár, mivel minden memóriaművelet egyetlen szó helyett egy egész gyorsítósor beolvasását vonja maga után.

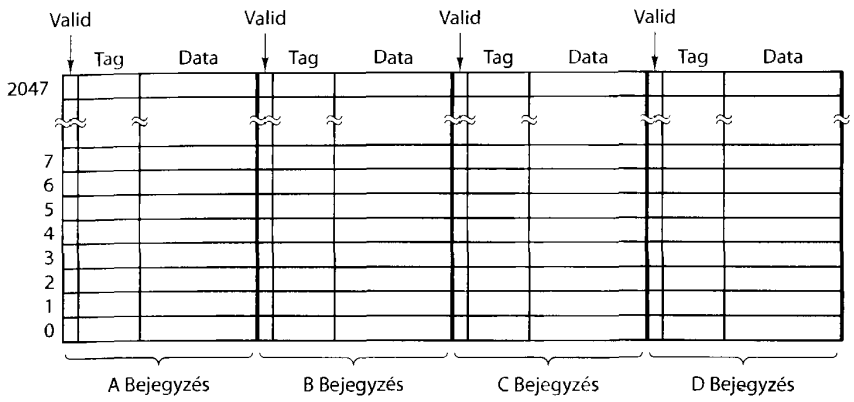
A direkt leképezésű gyorsítótárak a gyorsítótárak legelterjedtebb fajtái, és meglehetősen hatékonyan is működnek, mivel az olyan összeütközések, amit fentebb leírtunk, csak nagyon ritkán vagy egyáltalán nem fordulnak elő. Például, egy nagyon okos fordítóprogram számításba veheti a gyorsítótár-ütközéseket, amikor az utasításokat és adatokat elhelyezi a memóriában. Megjegyezzük, hogy a most leírt ütközés nem fordul elő olyan rendszerben, ahol külön van választva az utasítás- és az adatgyorsítótár, mivel az ütköző kéréseket különböző gyorsítótárak fogják kiszolgálni. Így láthatjuk a szétválasztott gyorsítótár második előnyét az egyesítettel szemben: nagyobb rugalmasság az egymásnak ellentmondó memóriasémák kezelésében.

Halmazkezelésű (csoportasszociatív) gyorsítótárak

Mint fentebb említettük, a memóriában sok különböző sor versenyez ugyanazért a gyorsítótár-bejegyzésért. Ha egy program a 4.38. (a) ábra gyorsítótárát alkalmazva erősen használja a 0 és 65 536 címeken lévő szavakat, állandó összeütközésbe kerül amiatt, hogy valószínűleg minden hivatkozás kilöki a másikat a gyorsítótárból. Ennek a problémának az a megoldása, hogy két vagy több sort is megengedünk minden gyorsítótár-bejegyzésen. Az olyan gyorsítótárat, amelynek minden egyes címhez n lehetséges bejegyzése (bejegyzéshalmaza) van, **n utas halmazkezelésű** (vagy **csoportasszociatív) gyorsítótár**nak nevezzük. Egy 4 utas halmazkezelésű gyorsítótárat mutat be a 4.39. ábra.

Egy halmazkezelésű gyorsítótár természetesen bonyolultabb, mint egy direkt leképezésű, mivel a hivatkozott memóriacímből csak a bejegyzéshalmaz címe számítható ki, és a gyorsítótár-bejegyzések n elemű halmazát kell ellenőrizni, hogy ott van-e a szükséges sor. Ennek ellenére a gyakorlat azt mutatja, hogy a 2 és 4 utas gyorsítótárak elég jól működnek ahhoz, hogy érdemes legyen ezt a bonyolultabb áramköri sémát elkészíteni.

A halmazkezelésű gyorsítótár használata a tervezőt megajándékozza egy választási lehetőséggel. Amikor egy új bejegyzést hozunk be a gyorsítótárba, a bent lévő tételek közül melyiket kell eldobni? Az optimális döntés, természetesen, egy kis jövőbelátást igényel, de legtöbbször elég jó algoritmus az **LRU (Least Recently**



4.39. ábra. *Egy 4 utas halmazkezelésű gyorsítótár*

Used, legrégebben használt). Ez az algoritmus rendezett listát készít minden bejegyzéshalmazhoz. Valahányszor a halmazban lévő sorok bármelyikéhez hozzányúlunk, frissíti a listát, megjelölve ezt a bejegyzést, mint legutoljára használtat. Ha elérkezik az idő, hogy egy bejegyzést kicseréljünk, a lista végén lévő – azaz a legrégebben használt – lesz az, amit eldobunk.

Szélsőséges esetben 2048 utas gyorsítómemória is lehetséges, amely egyetlen halmazban 2048 bejegyzést tartalmaz. Ekkor minden memóriacímet egyetlen halmazba képezünk le, így a keresés megköveteli a cím összehasonlítását a gyorsítótárban lévő mind a 2048 címkével. Megjegyezzük, hogy a halmazkezelésű gyorsítótár esetén minden egyes bejegyzésnek rendelkeznie kell egy címke-összehasonlító logikával. Mivel a LINE mező most 0 hosszúságú, a TAG mező a teljes cím, eltekintve a WORD és a BYTE mezőktől. Továbbá, ha egy gyorsítósort kicserélünk, mind a 2048 bejegyzés lehetséges jelölt a cserére. A 2048 bejegyzés rendezett listájának karbantartása jó sok könyvelési munkát követel, hogy az LRU-cseréket meg tudjuk valósítani. (Ne felejtsük el, hogy a listát minden memóriaműveletnél frissíteni kell, nemcsak akkor, ha nincs találat.) Meglepő, hogy a legtöbb esetben a sok utas halmazkezelésű gyorsítótárak nem javítanak a teljesítményen sokkal többet, mint a kevés utasok, és néhány esetben még rosszabb is a teljesítményük. Ezért a négyutas feletti halmazkezelés viszonylag szokatlan.

Az írások különleges problémát vetnek fel a gyorsítótárral kapcsolatban. Amikor egy processzor egy szót ír, és a szó a gyorsítótárban van, nyilvánvalóan vagy frissíteni kell a szót a gyorsítótárban, vagy el kell dobni a bejegyzést. Szinte minden konstrukció frissíti a gyorsítótárat. De mi a helyzet a főmemóriában lévő másolat frissítésével? Ezt a műveletet későbbre halaszthatjuk, amikor a bejegyzést ki kell cserélni. Ez nehéz döntés, és egyetlen választás sem egyértelműen kedvezőbb. A bejegyzés azonnali frissítését a főmemóriában **írásáteresztés**nek nevezzük. Ezt a megközelítést általában egyszerűbb megvalósítani, és megbízhatóbb is, mivel a memória mindig naprakész – hasznos például olyankor, ha hiba fordul elő,

és szükség van a memória állapotának helyreállítására. Sajnos, ez általában nagyobb írásforgalmat követel a memóriába, így sok kifinomult megvalósítás hajlik a másik megoldás alkalmazására, amelyet **késleltetett írás**nak vagy **visszaírás**nak neveznek.

Az írásokhoz kapcsolódóan még egy problémát kell felvetni: Mi a helyzet, ha olyan helyre történik írás, amely éppen nincs benn a gyorsítótárban? Be kell hozni az adatot a gyorsítótárba, vagy csak ki kell írni a memóriába? Ismét nincs olyan válasz, ami mindig helyes. A legtöbb, késleltetett memóriába írást alkalmazó konstrukció hajlik arra, hogy behozza az adatot a gyorsítótárba íráshiány esetén is. Ezt a módszert **írásallokálás**nak nevezzük. Másrészt a legtöbb írásáteresztő kivitelezés íráshoz nem tölt bejegyzést a gyorsítótárba, mert ez az egyébként egyszerű kivitelezést bonyolulttá tenné. Az írásallokálás csak akkor előnyös, ha ismételt írások történnek egy gyorsítósor ugyanazon vagy különböző szavaira.

A gyorsítótár teljesítménye meghatározó a rendszer teljesítménye szempontjából, mert a CPU és a memória sebessége között nagyon nagy a különbség. Következésképpen, az egyre jobb gyorsítótárazási stratégiák fejlesztése még mindig divatos téma; lásd (Alameldeen és Wood, 2004; Huh és társai, 2004; Min és társai, 2004; Nesbit és Smith, 2004; Suh és társai, 2004).

4.5.2. Elágazásjövendölés

A modern számítógépek magas szinten csővezetékezettek. A 4.35. ábra csővezetékének hét szakasza van; a csúcsteljesítményű számítógépeknek néha 10 vagy még több szakaszú csővezetéke van. A csővezetékek lineáris kóddal dolgoznak legjobban, így a betöltő egység éppen egymás utáni szavakat olvashat a memóriából, és küldhet a dekódoló egységhez, már mielőtt szükség lenne rá.

Ezzel a csodálatos modellel az egyetlen kis probléma az, hogy a legkevésbé sem valószerű. A programok nem lineáris kódsorozatok. Teli vannak elágazó utasításokkal. Tekintsük a 4.40. (a) ábra egyszerű utasításait. Egy *i* változót összehasonlítunk 0-val (talán a legáltalánosabb vizsgálat a gyakorlatban). Az eredménytől függően egy másik *k* változó a két lehetséges érték egyikét kapja.

```
if (i == 0)
    k = 1;
else
    k = 2;
```

(a)

```
        CMP i,0     ; összehasonlítja i-t 0-val
        BNE Else    ; elágazik Else-hez, ha nem egyenlő
Then:   MOV k,1     ; k-ba 1-et tesz
        BR Next     ; feltétel nélküli elágazás Next-hez
Else:   MOV k,2     ; k-ba 2-t tesz
Next:
```

(b)

4.40. ábra. *(a) Egy programrészlet. (b) Fordítása egy általános assembly nyelvre*

Ennek egy lehetséges assembly nyelvű fordítását a 4.40. (b) ábra mutatja. Az assembly nyelvet később még tárgyaljuk, most a részletek nem fontosak, de a géptől és a fordítóprogramtól függően a kód többé-kevésbé hasonló lehet a 4.40. (b) áb-

rán lévőhöz. Az első utasítás *i*-t összehasonlítja 0-val. A második elágazik az *Else* címkéhez (az else rész kezdetéhez), ha *i* nem 0. A harmadik utasítás *k*-nak 1 értéket ad. A negyedik utasítás a következő utasítás kódjához ágazik el. A fordítóprogram az elérhetőség kedvéért egy *Next* címkét helyezett itt el, így ez olyan hely, ahova lehet elágazni. Az ötödik utasítás a 2 értéket adja *k*-nak.

Figyeljük meg, hogy itt öt utasításból kettő elágazó. Továbbá egyikük, a BNE feltételes elágazás (olyan elágazás, ami akkor és csak akkor történik meg, amikor egy feltétel teljesül, ebben az esetben, ha az előző CMP-ben a két operandus nem egyezik meg). A leghosszabb lineáris kódsorozat itt két utasítás. Ennek következtében nagyon nehéz utasításokat nagy számban betölteni a csővezeték táplálására.

Első pillantásra úgy tűnhet, hogy nincs probléma a feltétel nélküli elágazásokkal, mint amilyen a 4.40. (b) ábra BR Next utasítása. Mindenesetre afelől nincs kétség, hogy hova kell menni. Miért ne tudná folytatni a betöltő egység az utasítások olvasását a célcímről (arról a helyről, ahová az elágazás történni fog)?

A nehézség a csővezeték jellegéből adódik. Például a 4.35. ábrán látjuk, hogy az utasítás dekódolás a második szakaszban történik. Így a betöltő egységnek el kell döntenie, hogy honnan töltse be a következőt, még mielőtt tudná, milyen utasítást kapott utoljára. Csak egy ciklussal később tudhatja meg, hogy pont egy feltétel nélküli elágazási utasítást kapott, és akkorra már elkezdte a feltétel nélküli elágazást követő utasítást betölteni. Ennek következtében a csővezetéket alkalmazó gépek (mint az UltraSPARC III) tekintélyes részének megvan az a tulajdonsága, hogy egy feltétel nélküli elágazást *követő* utasítás végrehajtódik, annak ellenére, hogy logikusan nem kellene. Az elágazás mögötti helyet **eltolási rés**nek nevezzük. A Pentium 4 [és a 4.40. (b) ábrán használt gép] nem rendelkezik ezzel a tulajdonsággal, de a probléma megkerülésének belső összetettsége gyakran óriási. Egy optimalizáló fordítóprogram megpróbál valamilyen hasznos utasítást találni, amit az eltolás résbe tehet, de gyakran nincs semmi felhasználható, így kénytelen egy NOP utasítást odatenni. Ezzel a program helyességét megőrizzük, de hosszabbá és lassabbá tesszük.

Bosszantók a feltétel nélküli elágazások, de a feltételes elágazások még roszszabbak. Nemcsak eltolási réseket hoznak létre, hanem a betöltő egység még azt sem tudja, hogy honnan olvasson, amíg a feltételes elágazás sokkal mélyebbre nem kerül a csővezetékbe. A korai csővezetékes gépek csaknem **bedugult**ak, amíg nem derült ki, hogy lesz elágazás, vagy sem. Három vagy négy ciklusra bedugulni minden feltételes elágazásnál, különösen, ha az utasítások 20%-a feltételes elágazás, a teljesítmény romlásával bosszulja meg magát.

Következésképpen a legtöbb gép, amikor talál egy feltételes elágazást, megjövendöli, hogy végre kell hajtani, vagy nem. Kellemes lenne, ha bedughatnánk egy kristálygömböt egy szabad PCI-csatlakozóba, hogy segítsen a jövendölésben, de mindeddig ez a megközelítés nem hozott eredményt.

Egy ilyen periféria hiányában különféle módszereket gondoltak ki a jövendölésre. Egy nagyon egyszerű jövendölés a következő: tételezzük fel, hogy minden visszafelé történő feltételes elágazást végre fogunk hajtani, és az összes előre irányulót nem. Az első rész mellett szól az az érv, hogy a visszafelé elágazások gyakran egy ciklus végén helyezkednek el. A legtöbb ciklust többször hajtjuk végre, így

általában beválik az a feltevés, hogy végre kell hajtani egy visszafelé elágazást a ciklus elejére.

A második rész rázósabb. Néhány előre irányuló elágazás akkor fordul elő, amikor hibát észlel a program (például egy fájlt nem tudunk megnyitni). A hibák ritkák, így a legtöbb velük összefüggő elágazást nem hajtjuk végre. Természetesen sok olyan előre irányuló elágazás van, ami nem kapcsolódik hibakezeléshez, így a siker aránya közel sem olyan jó, mint a visszafelé elágazásnál. Bár nem fantasztikus, de ez a szabály legalább jobb a semminél.

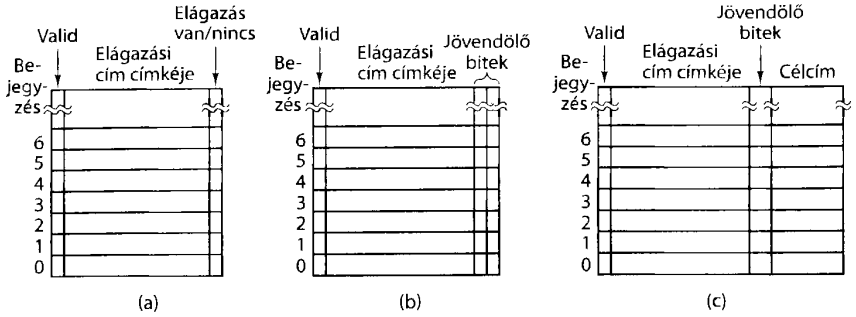
Ha egy elágazást pontosan megjósoltunk, nincs semmi különös tennivaló. A végrehajtás éppen a célcímnél folytatódik. Nehézségbe akkor ütközünk, amikor rosszul jósoltunk meg egy elágazást. Nem nehéz kigondolni, hogy hova kell menni, és nem nehéz odamenni sem. Az a nehéz, hogy visszacsináljuk azokat az utasításokat, amelyeket már végrehajtottunk, és nem kellett volna.

Ehhez két módon foghatunk. Az első, hogy megengedjük egy megjövendölt feltételes elágazás után betöltött utasítások végrehajtását, egészen addig, amíg nem próbálják megváltoztatni a gép állapotát (például egy regiszterbe tárolással). A regiszter felülírása helyett a számított értéket betesszük egy (rejtett) firkáló regiszterbe, és csak akkor másoljuk a valódi regiszterbe, amikor már tudjuk, hogy a jövendölésünk helyes volt. A második módszer, hogy feljegyezzük (például egy rejtett firkáló regiszterbe) minden olyan regiszter értékét, amely esetleg majd felülíródik, így a gép visszaforgatható a tévesen megjósolt elágazáskori állapotába. Mind a két megoldás bonyolult, és ipari erősségű könyvelést követel, hogy helyesen végezzük el. És ha egy második feltételes elágazást találunk, mielőtt kiderül, hogy az elsőt helyesen jósoltuk-e meg, a dolgok valóban összekeveredhetnek.

Dinamikus elágazásjövendölés

Világos, hogy nagyszámú pontos jövendölés lehetővé teszi a CPU-nak, hogy teljes sebességgel haladjon. Ennek következtében számos jelenlegi kutatás az elágazást jövendölő algoritmusok javítását célozza (Chen és társai, 2003; Falcon és társai, 2004; Jimenez, 2003; Parikh és társai, 2004). Az egyik megközelítés, hogy a CPU egy előzménytáblát tart fenn (egy különleges hardverben), amelyben feljegyzi az előforduló feltételes elágazásokat, így kikereshetők, ha ismét előfordulnak. Ennek a módszernek a legegyszerűbb változata látható a 4.41. (a) ábrán. Itt az előzménytábla minden feltételes elágazásra tartalmaz egy bejegyzést. A bejegyzés tartalmazza az elágazó utasítás címét egy bittel együtt, amely megmondja, hogy megtörtént-e az elágazás, amikor legutoljára végrehajtottuk. Ezt a rendszert használva a jövendölés egyszerűen az, hogy az elágazás ugyanazt az utat fogja követni, mint legutóbb. Ha a jövendölés rossz, a bit megváltozik az előzménytáblában.

Számos módszer van az előzménytábla szervezésére. Valójában ezek pontosan ugyanazok a módszerek, mint amiket a gyorsítótár szervezésénél használunk. Tekintsünk egy gépet 32 bites utasításokkal, amelyek szóhatáron kezdődnek, vagyis minden memóriacím alsó két bitje 00. Egy direkt leképezésű, 2^n bejegyzést tartalmazó előzménytábla esetén egy elágazó utasítás alsó $n + 2$ bitjét kivehetjük, és 2



4.41. ábra. *(a) Egy 1 bites elágazás előzménytáblája. (b) Egy 2 bites elágazás előzménytáblája. (c) Egy leképezés az elágazó utasítás címe és a célcím között*

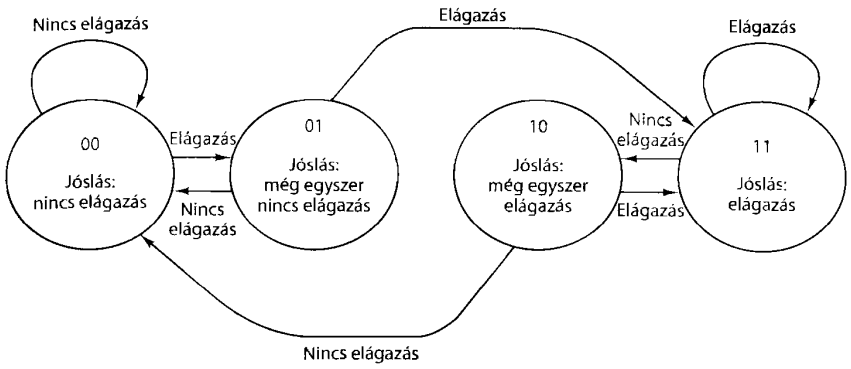
bittel jobbra léptethetjük. Az n bites számot indexként használhatjuk az előzménytáblához, ahol ellenőrzést végzünk annak megállapítására, hogy az ott tárolt cím megegyezik-e az elágazás címével. Akárcsak a gyorsítótárnál, nincs szükség az alsó $n + 2$ bit tárolására, így mellőzhetők (azaz csak a felső címbiteket – a címkét – tároljuk). Ha találat van, a jövendölő bitet használjuk az elágazás jóslására. Ha rossz címke van ott, vagy a bejegyzés érvénytelen, hiány lép fel éppúgy, mint a gyorsítótárnál. Ebben az esetben az előre/visszafelé elágazási szabályt alkalmazhatjuk.

Ha az elágazás előzménytábla, mondjuk, 4096 bejegyzést tartalmaz, a 0, 16 384, 32 768, ... címeken lévő elágazások összeütközésbe kerülnek, hasonlóan a gyorsítótár ugyanilyen problémájához. Ugyanaz a megoldás is lehetséges: egy kétutas, egy négyutas vagy egy n utas asszociatív bejegyzés. Ugyanúgy, mint a gyorsítótárnál, a határeset egyetlen n utas asszociatív bejegyzés, amely megköveteli a kikeresés teljes asszociativitását.

Ha elég nagy a táblaméret, és elegendő az asszociativitás, ez az elrendezés a legtöbb esetben jól működik. Viszont egy probléma rendszeresen előfordul. Amikor a ciklus végül kilép, a végén az elágazást tévesen jövendöli, és ami még rosszabb, a rossz jövendölés megváltoztatja az előzménytáblát, „nincs elágazás"-nak állítja be a következő jövendölést. Amikor legközelebb a ciklusba lépünk, az első ismétlés végén az elágazást rosszul fogja megjövendölni. Ha a ciklus egy külső ciklus belsejében van, vagy egy gyakran meghívott eljárásban, ez a hiba gyakran előfordulhat.

Hogy kiküszöböljük ezt a téves jövendölést, a táblázat bejegyzésének adhatunk egy második lehetőséget. Ezzel a módszerrel a jövendölés csak két egymás utáni helytelen jövendölés után változik meg. Ez a megközelítés két jövendölő bit meglétét követeli meg az előzménytáblában, egyet arra, hogy az elágazás „feltehetőleg" mit fog tenni, és egyet arra, hogy mit tett legutóbb, ahogy azt a 4.41. (b) ábra mutatja.

Kicsit más szemszögből nézve ezt az algoritmust, tekinthetjük úgy, mint egy véges állapotú gépet négy állapottal, ahogy azt a 4.42. ábrán is láthatjuk. Egy sorozat egymás után következő sikeres „nincs elágazás" jövendölés után FSM 00 állapotban lesz, és legközelebb „nincs elágazás"-t fog jövendölni. Ha ez a jövendölés



4.42. ábra. *Egy 2 bites véges állapotú gép elágazásjövendölésre*

rossz, átmegy 01 állapotba, de legközelebb ugyanúgy „nincs elágazás"-t jövendöl. Csak ha ez a jövendölés is rossz, akkor fog átmenni 11 állapotba, és folyamatosan elágazást jövendölni. Gyakorlatilag az állapot bal szélső bitje a jövendölés, a jobb szélső bit pedig az, hogy mi történt az elágazásban legutóbb. Míg ez az elgondolás csak 2 bitet igényel az előzménytáblából, olyan kivitelezés is lehetséges, amelyik 4 vagy 8 bittel követi nyomon az eseményeket.

Ez nem az első FSM-ünk. A 4.28. ábra szintén FSM volt. Valójában minden mikroprogramunk FSM-nek tekinthető, mivel minden sor a gép egy meghatározott állapotát képviseli, jól meghatározott átmenetekkel más állapotok egy véges halmazába. Az FSM-eket igen széles körben alkalmazzák a hardvertervezés minden területén.

Mindeddig feltételeztük, hogy minden feltételes elágazás célpontja ismert, jellemzően vagy egy határozott cím, ahova el kell ágazni (az utasítás tartalmazza a címet), vagy mint egy relatív eltolás a jelenlegi utasításhoz képest (például egy előjeles szám, amit az utasításszámlálóhoz kell hozzáadni). Gyakran jogos ez a feltételezés, de néhány feltételes elágazó utasítás a célcímet regisztereken végzett aritmetikai műveletekkel számolja ki, és úgy ugrik el. Még ha a 4.42. ábrán lévő FSM pontosan meg is jövendöli az elágazást, az olyan jövendölésnek nincs haszna, amelynél a célcím ismeretlen. E helyzet kezelésének egyik módszere, hogy az előzménytáblában tároljuk azt a tényleges címet, ahova legutóbb elágaztunk, ahogy az a 4.41. (c) ábrán látható. Ily módon, ha a tábla azt mondja, hogy legutóbb az 516-os címen lévő elágazás a 4000-es címre ugrott, akkor a munkahipotézis, ha a jövendölés most „van elágazás", ismét 4000-re történő elágazás lesz.

Az elágazásjövendölés egy másik megközelítése, hogy azt figyeljük, hogy az utolsó k feltételes elágazást végre kellett-e hajtani, függetlenül attól, hogy melyik utasítás volt az. Ezt a k bites számot, amelyet az **elágazási előzmények blokkos regiszteré**ben tárolunk, összehasonlítjuk párhuzamosan az előzménytábla minden bejegyzésének k bites kulcsával, és találat esetén az ott talált jövendölést használjuk. Némiképp meglepő, hogy ez a módszer egészen jól működik.

Statikus elágazásjövendölés

Minden eddig tárgyalt elágazásjövendölési módszer dinamikus, azaz futási időben jövendöl, miközben a program fut. Alkalmazkodik a program érvényes viselkedéséhez, ami jó dolog. A rossz oldala az, hogy speciális és drága hardvert igényel, és jókora lapkabonyolultságot.

Más utat követhetünk, ha a fordítóprogram segít bennünket. Ha a fordítóprogram egy ilyen utasítást lát:

```
for (i = 0; i < 1000000; i++) { ... }
```

nagyon jól tudja, hogy a ciklus végén lévő elágazást szinte állandóan végre kell hajtani. Bárcsak lenne arra mód, hogy ezt megmondjuk a hardvernek, sok erőfeszítést megtakarítanánk.

Bár ez architekturális változás (és nem csak egy kivitelezési kérdés), néhány gép, mint például az UltraSPARC III, a szokásoson felül (amelyek a visszafelé kompatibilitáshoz szükségesek) rendelkezik egy második feltételes elágazóutasítás-halmazzal (jövendölő elágazó utasítások). Az új utasítások tartalmaznak egy jövendölő bitet, amelyben a fordítóprogram jelezheti, úgy gondolja, az elágazást végre kell hajtani (vagy sem). Egy ilyen esetén, a betöltő egység pontosan azt teszi, amit az utasítás mond. Ezenkívül nem szükséges értékes helyet pazarolni az elágazás előzménytáblában ezekre az utasításokra, így csökkennek az összeütközések a táblában.

Végül az utolsó elágazásjövendölő módszerünk a megfigyelésen (profiling) alapul (Fisher és Freudenberger, 1992). Ez szintén statikus módszer, de nem a fordítóprogram feladata, hogy megpróbálja kitalálni, melyik elágazást hajtjuk végre, és melyiket nem. A program valójában fut (általában egy szimulátoron), és az elágazás viselkedését figyeljük meg. Ezt az információt betápláljuk a fordítóprogramba, ami azután jövendölő elágazó utasításokat használ, hogy a hardvernek megmondja, mit csináljon.

4.5.3. Sorrendtől eltérő végrehajtás és regiszterátnevezés

A legtöbb modern CPU mind csővezetékes, mind szuperskaláris, ahogy az a 2.6. ábrán látható. Ez általában azt jelenti, hogy van egy betöltő egység, amelyik kiveszi a memóriából az utasításszavakat, mielőtt még szükség lenne rájuk, hogy tápláljon egy dekódoló egységet. A dekódoló egység kiosztja a dekódolt utasításokat a megfelelő működési egységnek végrehajtásra. Néhány esetben a kiosztás előtt az egyedi utasításokat mikroutasításokra darabolhatja, attól függően, hogy a működési egységek mit képesek elvégezni.

Világos, hogy a számítógép-tervezés akkor a legegyszerűbb, ha minden utasítást abban a sorrendben hajtunk végre, ahogyan betöltöttük (pillanatnyilag tételezzük fel, hogy az elágazásjövendölő algoritmus soha nem ad rossz tippet). Mégis, a sorrend szerinti végrehajtás, az utasítások közötti függőség miatt, nem mindig

eredményez optimális teljesítményt. Ha egy utasításnak olyan értékre van szüksége, amit az előző utasítás számít ki, a második nem tudja elkezdeni a végrehajtást, amíg az első elő nem állítja a szükséges értéket. Ebben az esetben (RAW függőség) a második utasításnak várnia kell. A függőségnek más fajtái is léteznek, ahogy majd később látjuk.

Ennek a problémának a megkerülésére, és jobb teljesítmény elérésére, néhány CPU megengedi, hogy a függő utasításokat átugorjuk, és rátérjünk a későbbi utasításokra, amelyek nem függők. Szükségtelen mondani, hogy az alkalmazott belső utasítást ütemező algoritmusnak ugyanazt az eredményt kell szolgáltatnia, mintha a programot a leírt sorrendben hajtottuk volna végre. A következőkben egy részletes példán fogjuk megmutatni, hogyan működik az utasítások sorrendjének átrendezése.

A probléma természetének szemléltetésére egy olyan géppel kezdünk, amely mindig a program sorrendjében osztja ki az utasításokat, és megköveteli, hogy a végrehajtás is a program sorrendjében fejeződjön be. Az utóbbi kérdés jelentősége később fog világossá válni.

Példagépünknek van nyolc, a programozó számára látható regisztere, R0-tól R7-ig. Minden aritmetikai utasítás három regisztert használ: kettőt az operandusok és egyet az eredmény számára, ugyanúgy, mint a Mic–4-nél. Feltételezzük, hogy ha egy utasítást az n. ciklusban dekódolunk, a végrehajtás az $n + 1$. ciklusban kezdődik. Egy egyszerű utasításnál, mint amilyen az összeadás vagy a kivonás, a célregiszterbe történő visszaírás az $n + 2$. ciklus végére megtörténik. Összetettebb utasításnál, mint amilyen a szorzás, a visszaírás az $n + 3$. ciklus végére történik meg. Ahhoz, hogy valószerűbbé tegyük a példát, a dekódoló egységnek megengedjük, hogy óraciklusonként legfeljebb két utasítást kiosszon. A kereskedelemben lévő szuperskaláris CPU-k gyakran négy vagy akár hat utasítást is ki tudnak osztani óraciklusonként.

A példa végrehajtási sorrendje a 4.43. ábrán látható. Itt az első oszlop a ciklus sorszámát, a második pedig az utasítás sorszámát adja meg. A harmadik oszlop feltünteti a dekódolt utasítást. A negyedik mondja meg, melyik utasítást osztottuk ki (óraciklusonként legfeljebb kettőt). Az ötödik pedig azt adja meg, hogy most melyik utasítás készült el. Ne felejtsük el, ebben a példában mind a sorrend szerinti kiosztást, mind a sorrend szerinti befejezést megköveteltük, így a $k + 1$. utasítás nem osztható ki, amíg a k. utasítást nem osztottuk ki, és a $k + 1$. utasítást nem fejezhetjük be (ez a célregiszterbe való visszaírás elvégzését jelenti), míg a k. utasítás nem készül el. A többi 16 oszlopot az alábbiakban tárgyaljuk.

Egy utasítás dekódolása után a dekódoló egységnek el kell döntenie, hogy az utasítást ki tudja-e azonnal osztani, vagy sem. Ehhez a döntéshez a dekódoló egységnek ismernie kell az összes regiszter állapotát. Ha például a jelen utasításnak egy olyan regiszterre van szüksége, amelynek az értéke még nincs kiszámolva, nem tudja kiosztani, és várakoznia kell.

A regiszterhasználatot egy **eredményjelző**nek nevezett eszközzel fogjuk nyomon követni, ami először a CDC 6600-asban jelent meg. Az eredményjelzőnek minden egyes regiszterhez van egy kis számlálója, ami megmondja, hogy a regisztert hányszor használjuk forrásként az éppen végrehajtás alatt álló utasításoknál. Ha,

					Olvasott regiszterek								Írt regiszterek						
Cik	#	Dekódolt	Ind	Bef	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3 = R0 × R1	1		1	1										1				
	2	R4 = R0 + R2	2		2	1	1									1	1			
2	3	R5 = R0 + R1	3		3	2	1									1	1	1		
	4	R6 = R1 + R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7 = R1 × R2	5			2	1		1										1	1
6	6	R1 = R0 – R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1								1					
	7	R3 = R3 × R1	–		1		1								1					
10					1		1								1					
11				6																
12			7			1		1								1				
	8	R1 = R4 + R4	–			1		1								1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

4.43. ábra. *Egy szuperskaláris CPU működése sorrend szerinti kiosztással és sorrend szerinti befejezéssel*

mondjuk, maximum 15 utasítást lehet egyszerre végrehajtani, akkor megteszi egy 5 bites számláló (elvileg minden utasítás mindkét operandusa lehet ugyanaz a regiszter). Ha kiosztunk egy utasítást, megnövelődnek az operandus regisztereinek eredményjelző bejegyzései. Ha egy utasítás elkészült, a bejegyzések csökkennek.

Az eredményjelzőnek a célként használt regiszterek számára is van egy számlálója. Mivel egy időben csak egy írás megengedett, ezek a számlálók 1 bit szélesek lehetnek. A 4.43. ábra jobb oldali 16 oszlopa az eredményjelzőt mutatja.

A 4.43. ábra első sora I1-et (1. utasítást) mutatja, amelyik megszorozza R0-t R1-gyel, és az eredményt R3-ba teszi. Mivel ezek közül a regiszterek közül még egyik sincs használatban, az utasítás kiosztható. Az eredményjelzőt frissíteni kell, hogy tükrözze, R0-at és R1-et olvassuk, és R3-at írjuk. Egyetlen soron következő utasítás sem írhat ezek egyikébe sem, vagy nem olvashatja R3-at addig, amíg I1 el nem készült. Mivel ez az utasítás egy szorzás, a 4. ciklus végére fejeződik be. Minden egyes sorban az ott látható eredményjelző értékek azt az állapotot tükrözik, amely az abban a sorban lévő utasítás kiosztása után keletkezik. Az üres bejegyzések nullák.

Mivel a példánk olyan szuperskaláris gép, amely két utasítást tud kiosztani ciklusonként, egy második utasítás is (I2) kiosztódik az 1. ciklus alatt. Ez R0-t R2-höz adja, az eredményt R4-be tárolja. Ahhoz, hogy lássuk, hogy ez az utasítás kiosztható-e, a következő szabályokat alkalmazzuk:

1. Ha bármelyik operandust írjuk, nem osztjuk ki (RAW függőség).
2. Ha az eredményregisztert olvassuk, nem osztjuk ki (WAR függőség).
3. Ha az eredményregisztert írjuk, nem osztjuk ki (WAW függőség).

Már láttuk a RAW függőséget, ami akkor fordul elő, amikor egy utasításnak olyan eredményt kellene használnia forrásként, amit az előző utasítás még nem állított elő. A két másik függőség kevésbé súlyos. Alapvetően forrás-összeütközések. A **WAR függőség**ben (Write After Read; olvasás után írás) egy utasítás megpróbál felülírni egy regisztert, amelynek olvasását az előző utasítás lehet, hogy még nem fejezte be. A **WAW függőség** (Write After Write; írás után írás) hasonló. Ezek gyakran elkerülhetők, ha a második utasítás az eredményét valahova máshová teszi (esetleg ideiglenesen). Ha a fenti három függőség egyike sem áll fenn, és a szükséges működési egység elérhető, az utasítás kiosztható. Ebben az esetben I2 használ egy regisztert (R0), amelyet egy befejezetlen utasítás olvas, de ez az átfedés megengedett, így I2-t kiosztjuk. Hasonlóan, I3-t is kiosztjuk a 2. ciklus alatt.

Most elérkeztünk I4-hez, amelynek használnia kell R4-et. A 3. sorban látjuk, hogy R4-et, sajnos, éppen írjuk. RAW függőségünk van, így a dekódoló egység várakozik, amíg R4 elérhető nem lesz. Amíg várakozik, abbahagyja az utasítások kiszedését a betöltő egységből. Amikor a betöltő egység belső puffere megtelik, abbahagyja az előre betöltést.

Említésre érdemes, hogy a program soron következő utasításának, I5-nek nincs semmilyen összeütközése egyetlen befejezetlen utasítással sem. Dekódolhattuk és kioszthattuk volna, ha nem követelnénk meg az utasítások sorrend szerinti kiosztását.

Most nézzük meg, mi történik a 3. ciklus alatt. Mivel I2 összeadás (két ciklus), a 3. ciklus végére befejeződik. Sajnos, nem tudjuk befejezni (ezzel felszabadítani R4-et I4 számára). Hogy miért nem? Azért, mert most a sorrend szerinti befejezést is megköveteltük. De hát milyen baj származhat abból, hogy most végezzük el a tárolást R4-be, és megjelöljük mint elérhetőt?

A válasz szövevényes, de fontos. Tegyük fel, hogy az utasítások sorrendtől eltérően befejeződhetnek. Akkor, ha egy megszakítás következik be, nagyon nehéz lesz a gép állapotát eltárolni, hogy később visszaállítható legyen. Különösen azt lehetetlen megmondani, hogy a gép egy bizonyos címig végrehajtott-e minden utasítást, és a mögötte lévők közül egyet sem. Ezt **pontos megszakítás**nak nevezzük, és ez a CPU kívánatos tulajdonsága (Moudgill és Vassiliadis, 1996). Egy sorrendtől eltérő befejezés a megszakítást pontatlanná tenné, ezért követelik meg egyes gépek a sorrend szerinti utasításbefejezést.

Visszatérve példánkhoz, a 4. ciklus végén mind a három befejezetlen utasítás elkészül, így az 5. ciklusban I4 végre kiosztásra kerülhet az újonnan dekódolt I5-tel együtt. Valahányszor egy utasítás befejeződik, a dekódoló egységnek ellenőriznie kell, hogy van-e leállított utasítás, amely most kiosztható.

A 6. ciklusban I6 elakad, mivel írnia kellene R1-be, de R1 foglalt. Végül a 9. ciklusban indul el. A nyolc utasításból álló teljes sorozat befejezéséig 18 ciklus telik el a sok függőség miatt, még akkor is, ha a hardver képes minden ciklusban két utasítást kiosztani. Vegyük észre azonban, hogy végigolvasva a 4.43. ábra *Ind* oszlopát, minden az utasítások sorrendjében osztódik ki. Hasonlóan, a *Bef* oszlop mutatja, hogy a befejezés is sorrend szerint történt.

Most tekintsünk egy alternatív elgondolást: a sorrendtől eltérő végrehajtást. Ebben az elgondolásban az utasításokat sorrendtől eltérően lehet kiosztani, és ugyanígy sorrendtől eltérően lehet befejezni. Ugyanaz a nyolc utasításból álló sorozat látható a 4.44. ábrán, csak most sorrendtől eltérő kiosztás és sorrendtől eltérő befejezés is megengedett.

Az első különbség a 3. ciklusban van. Bár I4 elakadt, dekódolhatjuk és kioszthatjuk I5-öt, mivel ez nem ellenkezik egyik függőben lévő utasítással sem. Mindamellett, egy utasítás átugrása új problémát okoz. Tegyük fel, hogy I5 használt egy olyan operandust, amelyet az átugrott utasítás, I4 számol ki. A jelenlegi eredményjelzővel ezt nem vennénk észre. Következésképpen ki kell bővítenünk az eredményjelzőt, hogy nyomon kövesse azokat a tárolásokat, amelyeket az átugrott utasítások végeznek. Egy második, regiszterenként 1 bites bittérkép hozzáadásával nyomon követhetjük az elakadt utasítások által végrehajtott tárolásokat. (Ezek a számlálók nincsenek az ábrán.) Az utasításkiosztás szabályát most ki kell

					Olvasott regiszterek								Írt regiszterek							
Cik	**#**	**Dekódolt**	**Ind**	**Bef**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3 = R0 × R1	1		1	1										1				
	2	R4 = R0 + R2	2		2	1	1									1	1			
2	3	R5 = R0 + R1	3		3	2	1									1	1	1		
	4	R6 = R1 + R4	–		3	2	1									1	1	1		
3	5	R7 = R1 × R2	5		3	3	2									1	1	1		1
	6	S1 = R0 – R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3 = R3 × S1	–		3	4	2		1							1		1	1	1
	8	S2 = R4 + R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

4.44. ábra. *Egy szuperskaláris CPU működése sorrendtől eltérő kiosztással és sorrendtől eltérő befejezéssel*

terjesztenünk, hogy megakadályozzuk minden olyan utasítás kiosztását, amelynek operandusa ütemezés szerint egy olyan utasítással tárolódik, ami a sorban előtte van, de átugrottuk.

Most térjünk vissza a 4.43. ábra I6, I7 és I8 utasítására. Itt látjuk, hogy I6 kiszámol egy értéket R1-be, amit I7 használ. Azonban azt is látjuk, hogy az értéket soha többé nem használjuk, mivel I8 felülírja R1-et. Nincs valós ok arra, hogy R1-et I6 eredményének tárolására fenntartsuk. Ráadásul R1 borzasztó rossz választás átmeneti regiszternek, bár teljességgel elfogadható egy fordítóprogram vagy programozó számára, aki az utasítás átlapolás nélküli, sorrend szerinti végrehajtás elvét használja.

A 4.44. ábrán egy új módszert vezetünk be ennek a problémának a megoldására: a **regiszterátnevezést**. Az előrelátó dekódoló egység R1 használatát I6-ban (3. ciklus) és I7-ben (4. ciklus) egy titkos regiszterre, S1-re váltja, ami nem látható a programozó számára. Most I6-ot kioszthatjuk I5-tel párhuzamosan. A modern CPU-k gyakran titkos regiszterek tucatjait használják regiszterátnevezéssel. Ez a módszer gyakran kiküszöböli a WAR és WAW függőségeket.

I8-nál ismét regiszterátnevezést használunk. Ezúttal R1-et S2-nek nevezzük át, így az összeadás elkezdődhet, mielőtt R1 szabaddá válna, a 6. ciklus végére. Ha kiderül, hogy az eredménynek ekkor valóban R1-ben kell lennie, akkor tartalma még mindig idejében visszamásolható ide. Még jobb, ha minden olyan későbbi utasítás, amelynek R1-re szüksége van, a forrását abban az átnevezett regiszterben találja meg, ahova ténylegesen tárolva van (S2). Mindkét esetben ezzel a módszerrel az I8 utasítás előbb elindulhat, mint eddig.

Sok valódi gépnél az átnevezés mélyen be van ágyazva a regiszterek szervezésének rendszerébe. Sok titkos regiszter van, és egy táblázat, amely leképezi a programozó számára látható regisztereket a titkos regiszterekre. Mondjuk az R0-ként használt valódi regiszter helyét úgy állapítjuk meg, hogy megnézzük ennek a leképezési táblázatnak a 0. bejegyzését. Így nincs valódi R0 regiszter, csak az R0 név és a titkos regiszterek egyike közötti kapcsolat. Ez a kapcsolat gyakran változik a végrehajtás során, hogy elkerüljük a függőségeket.

A 4.44. ábrán a negyedik oszlopot tanulmányozva vegyük észre, hogy az utasításokat nem sorrend szerint osztottuk ki. Ugyancsak nem sorrend szerint fejeződtek be. Ebből a példából egyszerű a következtetés: sorrendtől eltérő végrehajtást és regiszterátnevezést használva képesek voltunk a számítások sebességét a kétszeresére emelni.

4.5.4. Feltételezett végrehajtás

Az előző részben bevezettük az utasításátrendezés elvét a teljesítmény javításának érdekében. Bár kifejezetten nem említettük, a középpontban az egyetlenegy alapblokkon belüli utasításátrendezés volt. Ideje ezt a kérdést közelebbről megvizsgálni.

A számítógépes programok **alapblokkok**ra bonthatók szét, ahol minden alapblokk lineáris kódsorozatból áll, a tetején egy belépési ponttal, az alján pedig egy

kilépéssel. Egy alapblokk nem tartalmaz egyetlen vezérlési szerkezetet sem (például if vagy while utasításokat), ezért gépi nyelvre fordítása semmilyen elágazást nem tartalmaz. Az alapblokkok vezérlési utasításokkal vannak összekötve.

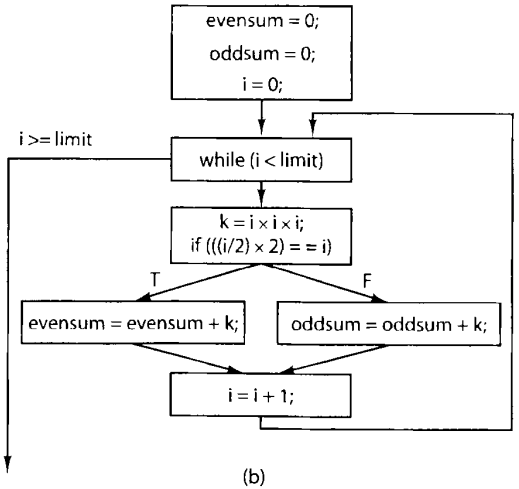
Egy program ebben a formában irányított gráfként ábrázolható, amint azt a 4.45. ábrán látjuk. Itt a páros és páratlan egész számok köbének összegét számoljuk egy bizonyos határig, és *evensum*-ban, illetve *oddsum*-ban gyűjtjük. Az egyes alapblokkokon belül az előző fejezet átrendező módszere nagyszerűen működik.

A gond az, hogy a legtöbb alapblokk rövid, és nincs bennük elegendő párhuzamosság, hogy azokat hatékonyan kihasználhassuk. Tehát a következő lépés az, hogy megengedjük, hogy az átrendezés az alapblokkok határain átnyúljon, megkísérelve ezzel az összes kiosztási hely feltöltését. A legnagyobb nyereségek akkor következnek be, ha egy feltehetően lassú művelet a gráfban felfelé mozdulhat, hogy korábban elinduljon. Ez lehet LOAD utasítás, lebegőpontos művelet vagy akár egy hosszú függőségi lánc kezdete. A kód egy elágazás fölötti előremozdítását **emelés**nek nevezzük.

Képzeljük el, hogy a 4.45. ábrán minden változót regiszterekben tartunk, kivéve *evensum*-t és *oddsum*-t (regiszterek hiányában). Akkor értelme lenne ezek LOAD utasításait a ciklus elejére vinni, *k* kiszámítása elé. Ha elég korán elindítjuk, akkor az értékek elérhetők lesznek, amikor szükség lesz rájuk. Természetesen minden egyes iterációnál közülük csak az egyik szükséges, így a másik LOAD felesleges, de ha a gyorsítótár és a memória csővezetékes, és van elérhető kiosztási hely, akkor még megérheti ezt elvégezni. Egy kód végrehajtását, mielőtt még tudnánk, hogy egyáltalán szükség lesz-e rá, **feltételezett végrehajtás**nak nevezzük. E módszer alkalmazásához támogatás kell a fordítóprogramtól, és a hardvertől is némi architekturális bővítés. Normális esetben az utasításoknak az alapblokk határokon ke-

```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i × i × i;
    if (((i/2) × 2) == i)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

(a)



(b)

4.45. ábra. *(a) Egy programrészlet. (b) A megfelelő alapblokk gráf*

resztül történő átrendezése meghaladja a hardver képességeit, így egyértelműen a fordítóprogramnak kell az utasítást mozgatnia.

A feltételezett végrehajtás néhány érdekes problémát vet fel. Először is, alapvető, hogy egyik feltételezett végrehajtásnak sem lehetnek visszavonhatatlan eredményei, hiszen később kiderülhet, hogy azokat nem kellett volna végrehajtani. A 4.45. ábrán nagyon jó, hogy betöltjük *evensum*-ot és *oddsum*-ot, és az is nagyon jó, hogy elvégezzük az összeadást, amint *k* elérhetővé válik (éppen az if utasítás előtt), de az eredményeket nem jó tárolni a memóriába. Bonyolultabb kódsorozatoknál gyakori módszer a feltételezett végrehajtás során használt összes célregiszter átnevezése, megelőzve ezzel, hogy a feltételezett végrehajtás felülírja a regisztereket, mielőtt még kiderülne, hogy egyáltalán szükség van-e rá. Ezáltal csak a firkáló regiszterek módosulnak, így nincs probléma, ha a kódra végső soron nincs szükség. Ha a kódra szükség van, a firkáló regisztereket átmásoljuk a valódi célregiszterekbe. Bárki el tudja képzelni, hogy eredményjelzővel mindezek nyomon követése nem egyszerű, de ha van elegendő hardver, meg lehet csinálni.

Mindamellett további probléma is felmerül a feltételezett végrehajtásnál, amit nem oldható meg regiszterátnevezéssel. Mi történik, ha a feltételezetten végrehajtott utasítás kivételes eseményt (exception) okoz? Fájdalmas, bár nem végzetes példa egy olyan LOAD utasítás, amelyik gyorsítótárhiányt okoz egy nagy gyorsítósorméretttel (mondjuk, 256 bájttal) rendelkező gépen, és a memória sokkal lassabb, mint a CPU és a gyorsítótár. Ha egy LOAD, ami valóban szükséges, hirtelen, több ciklusra megállítja a gépet, amíg a gyorsítósor betöltődik, nos, ilyen az élet, a szóra szükség van. Viszont egy olyan szó betöltése végett megállítani a gépet, amelyről kiderül, hogy nincs is rá szükség, nem kívánatos eredményre vezet. Ezekből az „optimalizálás"-okból túl sok a CPU-t lassabbá teheti, mintha egyáltalán nem lennének. (Ha a gépnek virtuális memóriája van, amiről a 6. fejezetben ejtünk majd szót, akkor egy feltételezett LOAD laphiányt is okozhat, ami a szükséges lap behozására egy lemezműveletet követel meg. A hamis laphiány szörnyű hatással van a hatékonyságra, ezért fontos, hogy elkerüljük.)

Számos modern gépnek van egy különleges SPECULATIVE-LOAD utasítása, amely megpróbálja a szót a gyorsítótárból betölteni, de ha nincs ott, akkor feladja. Ha az érték ott van, amikor tényleg szükség van rá, akkor használható, de ha nincs ott, és szükség lenne rá, a hardvernek később kell betöltenie. Ha az értékre történetesen nincs szükség, a gyorsítótárhiányért nem kell büntetést fizetni.

Sokkal rosszabb helyzet mutatható be a következő utasítással:

```
if (x > 0) z = y/x;
```

ahol *x*, *y* és *z* lebegőpontos változók. Tegyük fel, hogy az összes változót előre betöltöttük regiszterekbe, és hogy a (lassú) lebegőpontos osztást átemeltük az if vizsgálaton. Sajnos *x* éppen 0, és az ebből következő nullával való osztás csapdája befejezi a programot. A csapda eredménye az, hogy a feltételes végrehajtás egy helyes program meghibásodását okozta. Még rosszabb, ha a programozó világos kódot tett be, hogy elkerülje ezt a helyzetet, és az valahogyan mégis megtörtént. Ez a helyzet valószínűleg nem tesz boldoggá egyetlen programozót sem.

Az egyik lehetséges megoldás, hogy legyen speciális változatuk azoknak az utasításoknak, amelyek kivételes eseményt okozhatnak. Vegyünk hozzá minden regiszterhez egy bitet, nevezetesen a **mérgezésbit**et. Ha egy feltételezett utasítás hibás, akkor ahelyett, hogy csapdát okozna, állítsa be az eredményregiszter mérgezésbitjét. Ha azt a regisztert később egy rendes utasítás érinti, a csapda létrejön (ahogy kell). Mindazonáltal, ha az eredményt soha nem használjuk, a mérgezésbitet végül töröljük, és nem történik baj.

4.6. Példák a mikroarchitektúra-szintre

Ebben a részben rövid példákat mutatunk be három csúcsprocesszorra, megmutatva, hogy e processzorok hogyan alkalmazzák a jelen fejezetben tárgyalt elveket. Ezek szükségszerűen rövidek lesznek, mivel a valódi gépek roppant bonyolultak, kapuk millióit tartalmazzák. Példáink ugyanazok, mint eddig: a Pentium 4, az UltraSPARC III és a 8051.

4.6.1. A Pentium 4 CPU mikroarchitektúrája

Kívülről a Pentium 4 hagyományos CISC gépnek tűnik, nagy és nehezen kezelhető utasításrendszerrel, amely támogatja a 8, 16 és 32 bites egész műveleteket, valamint a 32 és 64 bites lebegőpontos műveleteket is. Csupán 8 látható regisztere van, és nincs két olyan, amely teljesen megegyezne. Az utasítások hossza 1 és 17 bájt között változik. Röviden, olyan örökölt architektúra, ami látszólag mindent rosszul csinál.

Belülről azonban a Pentium 4 egy modern, egyszerű, mélyen csővezetékezett RISC-magot tartalmaz, mely különlegesen gyors órafrekvencián fut, és ez valószínűleg a következő években tovább növelhető. Egészen bámulatos, hogy az Intel mérnökei képesek voltak egy régi architektúrára korszerű processzort építeni. Ebben a részben a Pentium 4 mikroarchitektúrájának működését vizsgáljuk meg.

A NetBurst mikroarchitektúra áttekintése

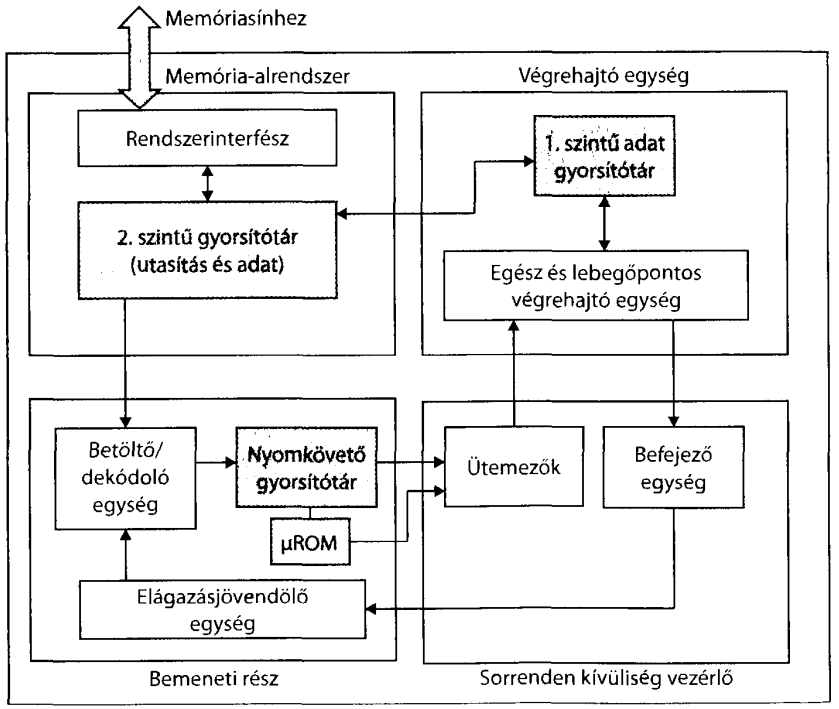
A Pentium 4 mikroarchitektúrája, melyet **NetBurst** mikroarchitektúrának neveznek, teljesen elszakad a Pentium Pro, Pentium II és Pentium III-nál használt korábbi P6 mikroarchitektúrától, és alapját képezi annak, amire az Intel a következő néhány évben építeni fog. A Pentium 4 mikroarchitektúrájának vázlatos áttekintése a 4.46. ábrán található. A diagram többé-kevésbé megfelel az 1.12. ábrának.

A Pentium 4 négy fő részből áll: a memória-alrendszer, a bemeneti rész, a sorrenden kívüliséget vezérlő és a végrehajtó egység. Vizsgáljuk meg ezeket egyenként, az ábrán a bal felsővel kezdve, az óramutató járásával ellentétesen haladva.

A memória-alrendszer tartalmazza az egyesített L2 (2. szintű) gyorsítótárat és a külső RAM-nak a memóriasínen való eléréséhez szükséges logikát. Az L2 az első generációs Pentium 4-ben 256 KB volt, a másodikban 512 KB, a harmadikban pedig 1 MB. Az L2 gyorsítótár 8 utas halmazkezelésű gyorsítótár, mely 128 bájtos gyorsítósoron alapszik. Ha egy keresett bejegyzés nem található meg az L2 gyorsítótárban, két 64 bájtos átvitelt kezdeményez a főmemóriából, hogy betöltse a szükséges blokkokat. Az L2 gyorsítótár egy visszaíró szervezésű gyorsítótár. Ez azt jelenti, hogy sor módosulásakor az új tartalom nem kerül vissza a memóriába, amíg a sor a memóriába nem töltődik.

A gyorsítótárhoz kapcsolódik egy előre betöltő egység (nem szerepel az ábrán), amely megpróbálja előre betölteni az adatokat a főmemóriából az L2 gyorsítótárba, még mielőtt arra szükség lenne. Az L2 gyorsítótárból az adatok nagy sebességgel áramolhatnak más gyorsítótárakba. Egy új L2 gyorsítótár betöltése minden második óraciklusban kezdődhet, így például 3 GHz-es órajel mellett, elméletileg az L2 gyorsítótár akár 1,5 milliárd 64 bájtos blokkot is szolgáltathat másodpercenként a többi gyorsítótárnak, ami 96 GB/s sávszélességet jelent.

A memória-alrendszer alatt a 4.46. ábrán a bemeneti rész található, amelyik betölti az utasításokat az L2 gyorsítótárból, és dekódolja a programnak megfelelő sorrendben. Minden Pentium 4 ISA-utasítást lebont RISC-szerű mikroműveletek sorozatára. Az egyszerűbb utasításokhoz a betöltő/dekódoló egység határoz-



4.46. ábra. *A Pentium 4 blokkdiagramja*

za meg mely mikroműveletek szükségesek. A bonyolultabbaknál a mikroművelet sorozatot a mikro-ROM-ból (μROM) keresi ki. Akár így, akár úgy, minden Pentium 4 ISA-utasítás mikroműveletek sorozatává konvertálódik a lapka RISC-magja számára. Ezzel a mechanizmussal hidalták át az ősi CISC-utasításhalmaz és a modern RISC-adatút közötti szakadékot.

A dekódolt mikroműveletek bekerülnek a **nyomkövető gyorsítótár**ba, amely az 1. szintű utasítás-gyorsítótár. Azáltal, hogy a dekódolt mikroműveleteket gyorsítótárazza, és nem a feldolgozatlan utasításokat, egy nyomkövető gyorsítótárbeli utasítás végrehajtásakor nincs szükség másodszori dekódolásra. Ez a megközelítés az egyik kulcsfontosságú különbség a NetBurst mikroarchitektúra és a P6 között (az utóbbi csak a Pentium 4 utasításokat tárolta az 1. szintű utasítás-gyorsítótárban). Az elágazásjövendölésre is itt kerül sor.

Az utasítások a nyomkövető gyorsítótárból a program által előírt sorrendben kerülnek az ütemezőbe, de onnan nem feltétlenül a program szerinti sorrendben kerülnek tovább. Ha egy nem végrehajtható mikroművelet következik, az ütemező tárolja, de folytatja az utasításfolyam feldolgozását, és a rákövetkező utasítások közül is kioszt olyanokat, melyek mindegyikének erőforrásai (regiszterek, működési egységek stb.) elérhetők. A regiszterátnevezés is itt történik, hogy késedelem nélkül továbbengedjük a WAR és WAW függőséggel rendelkező utasításokat.

Habár az utasításokat sorrenden kívül is ki lehet osztani, a Pentium 4 architektúrának pontos megszakításokra vonatkozó követelménye azt jelenti, hogy az ISA-utasításoknak sorrendben kell befejeződni (azaz elérhetővé tenni az eredményüket). Ezt a feladatot a befejező egység kezeli.

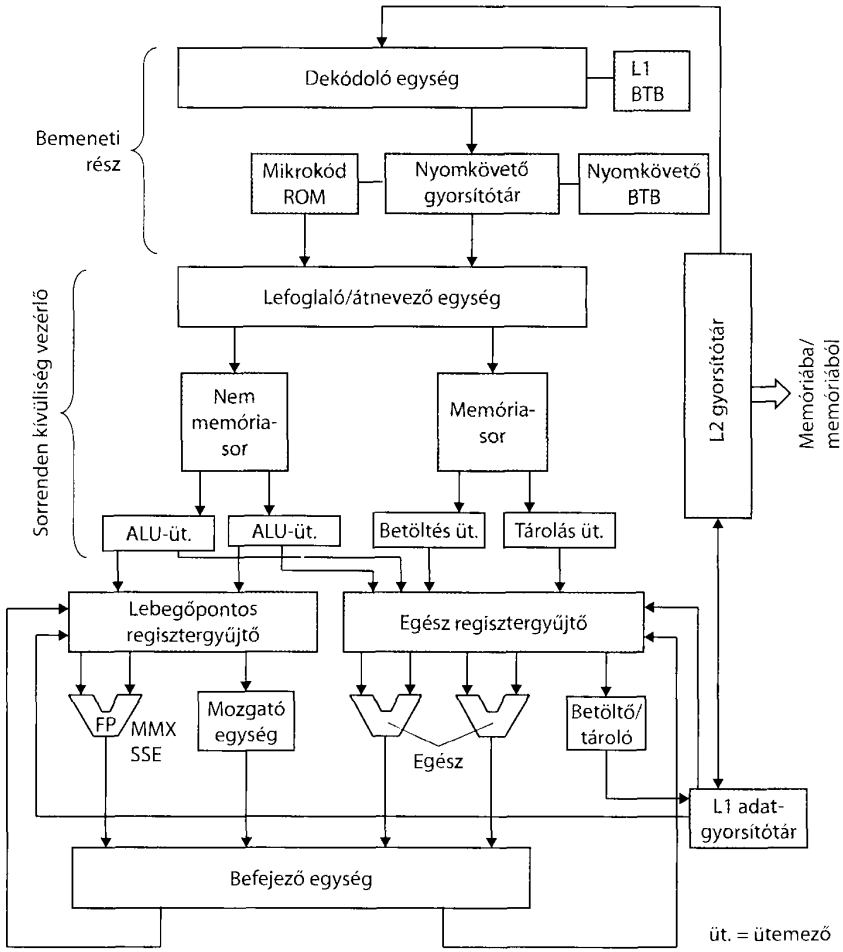
Az ábra jobb felső negyedében találhatók a végrehajtó egységek, amelyek az egész, a lebegő pontos, és a speciális műveleteket hajtják végre. Több végrehajtó egység is létezik, és ezek párhuzamosan futnak. Adataikat a regiszterekből és az L1 adat gyorsítótárból kapják.

A NetBurst csővezeték

A 4.47. ábra egy részletesebb változata a NetBurst mikroarchitektúrának, bemutatva a csővezetéket. Felül van a bemeneti rész, aminek az a feladata, hogy betöltse az utasításokat a memóriából, és előkészítse a végrehajtást. A bemeneti rész az új Pentium 4 utasításokat az L2 gyorsítótárból kapja, egyszerre 64 bitet. Ezeket dekódolja mikroműveletekre és tárolja a nyomkövető gyorsítótárban, mely 12K mikroműveletet tartalmaz. Ilyen méretű nyomkövető gyorsítótár teljesítménye összehasonlítható egy 8–16 KB méretű hagyományos L1 gyorsítótárral. A nyomkövető gyorsítótár hat mikroműveletet csoportosít minden nyomkövető sorba. A nyomkövető sor mikroműveletei sorrendben kerülnek végrehajtásra, annak ellenére, hogy olyan Pentium 4 ISA-utasításokból származnak, melyek több ezer bájt távolságra lehetnek. Hosszabb mikroművelet-sorozathoz több nyomkövető sort is össze lehet kapcsolni.

Ha egy Pentium ISA-utasítás több mint négy mikroműveletet igényel, nem dekódolódik a nyomkövető gyorsítótárba. Ehelyett egy jelzés kerül oda, mely utasítja a logikát, hogy keresse ki a mikroműveleteket a mikrokód ROM-ból. Ilyen módon a mikroműveletek úgy kerülnek a sorrenden kívüliséget vezérlő logikához, hogy vagy már a dekódolt ISA-utasítások kerülnek felhasználásra a nyomkövető gyorsítótárból, vagy pedig menet közben kerülnek ki a mikrokód ROM-ból az olyan bonyolult ISA-utasítások, mint például a string mozgatás.

Ha a dekódoló egység feltételes elágazáshoz ér, kikeresi a jövendölt címet az L1 **BTB**-ből (**Branch Target Buffer, elágazási célpuffer**) és a jövendölt címtől folytatja a dekódolást. Az L1 BTB tárolja az utolsó 4K elágazást. Ha az elágazó utasítás nem szerepel a táblázatban, statikus jövendölést használ. Egy visszafelé történő elágazásról feltételezi, hogy egy ciklusnak a része és, az ugrást végre is kell haj-



4.47. ábra. *A Pentium 4 adatútjának leegyszerűsített ábrája*

tani. Az ilyen statikus elágazásjövendölés pontossága rendkívül magas. Az előre irányuló elágazást úgy kezeli, mintha egy if utasítás része lenne, és feltételezi, hogy nem kell végrehajtani. Az ilyen statikus elágazásjövendölés pontossága sokkal alacsonyabb, mint a visszafelé történő elágazásé. A **nyomkövető BTB**-t arra használja, hogy megjósolja, hova ugranak az elágazó mikroműveletek.

A csővezeték második részét, a sorrenden kívüliséget vezérlő logikát a nyomkövető gyorsítótár táplálja, és 12K mikroműveletet tartalmaz. Ahogy a mikroműveletek a bemeneti részből érkeznek, ciklusonként három, a **lefoglaló/átnevező egység** egy 128 bejegyzést tartalmazó táblába veszi fel őket, amit **ROB**-nak **(ReOrder Buffer, átrendező puffer)** hívnak. Ez a bejegyzés követi nyomon a mikroművelet állapotát, amíg be nem fejeződik. Ezután a lefoglaló/átnevező egység megvizsgálja, hogy a mikroművelet által igényelt erőforrások rendelkezésre állnak-e. Ha igen, a mikroművelet bekerül az egyik végrehajtási sorba. Ha a mikroműveletet nem lehet végrehajtani, akkor felfüggeszti, de a rákövetkező mikroműveletek feldolgozása folytatódik, ami a mikroműveletek sorrenden kívüli végrehajtásához vezet. Ezt a stratégiát úgy tervezték meg, hogy foglalja le a működési egységeket, amennyire csak lehetséges. Akár 126 utasítás is lehet egyszerre feldolgozás alatt, melyek közül 48 végezhet memóriából betöltést, és 24 végezhet memóriába írást.

Néha egy mikroművelet elakad, mert olyan regiszterbe kell írnia, amelyet egy korábbi mikroművelet olvas vagy ír. Ezeket a konfliktusokat, ahogyan azt már korábban láttuk, rendre WAR és WAW függőségeknek nevezik. Egy mikroművelet célregiszterét átnevezhetjük, hogy az eredményét a 120 firkáló regiszter valamelyikébe írja a tervezett, ám még mindig foglalt célregiszter helyett. Ezáltal lehetővé válik, hogy azonnal ütemezhessük a mikroművelet végrehajtását. Ha nincsen elérhető firkáló regiszter, vagy a mikroművelet RAW függőséggel rendelkezik (amit soha nem lehet kiküszöbölni), a lefoglaló feljegyzi a probléma természetét a ROB bejegyzésben. Ha később az összes szükséges erőforrás rendelkezésre áll, a mikroművelet átkerül az egyik végrehajtási sorba.

A lefoglaló/átnevező egység a két várakozási sor valamelyikébe teszi át a mikroműveleteket, amikor készen állnak a végrehajtásra. A másik oldalon négy **ütemező** található, amelyek kiveszik a mikorműveleteket. Minden ütemező erőforrásokat ütemez az alábbiak szerint:

1. ütemező: ALU 1-et és a lebegőpontos mozgató egységet.
2. ütemező: ALU 2-t és a lebegőpontos végrehajtó egységet.
3. ütemező: betöltő utasításokat.
4. ütemező: tároló utasításokat.

Mivel az ütemezők és az ALU-k a névleges órafrekvencia kétszeresén futnak, az első két ütemező óraciklusonként két mikroműveletet tud elküldeni. Két, dupla sebességű, egész aritmetikájú ALU-val egy 3 GHz-es Pentium 4 12 milliárd egész műveletre képes másodpercenként. Ez a nagyon nagy sebesség az oka, hogy a sorrenden kívüliség vezérlője nagy nehézségekbe ütközik, hogy munkát találjon az ALU-k számára. A betöltő és tároló utasítások közösen használnak egy dupla frekvenciás végrehajtó egységet, amely egy-egy betöltést és tárolást képes végrehajtani minden óraciklusban. Tehát a lebegő pontos műveleteken felül legjobb esetben hat, egész típusú mikroműveletet lehet kiosztani óraciklusonként.

A két, egész aritmetikájú ALU nem teljesen egyezik meg. Az ALU 1 képes az összes aritmetikai és logikai művelet, valamint az elágazás végrehajtására. Az ALU 2 csak az összeadó, kivonó, léptető és forgató utasításokat képes végrehajtani. Hasonlóan a két lebegőpontos egység sem egyezik meg. Az első a mozgató és az SSE-műveleteket tudja végrehajtani. A második a lebegőpontos aritmetikát, az MMX- utasításokat és az SSE-utasításokat hajtja végre.

Az ALU-t és a lebegőpontos egységet két, 128 elemű regisztergyűjtő szolgálja ki: egy az egész, egy pedig a lebegőpontos egységet. Ezek szolgáltatják a végrehajtandó utasítások operandusait, és raktározzák az eredményeket. A regiszterátnevezés következtében nyolc tartalmazza az ISA-szinten is látható regisztereket (EAX, EBX, ECX, EDX stb.), de hogy melyik az a nyolc, amelyikben az „igazi" érték van, az időben változik, ahogyan a futás közben módosul a leképezés.

Az L1 adatgyorsítótár része a nagy sebességű (2×) áramkörnek. Ez 8 KB-os gyorsítótár, amely egész és lebegőpontos számokat, valamint más típusú adatokat tárol. A nyomkövető gyorsítótárral ellentétben ez egyáltalán nincs dekódolva. A memóriabeli bájtok pontos másolatát tartalmazza. Az L1 adatgyorsítótár 4 utas halmazkezelésű gyorsítótár, 64 bájtos gyorsítósorral. Írásáteresztő gyorsítótár, ami azt jelenti, hogy egy gyorsítósor módosításakor a sort azonnal visszamásolja az L2 gyorsítótárba. A gyorsítótár egy olvasó és egy író műveletet képes kezelni egy óraciklus alatt. Ha a keresett szó nem található meg az L1 gyorsítótárban, kérést küld az L2 gyorsítótárnak, amely vagy azonnal válaszol, vagy előbb betölti a gyorsítósort a memóriából, és csak azután. Egyszerre legfeljebb négy kérés lehet folyamatban az L1 gyorsítótárból az L2 gyorsítótárba.

Mivel a mikroműveletek sorrenden kívül is végrehajthatók, nem megengedett az L1 gyorsítótárba történő tárolás, amíg a tároló utasítást megelőző összes utasítás be nem fejeződött. A **befejező egység** feladata, hogy az utasítások sorrendben fejeződjenek be, és hogy helyüket nyilvántartsa. Megszakítás esetén a még befejezetlen utasításokat elveti, ezáltal a Pentium 4 megőrzi azt a tulajdonságát, hogy megszakításkor egy bizonyos ponttal bezárólag minden utasítás lezárult, és hogy ezen túl semelyik utasításnak sincsen hatása.

Ha egy tároló utasítás befejeződött, de még folyamatban vannak az azt megelőző utasítások, az L1 gyorsítótárat nem lehet módosítani, így az eredmények a folyamatban lévő tárolások pufferébe (pending-store buffer) kerülnek. Ennek a puffernek 24 bejegyzése van, ami megfelel a 24 tároló műveletnek, ami egyszerre végrehajtás alatt lehet. Ha egy későbbi betöltés tárolt adatot próbál olvasni, azt az utasítás még akkor is megkaphatja a folyamatban lévő tárolások pufferéből, ha az L1 gyorsítótárban még nem érhető el. Ezt a folyamatot **tárolás utáni betöltés**nek nevezzük.

Mostanra világosnak kell lennie, hogy a Pentium 4-nek nagyon összetett mikroarchitektúrája van, ahol a tervezést az vezérelte, hogy a régi Pentium-utasításhalmaz végrehajtható legyen egy modern, erősen csővezetékezett RISC-magon is. Ezt a célt úgy valósítja meg, hogy feldarabolja a Pentium-utasításokat mikroműveletekre, amelyeket gyorsítótárban tárol, majd hármasával átadja a csővezetéknek,

ahol a végrehajtás olyan ALU-kon történik, amelyek optimális körülmények között akár hat mikroműveletre is képesek ciklusonként. A mikroműveletek végrehajtása sorrendtől eltérő módon történik, de a befejezés és az eredmény tárolása az L1 és L2 gyorsítótárakban már sorrendben zajlik. A NetBurst architektúráról további információkat lásd (Hinton és társai, 2004).

4.6.2. Az UltraSPARC III Cu CPU-jának mikroarchitektúrája

Az UltraSPARC-sorozat a SPARC architektúra 9. változatának Sun megvalósítása. A felhasználó vagy a programozó szemszögéből (azaz ISA-szinten) a különböző modellek eléggé hasonlók, főleg teljesítményben és árban különböznek. A mikroarchitektúra szintjén azonban jelentősek a különbségek. Ebben a fejezetben az UltraSPARC III Cu processzort fogjuk bemutatni. A megnevezésben lévő Cu, a lapka vezetékeinél használt rézre utal, ellentétben az alumíniumvezetékezéssel, amit elődjénél használtak. A réznek kisebb az ellenállása, mint az alumíniumnak, ami vékonyabb vezetékeket és gyorsabb működést tesz lehetővé.

Az UltraSPARC III Cu egy teljesen 64 bites gép, 64 bites regiszterekkel és 64 bites adatúttal, bár a 8. változatú (vagyis 32 bites) SPARC-okkal való visszafelé kompatibilitás miatt 32 bites operandusokat is tud kezelni, és valóban változtatás nélkül futtat 32 bites SPARC-szoftvert. Bár a belső architektúra 64 bites, a memóriasín 128 bit széles, a Pentium 4-hez hasonlóan, amelynek 32 bites architektúrája és 64 bites memóriasínje van, de mindkét esetben a sín egy generációval későbbi, mint maga a CPU.

A Pentium 4-gyel ellentétben az UltraSPARC egy valódi RISC-architektúra, ami azt jelenti, hogy nincs szüksége arra a bonyolult mechanizmusra, ami a végrehajtáshoz átalakítja a régi CISC-utasításokat mikroműveletekre. A gépi utasítások maguk a mikroműveletek. Azonban az utóbbi években grafikai és multimédia-utasításokkal egészítették ki a processzort, melyek végrehajtása speciális hardverberendezéseket igényel.

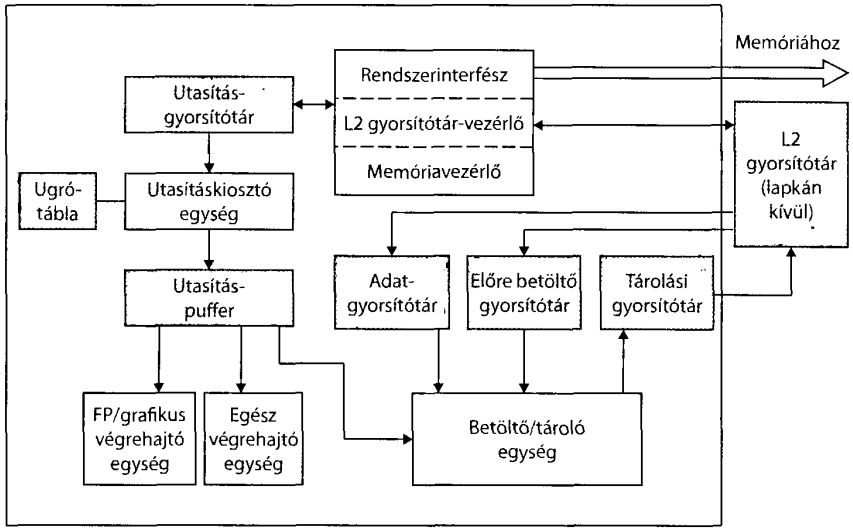
Az UltraSPARC III Cu mikroarchitektúrájának áttekintése

Az UltraSPARC III Cu blokkdiagramja a 4.48. ábrán látható. Egészében véve, sokkal egyszerűbb, mint a Pentium 4 NetBurst mikroarchitektúrája, mert az UltraSPARC-nak egyszerűbb ISA-architektúrát kell megvalósítania. Mindemellett néhány lényeges komponens hasonlít a Pentium 4-nél használtakhoz. Ezeket a hasonlóságokat többnyire a technológia, illetve a gazdaságosság vezérelte. Például, amikor ezeket a lapkákat tervezték, ésszerűnek számított a 8–16 KB méretű L1 adatgyorsítótár, ezért lettek ekkorák. Ha valamikor a jövőben technológiai és gazdasági szempontból egy 64 MB-os L1 gyorsítótár számít majd ésszerűnek, akkor minden CPU-ban az lesz. Ezzel ellentétben a különbségek többnyire a régi CISC-utasításhalmaz és a modern RISC-mag közötti különbségek áthidalásából, illetve ennek hiányából adódnak.

A 4.48. ábra bal felső részében található a 32 KB-os 4 utas halmazkezelésű utasítás-gyorsítótár, mely 32 bájtos gyorsítósort használ. Mivel a legtöbb UltraSPARC-utasítás 4 bájtos, így körülbelül 8K utasításnak van itt hely, ami valamivel kisebb, mint a NetBurst nyomkövető gyorsítótára.

Az **utasításkiosztó egység** négy utasítást is elő tud készíteni óraciklusonként. Ha gyorsítótárhiány lép fel az L1 gyorsítótárban, kevesebb utasítást fog kiosztani. Ha egy feltételes elágazó utasításba ütközik, utánanéz a 16K bejegyzést tartalmazó **ugrótáblá**ban, hogy az a következő utasítást jövendöli, vagy a célcímen szereplőt. Ezen felül, az utasítás-gyorsítótárban lévő szavakhoz kapcsolt extrabit is segíti az elágazásjövendölést. Az előkészített utasítások átkerülnek egy 16 elemű utasításpufferbe, amely kisimítja az utasítások csővezetékekbe áramlását.

Az utasításpuffer kimenete bekerül az egész, a lebegőpontos és a betöltő/tároló egységekbe, amint az a 4.48. ábrán látszik. Az egész aritmetikájú végrehajtó egység tartalmaz két ALU-t és egy rövid csővezetéket is az elágazó utasításokhoz. Az ISA-regiszterek és néhány firkáló regiszter is itt található.



4.48. ábra. *Az UltraSparc III Cu blokkdiagramja*

A lebegőpontos egység 32 regisztert és három külön ALU-t tartalmaz rendre az összeadás/kivonás, a szorzás és az osztás számára. A grafikai utasítások is itt kerülnek végrehajtásra.

A betöltő/tároló egység kezeli a különböző betöltő és tároló utasításokat. Adatúttal rendelkezik három különböző gyorsítótárhoz is. Az **adatgyorsítótár** egy hagyományos 64 KB-os 4 utas halmazkezelésű L1 adatgyorsítótár, mely 32 bájtos gyorsítósort használ. A 2 KB-os **előre betöltő gyorsítótár** azért van jelen, mert az UltraSPARC ISA tartalmaz előre betöltő utasításokat, melyek lehetővé teszik a

fordító számára, hogy adatszavakat töltsenek be, mielőtt még szükség lenne rájuk. Ha a fordító úgy gondolja, szüksége lehet egy bizonyos szóra, előre betöltő utasítást adhat ki, aminek eredményeképpen a megcímzett gyorsítósor idő előtt bekerül az előre betöltő gyorsítótárba, felgyorsítva ezzel a szó elérését, amikor néhány utasítás múlva szükség lesz rá. Bizonyos körülmények között hardveres előre betöltés is történik, hogy fokozódjon az olyan örökölt programok teljesítménye, amelyek nem végeznek előre betöltést. A **tárolási gyorsítótár** egy kisméretű (2 KB) gyorsítótár, melyet arra használunk, hogy összekapcsoljuk az eredmények írását, és jobban kihasználjuk az L2 gyorsítótárba menő 256 bit széles sínt. Egyedüli feladata a teljesítmény növelése.

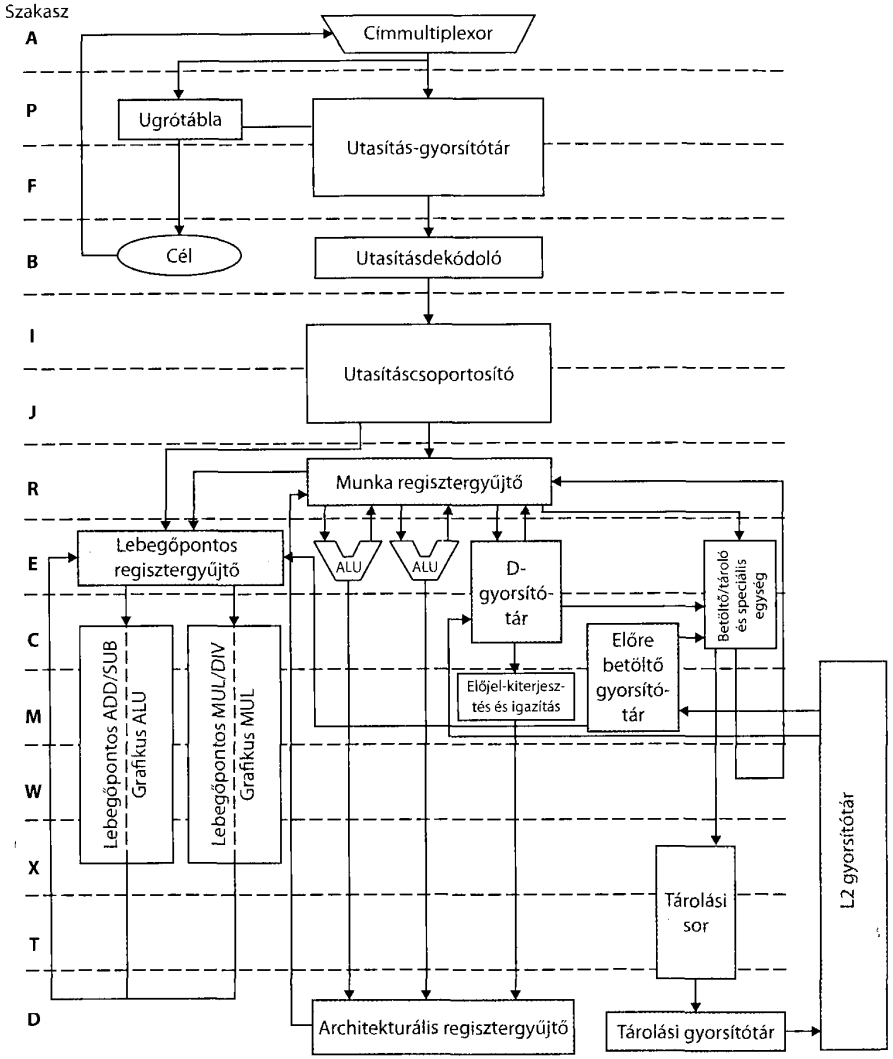
A lapka tartalmaz memóriavezérlő logikát is. Ez a logika három részre oszlik: a rendszerinterfészre, az L2 gyorsítótár-vezérlőre és a memóriavezérlőre. A rendszerinterfész 128 bit széles sínen keresztül kapcsolódik a memóriához. A külvilághoz érkező összes kérés, az L2 gyorsítótárat kivéve, ezen a felületen halad keresztül. A 43 bites fizikai memóriacímmel elméletben a főmemória egészen 8 TB-ig terjedhet, de az a nyomtatott áramkör, amire a processzort szerelték, 16 GB-ra korlátozza a memóriát. Az interfészt úgy tervezték, hogy több UltraSPARC-ot is lehessen ugyanahhoz a memóriához kapcsolni, hogy ezek **multiprocesszor**t alkossanak. A multiprocesszorokat a 8. fejezetben tárgyaljuk.

Az L2 gyorsítótár-vezérlő tartja a kapcsolatot az egyesített L2 gyorsítótárral, mely a CPU lapkáján kívül helyezkedik el. Azáltal, hogy az L2 gyorsítótár kívül van, 1, 4 és 8 MB is lehet a mérete. A gyorsítósor mérete függ a gyorsítótár méretétől, ami az 1 MB-os gyorsítótárra vonatkozó 64 bájttól a 8 MB-os gyorsítótárra vonatkozó 512 bájtig terjedhet. Összehasonlításul, a Pentium 4 L2 gyorsítótára a lapkán található, viszont helyhiány miatt 1 MB-ra korlátozódik. Két ellentétes nézet: az UltraSPARC jóval magasabb találati arányt érhet el az L2 gyorsítótárban, mint a Pentium 4 (mert nagyobb lehet a mérete), viszont az L2 gyorsítótár hozzáférése lassabb (mert nincs rajta a lapkán).

A memóriavezérlő képezi le a 64 bites virtuális címeket 43 bites fizikai címekre. Az UltraSPARC támogatja a virtuális memóriát (lásd 6. fejezet), ahol a lapok mérete 8 KB, 64 KB, 512 KB vagy 4 MB lehet. A leképezés gyorsítására speciális táblázatok, ún. **TLB**-k (**Translation Lookaside Buffer, lapkezelő segédpuffer**) állnak rendelkezésre, hogy össze lehessen hasonlítani az éppen hivatkozott virtuális címet a nemrégiben hivatkozottakkal. Három ilyen táblázat áll rendelkezésre az adatokhoz, hogy kényelmesen lehessen kezelni a különböző lapméreteket, és kettő az utasítások leképezéséhez.

Az UltraSPARC III Cu csővezetéke

Az UltraSPARC III Cu-nak tizennégy szakaszos csővezetéke van, mely egyszerűsített formában a 4.49. ábrán található. A 14 szakaszt – az ábra bal oldalán látható – *A*, ..., *D* betűkkel elnevezték el. Vizsgáljuk meg röviden az egyes szakaszokat. Az *A* (Address generation, címgeneráló) szakasz van a csővezeték elején. Azért van itt, hogy a következőnek betöltendő utasítás címét meghatározza. Normálisan



4.49. ábra. *Az UltraSPARC III Cu egyszerűsített ábrája*

ez az a cím, amelyik az aktuális utasítást követi. Azonban ez az egymást követő sorrend számos okból megszakadhat, például, ha egy korábbi utasítás egy elágazás, amiről azt jövendöltük, hogy végre kell hajtani, vagy egy csapda, vagy egy kiszolgálandó megszakítás. Mivel az elágazásjövendölés egy ciklus alatt nem hajtható végre, a feltételes elágazást követő utasítást mindig végrehajtja, függetlenül attól, hogy lesz-e elágazás.

A *P* (Preliminary fetch, előzetes betöltő) szakasz, az *A* szakasz által szolgáltatott címről kezd el betölteni ciklusonként legfeljebb négy utasítást az L1 I-gyorsítótárból. Itt utánanéz az ugrótáblában is, hátha az egyikük feltételes elágazás, és ha igen, akkor el kell-e ágazni. Az *F* (Fetch, betöltő) szakasz befejezi az utasítások betöltését az I-gyorsítótárból.

A *B* (Branch target, elágazási cél) szakasz dekódolja az imént betöltött utasításokat. Ha bármelyik közülük olyan elágazás, melyet a jövendölés szerint végre kell hajtani, akkor az információ ebben a szakaszban rendelkezésre áll, és vissza is kerül az *A* szakaszhoz, így vezérelve a további utasítások betöltését.

Az *I* (Instruction group formation, utasításcsoportosító) szakasz csoportokba sorolja a bejövő utasításokat attól függően, hogy a következő hat működési egység közül melyiket használják:

1. Egész ALU 1.
2. Egész ALU 2.
3. Lebegőpontos/grafikus ALU 1.
4. Lebegőpontos/grafikus ALU 2.
5. Elágazási csővezeték (nem szerepel az ábrán).
6. Betöltő, tároló és speciális műveletek.

A két egész aritmetikájú ALU nem teljesen egyforma, a két lebegőpontos ALU pedig jelentősen különbözik. Mindkét esetben különbözik az utasításhalmaz, amit az ALU-k végre tudnak hajtani. Az *I* szakasz az utasításokat aszerint rendezi, hogy melyik egységre van szükségük.

A *J* (Instruction stage grouping, utasításkiosztó) kiveszi az utasításokat az utasássorból, és előkészíti a végrehajtó egységek számára. Akár négy utasítást is tud továbbítani az *R* szakaszba ciklusonként. Az utasítások kiválasztása függ az elérhető működési egységektől. Például, két egész utasítást, egy lebegőpontos utasítást és egy betöltő vagy tároló utasítást ki lehet osztani egyszerre, de három egész utasítás nem bocsátható ki egy ciklusban.

Az *R* szakasz kikeresi az egész utasításokhoz szükséges regisztereket és továbbítja a lebegőpontos regiszter kéréseket a lebegőpontos regisztergyűjtőnek. A függőségi ellenőrzések is itt történnek. Ha egy szükséges regiszter nem áll rendelkezésre, mert még egy korábbi utasítás használja, úgy, hogy az ütközést eredményezne, az az utasítás, amelyiknek szüksége van a regiszterre, megakad, és a mögötte levőket is megállítja. A Pentium 4-gyel ellentétben az UltraSPARC III Cu soha nem oszt ki sorrendtől eltérően utasításokat.

Az *E* (Execution, végrehajtó) szakasz az, ahol az egész utasítások ténylegesen végre is hajtódnak. A legtöbb aritmetikai, logikai, és léptető utasítás az egész aritmetikájú ALU-t használja, és egy ciklus alatt be is fejeződik. Minden utasítás befejeztével azonnal frissül a regisztergyűjtő. Néhány bonyolultabb egész aritmetikájú utasítás a speciális egységbe kerül. A betöltő és tároló utasítások ebben a szakaszban kezdődnek, de nem itt fejeződnek be. A feltételes elágazó utasítások feldolgozása az *E* szakaszban kezdődik, és irányuk (elágazás/nem elágazás) is itt dől el. Hibás jövendölés esetén, jelet küld az *A* szakasznak, és érvényteleníti a csővezetéket.

A *C* (Cache, gyorsítótár) szakaszban zárul le az L1 gyorsítótár elérése. Azok az utasítások, amelyek a memóriából olvasnak (azaz a betöltő utasítások), itt szolgáltatják az eredményüket.

Az *M* (Miss, hiány) szakasz kezdi meg azoknak az adatoknak a megkercsését, amelyekre szükség van, de nincsenek az L1 gyorsítótárban. Ezután az L2 gyorsítótárban próbálkozik, és ha itt sincs, kibocsát egy memóriahivatkozást, mely számos ciklust igényel. Azok a bájtok, negyedszavak vagy félszavak is itt kerülnek feldolgozásra, melyek megtalálhatók az L1 gyorsítótárban, de igazítást és előjel-kiterjesztést kell rajtuk végezni. Az előre betöltő gyorsítótárból kiszolgálható lebegőpontos betöltések is itt kapják meg eredményüket. Az előre betöltő gyorsítótár bonyolult időzítési problémák miatt nem használatos egész típusú adatokhoz.

A *W* (Write, író) szakaszban a speciális egység eredményei bekerülnek a munka regisztergyűjtőbe.

Az *X* (eXtended, kiterjesztett) szakaszban fejeződik be a legtöbb lebegőpontos és grafikai utasítás. Az eredmények a következő utasítások számára a tárolás utáni betöltési technikával érhetők el, amíg az utasítások formálisan is le nem záródnak a *D* szakaszban.

A *T* (Trap, csapda) szakasz észleli az egész és lebegőpontos csapdákat. Ez a szakasz felelős azért, hogy a csapdák és megszakítások pontosan történjenek. Más szavakkal, csapda vagy megszakítás után a gép elmentett állapota olyan legyen, hogy minden utasítás a csapda vagy a megszakítás előtt teljesen befejeződött, és az utána jövők közül egy sem kezdődött el.

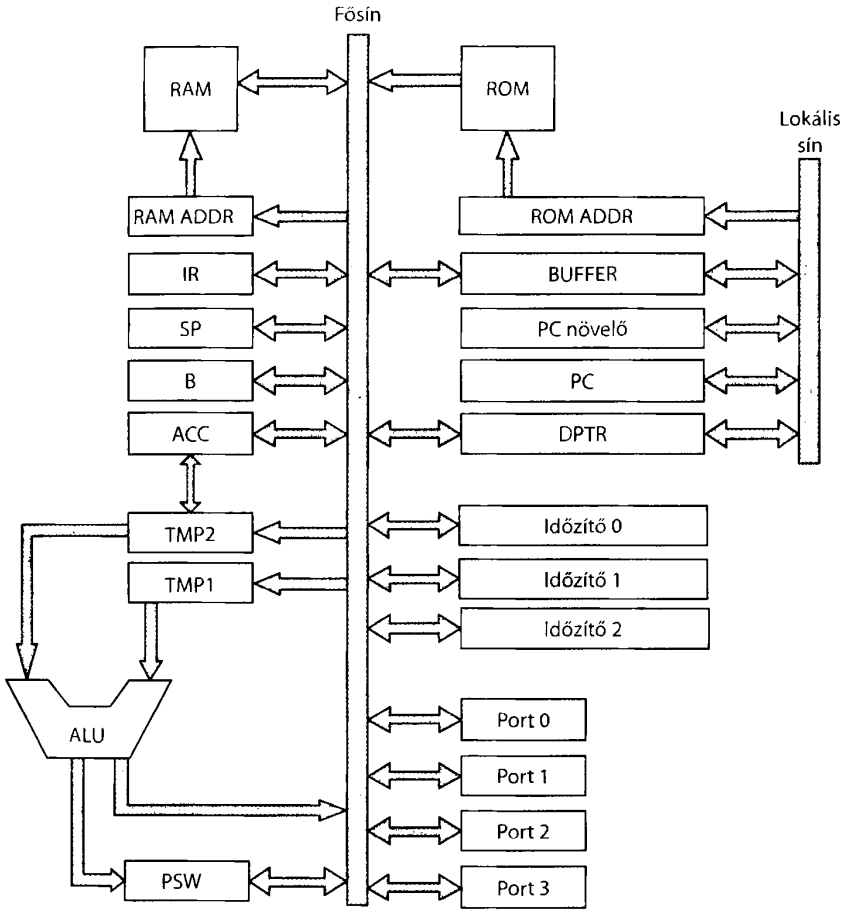
A *D* szakasz véglegesíti az egész és lebegőpontos regisztereket a megfelelő architekturális regisztergyűjtőkben. Csapda vagy megszakítás esetén ezek az értékek, valamint a nem a munka regiszterekben lévők lesznek láthatók. A regiszterek tárolása az architekturális gyűjtőben megegyezik a Pentiumnál használt befejezéssel. Ezen felül a *D* szakaszban befejeződő tároló utasítások a tárolási gyorsítótárba írják az eredményüket, és nem az L1 adatgyorsítótárba. Végül a gyorsítótár sorai visszakerülnek az L2 gyorsítótárba, elkerülve az L1 gyorsítótárat (L1-ben ez a gyorsítósor érvénytelenné válik). Ez a kezelés azt célozza, hogy könnyebb legyen UltraSPARC multiprocesszorokat építeni.

Az UltraSPARC III fenti leírása messze nem teljes, de elfogadható képet ad arról, hogyan működik, és miben tér el a Pentium 4 mikroarchitektúrától.

4.6.3. A 8051 CPU mikroarchitektúrája

Utolsó példánk a mikroarchitektúrára a 8051-es, amely a 4.50. ábrán látható. Ez jóval egyszerűbb a Pentiumnál és az UltraSPARC-nál. Az egyszerűség oka, hogy a lapka nagyon kicsi (60 000 tranzisztor), és még a csővezetékek elterjedése előtt tervezték. Továbbá az, hogy az elsődleges tervezési szempont az volt, hogy a lapka olcsó legyen, és nem az, hogy gyors. Az olcsóság és egyszerűség egymás jó barátai, míg az olcsóság és gyorsaság nem azok.

A 8051-es lelke a főSín. Számos regiszter kapcsolódik hozzá, melyek többségét a programok képesek írni és olvasni. Az ACC regiszter az **akkumulátor (ACCumula-**



4.50. ábra. *A 8051 mikroarchitektúrája*

tor), a fő aritmetikai regiszter, melyben a legtöbb számítás eredménye keletkezik. A legtöbb aritmetikai utasítás ezt használja. A B szorzás és osztás esetén kap szerepet, valamint ideiglenes eredmények tárolására is használható. Az SP regiszter a veremmutató, és mint a legtöbb gépben a verem tetejére mutat. Az IR regiszter az **utasításregiszter**. Az éppen végrehajtás alatt álló utasítást tartalmazza.

A TMP1 és TMP2 regiszterek az ALU tárolói. Egy ALU-művelet végrehajtásakor az operandusok először ezekbe a tárolókba kerülnek, az ALU csak ezután kezd működni. Az ALU kimenete bármelyik írható regiszterbe bekerülhet a fősínen keresztül. A PSW **(Program Status Word, programállapotszó)** -regiszterbe kerülnek a feltételkódok, melyek jelzik, ha az eredmény nulla, negatív stb.

A 8051-es külön memóriával rendelkezik az adatok és a kód számára. Az adat RAM 128 (8051) vagy 256 bájtos (8052), így a 8 bites RAM ADDR regiszter elegendően széles a címzésre. A RAM címzéséhez a kívánt bájt címét a RAM ADDR regiszterbe kell tenni, és elindítani a memóriaműveletet. A kódmemória 64 KB lehet (ha lapkán kívüli memóriát használunk), így a címzéséhez használt ROM ADDR 16 bit széles. Ugyanúgy a ROM ADDR regiszter a programkódot címzi a ROM-ban.

A DPTR **(Double PoinTeR, dupla szélességű mutató)** 16 bites regiszter a 16 bites címek kezelésére és összeállítására. A PC regiszter a 16 bites utasításszámláló, ami a következőnek betöltendő és végrehajtandó utasítás címét tartalmazza. A PC NÖVELŐ regiszter egy speciális hardver, amely pszeudoregiszterként működik. Ha belemásolják a PC-t, majd kiolvassák, az érték automatikusan növekszik. Sem a PC, sem a PC NÖVELŐ nem érhető el a fősínről. Végül, a PUFFER egy újabb 16 bites regiszter. Mindegyik 16 bites regiszter tulajdonképpen két 8 bites regiszterből áll, melyeket függetlenül is lehet kezelni, de a hatásuk 16 bites regiszterként érvényesül.

A 8051 rendelkezik még a lapkán elhelyezett három 16 bites időzítővel, melyek nélkülözhetetlenek valós idejű alkalmazások esetén. Van még négy 8 bites B/K portja, melyek lehetővé teszik a 8051 számára, hogy akár 32 külső gombot, lámpát, érzékelőt, indítókart stb. vezérelhessen. Éppen az időzítők és a B/K portok teszik lehetővé, hogy a 8051 kiegészítő lapkák nélkül is használható legyen beágyazott alkalmazások esetén.

A 8051 szinkronprocesszor, melynek legtöbb utasítása egy óraciklust igényel, bár némelyik többet. Minden óraciklust fel lehet osztani hat részre, melyeket **állapotok**nak nevezünk. Az első állapotban betöltődik a következő utasítás a ROM-ból, rákerül a fősínre és az IR regiszterbe. A második állapotban dekódolódik az utasítás, és növekszik a PC. A harmadik állapotban az operandusok előkészítése történik. A negyedik állapotban az egyik operandus rákerül a fősínre, általában azért, hogy TMP1-en keresztül az ALU felhasználhassa. Az ACC regisztert is ebben az állapotban lehet átmásolni TMP2-be, így az ALU mind a két bemenete készen áll. Az ötödik állapotban az ALU végrehajtja a műveletet. Végül a hatodik állapotban az ALU kimenete visszakerül a fősínen keresztül a rendeltetési helyére. Eközben a ROM ADDR regiszter előkészül a következő utasítás betöltésére.

Bár további részletekbe is lehetne bocsátkozni a 8051-essel kapcsolatban, a fenti leírás és a 4.50. ábra rávilágítanak az alapötletre. A 8051 egyetlen fősínnel rendelkezik (a lapka területének csökkentése miatt), regisztereinek halmaza heterogén, valamint három időzítő és négy port kapcsolódik a fősínre, továbbá van még néhány extra regisztere a lokális sínen. Minden adatútciklusban két operandus fut keresztül az ALU-n, és az eredmény visszakerül egy regiszterbe, ahogy az a korszerűbb számítógépeken is történik.

4.7. A Pentium, az UltraSPARC és a 8051 összehasonlítása

A három példánk nagyon különböző, de mégis van bennük némi közös. A Pentium 4-nek van egy régi CISC-utasításhalmaza, amelyet az Intel mérnökei legszívesebben bedobnának a San Franciscó-i öbölbe, ha ezzel nem szegnék meg Kalifornia vízszennyezési törvényeit. Az UltraSPARC III tiszta RISC-elképzelés, szegényes és egyszerű utasításhalmazzal. A 8051 egyszerű 8 bites processzor a beágyazott alkalmazások számára. Mégis mindegyik lelke a regiszterkészlet, és az egy vagy több ALU, amely elvégzi az egyszerű aritmetikai és logikai műveleteket a regisztereken.

Ezen nyilvánvaló külső különbségek ellenére a Pentium 4-nek és az UltraSPARC III-nak nagyon hasonlók a végrehajtó egységei. Mindkét végrehajtó egység olyan mikroműveleteket fogad el, amelyek egy műveleti kódot, két forrásregisztert és egy célregisztert tartalmaznak. Mindkettő végre tud hajtani egy mikroműveletet egy ciklus alatt. Mindkettőnek nagy tudású csővezetékei, elágazásjövendölése, valamint osztott I- és D-gyorsítótára van.

Ez a belső hasonlóság nem véletlen, és nem a Szilikon-völgy mérnökeinek végtelen munkaszeretetének köszönhető. Mint azt a Mic–3 és Mic–4 példákban láttuk, könnyű és természetes olyan csővezetékes adatutat építeni, amelyik vesz két forrásregisztert, átfuttatja az ALU-n, és az eredményt egy regiszterbe tárolja. A 4.34. ábra grafikusan mutatja ezt a csővezetéket. A jelenlegi technológiával ez a leghatékonyabb kivitelezés.

A fő különbség a Pentium 4 és az UltraSPARC III között az, ahogy ISA-utasításaikat a végrehajtó egységhez juttatják. A Pentium 4-nek szét kell bontania a CISC-utasításait, hogy – a végrehajtó egység elvárásainak megfelelően – háromregiszteres formátumúvá alakítsa őket. Ezt mutatja a 4.47. ábra – nagy utasítások szétszedése csinos, formás mikroműveletekre. Az UltraSPARC III-nak nem kell semmit tennie, mert eredeti utasításai már csinos, formás mikroműveletek. Ezért van az, hogy a legtöbb új ISA RISC típusú – így könnyebb a megfeleltetés az ISA-utasításhalmaz és a belső végrehajtó motor között.

Tanulságos, ha végső tervezésünket, a Mic–4-et összehasonlítjuk ezzel a két valóságos példával. A Mic–4-hez leginkább a Pentium 4 hasonlít. Mindkettőnek az a feladata, hogy egy nem RISC ISA-utasításhalmazt értelmezzen. Ezt mindkettő úgy csinálja, hogy az ISA-utasításokat egy műveleti kóddal szétbontja mikroműveletekre, két forrásregiszterrel és egy célregiszterrel. Mindkét esetben a mikroműveleteket lerakjuk egy sorba, későbbi végrehajtásra. A Mic–4 elgondolás szigorú sorrend szerinti kiosztást, sorrend szerinti végrehajtást és sorrend szerinti befejezést követel, míg a Pentium 4-nek sorrend szerinti kiosztás, sorrendtől eltérő végrehajtás és sorrend szerinti befejezés a vezérelve.

A Mic–4 és az UltraSPARC III valójában egyáltalán nem hasonlíthatók össze, mert az UltraSPARC III-nak RISC-utasításai vannak (vagyis háromregiszteres mikroműveletek), mint a saját ISA-utasításhalmaza. Ezeket nem kell szétbontani. Végrehajthatók úgy, ahogy vannak, mindegyik egyetlen adatútciklussal.

A Pentium 4-gyel és az UltraSPARC III-mal szemben a 8051 igazán egyszerű gép. Inkább RISC, mint CISC típusú, hiszen legtöbb utasítása egyszerű és szétbontás nélkül végrehajtható egy óraciklusban. Nincs csővezetéke és gyorsítótára, van viszont sorrend szerinti kiosztása, sorrend szerinti végrehajtása és sorrend szerinti befejezése. Egyszerűségében főleg a Mic–1-gyel rokon.

4.8. Összefoglalás

Minden számítógép lelke az adatút. Tartalmaz néhány regisztert, egy, két vagy három sínt, és egy vagy több működési egységet, mint az ALU-k és a léptetők. A fő végrehajtó ciklus tartalmazza néhány operandus betöltését a regiszterekből, és azok elküldését végrehajtásra a síneken keresztül az ALU-hoz és más működési egységekhez. Azután az eredményeket visszatároljuk a regiszterekbe.

Az adatutat egy sorba állító vezérelheti, amelyik mikroutasításokat tölt be egy vezérlőtárból. Minden mikroutasítás olyan biteket tartalmaz, amelyek egy cikluson keresztül vezérlik az adatutat. Ezek a bitek írják elő, hogy mely operandusokat kell kiválasztani, melyik műveletet kell végrehajtani, és mit kell tenni az eredményekkel. Ráadásul minden mikroutasítás előírja a követőjét, jellemzően úgy, hogy explicit módon tartalmazza annak címét. Néhány mikroutasítás módosítja ezt a címet OR művelettel, felhasználás előtt biteket téve a címhez.

Az IJVM gép egy veremgép, 1 bájtos műveleti kódokkal, amelyek beteszik a szavakat a verembe, kiveszik a szavakat a veremből, és kombinálják (például összeadják) a szavakat a veremben. Egy mikroprogramozott megvalósítást adtunk meg a Mic–1 microarchitektúrán. Utasításbetöltő hozzáadásával, amely előre betölti a bájtokat az utasításfolyamból, sok utasításszámlálóra való hivatkozást megszüntethetünk, és a gép nagymértékben felgyorsul.

A mikroarchitektúra-szint tervezésére sok módszer létezik. Sok a kompromisszum, ide tartozik a kétsínes vagy háromsínes tervezés, kódolt vagy dekódolt mikroutasítás mezők, előre betöltés megléte vagy hiánya, kevés- vagy sokszakaszos csővezetékek és még sok más. A Mic–1 egyszerű, szoftvervezérelt gép, egymást követő végrehajtással, párhuzamosság nélkül. Ezzel ellentétben a Mic–4 magas szinten párhuzamos mikroarchitektúra, hétszakaszos csővezetékkel.

A teljesítményt sokféle módon növelhetjük. A gyorsítótár a legfontosabb. A direkt leképezésű gyorsítótár és a halmazkezelésű gyorsítótár általánosan használatosak a memóriahivatkozások felgyorsítására. Mind a statikus, mind a dinamikus elágazásjövendölés fontos, ugyanúgy, mint a sorrendtől eltérő végrehajtás és a feltételezett végrehajtás.

Három példagépünk, a Pentium 4, az UltraSPARC III és a 8051, rendelkezik olyan mikroarchitektúrával, amely nem látható ISA assembly programozók számára. A Pentium 4 összetett módszerrel konvertálja az ISA-utasításokat mikroműveletekké, hogy a gyorsítótáron keresztül betáplálja a szuperskaláris RISC-magba, ahol sorrenden kívüli végrehajtással, regiszterátnevezéssel, és a könyvben szereplő számos egyéb trükkel kinyerje a hardverből a sebesség utolsó cseppjeit.

Az UltraSPARC III Cu mélyen csővezetékezett, de ezen felül viszonylag egyszerű, sorrend szerinti a kiosztás, sorrend szerinti a végrehajtás és sorrend szerinti a befejezés. A 8051 nagyon egyszerű, egyetlen sínjéhez maroknyi regiszter és egy ALU kapcsolódik.

4.9. Feladatok

1. A 4.6. ábrán a B sín regisztere egy 4 bites mezőben van kódolva, de a C sín bittérképként van feltüntetve. Miért?
2. A 4.6. ábrán van egy „Magas bit"-nek címkézett doboz. Adjunk meg rá egy áramköri rajzot.
3. Amikor a JMPC mező egy mikroutasításban engedélyezett, az MBR-en és a NEXT_ADDRESS-en OR műveletet végzünk, hogy kialakítsuk a következő mikroutasítás címét. Lehetnek-e olyan körülmények, ahol értelme van annak, hogy NEXT_ADDRESS = 0x1FF, és használjuk JMPC-t?
4. Tegyük fel, hogy a 4.14. (a) ábrán lévő példában a
   ```
   k = 5;
   ```
   utasítás kerül az if utasítás után. Mi lesz az új assembly kód? Tegyük fel, hogy a fordítóprogram optimalizáló fordítóprogram.
5. Adjunk meg két különböző IJVM-fordítást a következő Java-utasításra:
   ```
   i = k + n + 5;
   ```
6. Adjuk meg azt a Java-utasítást, amelyik a következő IJVM-kódot hozza létre:
   ```
   ILOAD j
   ILOAD n
   ISUB
   BIPUSH 7
   ISUB
   DUP
   IADD
   ISTORE i
   ```
7. A szövegben említettük, hogy az
   ```
   if (Z) goto L1; else goto L2
   ```
   utasítás binárisra fordításánál *L2*-nek a vezérlőtár alsó 256 szavában kell lennie. Ugyanígy nem lenne lehetséges, hogy *L1* legyen, mondjuk, 0x40-től és *L2* 0x140-től? Magyarázza meg válaszát.
8. Az Mic–1-re készült mikroprogramban, az if_icmpeq3-ban az MDR-t bemásoljuk a H-ba, és néhány sorral később kivonjuk a TOS-ból, hogy ellenőrizzük az egyenlőséget. Itt biztosan jobb egy utasítás:
   ```
   if_cmpeg3        Z = MDR – TOS; rd
   ```
   Miért nem így adtuk meg?
9. Mennyi ideig tart egy 2,5 GHz-es Mic–1-nek a következő Java-utasítást végrehajtani:
   ```
   i = j + k;
   ```


Válaszát nanoszekundumban adja meg.

10. Ugyanaz a kérdés, mint előbb, csak most 2,5 GHz-es Mic–2-re. Erre a számításra alapozva, mennyi ideig tartana egy olyan program futása Mic–2-n, amelyik Mic–1-en 100 szekundum?
11. Írjunk mikrokódot Mic–1 számára, amelyik a JVM POPTWO utasítását valósítja meg. Ez az utasítás két szót kivesz a verem tetejéről.
12. A teljes JVM gépen vannak speciális 1 bájtos műveleti kódok a 0–3 helyek verembe töltésére, az általános ILOAD utasítás használata helyett. Hogyan kell IJVM-et módosítani, hogy ezeknek az utasításoknak legjobban hasznát vegye?
13. Az ISHR utasítás (egész szám aritmetikai jobbra léptetése) létezik a JVM-ben, de nincs az IJVM-ben. A verem két felső értékét használja, a két értéket egyetlen értékkel, az eredménnyel helyettesíti. A verem tetejétől a második szó a léptetendő operandus. A tartalmát előjelesen jobbra léptetjük egy 0 és 31 közötti értékkel, attól függően, hogy mennyi az értéke a verem legfelső szava 5 legkisebb helyértékű bitjének (a legfelső szó többi 27 bitjét mellőzzük). Az előjelbitet jobbra annyi biten keresztül ismételjük, amennyi a léptetések száma. Az ISHR műveleti kódja 122 (0x7A).
    a) Mi az az aritmetikai művelet, amelyik ekvivalens a 2-vel való jobbra léptetéssel?
    b) Terjesszük ki úgy a mikrokódot, hogy ezt az utasítást az IJVM részének tekinthessük.
14. Az ISHL utasítás (egész szám balra léptetése) létezik a JVM-ben, de nincs az IJVM-ben. A verem két felső értékét használja, a két értéket egyetlen értékkel, az eredménnyel helyettesíti. A verem tetejétől a második szó a léptetendő operandus. A tartalmát előjelesen balra léptetjük egy 0 és 31 közötti értékkel, attól függően, hogy mennyi az értéke a verem legfelső szava 5 legkisebb helyértékű bitjének (a legfelső szó többi 27 bitjét mellőzzük). Jobbról annyi nulla bit lép be, amennyi a léptetések száma. Az ISHL műveleti kódja 120 (0x78).
    a) Mi az az aritmetikai művelet, amelyik ekvivalens a 2-vel való balra léptetéssel?
    b) Terjesszük ki úgy a mikrokódot, hogy ezt az utasítást az IJVM részének tekinthessük.
15. A JVM INVOKEVIRTUAL utasításnak tudnia kell, hogy hány paramétere van. Miért?
16. Valósítsuk meg a JVM DLOAD utasítását Mic–2 számára. Ennek van egy 1 bájtos indexe, ezen a helyen lévő lokális változót a verembe teszi. Ezután a következő címen lévő szót is beteszi a verembe.
17. Rajzoljunk egy véges állapotú gépet a tenisz pontozására. A tenisz szabályai a következők. A nyeréshez legalább négy pont szükséges, és legalább két ponttal több kell, mint az ellenfelünknek. Kezdjünk egy (0, 0) állapottal, ami azt mutatja, hogy még senkinek sincs pontja. Ezután adjunk hozzá egy (1, 0) állapotot, amelyik azt jelenti, hogy *A* pontot kapott. Címkézzük meg a (0, 0)-ból (1, 0)-ba mutató élet *A*-val. Most adjunk hozzá egy (0, 1) állapotot, amelyik azt jelenti, hogy *B* pontot kapott, és címkézzük meg a (0, 0)-ból induló élt *B*-vel.

Folytassuk az állapotok és az élek hozzáadását, amíg minden lehetséges állapotot hozzávettünk.

18. Vizsgáljuk meg újból az előző problémát. Van olyan állapot, amit elhagyhatunk anélkül, hogy bármelyik játék eredménye megváltozna? Ha van, melyek ekvivalensek?
19. Rajzoljunk egy véges állapotú gépet elágazásjövendölésre, amelyik megbízhatóbb, mint a 4.42. ábra. Csak három egymás után következő téves jövendölés után kellene megváltoztatni a jövendölést.
20. A 4.27. ábra léptető regisztere maximum 6 bájt kapacitású. El tudjuk készíteni az IFU olcsóbb változatát 5 bájtos léptető regiszterrel? Mi a helyzet a 4 bájtossal?
21. Megvizsgálva az olcsóbb IFU-kat az előző kérdésben, most nézzük meg a költségesebbeket. Lehet-e valamikor is olyan szempont, hogy mondjuk 12-nél nagyobb léptető regiszterünk legyen? Miért, vagy miért nem?
22. A Mic–2 számára készült mikroprogramban if_icmpeq6 kód T-re megy, amikor Z 1. Azonban, a T-nél lévő kód azonos a goto1-nél lévővel. Lehetséges volna a goto1-re közvetlenül menni? Ezzel gyorsabbá tennénk a gépet?
23. A Mic–4-ben a dekódoló egység az IJVM műveleti kódot leképezi egy ROM indexre, ahol a neki megfelelő mikroműveletek el vannak tárolva. Egyszerűbbnek tűnik, ha elhagynánk a dekódoló szakaszt, és az IJVM műveleti kódot közvetlenül küldenénk a sorba állítóhoz. Az IJVM műveleti kódot indexként használhatjuk a ROM-ba, ugyanúgy, ahogy a Mic–1 dolgozik. Miért rossz ez az elképzelés?
24. Egy számítógépnek van egy kétszintű gyorsítótára. Tegyük fel, hogy a memóriahivatkozások 80%-át az első szintű gyorsítótárban találjuk, 15%-át a második szinten, és 5% hiányzik. Az elérési idők rendre 5 ns, 15 ns és 60 ns, ahol a 2 szintű gyorsítótár és a memória számára abban a pillanatban kezdjük az időt számolni, amikor kiderül, hogy szükségesek (például egy 2 szintű gyorsítótárelérés addig nem kezdődik el, amíg nincs 1 szintű gyorsítótárhiány). Mi az átlagos elérési idő?
25. A 4.5.1. fejezet végén azt mondtuk, hogy az írásallokálás csak akkor nyerő, ha feltehetően többszörös írás van ugyanabba a gyorsítósorba. Mi a helyzet olyan írás esetén, amit többszörös olvasás követ? Ez nem járna ugyanakkora nyereséggel?
26. Ennek a könyvnek az első tervezetében a 4.39. ábra egy háromutas asszociatív gyorsítótárat mutatott be négyutas helyett. Az egyik bíráló dühkitörést kapott, azt bizonygatva, hogy a hallgatókat ez borzasztóan összezavarja, mivel a három nem kettő hatványa, a számítógépek pedig mindent binárisan végeznek. Mivel az ügyfélnek mindig igaza van, az ábrát megváltoztattuk négyutas asszociatív gyorsítótárra. Igaza volt a bírálónak? Fejtse ki a válaszát.
27. Egy ötszakaszos csővezetékkel rendelkező számítógép úgy foglalkozik a feltételes elágazásokkal, hogy egy találat után leállítja a következő három ciklust. Mennyire lassítja a leállítás a végrehajtást, ha az összes utasítás 20%-a feltételes utasítás? Tekintsünk el minden más leállásforrástól a feltételes elágazáson kívül.

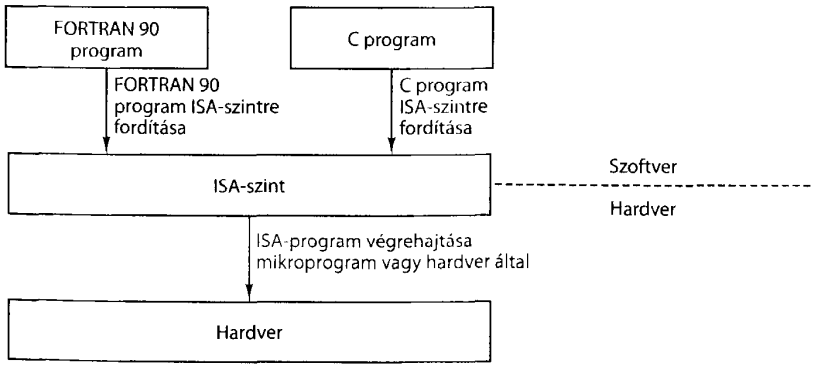
28. Tegyük fel, hogy egy számítógép legfeljebb 20 utasítást tölt be előre. Azonban ezek közül átlagosan négy feltételes elágazás, melyeket 90%-os valószínűséggel jósolunk meg. Mi a valószínűsége annak, hogy az előre betöltés jó úton jár?
29. Tegyük fel, hogy a 4.43. ábrán lévő számítógép tervét úgy változtatjuk meg, hogy 8 helyett 16 regisztere legyen. Akkor I6-ot úgy változtatjuk meg, hogy használja R8-at célként. Mi történik a 6. ciklustól kezdődő ciklusokban?
30. Általában a függőségek bajt okoznak a csővezetékes CPU-kban. Van valami optimalizálás, amit végrehajthatunk a WAW függőségekkel kapcsolatban, és ami valóban javíthat a bajokon? Mi ez?
31. Írjuk újra a Mic–1 értelmezőt, de az LV most mutasson az első lokális változóra, az összekötő mutató helyett.
32. Írjunk szimulátort egy egyutas direkt leképezésű gyorsítótárra. A szimulátor paramétere legyen a bejegyzések száma és a sor mérete. Kísérletezzünk vele, és írjuk le a tapasztalatainkat.

5. Az utasításrendszer-architektúra szintje

Ebben a fejezetben az utasításrendszer-architektúra (ISA) szintjét tárgyaljuk részletesen. Ez a szint a mikroarchitektúra szintje és az operációs rendszer gépi szintje között helyezkedik el, ahogy azt a 1.2. ábra mutatja. Történetileg ez a szint fejlődött ki elsőként, minden más szintet megelőzően. Valójában eredetileg ez volt az egyetlen szint. Nem meglepő, hogy napjainkig ezt tekintik a számítógépek „architektúrájának" vagy néha (helytelenül) „assembly nyelvnek".

Az ISA-szintnek különleges jelentősége van, ami fontossá teszi a rendszertervezők számára: ez a szint az összekötő kapocs a szoftver és a hardver között. Bár létezhetne olyan hardver, amely közvetlenül képes végrehajtani C, C++, Java vagy más magas szintű nyelven írt programot, de ez nem lenne igazán jó ötlet. Elveszne az a hatékonysági előny, ami a fordításból ered, szemben az értelmezéses végrehajtással. Továbbá, gyakorlati megfontolásból minden számítógépnek képesnek kell lennie nemcsak egy, hanem több különböző nyelven írt program végrehajtására is.

A rendszertervezők alapvetően úgy gondolkodnak, hogy a különböző magas szintű nyelven írt programokat először le kell fordítani egy közös közbülső – ISA-szintű – formára, majd olyan hardvert kell építeni, amely az ISA-szintű programokat közvetlenül végre tudja hajtani. Az ISA-szint a fordítóprogramok és a hardver közötti



5.1. ábra. *Az ISA-szint a fordítóprogramok és a hardver között van*

kapcsolati szintet definiálja. Ez az a nyelv, amelyet mindkettőnek értenie kell. A fordítóprogramok, az ISA-szint és a hardver közötti kapcsolatot mutatja az 5.1. ábra.

Elvileg, amikor egy új számítógépet terveznek, a tervezőknek ki kell kérniük mind a fordítóprogram-készítők, mind a hardvermérnökök véleményét, hogy külön-külön milyen ISA-szintű tulajdonságokat tartanak kívánatosnak. Ha a fordítóprogram-készítők olyan kívánsággal állnak elő, amelyet a hardvermérnökök nem tudnak árhatékonyan teljesíteni, akkor azt nem lehet bevenni a tervbe (például számlázó utasítás). Hasonlóan, ha a hardveresek valami remek új ötlettel állnak elő (például olyan memóriával, amely a prímszám című szavak elérésében szupergyors), de a szoftveresek nem tudják kitalálni, hogyan lehet olyan kódot készíteni, amely ezt kihasználja, az ötlet már a tervezőasztalon meghal. Az ISA sok konzultáció és szimuláció után alakul ki és kerül megvalósításra, miután teljesen optimalizálták a megcélzott programozási nyelvekre.

Ez az elmélet. De lássuk a kíméletlen valóságot. Valahányszor egy új számítógép születik, az első kérdés, amelyet minden potenciális vásárló feltesz: „Kompatibilis lesz az elődjével?" A második kérdés: „Képes futtatni a régi operációs rendszeremet?" A harmadik: „Módosítás nélkül tudja futtatni valamennyi régi alkalmazásomat?" A tervezőknek sok magyarázatra kell felkészülniük, ha csak egy kérdésre is „nem" a válasz. A vásárlók ritkán hajlandók kidobni összes régi szoftverüket, és teljesen elölről kezdeni mindent.

Ez a felfogás nagy nyomást gyakorol a tervezőkre, hogy az ISA-szint ne változzon a modellek között, vagy legalább **visszafelé kompatibilis** legyen. Ezen azt értjük, hogy az új gép módosítás nélkül képes végrehajtani minden régi programot. Az azonban teljesen elfogadható, hogy az új gép új utasításokkal rendelkezzen, és olyan új tulajdonságokkal, amelyet csak új szoftverrel lehet kihasználni. Az 5.1. ábra kifejezéseivel azt mondhatjuk, hogy mindaddig, amíg a tervezők biztosítják a visszafelé kompatibilitást, elég szabadon azt tesznek a hardverrel, amit csak akarnak. Nem kell törődniük a hardverrel (akár nem is kell ismerniük). Szabadon áttérhetnek mikropogramozott megoldásról közvetlen végrehajtásra, csővezeték-módszert vagy szuperskaláris lehetőséget vezethetnek be, bármi mást, feltéve, hogy fenntartják a visszafelé kompatibilitást a korábbi ISA-szinttel. A cél az, hogy biztosítsák a régi programok futtathatóságát az új gépen. A kihívás ezután az, hogy jobb gépet építsenek, kielégítve a visszafelé kompatibilitás követelményét.

A fentiek nem jelentik azt, hogy az ISA-tervezés nem számít. A jó ISA jelentős előnnyel rendelkezik a silánnyal szemben, különösen a nyers számítási teljesítmény/ár arány tekintetében. Az egyébként azonos tervezésű, de különböző ISA akár 25% teljesítménynövekedést is eredményezhet. Piaci nyomás miatt nehezen lehet (bár nem lehetetlen) kidobni a régi ISA-t és újat bevezetni. Ennek ellenére, időről időre új általános célú ISA keletkezik, és ez speciális piaci igények (például beágyazott rendszerek, multimédia-processzorok) esetén gyakran előfordul. Következésképpen, az ISA-tervezés megértése fontos dolog.

Mitől jó egy ISA? Két alapvető tényezőn múlik. Először is a jó ISA olyan utasításrendszert definiál, amely hatékonyan megvalósítható mai és jövőbeli technológiákkal, ami több generáción átívelő árhatékony tervezést eredményez. Gyenge tervezést nehéz megvalósítani, több áramköri elem kell a processzor megvalósí-

tásához, és több memóriára van szükség a futtatáshoz. Lassabban is futhat, mert nem ismeri a műveletek átlapolását, sokkal bonyolultabb tervezést igényelhet azonos teljesítmény elérése. Az a tervezés, amely egy bizonyos technológia sajátosságaira épül, szalmaláng lehet, egy generáció hatékony megvalósítására jó csupán, és felülmúlható előrelátóbb ISA-val.

Másodszor, a jó ISA világos célt biztosít a lefordított kód tekintetében. A szabályosság és a választások teljessége fontos jellemzők, melyek nem minden ISA tulajdonságai. Ezek a tulajdonságok a fordítóprogramok számára problémák okozói lehetnek, ha korlátozott lehetőségek közül kell a legjobbat kiválasztani, különösen, ha látszólag nyilvánvaló lehetőségeket megtilt az ISA. Röviden, mivel az ISA a kapocs a hardver és a szoftver között, boldoggá kell tennie a hardvertervezőket (könnyű hatékonyan megvalósítani) és a szoftverkészítőket (könnyen készíthető jó kód) is.

5.1. Az ISA-szint áttekintése

Kezdjük az ISA-szint tanulmányozását azzal a kérdéssel, hogy mi is az ISA. Ez egyszerű kérdésnek tűnhet, azonban több nehézséget tartogat, mint elsőre gondolnánk. A következő szakaszban felvetünk néhány tényezőt. Aztán a memóriamodellt, a regisztereket és az utasításokat tanulmányozzuk.

5.1.1. Az ISA-szint tulajdonságai

Alapvetően az ISA-szint azt jelenti, ami a gépi szintű programozó számára látszik a gépből. Mivel (épelméjű) ember nem sokat programoz már gépi szinten, mondjuk azt, hogy ami a fordítóprogramok kimenete (eltekintve az operációsrendszer-hívásoktól és a szimbolikus assemblytől). ISA-szintű kód készítése céljából a fordítóprogram írójának ismernie kell a memóriamodellt, hogy milyen regiszterek vannak, milyen adatokat és utasításokat használhat és így tovább. Mindezen információk összessége alkotja az ISA-szintet.

E definíció szerint minden olyan kérdés, amely arra vonatkozik, hogy a mikroarchitektúra mikroprogramozott-e, csővezetékes-e, szuperskaláris-e stb., nem része az ISA-szintnek, mert a fordítóprogram készítője számára nem látható. Tekintsünk például egy szuperskaláris tervezésű architektúrát, amely képes végrehajtani két egymás utáni utasítást egyetlen ciklusban, ha az egyik egész, a másik pedig lebegőpontos utasítás. Így ha a fordító váltakozva adja a kétféle utasítást, hatékonyabb kódot készít, mint ha nem ezt tenné. Tehát a szuperskaláris végrehajtás részletei láthatók az ISA-szinten, így a szintek elválasztása nem is olyan egyszerű, mint elsőre látszik.

Néhány architektúra ISA-szintjét formális dokumentum definiálja, amelyet gyakran ipari konzorciumok készítenek. Mások esetén ilyen dokumentum nem létezik. Például a V9 SPARC (SPARC 9 verzió) rendelkezik hivatalos dokumentációval (Weaver és Germond, 1994). Az ilyen definiáló dokumentumnak az a célja,

hogy lehetővé tegye a különböző megvalósítók számára, hogy olyan gépeket építsenek, amelyek ugyanazt a szoftvert futtatják ugyanolyan eredményt produkálva.

A SPARC esetében az elv az, hogy lehetővé tegyék különböző processzorgyártók számára, hogy olyan SPARC processzorokat gyártsanak, amelyek funkcionálisan ekvivalensek, csak teljesítményben és árban különbözők. Ez az elv akkor kivitelezhető, ha a gyártók tudják, mit kell csinálnia a SPARC processzornak (ISA-szinten). Ezért a definiáló dokumentum megmondja, hogy mi a memóriamodell, milyen regiszterek vannak, mit csinálnak az utasítások és így tovább, de nem beszél arról, hogy milyen a mikroarchitektúra szintje.

Az ilyen definiáló dokumentum tartalmaz **normatív** fejezeteket, amelyek követelményeket állítanak fel, valamint **informatív** fejezeteket, amelyek az olvasót hivatottak segíteni, de nem része a formális definíciónak. A normatív fejezetekben állandóan olyan kifejezéseket találunk, mint *kell*, *nem lehet*, illetve *lehetőleg*, amelyek az architektúrára vonatkozó követelményt, tiltást, illetve javaslatot fejeznek ki. Példa a következő mondat, amely azt fejezi ki, hogy ha a program olyan kódot hajtana végre, amely nincs definiálva, akkor az csapdát eredményezzen ahelyett, hogy figyelmen kívül hagyná a CPU:

Fenntartott műveleti kód végrehajtása csapdát eredményezzen.

Ha alternatív megoldásként nyitva hagyná az esetet, akkor az alábbi mondat kerülne a dokumentumba:

Fenntartott műveleti kód hatása a megvalósítástól függ.

Ez azt jelenti, hogy a fordítóprogram készítője nem feltételezhet semmilyen eredményt, de megadja a lehetőséget, hogy különböző gyártók különbözőképpen valósítsák meg az utasítást. A legtöbb architektúraspecifikációt ellátnak olyan tesztkészlettel, amely ellenőrzi, hogy az adott megvalósítás valóban teljesíti-e a specifikáció követelményeit.

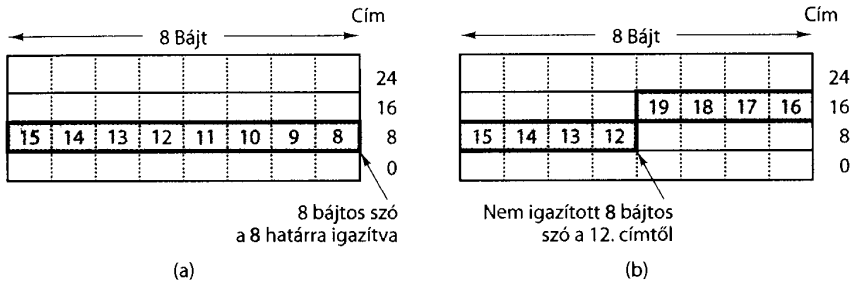
Világos, hogy a V9 SPARC miért rendelkezik az ISA-szintet definiáló dokumentummal: mert minden V9 SPARC processzor ugyanazt a szoftvert fogja futtatni. A Pentium 4 ISA-szintjének nem létezik definiáló dokumentuma, mert az Intel nem akarja, hogy más gyártók is képesek legyenek Pentium 4 processzort gyártani. Tulajdonképpen az Intel bírósághoz fordult, hogy megakadályozza, hogy más gyártók klónt állítsanak elő, bár a pert elvesztette.

Az ISA-szint másik fontos tulajdonsága, hogy a legtöbb gép legalább két módban működhet. A **kernelmód** az operációs rendszer futtatására szánt mód, amikor is minden utasítás végrehajtható. A **felhasználói mód** a felhasználói programok futtatását célozza, és nem teszi lehetővé bizonyos érzékeny utasítások végrehajtását (mint amelyek közvetlenül manipulálják a gyorsítótárat). Ebben a fejezetben elsősorban a felhasználói mód utasításait és tulajdonságait vizsgáljuk.

5.1.2. Memóriamodellek

Minden számítógép memóriája cellákra van osztva, és a cellák címei összefüggő tartományt képeznek. A leggyakoribb cellaméret napjainkban 8 **bit**, de régebben alkalmaztak 1–60 bit cellaméretet is (lásd 2.10. ábra). Egy 8 bites cella neve **bájt**. A 8 bites cellaméret oka az, hogy egy ASCII karakter 7 bit, így a paritásbittel együtt éppen belefér egy bájtba. Ha az UNICODE kódolás általánossá válik a jövőben, akkor a számítógépek 16 bites cellát alkalmazhatnak. Végül is 2^4 még szebb szám, mint a 2^3, mivel 4 kettő hatványa, a 3 pedig nem.

A bájtok általában 4 bájtos (32 bit) vagy 8 bájtos (64 bit) csoportokba, szavakba rendezettek, lehetővé téve, hogy az utasítások teljes szavakon végezzenek műveleteket. Sok architektúra megköveteli, hogy a szavak természetes határokhoz legyenek igazítva, például hogy a 4 bájtos szavak címe 0, 4, 8 stb. legyen. Hasonlóan a 8 bájtos szavak címe 0, 8, 16 stb. legyen, tehát nem lehet például 4 vagy 6. A 8 bájtos szavak igazítását mutatja az 5.2. ábra.



5.2. ábra. *8 bájtos szó kis endián memóriában. (a) Igazított. (b) Nem igazított. Néhány gép megköveteli, hogy a szavak igazítottak legyenek*

Az igazítást gyakran azért követelik meg, mert a memória így hatékonyabban működhet. A Pentium 4 például, amely 8 bájtot olvas ki a memóriából, 36 bites fizikai címet használ, de csak 33 bites címe van, ahogy az a 3.44. ábrán látható. Tehát a Pentium 4 nem is tudna nem igazított memóriahivatkozást végezni, mivel a cím alsó 3 bitjét nem lehet ténylegesen megadni. Ezek értéke mindig 0, így minden hivatkozott cím 8 többszöröse lehet csak.

Az igazítás kényszere azonban néha problémákat okoz. A Pentium 4 esetén az ISA-programok bármely címen kezdődő szóra hivatkozhatnak, ami a 8088-ra vezethető vissza, amelynek adatsínje 1 bájtos volt (és így nem volt 8 bájtos igazítási kényszer). Ha a Pentium 4 a 7. címről olvas egy 4 bájtos szót, akkor a hardvernek először egy memóriahivatkozással be kell olvasnia a 0–7. című bájtokat, majd egy másik hivatkozással a 8–15. címűeket. Ezután a CPU kiveszi a beolvasott 16 bájtból a kívánt 4 bájtot, és beilleszti a megfelelő sorrendbe, hogy 4 bájtos szót kapjon.

A tetszőleges címről való olvashatóság követelménye extra áramkört igényel a processzorban, tehát a processzor nagyobb és drágább lesz. A tervezőmérnökök

örömmel megszabadulnának ettől, és egyszerűen megkövetelnék minden programtól, hogy csak szóhatárú memóriahivatkozást végezzen. Az a gond, hogy valahányszor a mérnökök azt mondják, hogy „Ki törődik azzal, ha ósdi 8088 program helytelen memóriahivatkozást végez?", a marketinges szakemberek szűkszavú válasza: „A vásárlóink."

A legtöbb gép egyetlen lineáris címtartományt alkalmaz, amely 0-tól valamiféle maximumig (gyakran 2^{32} vagy 2^{64} bájt) terjed. Azonban néhány gép külön címtartománnyal rendelkezik az adatok, és külön az utasítások számára. Így a 8-as címről történő utasítás kiolvasása más címtartományra vonatkozik, mint a 8-as címről történő adat kiolvasása. Ez a séma bonyolultabb, mint amikor csak egyetlen címtartomány van, azonban két előnye is van. Először, lehetséges 2^{32} méretű program és 2^{32} méretű adat csak 32 bites címzést használva. Másodszor, mivel minden írás automatikusan az adatcímtartományra vonatkozik, nem lehet a programot ily módon felülírni, ami egy programozási hibaforrás kiküszöbölését jelenti.

Megjegyzendő, hogy a külön utasítás- és külön adatcímtartomány nem egyenértékű az 1. szintű gyorsítótárral. Az előbbi esetben a címtartomány duplázódik, és minden olvasás mást eredményez, attól függően, hogy utasításról vagy adatról van szó. Gyorsítótár esetén csak egy címtartomány van, amely annak csupán különböző részeit tartalmazza.

Még egy aspektusa van az ISA-szintű memóriamodellnek – ez a memória-szemantika. Természetes elvárás, hogy egy LOAD utasítás, amelyet ugyanazon címre vonatkozó STORE utasítás után hajtunk végre, az éppen eltárolt adatot eredményezze. Azonban, mint azt a 4. fejezetben láttuk, a mikroutasítások sorrendje átrendeződhet. Tehát valós a veszély, hogy a memória nem az elvárt módon viselkedik. A probléma még rosszabbá válhat multiprocesszor esetén, amikor a különböző CPU-k (esetleg átrendezett) olvasási sorozatot bocsátanak ki osztott memóriára vonatkozóan.

A tervezők a probléma különböző megoldásai közül választhatnak. Az egyik szélsőséges megoldás, ha minden memóriahivatkozást sorba rendeznek, így mindegyik előbb befejeződik, mint egy másik elkezdődne. Ez a stratégia sérti a hatékonyságot, de a legegyszerűbb memóriaszemantikát adja (az utasítások szigorúan a programban megadott sorrendben hajtódnak végre).

A másik szélsőséges megoldás esetében semmi garancia nincs általánosan biztosítva. A memóriahivatkozások sorrendjének kikényszerítésére SYNC utasítást kell a programnak végrehajtania, amely blokkolja minden memóriaművelet kibocsátását mindaddig, amíg az összes eddig kiadott memóriaművelet be nem fejeződött. Ez a megoldás nagymértékben korlátozza a fordítóprogram-írókat, mert részleteiben kell ismerniük a mikoarchitektúra működését, azonban a legnagyobb szabadságot adja a hardver tervezőinek a memória működésének optimalizálására.

Közbülső megoldás is lehet a memóriamodellre, amikor a hardver automatikusan blokkol bizonyos memóriahivatkozásokat (többek között a RAW és WAR függőséget tartalmazókat), de nem mindet. Ugyan a mikroachitektúra ezen furcsaságainak ISA-szintű következményei bosszantók (legalábbis a fordítóprogramkészítők és assembly programozók számára), mégis leginkább ez a trend. Ennek

az oka az olyan megvalósításokban rejlik, mint a mikroutasítások sorrendjének felcserélése, csővezeték alkalmazása, többszintű gyorsítótár stb. Ebben a fejezetben később több ilyen természetellenes hatást fogunk látni.

5.1.3. Regiszterek

Minden számítógép rendelkezik ISA-szinten látható regiszterekkel. Ezek célja a program végrehajtásának vezérlése, közbülső eredmény tárolása és még egyebek. Általában azok a regiszterek, amelyek a mikroarchitektúra szintjén láthatók, mint a TOS és a MAR a 4.1. ábrán, nem láthatók ISA-szinten. Azonban néhány, mint például az utasításszámláló vagy a veremmutató, mindkét szinten látható. Másrészt, minden ISA-szinten látható regiszter látható a mikroarchitektúra szintjén is, hiszen ezen a szinten van megvalósítva.

Az ISA-regiszterek durván két kategóriába sorolhatók: speciális célú és általános célú regiszterek. A speciális célú regiszterek közé tartoznak az utasításszámláló, a veremmutató és a speciális funkciójú regiszterek. Ezzel ellentétben az általános célú regiszterek azért vannak, hogy fontos lokális változókat és időleges eredményeket tároljanak. Ezek fő célja az, hogy a gyakran használt adatok gyors elérését biztosítsák (kiküszöbölve a memóriahivatkozást). A RISC gépek, a gyors CPU és a (viszonylag) lassú memória miatt általában 32 általános célú regiszterrel rendelkeznek, és az a tendencia az új processzorok tervezésénél, hogy még ennél is több regiszter legyen.

Néhány gép esetén az általános célú regiszterek teljesen szimmetrikusak és felcserélhetők. Ha a regiszterek ekvivalensek, a fordítóprogram akár az R1, akár az R25 regisztert választhatja közbülső eredmény tárolására. A regiszter választása nem számít.

Azonban más gépek esetén néhány általános célú regiszter mégis kicsit speciális. Példa erre a Pentium 4, ahol az EDX nevű regiszter, amely általános célra is használható, de szorzás esetén a szorzat felét, osztás esetén az osztandó felét tartalmazza.

Még ha az általános célú regiszterek teljesen felcserélhetők is, az operációs rendszerek és fordítóprogramok rögzített konvenció szerint használják ezeket. Például adott regiszterek mindig a hívott eljárás paramétereit tárolják. Így ha a fordító az R1 regiszterben tárol lokális változót, majd hív egy olyan könyvtári eljárást, amely azt feltételezi, hogy az R1 szabad, akkor az eljáráshívás után az R1 szemetet tartalmazhat. Ha van rendszerszinten elfogadott regiszterhasználati konvenció, akkor azt mind a fordítóprogramoknak, mind az assembly programozóknak célszerű betartaniuk.

ISA-szinten a felhasználói programok által látható regiszterek mellett mindig van szép számban olyan speciális célú regiszter, amely csak kernelmódban használható. Ezek a regiszterek vezérlik a különböző gyorsítótárakat, memóriát, B/K eszközöket és a gép más hardvertulajdonságait. Ezeket csak az operációs rendszer használja, így a fordítóprogramoknak és felhasználóknak nem kell tudniuk róluk.

Van egy vezérlőregiszter, amely a kernel/felhasználói mód keveréke, ez a **jelzők regisztere (flags register)** vagy PSW **(Program Status Word, programállapotszó)** regiszter. Ez a regiszter jó néhány olyan kiegészítő bitet tartalmaz, amelyet a CPU használ. A legfontosabb bitek a **feltételkódok**. Ezeket a biteket minden ALU-ciklus beállítja, és az aktuális utasítás eredményétől függenek. A tipikus feltételkódok az alábbiak:

N – beállítva, ha az eredmény Negatív volt
Z – beállítva, ha az eredmény Zérus volt
V – beállítva, ha az eredmény túlcsordulást (oVerflow) okozott
C – beállítva, ha az eredmény a bal szélső bit átvitelét okozta (Carry)
A – beállítva, ha átvitel történt a 3. bitről (Auxiliary carry)
P – beállítva, ha az eredmény paritása páros (Parity)

A feltételkódok azért fontosak, mert az összehasonlító és elágazó utasítások ezeket használják (például feltételes ugró utasítás). Például a CMP utasítás tipikusan kivonást végez két operandus között, majd beállítja a feltételkódot a különbség alapján. Ha az operandusok egyenlők, akkor a különbség 0, és a PSW Z bitjét beállítja. Az ezt követő BEQ (Branch Equal/elágazás egyenlő esetén) utasítás a Z bitet ellenőrzi, és ugrás következik, ha beállítva találja.

A PSW nemcsak a feltételkódokat tartalmazza, de a tényleges tartalom gépről gépre eltérő. Tipikus egyéb mezők közé tartozik a kernel/felhasználói mód jelzése, a nyomkövető bit, CPU prioritási szint, megszakítást engedélyező állapotjelző. A PSW gyakran olvasható felhasználói módban is, de néhány bit csak kernelmódban módosítható (többek között a kernel/felhasználói mód bitje).

5.1.4. Utasítások

Az ISA-szint legfontosabb tulajdonsága a gépi utasításkészlet. Ez határozza meg, hogy a gép mit csinálhat. Mindig van LOAD és STORE utasítás (egy vagy több formában) az adatoknak a memória és a regiszterek közötti mozgatására, és van MOVE utasítás az adatok regiszterek közötti másolására. Mindig vannak aritmetikai utasítások, csakúgy mint logikai utasítások, adatok összehasonlítására és elágazásra szolgáló utasítások. Már láttunk néhány ISA-utasítást (lásd 4.11. ábra), és ebben a fejezetben még többet fogunk tanulmányozni.

5.1.5. A Pentium 4 ISA-szintjének áttekintése

Ebben a fejezetben három nagyon különböző utasításrendszer-architektúrát vizsgálunk, ezek: Intel IA–32, amelyet a Pentium 4 processzor, a SPARC V9, amelyet az UltraSPARC processzor valósít meg, és a 8051 processzor. Nem az a szándékunk, hogy ezek kimerítő leírását adjuk, hanem az, hogy bemutassuk az utasításrendszer-architektúra fontos aspektusait, és hogy ezek milyen mértékben különböznek egymástól. Kezdjük a Pentium 4-gyel.

A Pentium 4 processzor több generáción keresztül fejlődött ki. Származása visszavezethető az egyik legelsőként épített mikroprocesszorra, mint azt az 1. fejezetben

már megismertük. Az alapvető ISA teljesen támogatja a 8086 és a 8088 processzorokra írt programok végrehajtását (amelyek ISA-szintje azonos volt), sőt még a 70-es évek népszerű 8 bites processzorának, a 8080-nak a maradványai is megtalálhatók. A 8080 tervezését viszont kompatibilitási kényszer miatt nagymértékben befolyásolta a korábbi 8008-es, amelynek az alapja a 4004-es volt, az a 4 bites processzor, amelyet még akkoriban használtak, amikor dinoszauruszok népesítették be a Földet.

Szoftverszempontból a 8086 és a 8088 16 bites gépek voltak (bár a 8088-nak 8 bites adatsínje volt). Ezek leszármazottja a 80286, amely szintén 16 bites gép. Nagy előnye a nagyobb memória-címtartomány volt, bár kevés program használta ki, mivel a memória 16 384 darab 64 KB méretű szegmensből állt, a lehetséges 2^{30} méretű lineáris memória helyett.

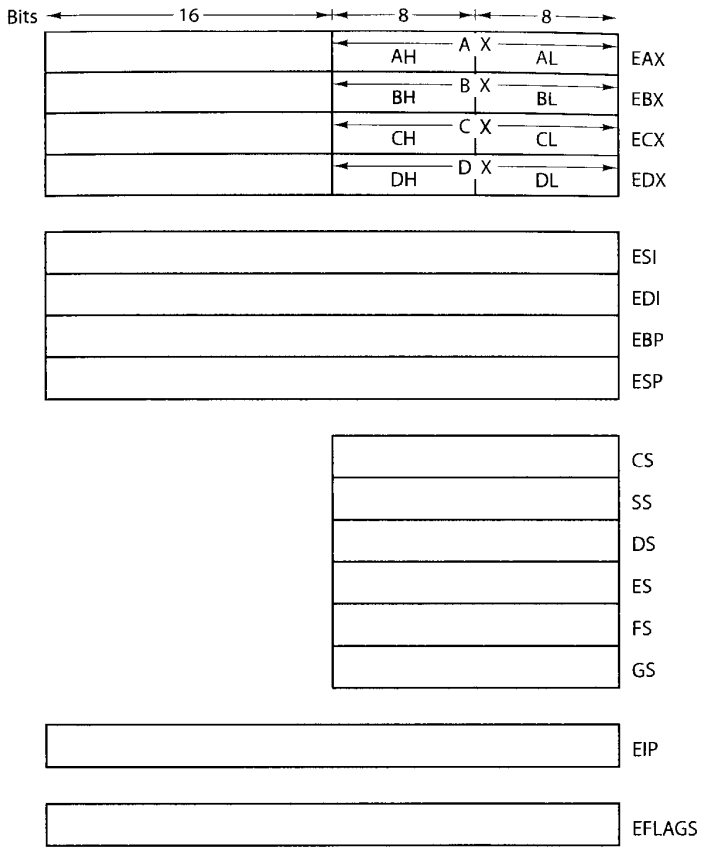
A 80386 volt az Intel család első 32 bites processzora. Az összes további leszármazott (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino stb.) architektúrája alapvetően megegyezett a 80386-os, úgynevezett **IA–32** architektúrával. Ennek tanulmányozására koncentrálunk a következőkben. Az architektúra szempontjából az egyetlen nagyobb változást a Pentium későbbi verzióiban bevezetett MMX, SSE és SSE2 utasítások jelentették a 80386-hoz képest.

A Pentium 4-nek három működési módja van, kettőben ezek közül a gép úgy viselkedik, mintha 8088-as lenne. **Valós mód**ban minden olyan tulajdonság ki van kapcsolva, amelyet a 8088 után vezettek be, a gép úgy működik, mint egy egyszerű 8088-as. Ha a program valami szabálytalant csinál, a gép egyszerűen összeomlik. Emberi hasonlattal ez annyit tesz, mintha az embert visszaminősítenék csimpánzzá (az agy nagy része eltűnik, nincs beszéd, jórészt banánt eszik stb.).

Egy lépés előre a **virtuális 8086 mód**, amely lehetővé teszi régi 8088-as programoknak védett módon történő futtatását. Ebben a módban valódi operációs rendszer vezérlete alatt működik a teljes gép. A 8088-as program futtatása úgy történik, hogy az operációs rendszer speciális izolált környezetet épít fel, amely úgy viselkedik, mint a 8088-as, és ebben futtatja a programot. A lényeges kivétel, hogy a program összeomlásakor értesíti az operációs rendszert ahelyett, hogy a gép maga összeomlana. Amikor a Windows-felhasználó MS-DOS ablakot nyit, akkor az abban indított program virtuális 8086 módban fut, megvédve magát a Windowst az MS-DOS program rendellenes viselkedésétől.

Végül a harmadik mód a **védett mód**, amelyben a Pentium 4 valódi Pentium 4-ként viselkedik, és nem úgy, mint egy erősen kiterjesztett 8088. Négy privilegizált szint van, amelyeket a PSW bitjei vezérelnek. A 0-s szint más gépek kernelmódjának felel meg, amikor is a gép korlátozás nélküli elérése biztosított. Ezt a szintet használja az operációs rendszer. A 3-as szint a felhasználói programoké. Ekkor bizonyos kritikus utasítások és regiszterek elérése blokkolt, hogy megvédje a gépet a huncut programozóktól. Az 1-es és 2-es szintet ritkán használják.

A Pentium 4 hatalmas címtartománnyal rendelkezik, a memória 16384 darab szegmensre van osztva, a címek mindegyikben 0-tól $2^{32}-1$-ig terjednek. A legtöbb operációs rendszer (beleértve a UNIX-ot és minden Windows-változatot) azonban csak egy szegmenst támogat, így a legtöbb felhasználói program csak egy 2^{32} nagyságú lineáris címtartományt lát, amelynek egy részét maga az operációs



5.3. ábra. *A Pentium 4 fő regiszterei*

rendszer foglalja el. A címtartományban minden bájtnak saját címe van, a szavak 32 bitesek. A szavak tárolása kis endián kódban történik (a szó legalacsonyabb helyértékű bájtjának legkisebb a címe).

A Pentium 4 regisztereit az 5.3. ábra mutatja. Az első négy, az EAX, az EBX, az ECX és az EDX többé-kevésbé általános célú regiszter, bár mindegyiknek van sajátossága. Az EAX a fő aritmetikai regiszter, az EBX jól használható mutató (memóriacím) tárolására, az ECX-nek pedig a ciklusszervezésben van szerepe. Az EDX a szorzás és osztás utasításoknál kell, amikor is az EAX-szel együtt tartalmazzák a 64 bites szorzatot, illetve osztandót. A felsorolt regiszterek mindegyikének első 16 bitje egy 16 bites, első 8 bitje pedig egy 8 bites regisztert alkot. Ezek lehetővé teszik műveletek egyszerű elvégzését 8 és 16 bites adatokon. A 8088 és 80286 csak 8, illetve 16 bites regisztert tartalmazott. A 32 bites regisztereket a 80386-tal vezették be, E betűvel jelölve őket, ami Extended-et jelent.

A következő három szintén valamelyest általános célú regiszter, de több különlegességgel. Az ESI-t és EDI-t memóriába mutató mutatók tárolására szánták, különösen a hardveres karakterlánc műveletek esetére, amikor is az ESI a forrás karakterláncra, az EDI pedig a cél karakterláncra mutat. Az EBP is mutatóregiszter, tipikusan az aktuális veremkeret címét tartalmazza, mint az IJVM esetén az LV regiszter. Ha egy regiszter (mint az EBP) a lokális veremkeret címét tartalmazza, akkor azt mondjuk, hogy **keretmutató**. Végül az ESP a veremmutató.

A regiszterek következő csoportja CS-től GS-ig bizonyos értelemben elektronikus őskövületek, amelyek abból az időből maradtak fenn, amikor a 8088 esetén a 2^{20} méretű memória 16 bites címzéssel történő elérésére használták. Elég megjegyezni, hogy a Pentium 4-nél ezek biztosan elhagyhatók, ha a gép egyetlen 32 bites lineáris címtartományt használ. A következő az EIP, amely az utasításszámláló (Extended Instruction Pointer, kiterjesztett utasításmutató). Végül az EFLAGS regiszter látható, amely a PSW.

5.1.6. Az UltraSPARC III ISA-szintjének áttekintése

A SPARC architektúrát először a Sun Microsystem vezette be 1987-ben. Ez volt az egyik első kereskedelmi forgalomba került RISC címkét viselő architektúra. Nagyrészt a Berkeley-ben az 1980-as években folytatott kutatások eredményeire épült (Patterson, 1985; Patterson és Séquin, 1982). Eredetileg a SPARC 32 bites architektúra volt, de az UltraSPARC III már 64 bites, ami a 9-es verzióra épül, és ezt fogjuk bemutatni ebben a fejezetben. A könyv többi részével összhangban kívánunk lenni, ezért az UltraSPARC III-ra hivatkozunk ebben a fejezetben, de az ISA-szintet tekintve valamennyi UltraSPARC azonos.

Az UltraSPARC III memóriaszerkezete tiszta és világos: a címezhető memória 2^{64} bájt méretű lineáris tömb. Szerencsétlen módon ez olyan nagy méret (18 446 744 073 709 551 616 bájt), hogy jelenleg nincs olyan gép, amely megvalósítaná. A jelenlegi megvalósítások korlátozzák a címtartományt (2^{44} bájt az UltraSPARC III esetén), de ez növekedni fog a jövőbeli modelleknél. Az alapértelmezett bájtsorrend a nagy endián, de kis endián sorrend is beállítható a PSW megfelelő bitjével.

Fontos, hogy az ISA magasabb korlátot tartalmaz, mint amire a megvalósításnál szükség van, mert a jövőbeli megvalósítások bizonyára nagyobb memóriát igényelnek majd. Az egyik legkomolyabb probléma, amely a sikeres architektúrákkal előfordulhat, az a címezhető memória nagysága. Egy napon unokáink azt fogják kérdezni, hogyan lehetett valamit is csinálni azokkal a számítógépekkel, amelyeknek csak 32 bites címe és legfeljebb 4 GB memóriája volt, amikor egy átlagos játékprogram elindítása 8 GB memóriát kíván.

A SPARC ISA világos, bár a regiszterek szervezése kissé bonyolult, hogy az eljáráshívásokat hatékonyabbá tegyék. A tapasztalat szerint a regiszterek szervezése több problémát jelent, mint gondolták, de a visszafelé kompatibilitás kényszere miatt nem lehet megváltoztatni.

Az UltraSPARC III regiszterek két csoportját tartalmazza: 32 darab 64 bites általános célú regiszter és 32 darab lebegőpontos regiszter. Az általános célú regiszterek jelei: R0, ..., R31, bár bizonyos környezetben más elnevezések is előfordulnak. Az alternatív nevek és funkciók az 5.4. ábrán láthatók.

Regiszter	Alternatív név	Funkció
R0	G0	Hardveres 0-t tartalmaz, minden tárolás eredménytelen.
R1–R7	G1–G7	Globális változók
R8–R13	O0–O5	A hívott eljárás paraméterei
R14	SP	Veremmutató
R15	O7	Ideiglenes regiszter
R16–R23	L0–L7	Az aktuális eljárás lokális változói
R24–R29	I0–I5	A bejövő paraméterek
R30	FP	Az aktuális veremkeret-mutató
R31	I7	Az aktuális eljárás visszatérési címe

5.4. ábra. *Az UltraSPARC III általános regiszterei*

Minden általános regiszter 64 bites. Az azonosan 0 tartalmú R0 kivételével a regisztereket sokféle író és olvasó utasítással lehet átírni, illetve kiolvasni. Az 5.4. ábrán látható regiszterhasználat részben konvención alapszik, részben azon, ahogyan a hardver kezeli őket. Általában nem bölcs dolog eltérni a konvencióktól, kivéve, ha valaki fekete öves SPARC-guru, és kétséget kizáróan tudja, hogy mit csinál. A fordítóprogram és a programozó felelőssége, hogy a regisztereket megfelelően használja. Például betölthetünk általános regiszterekbe lebegőpontos számokat, majd egészként összeadhatjuk őket, az eredmény értelmetlen dolog lesz, de a CPU szívélyesen elvégzi nekünk.

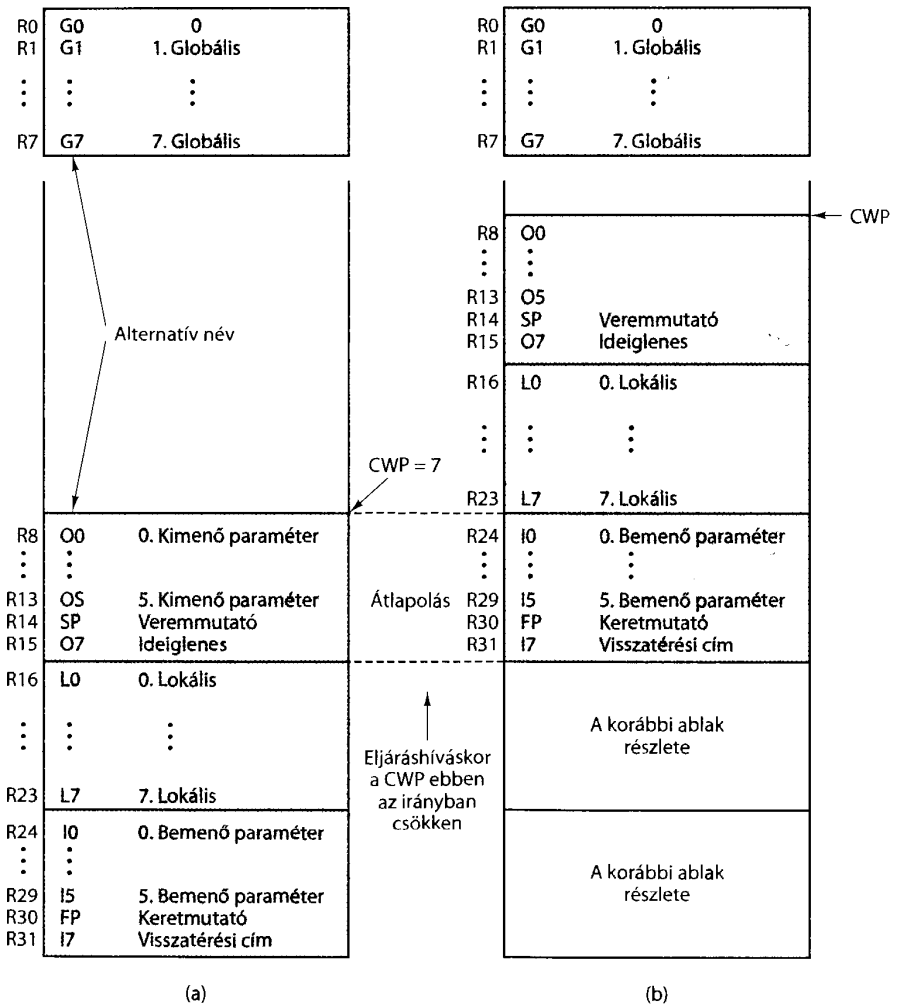
A globális változók olyan konstansok, változók és mutatók, amelyeket minden eljárás használ, bár szükség esetén elmenthetők és visszatölthetők eljárásba való belépéskor, illetve kilépéskor. Az Ix és Ox regisztereket paraméterek átadására használják, hogy elkerüljék a memóriahivatkozást. A későbbiekben elmagyarázzuk, hogyan működik ez.

Három dedikált regisztert speciális célra használnak. Az FP és SP regiszterek az aktuális keret határait tartalmazzák. Az FP regiszter az aktuális keret kezdőcímét tartalmazza, és a lokális változók címzésére szolgál, éppen úgy, mint azt a 4.10. ábrán bemutattuk az LV regiszter esetében. Az SP az aktuális verem tetejére mutat, és a verembe, illetve veremből műveletek hatására változik. Ezzel ellentétben, az FP csak eljáráshíváskor és visszatéréskor változik. A harmadik speciális célra használt regiszter az R31, amely az aktuális eljárás visszatérési címét tartalmazza.

Az UltraSPARC III ténylegesen 32-nél több általános regisztert tartalmaz, de csak 32 látszik a programozó számára egy adott pillanatban. Ezt a tulajdonságot **regiszterablak**nak nevezik, és az eljárások hatékony megvalósítását célozza. Ezt illusztrálja az 5.5. ábra. Az alapötlet a verem szimulálása, de regiszterek használatával. Vagyis ténylegesen több regiszter van, ahogy ugyancsak több keret van a veremben. Mindenkor pontosan 32 általános regiszter használható. A CWP (Current Window Pointer, aktuális ablakmutató) regiszter mutatja, hogy aktuálisan mely regiszterek használhatók.

Az eljáráshívás utasítás elrejti a régi regiszterkészletet, és egy újat nyit úgy, hogy csökkenti a CWP értékét. Azonban néhány régi regiszter az új készletben is benne lesz, hogy hatékonyabbá tegyük az eljáráshívások megvalósítását. Ez a technika úgy működik, hogy néhány regiszter más nevet kap, eljáráshívás után a régi R8–R15 kimeneti regiszterek továbbra is láthatók lesznek, de most ezek az R24–R31 bemeneti regiszterek. A nyolc globális regiszter sohasem változik, ezek mindig ugyanazt jelentik.

A memóriával ellentétben, amely majdnem végtelen (legalábbis ameddig a verem növekszik), ahogy az eljáráshívások egyre mélyebben egymásba ágyazódnak,



5.5. ábra. *Az UltraSPARC III regiszterablak technikája*

úgy fogyhat ki a regiszterkészlet. Ekkor a régi, betelt készletet ki kell írni a memóriába, ezzel felszabadítva újabb használatra. Hasonlóan, ha sok eljárásból következik be visszatérés, a memóriába kimentett regiszterkészletet be kell olvasni. Ez az egész túl bonyolult és több gondot okoz, mint amennyi a haszna. Ez a technika csak akkor segít, ha az eljáráshívások nem túl mélyen egymásba ágyazottak.

UltraSPARC III 32 lebegőpontos regiszterrel is rendelkezik, amelyek mindegyike vagy 32 bites (egyszeres pontosságú), vagy 64 bites (dupla pontosságú) számot tartalmazhat. Lehetséges két regiszterben tárolni egy 128 bites (négyszeres pontosságú) számot is.

Az UltraSPARC architektúrája **betöltő/tároló (load/store) architektúra**. Ez azt jelenti, hogy csak a regiszterek és a memória közötti adatátvitelt megvalósító betöltő (LOAD) és tároló (STORE) utasítások hivatkoznak közvetlenül a memóriára. Az összes többi utasítás operandusa mindig valamelyik regiszterben (nem a memóriában) vagy az utasításban van, és az eredmény is mindig regiszterbe kerül (nem a memóriába).

5.1.7. A 8051 ISA-szintjének áttekintése

Harmadik példánk a 8051-es processzor. A Pentium 4 (amelyet elsősorban asztali gépekben és kiszolgálókban alkalmaznak) és az UltraSPARC III (amelyet elsősorban nagyobb kiszolgáló konfigurációkban, különösen többprocesszoros rendszerekben alkalmaznak) processzorokkal ellentétben a 8051-et beágyazott rendszerekben használják. Például közlekedési lámák és időzített órák esetében az eszközök vezérlésére, valamint nyomógombok, lámpák és egyéb felhasználói felületelemek kezelésére. Története egyszerű és egyenes vonalú. Az Intel azonnal sikert ért el, amikor 1974-ben kijött az egy lapra szerelt 8080-as CPU-val. A legkülönbözőbb készülékek gyártói építették be ezt a processzort termékeikbe. Majd kérték az Intelt, hogy építsen olyan egy lapra szerelt processzort, amely nem csak a CPU-t, hanem a memóriát és az B/K eszközvezérlőt is tartalmazza. Az Intel válaszolt, és megalkotta a 8048-ast, amelyből hamarosan megszületett a 8051-es. Éltes kora ellenére (vagy éppen ezért) még ma is széles körben használatos, mert nagyon olcsó, ami igen fontos a beépített rendszereknél. Ebben a fejezetben rövid technikai ismertetést adunk a 8051-esről, valamint fivéreiről és nővéreiről.

A 8051-es egyetlen módban működik, és nincs védelmi hardvere, mivel sohasem futtat egyszerre több, különböző felhasználói programot. Memóriamodellje nagyon egyszerű. Külön 64 KB-os címtartományú program- és 64 KB címtartományú adatmemóriája van. Különválasztották a program- és az adatmemóriát, hogy az előbbit ROM, az utóbbit pedig RAM valósíthassa meg.

A memóriának sokféle megvalósítása lehetséges. A legegyszerűbb esetben 4 KB ROM van a program, és 128 bájt RAM az adat számára. A ROM és a RAM a lapkára integrált. Kisebb alkalmazások esetén ennyi memória elegendő, és hatalmas nyereség, hogy egy lapkán van a CPU-val. A 8052-esnek kétszer ennyi memóriája van, 8 KB ROM és 256 bájt RAM a lapkán. E modellek használatakor a programot gyárilag beégetik a ROM-ba, amit aztán a felhasználó nem változtat.

A másik véglet az olyan 8051-es, amelynek 64 KB külső ROM vagy EPROM programmemóriája, és 64 KB külső RAM adatmemóriája van. Az is lehetséges, hogy egyetlen 64 KB külső RAM tartalmazza a programot is és az adatot is.

A 8051-es támogat olyan közbülső modellt is, amikor az alsó 4 KB programmemória és 128 bájt adatmemória a lapkán belül van, a maradék memória pedig külső. Bizonyos lábaira adott feszültség határozza meg, hogy melyik modell érvényesül.

A 8051-es regiszterkezelése szokatlan. A legtöbb 8051-es program úgy készül, hogy nyolc darab 8 bites regisztert használ. Ez a természetes használata a CPU-nak, mert a legtöbb utasítás egy 3 bites mezőn határozza meg a használt regisztert. A regiszterekre R0,..., R7 nevekkel hivatkozunk. Mindamellett négy regiszterkészlete van, de egyszerre csak az egyik használható. A PSW 2 bites mezője mondja meg, éppen melyik készlet az aktuális. A több regiszterkészlet célja az, hogy lehetővé tegye a nagyon gyors megszakításfeldolgozást. Ha megszakítás keletkezik, a feldolgozó programnak nem kell elmentenie az összes regisztert, hanem csak átvált egy másik készletre. A 8051-esnek ez a tulajdonsága lehetővé teszi, hogy másodpercenként nagyon nagy számú megszakítást tudjon feldolgozni, ami nagyon fontos tulajdonsága minden olyan processzornak, amelyet beágyazott valós idejű rendszerben akarnak használni.

Egy másik sajátossága a 8051 regisztereinek, hogy a memóriában vannak. A memória 0. bájtja a 0. regiszterkészlet R0 regisztere. Ha egy utasítás megváltoztatja az R0 regiszter tartamát, majd később egy utasítás a 0. bájtot olvassa, akkor ott a megváltozott R0 értékét találja. Hasonlóan az 1. bájt az R1 regiszter és így tovább. A 8. bájttól a 15. bájtig vannak az 1. regiszterkészlet regiszterei, és így tovább 31-ig, amely a 3. készlet R7 regisztere. Az elrendezés az 5.6. (a) ábrán látható.

Közvetlenül a négy regiszterkészlet (regiszterbank) felett a 32. memóriacímtől a 47-ig terjedő 16 bájt a bit-címezhető memóriamező. A bitek címzése 0-tól 127-ig terjed. A 8051-esnek utasításai vannak a beállítás, törlés, ÉS, VAGY és tesztelés elvégzésére az utasításban megadott címen, a 0-tól 127-ig terjedő bitcímtartományban. Ezek tehát a 32 és 47 közötti memóriacellákon operálnak. Az ilyen utasítások hasznosak, mert beágyazott processzorok gyakran alkalmaznak bitváltozót kapcsolók, lámpák és más B/K eszközök állapotának lekérdezésére vagy beállítására. A speciális memória mező lehetővé teszi bitváltozók kezelését anélkül, hogy a teljes bájtot be kellene olvasni, majd léptetni, és kimaszkolni a nem kívánt biteket. Egy ilyen kicsi processzor esetén ez rendkívül hasznos a logikai változók kezelésére.

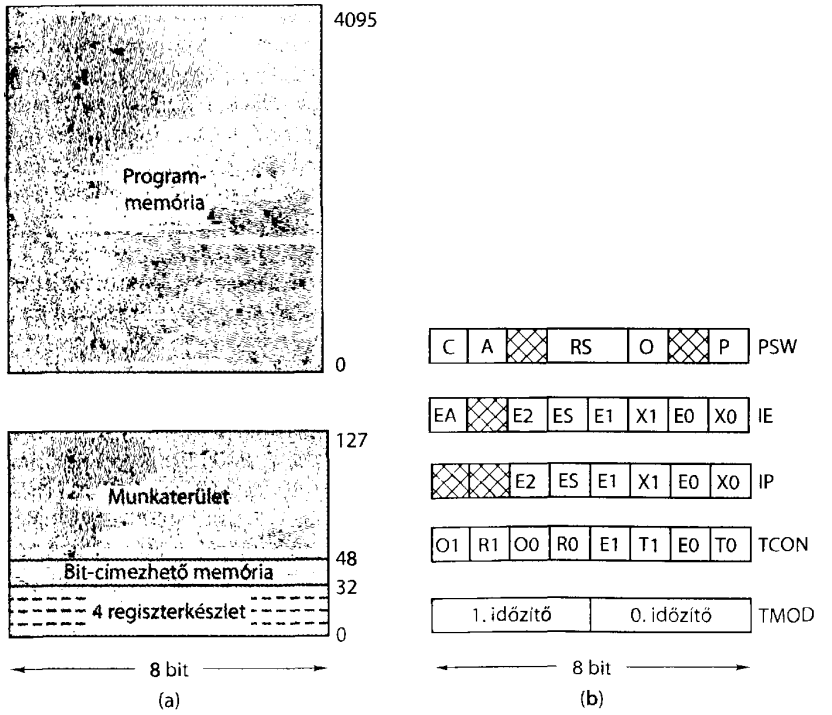
A 8051 a négy általános regiszterkészleten kívül tartalmaz néhány speciális célú regisztert is, amelyek közül a legfontosabbakat az 5.6. (b) ábra szemlélteti. A PSW-ben balról jobbra haladva van az átvitelbit, kiegészítő átvitelbit, a használt regiszterkészletre utaló bitek, a túlcsordulásbit és a paritásbit. A használt regiszterkészletre utaló bitek kivételével minden bitet az aritmetikai utasítások állítanak be. Az ábrán látható keresztvonalkázott mezők nem definiáltak.

Az IE regiszter a megszakítások egyedi és kollektív engedélyezését és tiltását kezeli. Ha az EA bit 0, akkor minden megszakítás tiltott. Ezen bit törlése lehetővé teszi minden további megszakítás tiltását egyetlen utasítással. Az EA bit 1-re állítása engedélyez minden olyan megszakítást, amelynek egyedi bitje 1. Az E2, E1 és E0 bitek engedélyezik vagy tiltják a három időzítő csatornát. Legfeljebb három csatornához tartozó számláló futhat egyszerre, és ezek mindegyike megszakítást vált ki, ha letelt az ideje. Az ES bit engedi vagy tiltja a soros vonali megszakítást. A további két bit a külső eszközök megszakítását engedi vagy tiltja. Ha engedi, akkor a 8051 két lábához kötött külső eszköz megszakítást generálhat. Ha tiltott, akkor az eszköz nem okozhat megszakítást.

Az IP regiszter határozza meg a megszakítások prioritási szintjét. Két szint van, alacsony és magas. Ha alacsony prioritási szintű megszakítás kiszolgálása közben magas prioritású keletkezik, akkor az alacsony feldolgozása felfüggeszthető, de fordítva nem. Ha a bit 1, akkor a hozzá tartozó megszakítás prioritási szintje magas, egyébként alacsony.

A TCON regiszter vezérli a 0. és az 1. időzítőt, ezek a fő időzítők. Az O1 és O0 biteket hardver állítja be, ha a hozzá tartozó időzítő túlcsordul. Az R1 és R0 bitek a futásvezérlő bitek, amelyek lehetővé teszik, hogy a program szoftveresen ki- vagy bekacsolja az időzítőt. A többi bit a két időzítő él- vagy szintvezérlésével kapcsolatos.

Az utolsó regiszter, a TMOD az időzítők üzemmódját határozza meg (8, 13 vagy 16 bites), hogy valódi időzítő avagy számláló, továbbá a fokozatot, hogy mely hardverjelek vezérelhetik az időzítőt. Az ábrán nem látható további regiszterek az áramellátással, és a soros port vezérlésével kapcsolatosak.



5.6. ábra. *(a) A 8051 memória szervezése. (b) A 8051 fő regiszterei*

Az összes eddig említett speciális regiszter és még néhány, mint az akkumulátor, a B/K-portok, mind a 128–255 memóriatartományban vannak. Ezek ugyanúgy címezhetők, mint bármely memóriacella, ahogy az R0–R7 regiszterek is. Például az akkumulátor, amely szerepel szinte minden aritmetikai utasításban, a 240-es címen található. A 8052 esetén, amely valódi memóriát tartalmaz a 128–255 tartományban, a speciális regiszterek átlapolják a memóriateret. Direkt címzés esetén a 8052-es a speciális regisztereket címzi, míg indirekt hivatkozáskor (regiszterben lévő mutatóval) a RAM-ot.

5.2. Adattípusok

A számítógépnek adatok kellenek. Valójában sok számítógépes rendszer nem tesz mást, mint pénzügyi, kereskedelmi, tudományos, mérnöki vagy egyéb adatokat dolgoz fel. Az adatokat a számítógépen belül speciális formában kell ábrázolni. Az ISA-szint adattípusok széles választékát nyújtja erre. Ezeket az adattípusokat vizsgáljuk a továbbiakban.

A kulcskérdés, hogy vajon a hardver mely konkrét adattípusokat támogatja. A hardvertámogatás azt jelenti, hogy vannak utasítások, amelyek megadott adatformátumot feltételeznek, a programozó nem választhat más adatábrázolást. Például, a könyvelők előszeretettel teszik a negatív számok esetén a – jelet a szám után, nem pedig elé, mint a számítógépes emberek. Képzeljük el, hogy egy könyvelőcég számítóközpontjának vezetője elintézte, hogy a gépen minden számot úgy ábrázolnak, hogy az előjelbit az utolsó bit, és nem az első, mint szokásosan. Ez kétségtelenül nagy benyomást tenne a főnökre, mivel minden szoftver korrektül dolgozna. A hardver megadott formátumban feltételezi az egész számok ábrázolását, és nem működik helyesen, ha bármilyen más formában ábrázoljuk a számokat.

Tekintsünk egy másik könyvelőcéget, amelyik éppen most kapta a megbízást, hogy ellenőrizze a szövetségi költségvetést (hogy az USA kormánya mennyivel tartozik egy polgárnak). A 32 bites aritmetika nem lenne elég, mert olyan számokkal kell dolgozni, amelyek nagyobbak, mint 2^{32} (kb. 4 milliárd). Az egyik megoldás lehetne, hogy két 32 bites egész számot használjunk minden szám ábrázolására, azaz számonként 64 bitet. Ha a gép nem támogatja az ilyen **dupla pontosságú** számokat, akkor minden aritmetikát szoftverrel kell megvalósítani, de ekkor a két 32 bites szám sorrendje tetszőleges lehet, hiszen a hardver ezzel nem törődik. Ez olyan adattípusra példa, amelynek nincs hardvertámogatása, és így nem is igényel megadott hardverábrázolást. A következő részben olyan adattípusokat tanulmányozunk, amelyeknek van hardvertámogatása, ezért speciális ábrázolási formát igényelnek.

5.2.1. Numerikus adattípusok

Az adattípusok két kategóriába sorolhatók: numerikus és nem numerikus adattípusok. Az egész számok kitüntetettek a numerikus adattípusok között. Méretük sokféle lehet, tipikusan 8, 16, 32 és 64 bit. Az egész számok tárgyak megszámlálására (például

a csavarhúzók száma egy vaskereskedésben), tárgyak azonosítására (például banksszámlaszám) és még sok minden másra alkalmazhatók. A modern számítógépek az egész számokat bináris kettes komplemens kódban ábrázolják, habár korábban más rendszereket is alkalmaztak. A bináris számokat az A) függelékben tárgyaljuk.

Vannak olyan számítógépek, amelyek mind az előjel nélküli, mind az előjeles egész számokat támogatják. Az előjel nélküli egészek esetében nincs előjelbit, minden bit számjegyet tartalmaz, így egy 32 bites szó értéke 0 és $2^{32} - 1$ közötti lehet (beleértve a határokat is). Ellenben, a 32 bites kettes komplemens kódban a legnagyobb szám $2^{31} - 1$ lehet, de képes negatív számokat is kezelni.

Olyan számok ábrázolására, amelyek nem fejezhetők ki egészként, mint például a 3,5, lebegőpontos számábrázolást alkalmaznak. Ezeket a B) függelékben tárgyaljuk. A lebegőpontos számok lehetnek 32, 64 vagy néha 128 bitesek. A legtöbb számítógép rendelkezik lebegőpontos aritmetikai utasításokkal. Sok gép különkülön regisztereket alkalmaz az egész és a lebegőpontos operandusok tárolására.

Néhány programozási nyelv, különösképpen a COBOL, decimális adattípust is megenged. Azok a gépek, amelyek COBOL-barátok akarnak lenni, gyakran hardvertámogatást biztosítanak erre, tipikusan 4 biten ábrázolva egy decimális számjegyet, aztán egy bájtba pakolnak két ilyen jegyet (binárisan kódolt decimális forma). Azonban az aritmetika nem működik helyesen a pakolt decimális számokon, ezért decimális aritmetikai korrekciós utasítások kellenek. Ezek az utasítások felismerik a 3. bitről történő átvitelt. Ez az oka annak, hogy a 3. bitről történő átvitel felismerésére külön feltételbit van a PSW-ben. Mellékesen, a hírhedt Y2K (2000. év) problémát nagyrészt COBOL-programozók okozták, mondván, elég és olcsóbb a dátum utolsó két jegyét tárolni, nem kell 16 bit. Némi optimalizálás.

5.2.2. Nem numerikus adattípusok

Amíg a korai számítógépek zöme életét számolással töltötte, a modern gépeket gyakran nem numerikus alkalmazásokra használják, mint az elektronikus levelezés, szörfözés a világhálón, digitális fényképezés, multimédia-készítés és lejátszás. Az ilyen alkalmazások más adattípusokat igényelnek, amit gyakran az ISA-szintű utasítások is támogatnak. A karakterek különösen fontosak, bár nem minden számítógép biztosít erre hardvertámogatást. A leggyakoribb karakterkódok az ASCII és a UNICODE. Az előbbi 7 bites, az utóbbi 16 bites ábrázolást alkalmaz. Mindkettőt a 2. fejezetben tárgyaltuk.

Nem szokatlan, hogy az ISA-szint rendelkezik olyan speciális utasításokkal, amelyek karakterláncokon operálnak. A karakterláncok néha speciális karakterrel végződnek. Alternatív megoldásként a lánc hosszát is tárolhatjuk, hogy tudjuk, hol a vége. Az utasítások másolást, keresést, szerkesztést és még néhány más funkciót valósítanak meg karakterláncokon.

A logikai (Boolean-) értékek különösen fontosak. Két különböző logikai érték van: az igaz és a hamis. Elméletileg egyetlen bit elég lenne logikai érték tárolására, 0 a hamis és 1 az igaz számára (vagy fordítva). A gyakorlatban azonban egy bájtot vagy egy szót alkalmaznak, mert az egyedi bitek nem címezhetők, így nehéz az elérésük. Szokásos, hogy a 0 a hamis, és minden más az igaz értéket ábrázolja.

Van olyan helyzet, amikor normális, hogy egy biten ábrázoljunk egy logikai értéket, nevezetesen, amikor logikai értékek teljes tömbjét kell tárolni, így 32 bit 32 logikai értéket tartalmazhat. Az ilyen adattípust **bittérkép**nek nevezzük, és több helyzetben is előfordul. Például, egy bittérkép tárolhatja a mágneslemez szabad blokkjait. Ha a lemez n blokkból áll, akkor a térkép n bitet tartalmaz.

Utolsó adattípusunk a mutató, amely egyszerűen egy memóriacímet jelent. Mutatókkal már többször találkoztunk. A Mic–x gép esetén az SP, PC, LV és a CPP mind példák mutatókra. Mutatóhoz adott távolságra lévő változó elérése, mint ahogyan az ILOAD utasítás dolgozik, minden gépen nagyon gyakori.

5.2.3. A Pentium 4 adattípusai

A Pentium 4 támogatja az előjeles 2-es komplemens egész, az előjel nélküli egész, a binárisan kódolt decimális egész és az IEEE 754 szabványnak megfelelő lebegőpontos számokat, amint az 5.7. ábra mutatja. A 8/16 bites eredetéből következően kezelni tudja az ilyen méretű egészeket, számos aritmetikai, logikai és összehasonlító utasítást kínálva. Az operandusokat nem kell igazítani a memóriában, bár hatékonysági okok miatt célszerű 4-gyel osztható címen tárolni az adatokat.

Típus	1 Bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit						
Előjeles egész		×	×	×		
Előjel nélküli egész		×	×	×		
Binárisan kódolt decimális		×				
Lebegőpontos				×	×	

5.7. ábra. *A Pentium 4 numerikus adattípusai. A támogatott típusokat × jelöli*

A Pentium 4 jó a 8 bites ASCII karakterek manipulálása terén is: vannak speciális utasítások karakterláncok másolására és keresésére. Ezek az utasítások használhatók mind ismert hosszú, mind végjellel megadott karakterláncokra. Ezeket az utasításokat gyakran használják a karakterláncot feldolgozó könyvtári eljárások.

5.2.4. Az UltraSPARC III adattípusai

Az UltraSPARC III adatformátumok széles skáláját nyújtja, mint azt az 5.8. ábrán is láthatjuk. Csak az egészek esetén lehet 8, 16, 32 és 64 bites operandus, mind előjeles, mind előjel nélküli változatban. Az előjeles egészeket 2-es komplemens kódban ábrázolja. A lebegőpontos operandusok lehetnek 32, 64 és 128 bitesek, és kielégítik az IEEE 754 szabványt (a 32 és 64 bitesek). Binárisan kódolt decimális ábrázolást nem támogat. Minden operandusnak a memóriában igazítottnak kell lennie.

Az UltraSPARC III erősen regiszterorientált gép, majdnem minden utasítás 64 bites regiszterre vonatkozik. A karakterlánc adattípust nem támogatja speciális utasítással.

Típus	1 Bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit						
Előjeles egész		×	×	×	×	
Előjel nélküli egész		×	×	×	×	
Binárisan kódolt decimális						
Lebegőpontos				×	×	×

5.8. ábra. *Az UltraSPARC III numerikus adattípusai. A támogatott típusokat × jelöli*

5.2.5. A 8051 processzor adattípusai

A 8051-es processzor nagyon korlátozott számú adattípussal rendelkezik. Egy kivételtől eltekintve minden regiszter 8 bites, így az egész számok is 8 bitesek. A karakterek is 8 biten tárolódnak. Lényegében az egyetlen adattípus, amelyet ténylegesen támogat a hardver, az a 8 bites bájt. Ez látható az 5.9. ábrán.

Típus	1 Bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit	×					
Előjeles egész		×				
Előjel nélküli egész						
Binárisan kódolt decimális						
Lebegőpontos						

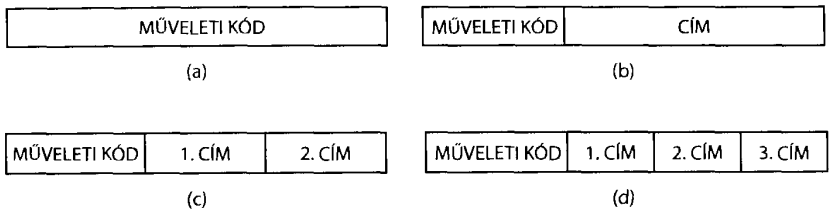
5.9. ábra. *A 8051 numerikus adattípusai. A támogatott típusokat × jelöli.*

A 8051 egy másik olyan adattípussal is rendelkezik, amely hardvertámogatást élvez, ez a bit. A 32. címtől kezdődő 16 bájtos memóriablokk bit-címezhető memória. Minden bit egyedileg címezhető 0-tól 127-ig terjedő eltolással. A 0. bit a jobb szélső bit a 32. bájtban, az 1. bit a következő és így tovább. Létezik egyedi bitekre vonatkozó utasítás olyan műveletek elvégzésére, mint a beállítás, törlés, ÉS, VAGY, komplementer, másolás és tesztelés. Beágyazott rendszerekben egyedi biteket használnak kapcsolók, lámpák és hasonlók állapotainak tárolására, ezért ezek közvetlen manipulálási lehetősége nagyon hasznos.

5.3. Utasításformátumok

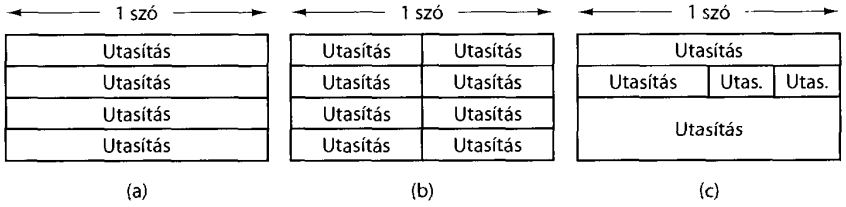
Minden utasítás tartalmaz egy műveleti kódot és általában valamilyen kiegészítő információt, mint hogy hol vannak az operandusok, és az eredmény hova kerül. Azt az általános módszert, amely meghatározza az operandusok helyét (tehát a címüket), **címzés**nek nevezzük, és ezt tanulmányozzuk a továbbiakban.

Az 5.10. ábra számos 2. szintű utasításformát mutat. Az utasítás mindig tartalmaz műveleti kódot, amely megmondja, hogy mit csinál az utasítás. Ezenkívül tartalmazhatja 0, 1, 2 vagy 3 operandus címét.



5.10. ábra. *Négy gyakori utasításforma: (a) 0 címes utasítás. (b) Egycímes utasítás. (c) Kétcímes utasítás. (d) Háromcímes utasítás*

Vannak gépek, amelyeknél minden utasítás azonos hosszúságú, mások esetén több, különböző hosszúságú utasítás lehet. Az utasítások hossza lehet a szóhosszal azonos, rövidebb vagy hosszabb is. Ha minden utasítás azonos hosszúságú, akkor könnyebb a dekódolás, de pazarló a tárolás, mert a közös hossz megegyezik a leghosszabb hosszával. Az 5.11. ábra néhány lehetséges összefüggést mutat a szóhossz és az utasításhossz között.



5.11. ábra. *Lehetséges szóhossz és utasításhossz*

5.3.1. Utasításformák tervezésének követelményei

Amikor egy számítógépet tervező csapat utasításformát választ, számos tényezőt kell figyelembe vennie. A döntés bonyolultságát nem szabad lebecsülni. Az új gép tervezésének korai fázisában kell dönteni az utasításformáról. Ha a számítógép kereskedelmileg sikeres, az utasításrendszer akár 20 évig is életben maradhat. Nagy jelentősége van annak a képességnek, hogy új utasítást lehessen bevezetni és kihasználni az új lehetőségeket, de csak akkor, ha az architektúra eléggé sikeres ahhoz, hogy hosszabb időre életben maradjon.

Egy adott architektúra hatékonysága nagymértékben függ a számítógép megvalósítására használt technológiától. Hosszú idő elteltével ez a technológia hatalmas változáson megy keresztül, így néhány ISA-választásról kiderül, hogy szerencsétlen volt. Például, ha a memória elérése gyors, akkor a veremelvű tervezés (mint a JVM) szerencsés, de ha lassú, akkor a regiszterelvű megoldás a jó (mint az UltraSPARC III). Az az olvasó, aki úgy gondolja, hogy ez a választás könnyű, javaslom, írja fel egy papírlapra a következők előrejelzését: (1) tipikus CPU órajelsebesség, (2) tipikus RAM-elérési idő 20 év múlva. Gondosan őrizze meg, 20 év elteltével bontsa fel, és olvassa el, mit jelzett előre.

Természetesen még az előrelátó tervezők sem mind képesek a jó választásra. De még ha képesek is lennének, meg kell birkózniuk a rövid távú kihívásokkal is. Ha az elegáns architektúra csak egy kicsit is drágább, mint a kellemetlen vetélytársaké, a cég nem lesz képes elég sokáig életben maradni, hogy élvezze az elegáns ISA előnyeit.

Ha minden azonos, a rövid utasítás jobb, mint a hosszú. Az a program, amely n számú 16 bites utasítást tartalmaz, csak feleakkora memóriát igényel, mint amelyik n darab 32 bites utasításokból áll. Az állandóan csökkenő memóriaárak mellett ez a tényező lehet, hogy kevésbé fontos a jövőben; így lenne, ha a szoftver nem terpeszkedne szét még gyorsabban, mint ahogy a memória ára esik.

Továbbá, az utasítás hosszának minimalizálása nehézzé teheti a dekódolást és az átlapolást. Tehát az utasítás méretének minimalizálásával több idő kell a dekódolásra és a végrehajtásra.

Egy másik szempont – az utasításhossz minimalizálása mellett – már most fontos, és a jövőben egyre fontosabb lesz a gyorsabb processzorok esetén, nevezetesen a memória sávszélessége (a memória által szolgáltatott bitek száma/másodperc). A processzorok elmúlt tíz évben bekövetkezett hatalmas sebességnövekedése nincs arányban a memóriák sávszélességének növekedésével. A processzorok feldolgozó képességének egyik súlyosbodó általános korlátja, hogy a memória nem tudja olyan ütemben szolgáltatni az adatokat és az utasításokat, amilyen mértékben a processzor képes feldolgozni. Minden memória sávszélességét az alkalmazott technológia és a mérnöki tervezés határozza meg. A sávszélesség mint szűk keresztmetszet nem csak a memóriára, hanem a gyorsítótárakra is vonatkozik.

Ha az utasítás-gyorsítótár sávszélessége t bps, az átlagos utasításhossz pedig r bit, akkor a gyorsítótár legfeljebb t/r utasítást képes szolgáltatni másodpercenként. Megjegyzendő, hogy ez *felső korlátja* annak, amilyen mértékben a processzor képes az utasításokat végrehajtani. Bár folynak olyan kutatási kísérletek, amelyek ezen látszólag leküzdhetetlen korlát leküzdését célozzák. Valóban, az utasításhossz korlátozza a végrehajtási sebességet (azaz a processzor sebességét). Mivel a modern processzorok egy órajel alatt több utasítást is képesek végrehajtani, ezért szükséges, hogy képesek legyenek egy órajel alatt több utasítást beolvasni. Az utasítás-gyorsítótárnak ez a tulajdonsága fontos tervezési kritériummá vált.

A másik tervezési kritérium, hogy elég hely legyen az összes megkívánt utasítás számára. Az utasítások száma nem lehet több, mint 2^n, ha az utasítások hossza n bit. Egyszerűen nincs elég hely a műveleti kód számára, hogy megkülönböztessünk több utasítást. A történelem időről időre bebizonyítja, hogy elegendő helyet kell hagyni a jövőbeli utasítások bevezetésére.

A harmadik tervezési szempont a címrész bitjeinek számát érinti. Tekintsük egy olyan gép tervezését, amely 8 bites karakterekkel dolgozik, és 2^{32} számú karaktert akar tárolni a memóriájában. A tervezők választhatják címzési egységként a 8, 16, 24, 32 stb. bitet, de más lehetőség is kínálkozik.

Képzeljük el, hogy a tervezőcsapat két részre oszlik, az egyik a 8 bites egységre szavaz, a másik pedig a 32 bites szavakra mint memóriaegységre. Az első csapat

2^{32} számú egybájtos memóriát kínál a 0, 1, 2, 3, ..., 4 294 967 295 címekkel. A másik csapat 2^{30} méretű memóriát kínál, ahol a szavak címei 0, 1, 2, 3, ..., 1 073 741 823.

Az első csapat kimutatja, hogy két karakter összehasonlítása a szószervezésű gép esetén úgy oldható meg, hogy a programnak nemcsak a karaktereket tartalmazó szavakat kell beolvasnia, hanem még a szavakból is ki kell vennie a karaktereket, hogy össze tudja őket hasonlítani. Ez extra utasítást igényel és helypazarló. Ellenben a bájtos szervezés esetén minden karakter külön címezhető, ami könnyűvé teszi az összehasonlítást.

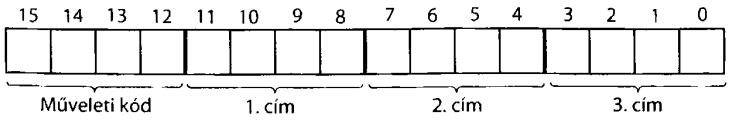
A 32 bites szavakat támogató csapat pedig azzal érvelhet, hogy esetükben csak 2^{30} cím van, így 30 bites cím elég az utasításokban, míg a 8 bitesek esetében 32 bit kell ugyanakkora memória megcímzésére. Rövidebb cím rövidebb utasítást eredményez, ami nem csak helytakarékosabb, de gyorsabban ki lehet olvasni. Alternatívaként megtartható a 32 bites cím, de ekkor a második csapat 16 GB memóriát tud címezni, az első csapat meg csak 4 GB-ot.

Ez a példa demonstrálta, hogy a finomabb memóriafelbontás ára a hosszabb cím és hosszabb utasítás. A memóriafelbontás szélsőséges megoldása, ha minden bit külön címezhető (ilyen volt a Burroughs B1700 gép). A másik véglet, ha a memória nagyméretű szavakból áll (ilyen volt a CDC Cyber 60 bites szavával).

A modern számítógépek kompromisszumra jutnak, bizonyos értelemben mindkét végletből a rosszat megtartva. Minden bit szükséges az egyedi bájtok címzése miatt, de minden memóriaművelet 1, 2, 3, néha 4 bájtos szavakat olvas egyszerre. Egy memóriabájt kiolvasása az UltraSPARC III esetén például minimum 16 bájtot érint, de valószínűleg egy teljes 64 bájtos gyorsítósort.

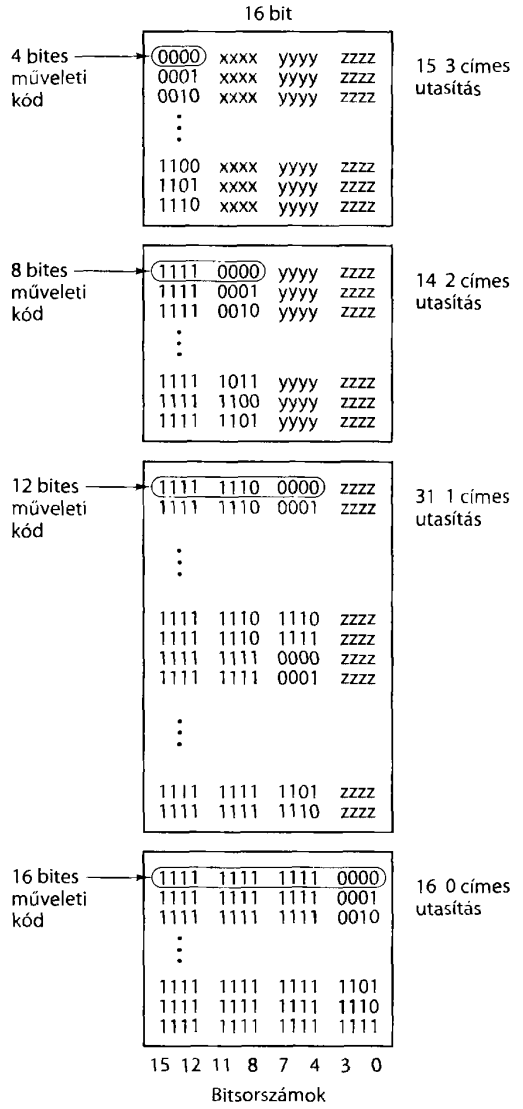
5.3.2. A műveleti kód kiterjesztése

Az előző szakaszban megmutattuk, hogy a rövid címzés és a finom memóriafelbontás milyen ellentétben áll. Ebben a szakaszban a műveleti kód és a címrész kompromisszumát vizsgáljuk. Tekintsünk egy $(n + k)$ bites utasításkészletet, ahol k bit a műveleti kód és n bit a címrész. Ez 2^k különböző utasítást és 2^n memóriacímet tesz lehetővé. Egy változat lehetne az $n + k$ bit felosztására, ha $k - 1$ bites műveleti kódot és $n + 1$ bit címrészt választanánk. Ekkor feleannyi utasításunk lenne, de kétszer akkora memóriát tudnánk címezni, vagy azonos nagyságú memória, de kétszer akkora felbontás lehetne. A $k + 1$ bites műveleti kód és $n - 1$ bites cím több utasítást, de kisebb vagy durvább felbontású memóriát eredményezne. Igen bonyolult kiegyensúlyozás lehetséges a műveleti kód és a címrész bitjeinek száma között. Az a séma, amelyet a továbbiakban tárgyalunk, az ún. **műveletikód-kiterjesztés**.



5.12. ábra. *Utasítás 4 bites műveleti kóddal és 4 bites címekkel*

A műveleti kód kiterjesztésének elvét világosan láthatjuk a következő példából. Tekintsünk egy olyan gépet, amelynek utasításai 16 bitesek, a címrész pedig 4 bites, amint az 5.12. ábra mutatja. Ez a helyzet elfogadhatónak tűnik, ha a gépnek



5.13. ábra. *Műveleti kód kiterjesztése: 15 3 címes, 14 2 címes, 31 1 címes és 16 0 címes műveleti kód. Az xxxx, yyyy és zzzz mezők 4 bites címeket tartalmaznak*

16 regisztere van (tehát 4 bittel lehet a regisztereket címezni), és minden aritmetikai műveletet regiszterekkel lehet végezni. Az egyik tervezési elgondolás szerint 4 bites lenne a műveleti kód, így 16 darab 3 címes utasításunk lenne.

Azonban, ha a tervezőknek olyan utasításrendszer kell, ahol 15 háromcímes, 14 kétcímes, 31 egycímes és 16 0 címes utasítás van, akkor 0-tól 14-ig kódolhatnák a háromcímes utasításokat, de a 15. műveleti kódot másként értelmezhetnék (lásd 5.13. ábra).

A 15. műveleti kódot úgy kell értelmezni, hogy a tényleges műveleti kódot a 8–15. bitek tartalmazzák, nem pedig a 12–15. bitek. A 0–3. és 4–7. bitek két címet alkotnak, mint egyébként is. A 14 darab kétcímes utasítás mindegyikében a bal oldali 4 bit értéke 1111, és a 8–11. biteken rendre a 0000-tól 1101-ig terjedő számok vannak. Azok az utasítások, amelyekben a bal szélső 4 biten 1111, és a 8–11. biteken 1110 vagy 1111 van, speciálisan kezelendők. Ezeket úgy kell értelmezni, hogy a tényleges műveleti kódot a 4–15. bitek tartalmazzák. Ez azt eredményezi, hogy további 32 műveleti kódunk lesz, ezek lesznek az egycímes utasítások. Mivel csak 31 darab egycímes utasítás kell, így az 1111 1111 1111 műveleti kódot úgy értelmezzük, hogy a tényleges műveleti kód a 0–15. biten van, és nincs címrész.

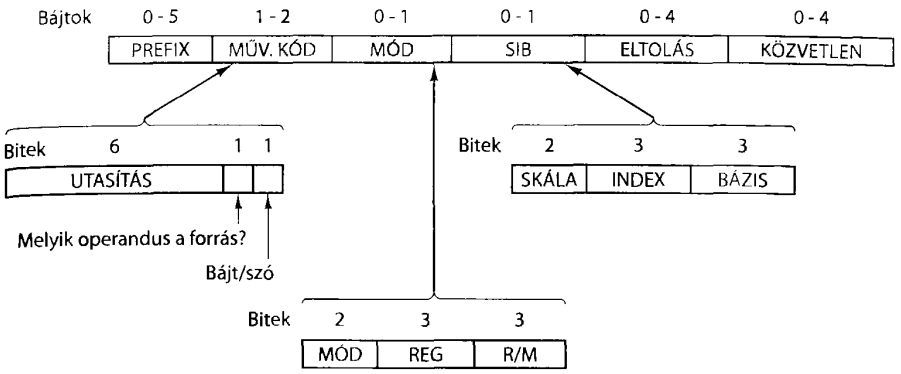
Ahogy haladtunk előre az eljárásban, egyre hosszabb műveleti kódot kaptunk: a háromcímesek hossza 4 bit, a kétcímeseké 8 bit, az egycímeseké 12 bit, a 0 címeseké pedig 16 bit lett.

A műveleti kód kiterjesztésének ötlete megmutatta a kompromisszumot a műveleti kód és az egyéb információk számára lefoglalt mezők mérete között. A gyakorlatban a műveleti kód kiterjesztése nem olyan világos és szabályos, mint azt a példánkban láthattuk. Valójában a változó hosszúságú műveleti kód kétféle módon is kihasználható. Először, az utasítások hosszát azonosra lehet venni, feltételezve, hogy a kevesebb bitet igénylő műveleti kódok esetén a fennmaradó bitek más információt tartalmaznak. Másodszor, az átlagos utasításhosszt minimalizálni lehet úgy, hogy a gyakran használt kódok hossza rövidebb, a ritkán használtaké pedig hosszabb lesz.

A változó hosszú műveleti kód ötletét szélsőségesen kihasználva elérhetjük, hogy az átlagos utasításhossz minimális legyen azáltal, hogy minden utasítást a legkevesebb szükséges biten kódolunk. Szerencsétlen módon ez azt eredményezné, hogy az utasításokat nem lehetne még bájthatárra sem igazítani. Bár volt olyan ISA (például az Intel 432), amely ilyen volt, az igazítás annyira fontos az utasítások gyors dekódolása miatt, hogy a fenti optimalizálás majdnem biztosan gazdaságtalan. Azonban gyakran használják ezt a módszert bájtszinten.

5.3.3. A Pentium 4 utasításformátumai

A Pentium 4 utasításformátumai nagyon bonyolultak és szabálytalanok, hatféle változó hosszúságú mezővel, amelyek közül öt opcionális. Az általános sémát az 5.14. ábra mutatja. Ez a helyzet azért alakult ki, mert az architektúra több generáción keresztül fejlődött, és a korai fázisban nem éppen jó döntéseket hoztak. Továbbá, a visszafelé kompatibilitás kényszere miatt később nem lehetett változtatni. Általánosan érvényes, hogy ha egy kétoperandusú utasítás egyik operandusa a memóriában van, akkor a másik nem lehet a memóriában (csak regiszterben). Tehát léteznek utasítások, amelyek regiszter-regiszter, regiszter-memória, memória-regiszter operandusokon dolgoznak, de nincs olyan utasítás, amelynek mindkét operandusa a memóriában lenne.



5.14. ábra. *A Pentium 4 utasításformátumai*

A korai Intel-architektúrákban minden utasítás műveleti kódja 1 bájtos volt, bár utasítások módosítására intenzíven használták az úgynevezett **prefix bájt** módszert. Ez azt jelenti, hogy van egy extra bájt az utasítás elején, ami módosítja a jelentését. A prefix bájtra példa a WIDE utasítás az IJVM esetén. Szerencsétlen módon a fejlődés során egyszer csak az Intel kifogyott a műveleti kódokból, és bevezette a 0xFF **kiterjesztő kódot (escape code)**, hogy lehetővé tegyen egy második műveletikód-bájtot.

A Pentium 4 egyedi bitjei nem sok információt adnak magáról az utasításról. Az egyetlen szerkezet a műveleti kód mezőjében bizonyos utasításoknál, hogy a szélső bit jelzi, hogy bájt vagy szó az operandus, egy másik bit pedig azt, hogy a memóriába vagy regiszterbe kerül a művelet eredménye. Tehát a műveleti kódot teljesen dekódolni kell annak meghatározásához, hogy milyen osztályba tartozik a végrehajtandó utasítás, vagy hogy mekkora a hossza. Ez megnehezíti a nagy hatékonyságú megvalósítást, mivel még ahhoz is intenzív dekódolást kell végrehajtani, hogy meghatározzák a következő utasítás címét.

Sok memóriára hivatkozó utasításban a műveleti kód bájtot a következő bájt adja meg ténylegesen az utasítást. Ez a 8 bit egy 2 bites MOD és két 3 bites REG és R/M mezőkre oszlik. Néha az első 3 bit a műveleti kódhoz tartozik, így összesen 11 bites a kód. Azonban a 2 bites mód mező csak négyféle operandusecímzési módot határozhat meg, és az egyik operandusnak regiszternek kell lennie. Logikusan az EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP regiszterek mindegyike megadható lenne, de a módszer miatt néhány kombináció tiltott, és speciális esetekre használják. Néhány mód további bájtokat igényel, ezek az **SIB (Scale, Index, Base – skála, index, bázis)**. Ez a séma nem ideális, de kompromisszumot képez a visszafelé kompatibilitás kényszere és az új lehetőségek bevezethetősége között.

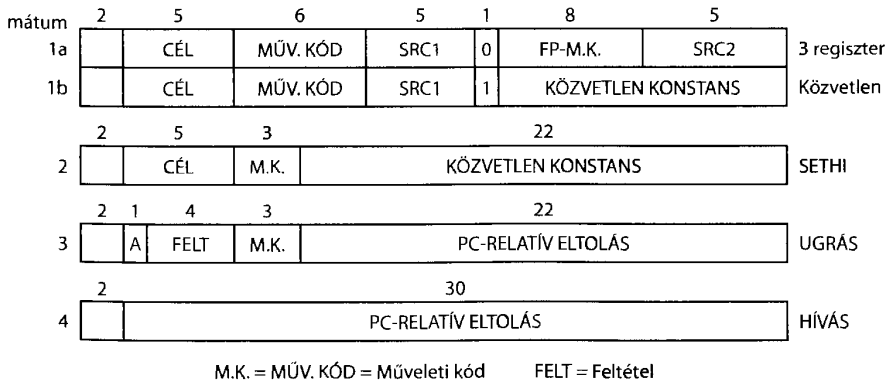
Mindezekhez vegyük még hozzá, hogy néhány utasítás 1, 2 vagy 4 további bájtot tartalmaz memóriacím (eltolás) megadására, és további 1, 2 vagy 4 bájtot a konstans számára.

5.3.4. Az UltraSPARC III utasításformátumai

Az UltraSPARC III ISA minden utasítása pontosan 32 bit és szóhatárra igazított. Az utasítások általában egyszerűek, és csak egy akciót határoznak meg. A tipikus aritmetikai utasítás két regisztert specifikál a bemenő operandusok és egyet az eredmény számára. Van olyan változat, amikor az egyik regiszter helyett egy 13 bites előjeles konstanst lehet megadni. A LOAD utasítás összeadja két regiszter (vagy egy regiszter és egy 13 bites konstans) tartalmát, így képezve a betöltendő cella címét.

Az eredeti SPARC nagyon korlátozott utasításformákat tartalmazott, mint azt az 5.15. ábra mutatja. Később új formák keletkeztek. E könyv írásának idején 31 a számuk, és ez növekszik. (Mikor áll elő vajon egy eég, hogy a világ legösszetettebb RISC gépét reklámozza?) A legtöbb új változat úgy keletkezett, hogy bizonyos mezőkből néhány bitet elhagytak. Például az eredeti elágazó utasítás a 3. formát használta 22 bites eltolással. Amikor a jövendölő elágazást bevezették, a 22 bitből 3-at levágtak, egyet a jövendölésre használtak (lesz/nem lesz elágazás), kettőt pedig a feltételkód megadására alkalmaztak. Így maradt 19 bit az eltolásra. Egy másik példa erre a sok adattípus-konvertáló utasítás (például egészből lebegőpontosba). Sok ezek közül az 1b formájú variánsa: az IMMEDIATE mezőt felbontották egy 5 bites részre (a forrásregiszter specifikálására) és egy 8 bites részre (további műveleti kódokra). Az utasítások többsége azonban továbbra is az ábrán látható formájú maradt.

Minden utasítás első két bitje segít meghatározni az utasítás formáját, és megadja a hardvernek, hol találja a műveleti kód maradék részét, ha van egyáltalán. Az 1a formában regiszter mind a két forrás; az 1b-ben az egyik forrásregiszter, a má-



5.15. ábra. *Az eredeti SPARC utasításformátumai*

sik egy –4096 és +4095 közötti konstans. A 13. bit a két forma közül választ (a jobb szélső bit a 0.). Mindkét esetben regiszter a cél. 64 utasítás kódolásához elegendő hely van az 1a formában; ezek pillanatnyilag későbbi felhasználásra foglaltak.

32 bites utasítás esetén nem lehet 32 bites konstanst megadni az utasításban. A SETHI utasítás 22 bitet ad meg, a maradék 10 bitet másik utasítás adja meg. Ez az egyetlen utasítás, amelynek ez a formája.

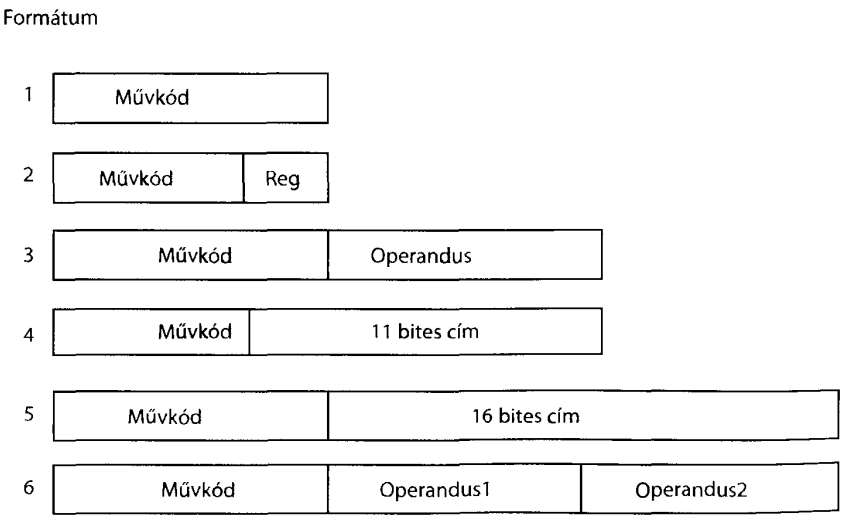
A nem jövendölő ugró utasítások 3-as formájúak, ahol a FELT mező mondja meg, hogy melyik feltételt kell ellenőrizni. Az A bit hivatott megakadályozni a késleltetési rés létrejöttét bizonyos feltételek esetén. A jövendölő ugrások formája ugyanilyen, kivéve a 19 bites eltolást, amelyet már említettünk.

Az utolsó forma a CALL utasításé, amely eljáráshívást végez. Ez az utasítás speciális, mert ez az egyetlen, amelyben 30 bit kell a cím megadására. Erre az ISA-ban csak 2 bites a műveleti kód. A célcím a megadott cím négyszerese, ami azt eredményezi, hogy az adott utasítás helyétől (közelítőleg) $\pm 2^{31}$ bájt tartományt lehet címezni.

5.3.5. A 8051 utasításformátumai

A 8051-es processzornak hatféle egyszerű utasításformája van, amelyek az 5.16. ábrán láthatók. Az utasítások hossza 1, 2 vagy 3 bájt. Az első forma csak egyetlen műveleti kódot tartalmaz. Például az akkumulátor tartalmát növelő utasítás használja ezt a formátumot.

A 2. formátum hossza szintén egy bájt, amely 5 bites műveleti kódot és 3 bites regisztercímet tartalmaz. Ilyen formátumú sok utasítás, amely az akkumulátor és



5.16. ábra. *A 8051 utasításformátumai*

egy regiszter tartalmán végez valamilyen műveletet (például regiszter és akkumulátor tartalmának összeadása, adatmozgatás regiszter és akkumulátor között).

A 3. formátumnak egy 1 bájtos operandusa van. Az operandus lehet közvetlen konstans, például amit akkumulátorba tölt, lehet eltolás, például az ugrás távolsága, vagy lehet bitsorszám, például az *n*. bitet beállító, törlő vagy ellenőrző utasításban.

A 4. és 5. formátum ugró és szubrutinhívó utasítások formája. A 11 bites változat akkor használható, ha nincs külső memória, tehát minden programcím 4096 alatti (8051) vagy 8192 alatti (8052). 8 KB-nál nagyobb külső memória esetén a 16 bites forma szükséges.

A 6. formátumú utasításoknak két 8 bites operandusa van. Sokféle utasítás ilyen formátumú, például egy 8 bites közvetlen operandusnak a lapkán lévő memóriacellába töltése.

5.4. Címzési módszerek

A legtöbb utasításnak van operandusa, ezért kell valamilyen módszer az operandus helyének megadására. Ezt nevezzük **címzés**nek, és ezt fogjuk a következőkben vizsgálni.

5.4.1. Címzési módok

Eddig nem sok figyelmet szenteltünk annak, hogy a címrész bitjeit hogyan értelmezik az operandusok címének meghatározásához. Itt az idő, hogy rátérjünk ennek a témának tanulmányozására, amelyet **címzési mód**nak nevezünk.

5.4.2. Közvetlen címzés

Az operandus specifikációjának a legegyszerűbb módja, ha az operandust közvetlenül az utasításban adjuk meg, nem pedig a címét vagy más információt, amely leírja, hogy az operandus hol található. Az ilyen operandust **közvetlen operandus**nak nevezzük, mert automatikusan betöltődik az utasítással, tehát azonnal hozzáférhető felhasználásra. Az 5.17. ábrán egy lehetséges közvetlen utasítás látható, amely az R1 regiszterbe a 4 konstanst tölti.

A közvetlen címzésnek az a lényege, hogy nem kíván külön memóriahivatkozást az operandus kiolvasására. A hátránya, hogy ily módon csak konstanst lehet megadni. Továbbá, az értékek száma korlátozott a mező méretével. Mégis sok architektúra használja ezt a technikát kis egész konstansokkal.

MOV	R1	4

5.17. ábra. *Utasítás a 4 közvetlen konstansnak az R1 regiszterbe töltésére*

5.4.3. Direkt címzés

A memóriabeli operandus megadása egyszerűen teljes címének megadásával megtehető. Ezt a módot **direkt címzés**nek nevezzük. Mint a közvetlen címzés, a direkt címzés használata is korlátozott: az utasítás minden végrehajtása ugyanazt a memóriamezőt érinti. Tehát amíg az érték változhat, a hely nem, vagyis a direkt címzés csak olyan globális változók elérésére használható, amelyek címe fordításkor ismert. Azonban sok program használ globális változókat, így ez a módszer széles körben használatos. Annak részletezése a későbbiekben következik, hogy a gép honnan tudja, hogy mely címek közvetlenek és melyek direktek.

5.4.4. Regisztercímzés

A regisztercímzés alapvetően azonos a direkt címzéssel, azzal a különbséggel, hogy memóriacím helyett regisztert határoz meg. Mivel a regiszterek (a gyors elérés és a rövid cím miatt) nagyon fontosak, a legtöbb számítógépen ez a címzési mód a leggyakoribb. Sok fordítóprogram alaposan megnézi, hogy mely változókat használják a legtöbbet (például ciklusváltozók), és ezeket regiszterekbe tölti.

Ez a címzési mód egyszerűen **regisztermód**ként ismert. A töltő/tároló architektúrák, mint az UltraSPARC III, majdnem kizárólag ezt a címzési módot alkalmazzák. Csak az olyan utasítások térnek el ettől, amelyek memóriából regiszterbe (LOAD), illetve regiszterből memóriába töltenek (STORE). De még ezeknél az utasításoknál is az egyik operandus regiszter, ahonnan, illetve ahova az operandus töltődik.

5.4.5. Regiszter-indirekt címzés

Ebben a címzési módban is a memóriából olvassuk a specifikált operandust, vagy oda írjuk, mint a direkt címzés esetén, de nem közvetlenül a címe van adva az utasításban. Helyette a címet egy regiszter tartalmazza. Ha egy címet ilyen módon adunk meg, **mutató**nak hívjuk. A regiszter-indirekt címzés nagy előnye, hogy úgy hivatkozik a memóriára, hogy annak címét nem kell az utasításban tárolni. Továbbá, az utasítás különböző végrehajtása más-más memóriamezőre hivatkozhat.

Annak bemutatására, hogy miért előnyös ez, tekintsük azt a ciklust, amely 1024 elemű egydimenziós tömb elemein megy végig, hogy kiszámítsa összegüket az R1

```
        MOV  R1,#0          ;gyűjtsük az eredményt R1-ben, kezdetben ez 0
        MOV  R2,#A          ;R2 = az A tömb címe
        MOV  R3,#A+4096     ;R3 = az A tömb utolsó eleme utáni cím
LOOP:   ADD  R1,(R2)        ;regiszter-indirekt címzés az operandus elérésére
        ADD  R2,#4          ;növeljük R2 tartalmát 4-gyel
        CMP  R2,R3          ;végeztünk?
        BLT  LOOP           ;ha R2 < R3, akkor nem végeztünk, folytassuk tovább
```

5.18. ábra. *Általános assembly program tömb elemeinek összegzésére*

regiszterben. A cikluson kívül valamely regiszter, mondjuk R2 vegye fel a tömb első elemének címét, egy másik, mondjuk R3 pedig a tömb utolsó eleme utáni címet. Ha az elemek 4 bájtos egész számok, és az első elem címe A, akkor az R3 tartalmának A + 4096-nak kell lennie. E számítás tipikus assembly kódját láthatjuk az 5.18. ábrán, kétcímes gép esetére.

Ebben a kis programban több címzési módot is használtunk. Az első három utasítás regisztermódot alkalmaz az első (cél) operandusra, és közvetlen címzést a második operandusra (a # jel konstanst jelöl). A második utasítás az A tömb *címét* teszi az R2 regiszterbe, és nem az A tartalmát. Hasonlóan, a harmadik utasítás a tömb utáni első címet teszi az R3-ba.

Érdemes megjegyezni, hogy a ciklusmag utasításai nem tartalmaznak memóriacímet. A negyedik utasításban regiszteres és regiszter-indirekt címzést használunk. Az ötödik utasítás regiszteres és közvetlen címzésű, a hatodik pedig kétszeresen regiszteres. A BLT utasítás lehet, hogy memóriacímet tartalmaz, de valószínűbb, hogy az ugrás helyéhez relatívan specifikálja, mert 8 bites eltolással megadható távolságra ugrik a BLT helyétől. A memóriahivatkozásokat elkerülve rövid és gyors ciklust készítettünk. Mellékesen, ez a program valódi Pentium 4 program, eltekintve attól, hogy a regisztereket és az utasításokat a könnyebb olvashatóság kedvéért átneveztük, mert a Pentium 4 assembly nyelve bizarr, a gép korábbi 8088-as életéből maradt ránk.

Csak elméletileg érdekes, hogy ezt a számítást regiszter-indirekt címzés nélkül is elvégezhetjük. A ciklus tartalmazhatná A-nak R1-hez való hozzáadását:

```
ADD  R1,A
```

Aztán minden ismétlésben az utasítást magát növelnénk 4-gyel, és ezzel a következőt kapnánk:

```
ADD  R1,A+4
```

És így tovább, amíg végig nem értünk.

Az olyan programot, mint ez is, amely saját magát módosítja, **önmódosító programnak** nevezzük. Az ötlet nem mástól, mint Neumann Jánostól ered, és nagyon is értelme volt a korai gépeken, ahol nem volt regiszter-indirekt címzés. Napjainkban az önmódosító program rettenetes stílusnak tekinthető, és nagyon nehéz megérteni a működését. Megjegyzendő, hogy az ilyen programot nem lehet megosztani több processzor között. Továbbá nem működik helyesen 1. szintű gyorsítótárral rendelkező gépeken, ha az I gyorsítótárból nem történik visszaírás (mert a tervezők feltételezték, hogy senki nem ír önmódosító programot).

5.4.6. Indexelt címzés

Gyakran jól kihasználható, ha ismerjük egy memória címét egy regiszter tartalmához képest. Láttunk már erre példát az IJVM esetén, ahol a lokális változókat az LV regiszterhez képest címzik. Azt a címzési módot, amikor a hivatkozott memória címét egy regiszter értéke és egy konstans határozza meg, **indexelt címzésnek** nevezzük.

IJVM esetén a lokális változók elérése egy regiszterben található memóriába mutató mutató (LV) és egy kicsi eltolási érték megadásával történik, mint az a 4.18. ábrán látható. Azonban másképpen is használható, lehet a mutató az utasításban és az eltolási érték a regiszterben. Tekintsük a következő számítást annak bemutatására, hogy ez hogyan is működik. Adott az A és B, egyenként 1024 elemet tartalmazó tömb. Ki akarjuk számítani, hogy van-e legalább egy olyan A_i, B_i pár, hogy egyikük sem 0. Ehhez kiszámítjuk az A_i AND B_i logikai szorzatokat, majd képezzük ezek OR művelettel vett összegét. Az egyik lehetőség az lenne, hogy az egyik regiszterbe raknánk az A, egy másikba a B tömb címét, és ciklusban együtt mennénk végig a két tömb elemein, mint ahogy azt egy tömb esetére az 5.18. ábrán láttuk. Ez biztosan járható út, de jobbat is tudunk; ezt szemlélteti az 5.19. ábra.

```
        MOV  R1,#0          ;gyűjtsük az OR eredményt R1-ben, kezdetben 0
        MOV  R2,#0          ;R2 = az A[i] AND B[i] szorzat indexe
        MOV  R3,#4096       ;az első túl nagy index
LOOP:   MOV  R4,A(R2)       ;R4 = A[i]
        AND  R4,B(R2)       ;R4 = A[i] AND B[i]
        OR   R1,R4          ;R1-be a logikai szorzatok OR összege
        ADD  R2,#4          ;i = i + 4 (a lépésköz egy szó = 4 bájt)
        CMP  R2,R3          ;végeztünk?
        BLT  LOOP           ;ha R2 < R3, folytassuk
```

5.19. ábra. *Általános assembly program 1024 elemű tömbök A_i AND B_i logikai szorzatának, majd ezek OR összegének kiszámítására*

A program működése nyilvánvaló. Négy regisztert használunk:

1. R1: a szorzatok OR összegét tartalmazza.
2. R2: az i index, amely a tömbök elemein való végighaladáshoz kell.
3. R3: a 4096 konstans, amely a legkisebb nem érvényes index.
4. R4: segédregiszter, amely a logikai szorzatok kiszámításához kell.

A regiszterek inicializálása után lépünk be a hat utasítást tartalmazó ciklusmagba. A *LOOP* címkéjű utasítás betölti az $A[i]$ értékét az R4 regiszterbe. A forrás címzése itt indexelt. Az R2 regiszter és az A címének (konstans, közvetlen operandus) az összege adja a címet. A két érték összege adja a memória címét, de nem tárolódik a felhasználó által látható egyetlen regiszterben sem. A

```
MOV  R4,A(R2)
```

jelölés azt jelenti, hogy a cél címzése regisztermódú az R4-gyel, a forrás pedig indexelt módú az A eltolással és R2 regiszterrel. Ha mondjuk az A értéke 124 300, akkor az utasítás az 5.20. ábrán látható formájú lehet.

MOV	R4	R2	124300

5.20. ábra. *A MOV R4,A(R2) utasítás egyik lehetséges reprezentációja*

A ciklusmag első végrehajtásakor R2 értéke 0 (mert így inicializáltuk), tehát a memóriacím az A_0 címe lesz, amely 124300. Ez a szó töltődik be R4-be, a ciklusmag következő végrehajtásakor az A_1 az 124304 címről és így tovább.

Mint ahogy korábban ígértük, itt az utasításban van az eltolási érték: a memóriába mutató cím, és a regiszterben egy kis egész szám van, amelyet lépésenként növelünk. Ez a forma természetesen megkívánja, hogy az utasításban az eltolási értéket tartalmazó mező elég nagy legyen, hogy a memóriacím beleférjen. Ez a módszer nem olyan hatékony, mint a másik, mégis gyakran ez a legjobb megoldás.

5.4.7. Bázis-index címzési mód

Néhány gép rendelkezik olyan címzési móddal, amely úgy határozza meg a hivatkozott memória címét, hogy összeadja két regiszter tartalmát, és (esetleg) ehhez hozzáad egy eltolási értéket. Néha ezt a módot **bázis-index címzés**nek nevezik. Az egyik regiszter a bázis, a másik az index. Ez a mód hasznos lenne esetünkben. A cikluson kívül az R5-be tennénk A címét, az R6-ba pedig B címét. Ekkor a LOOP-nál lévő és az azt követő utasítást a következőre cserélhetnénk.

```
LOOP:   MOV  R4,(R2+R5)
        ADD  R4,(R2+R6)
```

Ha lenne olyan címzési mód, amely két regiszter összegéhez képest eltolás nélküli indirekt hivatkozást valósítana meg, az ideális lenne. Vagy még az is segítene, ha 8 bites eltolást lehetne megadni, mert akkor 0-t adhatnánk meg. Ha azonban az eltolás mindig 32 bites, akkor nem nyerünk semmit. A gyakorlatban azok a gépek, amelyek alkalmazzák ezt a módot, általában 8 és 16 bites eltolást is biztosítanak.

5.4.8. Veremcímzés

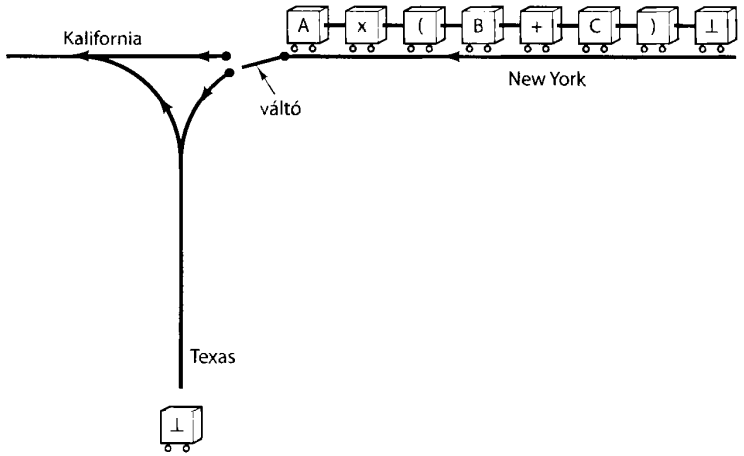
Már korábban megjegyeztük, mennyire kívánatos, hogy a gépi utasítás annyira rövid legyen, amennyire csak lehet. A végsőkig redukálva a cím hosszát, eljutunk a 0 címes utasításhoz. Mint már láttuk a 4. fejezetben, 0 címes utasítás lehetséges a veremmel kapcsolatban, például ilyen az IADD. Ebben a szakaszban alaposabban megvizsgáljuk a veremcímzést.

Fordított lengyel jelölés

A matematikában régi hagyomány, hogy a művelet jelét az operandusok közé írják, mint $x + y$, nem pedig utánuk, mint $x\ y\ +$. Ha a műveleti jel az operandusok között van, **infix** jelölésről beszélünk. Ha a műveleti jel az operandusok után áll, akkor **postfix** vagy **fordított lengyel jelölés** az elnevezés, a lengyel logikatudós J. Lukasiewicz (1958) után, aki kimutatta ennek a jelölésmódnak a jó tulajdonságait.

A fordított lengyel jelölésnek számos előnye van az infix jelöléssel szemben algebrai kifejezések esetén. Először is, minden formula kifejezhető zárójel nélkül. Másodszor, kényelmesen kiértékelhetők a formulák számítógéppel, verem használatával. Harmadszor, az infix operátorokra elsőbbségi szabály vonatkozik, amely tetszőleges és nem kívánatos. Például, tudjuk, hogy az $a \times b + c$ jelentése $(a \times b) + c$ és nem $a \times (b + c)$, mivel a szorzás nagyobb prioritású, mint az összeadás. Vajon nagyobb prioritású-e a balra léptetés, mint a logikai AND? Ki tudja. A fordított lengyel jelölés kiküszöböli ezt a kellemetlenséget.

Számos algoritmus ismert infix formulák fordított lengyel jelölésre konvertálására. Az itt következő algoritmus E. W. Dijkstra ötletén alapszik. Tegyük fel, hogy a formula a következő jeleket tartalmazhatja: változók, diadikus (kétoperandusú) műveletek: + – * /, továbbá a bal és jobb zárójelek. A formula végeinek jelzésére a formula elé és után a ⊥ jelet tesszük.



5.21. ábra. *Minden vasúti kocsi a fordított lengyel jelölésre konvertálandó infix formula egy jelét tartalmazza*

Az 5.21. ábra olyan vasúti pályát mutat, amely New Yorkból Kaliforniába vezet, középen Texas felé elágazással. A formula minden jelét egy kocsi reprezentálja. A vonat nyugat felé (balra) halad. Amikor egy kocsi a váltóhoz érkezik, meg kell állnia előtte, hogy megkérdezze, közvetlenül menjen-e Kaliforniába, vagy Texas felé kerüljön. Azok a kocsik, amelyek változót tartalmaznak, mindig közvetlenül

Kalifornia felé mennek, sohasem Texas felé. Az összes többi kocsinak meg kell néznie, hogy mi a Texas felé menő szerelvény utolsó kocsija, mielőtt a váltón áthaladna.

Az 5.22. ábra táblázata mutatja, hogy mi a teendő, annak függvényében, hogy milyen kocsi áll a váltónál, és mi az utolsó kocsija a Texas felé menő szerelvénynek. Az első ⊥ jel mindig Texas felé megy. A táblázatban a számok jelentése a következő:

1. A váltó előtt álló kocsi Texas felé menjen.
2. A Texas felé irányított szerelvény utolsó kocsija jöjjön vissza, és Kalifornia irányába menjen.
3. A váltó előtt álló és a Texas felé irányított szerelvény utolsó kocsija tűnjön el (törlődjön).
4. Stop. A Kaliforniába irányított szerelvény a fordított lengyel jelölést reprezentálja.
5. Stop. Hibás az eredeti formula, nem szabályos a zárójelezés.

A Texas felé menő szerelvény utolsó kocsija \ A váltó előtt álló kocsi	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

5.22. ábra. *Döntési táblázat a fordított lengyel jelölésre konvertáló algoritmushoz*

Minden lépés után újra össze kell hasonlítani a váltó előtt álló kocsit és a Texas felé irányított szerelvény utolsó kocsiját. A váltó előtt álló kocsi lehet ugyanaz, mint az előző összehasonlításban, vagy lehet az utána következő. Az eljárás a 4. lépéssel ér véget. Vegyük észre, hogy a Texas felé vezető pálya a verem, a Texas felé irányítás a verembe művelet (push), a Texas felé irányított szerelvény utolsó kocsijának Kalifornia felé irányítása (vagy eltüntetése) pedig a veremből művelet (pop).

Infix	Fordított lengyel jelölés
A + B × C	ABC × +
A × B + C	AB × C +
A × B + C × D	AB × CD × +
(A + B) / (C − D)	AB + CD − /
A × B / C	AB × C /
((A + B) × C + D)/(E + F + G)	AB + C × D + EF + G + /

5.23. ábra. *Néhány példa infix és a neki megfelelő fordított lengyel jelölésű formulára*

A változók sorrendje az infix és a prefix jelölésben megegyezik. A műveleti jelek sorrendje azonban nem mindig azonos. A fordított lengyel jelölésben a műveleti jelek olyan sorrendben szerepelnek, amilyen sorrendben azok végrehajtódnak a kifejezés kiértékelésekor. Az 5.23. ábra néhány példát mutat infix és a neki megfelelő fordított lengyel jelölésre.

Fordított lengyel jelölésű formulák kiértékelése

A fordított lengyel jelölés ideális formulák számítógépes kiértékelésére verem használatával. A formula *n* jelből áll, mindegyik vagy operandus, vagy műveleti jel. A fordított lengyel jelölésű formulának veremmel való kiértékelése egyszerű algoritmus. Olvassuk a formulát balról jobbra. Ha operandushoz érünk, rakjuk a verembe. Ha az aktuális jel műveleti jel, hajtsuk végre a megfelelő műveletet.

Az 5.24. ábra a

(8 + 2 × 5) / (1 + 3 × 2 − 4)

formula kiértékelését mutatja IJVM-ben. Az ennek megfelelő fordított lengyel jelölés a következő:

8 2 5 × + 1 3 2 × + 4 − /

Lépés	A maradék formula	Utasítás	Verem
1	8 2 5 × + 1 3 2 × + 4 − /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 − /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 − /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 − /	IMUL	8, 10
5	+ 1 3 2 × + 4 − /	IADD	18
6	1 3 2 × + 4 − /	BIPUSH 1	18, 1
7	3 2 × + 4 − /	BIPUSH 3	18, 1, 3
8	2 × + 4 − /	BIPUSH 2	18, 1, 3, 2
9	× + 4 − /	IMUL	18, 1, 6
10	+ 4 − /	IADD	18, 7
11	4 − /	BIPUSH 4	18, 7, 4
12	− /	ISUB	18, 3
13	/	IDIV	6

5.24. ábra. *Fordított lengyel jelölésű formula kiértékelése veremmel*

Az ábrán bevezettük az IMUL és IDIV utasításokat, amelyek a szorzás, illetve az osztás műveletét végzik. A verem tetején lévő operandus a jobb, nem pedig a bal operandus. Ez fontos az osztás (és kivonás) esetén, mivel az operandusok sorrendje lényeges (nem úgy, mint az összeadásnál és a szorzásnál). Más szóval, az IDIV uta-

sítást körültekintően definiáltuk, először az osztandót kell a verembe tenni, majd az osztót, és ezután kell elvégezni a műveletet, hogy helyes eredményt kapjunk. Vegyük észre, hogy milyen egyszerű kódot generálni az IJVM-re fordított lengyel jelölésből: csak olvassuk balról jobbra a formulát, és jelenként adjunk ki egy utasítást. Ha a jel konstans vagy változó, akkor olyan utasítást kell kiadni, amely azt a verembe teszi. Ha a jel műveleti jel, akkor a műveletet megvalósító utasítást kell kiadni.

5.4.9. Címzési módok elágazó utasításokban

Eddig olyan utasításokat vizsgáltunk, amelyek adatokon végeznek műveleteket. Az elágazó (és eljáráshívó) utasítások szintén igényelnek olyan címzési módot, amelylyel meg lehet adni az ugrási hely címét. Az eddig tanulmányozott címzési módok nagyrészt elágazó utasításokra is alkalmazhatók. A direkt címzés biztosan lehetséges, amikor a célcímet teljes egészében közvetlenül az utasításban adjuk meg.

Azonban más címzési módok is értelmesek. A regiszter-indirekt címzés lehetővé teszi, hogy a célcímet előre kiszámítsuk, regiszterben tároljuk, és így végezzük az ugrást. Ez a mód nyújtja a legnagyobb hajlékonyságot, mert a célcímet futási időben számíthatjuk ki. Ez persze a legjobb módja annak is, hogy olyan hibát kövessünk el, amelyet majdnem lehetetlen kideríteni.

További lehetőség az indexelt címzés, amikor a célcím egy regiszterhez képest adott eltolási érték. Ennek ugyanazok a tulajdonságai, mint a regiszter-indirekt címzésnek.

A PC-relatív címzés a másik lehetőség. Ebben a módban az eltolási érték (előjeles egész) hozzáadódik az utasításszámláló értékéhez, így képezve az ugrási címet. Valójában ez egyszerűen indexelt címzés, ahol a PC a regiszter.

5.4.10. A műveleti kód és a címzési mód ortogonalitása

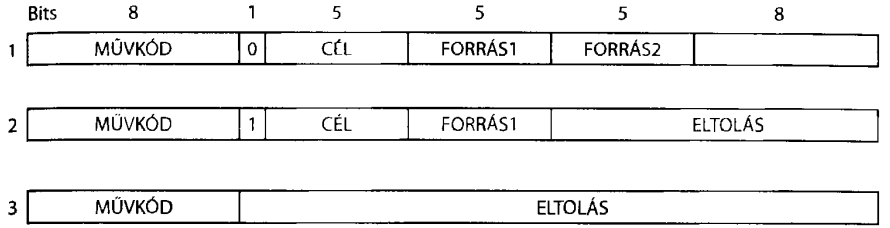
Szoftverszempontból az utasítások és a címzési módok szabályos szerkezete lenne kívánatos a legkevesebb utasítás formátummal. Ilyen szerkezet könnyűvé tenné a fordítóprogramok számára, hogy jó kódot generáljanak. Minden műveleti kód megengedne minden értelmes címzési módot. Továbbá, minden regiszter elérhető lehetne minden regisztermódban [beleértve a keretmutatót (FP), a veremmutatót (SP) és az utasításszámlálót (PC)].

Egy háromcímes gép világos tervezésére mutat példát az 5.25. ábra 32 bites utasításokkal. A lehetséges műveleti kódok száma 256. Az 1. formátumban minden utasítás egy cél- és két forrásregisztert tartalmaz. Minden aritmetikai-logikai utasítás ilyen formátumú.

A fel nem használt 8 bites mező az utasítás végén további utasításmegkülönböztetést tesz lehetővé. Például, egyetlen műveleti kód lehetne minden lebegőpontos utasításra, és ez a mező jelezhetné a különbségeket. Ha a 23. bit 1-es, akkor ez azt jelentené, hogy a 2. utasításforma érvényes, azaz a második operandus most nem

regiszter, hanem egy 13 bites közvetlen adat. A LOAD és STORE utasítások is használhatnák ezt a formát, hogy indexmódban hivatkozzanak a memóriára.

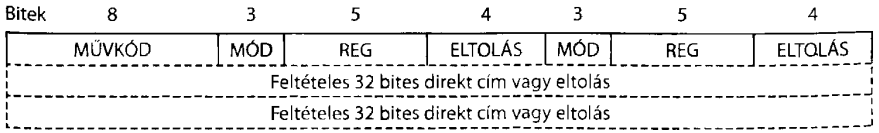
Kellene néhány speciális utasítás, például feltételes elágazások, de ezek beilleszthetők lennének a 3. formába. Például egy-egy műveleti kód kellene minden (feltételes) elágazás, eljáráshívás stb. számára, 24 bites helyet hagyva a PC relatív eltolási értéknek. Feltéve, hogy az eltolási érték szóban értendő, így az átfogható tartomány ±32 MB lenne. Ugyancsak kellene ilyen 3. formátumú LOAD és STORE utasítás is. Ez nem lenne teljesen általános (például csak R0-ba lehetne betölteni, vagy R0-ból lehetne tárolni), de csak ritkán használnák.



5.25. ábra. *Egy egyszerű háromcímes gép utasításformáinak tervezete*

Tekintsük most egy kétcímes gép tervezését, amelynek mindkét operandusa memóriaszó lehetne. Ezt mutatja az 5.26. ábra. Ez a gép képes összeadni regisztert és memóriaszót, memóriaszót és regisztert, regisztert és regisztert, illetve memóriaszót és memóriaszót. Jelenleg a memóriaelérés viszonylag költséges, így ez a terv nem túl népszerű. Azonban, ha a jövőbeli technológiai fejlődés olcsóvá teszi a gyorsítótár és a memória elérését, akkor különösen egyszerű és hatékony fordítást tesz majd lehetővé. A PDP–11 és VAX gépek, amelyek két évtizeden keresztül uralták a miniszámítógépek világát, hasonló felépítésűek voltak.

Most is 8 bit van a műveleti kód számára, de most 12-12 biten lehet megadni a forrás és a cél címét. Minden operandus számára 3 bit mondja meg a módot, 5 bit a regisztert és 4 bit az eltolást. A 3 módbittel megkülönböztethetnénk a közvetlen, direkt, regiszter-, regiszter-indirekt, index- és veremcímzési módokat, továbbá maradna még két lehetőség később bevezetendő mód számára. Ez világos és szabályos tervezet, amely hajlékony, és könnyű rá fordítani, különösen, ha az utasításszámláló, a veremmutató és a lokális változók regisztere mind általános regiszter, amelyek egyformán elérhetők.



5.26. ábra. *Egy egyszerű kétcímes gép utasításformáinak tervezete*

Az egyetlen probléma, hogy a direkt címzéshez több bit kell. A PDP–11 és a VAX azt a megoldást választotta, hogy további szót illesztett az utasításhoz minden közvetlenül címzett operandus számára. Szintén megtehetnénk, hogy valamelyik indexmód esetén 32 bites eltolást alkalmazzunk. Így a legrosszabb esetben, például memória-memória összeadásánál (ADD), ha mindkét operandus közvetlenül címzett, az utasítás 96 bit hosszú lenne, ami három sínciklust igényelne. Másrészt a legtöbb RISC-terv legalább 96 bitet igényel két memóriaszó összeadására és legalább négy sínciklust.

Sok alternatív megoldása lehet az 5.26. ábrán bemutatottnak. Tervezetünk szerint az

```
i = j;
```

utasítás végrehajtása egyetlen 32 bites utasítással megoldható, feltéve, hogy *i* és *j* mindegyike az első 16 lokális változó között van. Másrészt, minden olyan változó, amely nincs az első 16 lokális változó között, 32 bites eltolási értéket követel. Egy másik megoldás lehetne egy további utasításforma egyetlen 8 bites eltolással a két 4 bites helyett, plusz egy szabály, amely szerint a forrás és a cél közül csak az egyik használhatná ezt a módot. A lehetőségek nem korlátozottak, és a géptervezőknek sok tényezőt kell figyelniük ahhoz, hogy jó eredményhez jussanak.

5.4.11. A Pentium 4 címzési módjai

A Pentium 4 címzési módjai nagyon szabálytalanok, és attól is függnek, hogy 16 vagy 32 bites az utasítás. A továbbiakban eltekintünk a 16 bites módtól, a 32 bites is eléggé rossz. A támogatott módok: közvetlen, direkt, regiszter-, regiszter-indirekt, indexelt címzés és még egy speciális mód tömbelemek címzésére. Az a gond, hogy nem minden mód alkalmazható minden utasításban, és nem minden regiszter használható minden módban. Ez nagyon megnehezíti a fordítóprogram-készítők dolgát, és gyenge kódot eredményez.

Az 5.14. ábrán a MODE bájt vezérli a címzési módot. Az egyik operandust a MOD és az R/M mezők határozzák meg. A másik operandus mindig regiszter, hogy melyik, azt a REG mező tartalmazza. Az 5.27. ábra felsorolja azt a 32 kombinációt, amelyet a 2 bites MOD és 3 bites R/M mező szolgáltat. Például, ha mindkét mező 0, akkor az operandust a memóriából olvassa, amelynek címe az EAX regiszterben van.

A 01 és 10 oszlopok tartalmazzák azokat a módokat, amelyekben 8, illetve 32 bites eltolást lehet megadni a (gépi) utasítás végén. Ha 8 bitest választunk, akkor a hozzáadás előtt előjelesen kiterjeszti 32 bitesre. Például az ADD utasítás az R/M = 011, MOD = 01 és 6 eltolás esetén kiszámítja az EBX és 6 összegét, és az így kapott címről veszi az egyik operandust. Az EBX nem módosul.

A MOD = 11 oszlopnál két regiszter közül választhatunk. Szavas utasítások az első változatot, a bájtos utasítások pedig a második változatot használják. Megjegyzendő, hogy a táblázat nem teljesen szabályos. Például nem lehet az EBP indirekt és az ESP relatív címzés.

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX vagy AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX vagy CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX vagy DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX vagy BL
100	SIB	SIB OFFSET8-cal	SIB OFFSET32-vel	ESP vagy AH
101	Direkt	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP vagy CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI vagy DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI vagy BH

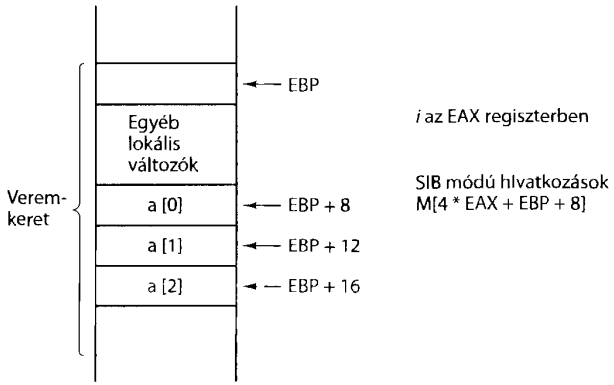
5.27. ábra. *A Pentium 4 32 bites címzési módjai. M[x] az x című memóriaszó*

Néhány módban egy további bájt, az úgynevezett **SIB (Scale, Index, Base)** követi a MODE bájtot (lásd 5.14. ábra). Ha a SIB bájt meg van adva, akkor az operandus címét a következőképpen kell kiszámítani: szorozni kell az indexregisztert 1-gyel, 2-vel, 4-gyel vagy 8-cal (a SCALE-től függően), ezt hozzá kell adni a bázisregiszterhez, és MOD-tól függően ehhez kell adni a 8 vagy 32 bites eltolási értéket. Majdnem minden regiszter használható bázis- és indexregiszterként.

A SIB mód tömbelemek címzése esetén hasznos. Példaként tekintsük a következő Java-utasítást:

```
for (i = 0; i < n; i++) a[i] = 0;
```

ahol az *a* tömb az aktuális eljárásra lokális, és 4 bájtos egészeket tartalmaz. Tipikusan az EBP használatos a lokális változókat tartalmazó veremkeret kezdőcímének megadására; ezt mutatja az 5.28. ábra. A fordító *i* értékét az EAX regiszterben tarthatja. Az *a[i]* elérése végett a SIB módot használhatja, ahol az operandus 4 × EAX, EBP és 8 összege. Így egyetlen utasítás elvégezné a munkát.



5.28. ábra. *Az a[i] tömbelem elérése*

Megéri ez a mód a fáradságot? Nehéz válaszolni a kérdésre. Kétségtelen, hogy ha jól használják, akkor ezzel a móddal több ciklus megtakarítható. Használatának gyakorisága függ a fordítóprogramtól és az alkalmazástól. Az a gond, hogy ennek a módnak a megvalósítása a lapka egy részét elviszi, amelyet másra is lehetne használni, ha nincs ez az utasítás. Például az 1. szintű gyorsítótár nagyobb lehetne, vagy a lapka mérete kisebb, ami nagyobb órajelsebességet engedne.

Ezek azok a kompromisszumok, amelyekkel a tervezők állandóan szembesülnek. Általában intenzív tesztelést végeznek mielőtt szilikonba öntenék a tervet. Azonban nagyon jól megalapozott elképzeléssel kell rendelkezni a szimulációhoz, figyelembe véve a várható terhelést. A 8088 tervezői biztosan nem teszteltek webböngészővel. Mindazonáltal, nem sok számítógép elődjét használták webböngészésre, így a 20 évvel ezelőtti döntések teljesen hamisak lehetnek a mai alkalmazások szempontjából. Azonban a visszafelé kompatibilitás kényszere miatt nehéz kidobni azt, ami egyszer már bekerült.

5.4.12. Az UltraSPARC III címzési módjai

Az UltraSPARC ISA minden utasítása közvetlen vagy regisztercímzést használ, kivéve azokat, amelyek memóriára hivatkoznak. Regisztermódban 5 biten van adva a regiszter. A közvetlen módban 13 bites (előjeles) adat lehet a konstans. Nincs más módú aritmetikai, logikai vagy hasonló utasítás.

Háromféle utasítás hivatkozik memóriára: a betöltő (LOAD), a tároló (STORE) és egy multiprocesszor-szinkronizáló utasítás. A LOAD és STORE utasítások kétféle módban dolgozhatnak. Az egyik módban két regiszter tartalmának összege adja a címet. A másik a hagyományos indexelés 13 bites eltolási értékkel.

5.4.13. A 8051 címzési módjai

A 8051 címzési módja nagyon szabályos. Öt alapvető címzési módja van. A legegyszerűbb az implicit mód, amely az akkumulátort használja. Sok utasítás végez műveletet egy operandus és az akkumulátor tartalmán, ilyen az összeadás, kivonás, ÉS, VAGY. Az akkumulátort használó utasításokban nincs speciális bit, amely azt jelezné, hogy az akkumulátort kell használni. Ezt a műveleti kód magába foglalja.

A másik a regisztermód, amikor az egyik operandus regiszter. Regiszter lehet akár forrás, akár cél. A harmadik mód a direkt címzés, amikor az operandus a memóriában van, amelynek címe az utasítás része. A negyedik mód a regiszter-indirekt címzés, amikor az operandus címe regiszterben van. Mivel minden normál regiszter 8 bites, ezért az operandusnak a memória alsó 256 címén kell lennie. Az ötödik mód a közvetlen mód, amikor az operandus magában az utasításban van megadva.

A 8051 speciális címzési módot tartalmaz néhány olyan utasításban, amely memóriaelérést végez. Emlékeztetünk, hogy a 8051-nek lehet 64 KB külső programés 64 KB külső adatmemóriája is. Ezért szükség van valamilyen címzési módra, amely lehetővé teszi ezek elérését. A külső memória elérését két utasítás biztosítja, az LJMP, amely 16 bites eltolási értéket tartalmaz, így lehetővé tesz bármely memóriacímre ugrást, és az LCALL, amely szintén 16 bites eltolást tartalmaz, így bármely memóriacímen lévő eljárást tud hívni.

A külső adatmemória elérése más módszerrel történik. A 8051 rendelkezik egy 16 bites mutató regiszterrel, ez a DPTR, amely 16 bites memóriacímet tartalmazhat. Ezt a programok betölthetik, aztán a betöltött érték indirekt címzésre használható, így elérhető a teljes 64 KB-os tartományt.

5.4.14. A címzési módok összefoglalása

Jó néhány címzési módot tanulmányoztunk eddig. A Pentium 4, az UltraSPARC III és a 8051 által használtakat az 5.29. ábrán látható táblázatban foglaltuk össze. Mint már korábban megjegyeztük, nem minden mód használható minden utasításban.

Címzési mód	Pentium 4	UltraSPARC III	8051
Akkumulátor			×
Közvetlen	×	×	×
Direkt	×		×
Regiszter	×	×	×
Regiszter-indirekt	×	×	×
Index	×	×	
Bázis-index		×	
Verem			

5.29. ábra. *A címzési módok összehasonlítása*

A gyakorlatban hatékony architektúrához nem kell sokféle címzési mód. Mivel az ISA-szinten írt kód nagy részét fordítóprogramok generálják (a 8051 esetleg kivétel), ezért a legfontosabb szempont az architektúra címzési módjánál, hogy kevés és világos legyen a választás, könnyen kiszámítható költséggel (a végrehajtási időt és a kód méretét tekintve). Ez általában azt jelenti, hogy a számítógép szélsőséges pozíciót válasszon: vagy mindent, vagy csak egy választást nyújtson. Minden közbülső esetben a fordítóprogram választásra kényszerül, esetleg kellő ismeret hiányában.

Tehát a legtisztább architektúrában általában nagyon kevés címzési mód van, amelyek használata szigorúan korlátozott. A gyakorlatban a közvetlen, direkt, regiszter- és indexelt mód elegendő a legtöbb alkalmazáshoz. Hasonlóan, minden regiszter használata megengedett (beleértve a lokális változók keretének mutatóját, a veremmutatót és az utasításszámlálót) minden esetben, amikor regiszter szóba kerülhet. Bonyolultabb címzési módok csökkenthetik az utasítások számát, de ennek az az ára, hogy olyan utasítássorozat alakul ki, amely nehezen párhuzamosítható más utasítássorozatokkal.

Ezzel befejeztük a különböző műveleti kódok és címzések közötti lehetséges kompromisszumok tanulmányozását. Egy új számítógép megítélésénél érdemes

megvizsgálni nemcsak az utasításkészletet és a címzési módokat, hanem azt is, hogy mi az oka az adott választásnak, és mi lett volna a következménye egy ettől eltérő választásnak.

5.5. Utasítástípusok

Az ISA-szintű utasításokat megközelítőleg fél tucat csoportba lehet osztani, amelyek eléggé hasonlók, akármilyen gépet is tekintünk, még akkor is, ha a gépek részleteiben nagyon eltérők. Azonban minden gépnek van néhány szokatlan utasítása, amelyeket vagy kompatibilitási kényszer miatt, vagy azért vezettek be, mert a tervezőknek valami zseniális ötletük volt, vagy mert kormányzati hivatal fizetett érte a gyártónak. A továbbiakban a teljesség igénye nélkül röviden áttekintjük az általános kategóriákat.

5.5.1. Adatmozgató utasítások

Adat másolása egyik helyről a másikra a legalapvetőbb az összes művelet között. Másoláson azt értjük, hogy létrehozunk egy új objektumot, amelynek a bitképe azonos az eredetivel. A „mozgatás" szót itt nem a köznapi értelemben használjuk. Amikor azt mondjuk, hogy a szekrényt a szoba másik sarkába mozgattuk, az nem azt jelenti, hogy a szekrény ott maradt az eredeti helyén, és az új helyen keletkezik a szekrénynek egy azonos másolata. Ha azt mondjuk, hogy a 2000 memóriacím tartalmát egy regiszterbe mozgatjuk, akkor ezen mindig azt értjük, hogy az eredetivel azonos másolat keletkezik a regiszterben, és a 2000 memóriacím tartalma változatlan marad. Az adatmozgató utasításokat szerencsésebb lenne adatmásoló utasításoknak hívni, de az „adatmozgató" kifejezés már régóta elterjedt.

Annak, hogy adatot egyik helyről a másikra kell másolni, két indoka van. Az egyik alapvető: értékadás a változónak. Az

```
A = B
```

értékadás megvalósítása az, hogy az *B* memóriacímről az *A* memóriacímre másolódik az ottani érték, mert a programozó ezt kérte. A másik indok a másolásra az, hogy az adat hatékony elérését és felhasználását biztosítsuk. Mint már láttuk, sok utasítás csak regiszterben képes elérni az adatokat. Mivel az adatoknak forrásuk szerint két lehetséges helye van: a memória vagy a regiszter, és a cél is vagy a memória, vagy a regiszter, ezért négy esete lehet a másolásnak. Néhány gépnek négy külön utasítása van a négy esetre. Mások mind a négy esetre ugyanazt az utasítást nyújtják. Még mások a LOAD utasítást adják a memóriából regiszterbe, a STORE utasítást a regiszterből memóriába, a MOVE utasítást pedig a regiszterből regiszterbe történő átvitelre, és nincs a memóriából memóriába közvetlenül mozgató utasítás.

Az adatmozgató utasításoknak valahogy meg kell mondani, hogy mekkora mennyiségű adatot mozgassanak. Bizonyos gépeken létezik olyan utasítás, amely képes változó mennyiség átvitelére 1 bájttól akár a teljes memóriáig. A rögzített szóméretű gépeken az egy utasításban átvihető mennyiség általában pontosan egy szó. Ennél nagyobb vagy kisebb mennyiség átvitelét szoftver segítségével kell megoldani léptetések és összevonások alkalmazásával. Néhány ISA lehetővé teszi egy szónál kisebb (általában bájt-többszörös) és a szó többszörösének az átvitelét is. Több szó másolása trükkös, különösen, ha a szavak száma nagy, mert hosszú ideig tarthat, és esetleg közben fel kell függeszteni a műveletet. Néhány változó szóhosszúságú gépnek van olyan másoló utasítása, ahol csak a forrás és a cél címét kell megadni, a hosszt nem. Ekkor a másolás a forrásadatban található adatvége jelig történik.

5.5.2. Diadikus műveletek

A kétoperandusú, egyetlen eredményt szolgáltató műveleteket nevezzük diadikus műveleteknek. Minden ISA rendelkezik egész számokon végezhető összeadás és kivonás művelettel. A szorzás és az osztás szintén majdnem általános az egész számok körében. Bizonyára nem szükséges magyarázni, hogy a számítógépek miért rendelkeznek aritmetikai utasításokkal.

A diadikus műveletek másik csoportjába tartoznak a logikai utasítások. Habár 16 kétváltozós logikai művelet van, nagyon kevés vagy talán egyetlen gép sem tartalmazza mind a 16 műveletet. Szokásos az AND, OR és NOT*, néha a XOR kizáró vagy, a NOR, valamint a NAND is megtalálható.

Az AND művelet egyik fontos alkalmazása bitek kivágása szavakból. Tekintsünk például egy 32 bites szóval rendelkező gépet, ahol négy 8 bites karaktert tárolnak egy szóban. Tegyük fel, hogy a második karaktert nyomtatáskor el kell különíteni a többitől. Tehát olyan szót kell létrehozni, amelyben az utolsó 8 biten van a kívánt karakter, a többi 24 bitnek meg 0-nak kell lennie **(jobbra igazított)**.

Karakter kivágása úgy történik, hogy a karaktert tartalmazó szónak és egy konstansnak, a **maszk**nak az AND művelettel vett eredményét képezzük. A művelet eredményeként az összes nem kívánt bit értéke 0 lesz, kimaszkolódik, mint azt az alábbi példán láthatjuk.

```
10110111 10111100 11011011 10001011 A
00000000 11111111 00000000 00000000 B (maszk)
00000000 10111100 00000000 00000000 A AND B
```

Ezután 16 bittel jobbra léptetve megkapjuk a kívánt szót, amely most már a jobb szélén tartalmazza a kívánt karaktert.

Az OR művelet egyik fontos felhasználása az, amikor biteket építünk be egy szóba, tehát a kivágás ellentettje. Változtassuk meg egy 32 bites szó utolsó 8 bitjét

* A megvalósított logikai műveletek között csaknem mindig szerepel a NOT művelet, ezért szerepel a felsorolásban. A NOT művelet monadikus. *(A lektor)*

adott bájtra, de a szó többi bitje ne változzon. Először maszkoljuk ki a nem kívánatos biteket, majd OR művelettel vegyük be a beépítendő bájtot.

```
10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (maszk)
10110111 10111100 11011011 00000000 A AND B
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A AND B) OR C
```

Az AND művelet az 1-esek eltüntetését végzi, mert az eredmény akkor és csak akkor 1-es, ha mindkét operandus megfelelő bitje 1-es. Az OR művelet 1-es bitet épít be, mert az eredmény akkor és csak akkor 1-es, ha legalább az egyik operandus 1-es. A XOR kizáró vagy művelet szimmetrikus, azaz az eredmény akkor és csak akkor 1-es, ha pontosan az egyik operandus 1-es. Ez a szimmetria sokszor hasznos, például véletlen számok generálásakor.

A legtöbb számítógépnek vannak lebegőpontos utasításai, nagyjából olyanok, mint az egész számok aritmetikájának utasításai. A legtöbb gép legalább kétféle hosszúságú lebegőpontos számokat támogat, a rövidebbet a gyorsaság, a hosszabbat pedig a nagyobb pontosság érdekében. Sokféle lebegőpontos számábrázolási forma létezik, de manapság az IEEE 754 szabvány az egyetlen széles körben elfogadott. A lebegőpontos számokat és az IEEE 754 szabványt a B) függelék tárgyalja.

5.5.3. Monadikus műveletek

A monadikus műveletnek egy operandusa van, és egy eredményt ad. Mivel eggyel kevesebb címet kell megadni a monadikus utasításokban a diadikusokhoz képest, ezért ezek az utasítások általában rövidebbek, bár néha más egyéb információt is meg kell adni.

Szó vagy bájt léptetése, illetve forgatása sokszor hasznos művelet, és gyakran több különböző változatuk is van. A léptető művelet hatására minden bit azonos mértékben balra vagy jobbra léptetődik, a szélén kilépő bitek elvesznek. Forgatás esetén a kilépő bitek a másik oldalon bejönnek. A két művelet közötti különbséget illusztrálja az alábbi példa.

```
00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 A 2 bittel jobbra léptetve
11000000 00000000 00000000 00011100 A 2 bittel jobbra forgatva
```

Mind a balra és jobbra léptetés, mind a forgatás hasznos művelet. Egy n bites szót k bittel balra forgatva vagy $n - k$ bittel jobbra forgatva az eredmény megegyezik.

A jobbra léptetést gyakran előjel-kiegészítő (röviden előjeles) módon is végzik. Ekkor a bal oldalon az eredeti előjelbit jön be az üres helyekre. Többek között ez azt jelenti, hogy negatív szám mindig negatív marad. A kétféle léptetést illusztrálja az alábbi példa.


```
11111111 11111111 11111111 11110000 A
00111111 11111111 11111111 11111100 A előjel nélküli jobbra léptetése 2 bittel
11111111 11111111 11111111 11111100 A előjeles jobbra léptetése 2 bittel
```

A léptetés művelet fontos alkalmazása a 2 hatvánnyal való szorzás megvalósítása. Ha egy pozitív egész számot k bittel balra léptetünk, akkor az eredmény az eredeti szám 2^k-val való szorzásának felel meg, kivéve, ha túlcsordul. Ha egy pozitív egész számot k bittel jobbra léptetünk, akkor a kapott szám az eredeti szám 2^k-val való osztásának az eredménye lesz.

A léptetések felhasználhatók bizonyos aritmetikai műveletek felgyorsítására. Tekintsük például a $18 \times n$ kifejezés kiszámítását. Mivel $18 \times n = 16 \times n + 2 \times n$, így a $16 \times n$ kiszámítható n-nek 4 bittel való balra léptetésével, $2 \times n$ pedig 1 bittel való balra léptetéssel. A két kifejezés összege $18 \times n$. Tehát a szorzást egy másolással, két léptetéssel és egy összeadással oldottuk meg, ami gyakran gyorsabb, mint a szorzás. Természetesen a fordítóprogram csak akkor tudja alkalmazni ezt a trükköt, ha az egyik operandus konstans.

Negatív számok – akár előjeles – léptetése azonban egészen más eredményt ad. Tekintsük például a –1 szám 1-es komplemens ábrázolását. Balra léptetve 1 bittel az eredmény –3 lesz. Még egyszer balra léptetve 1 bittel –7-et eredményez:

```
11111111 11111111 11111111 11111110 – 1 1-es komplemense
11111111 11111111 11111111 11111100 – 1 balra léptetve 1 bittel = –3
11111111 11111111 11111111 11111000 – 1 balra léptetve 2 bittel = –7
```

1-es komplemensben adott negatív szám balra léptetése 1 bittel nem 2-vel való szorzást valósít meg. A jobbra léptetés sem szimulálja helyesen az osztást.

Tekintsük a –1 szám 2-es komplemens ábrázolását. Ha 6 bittel előjeles jobbra léptetést végzünk, az eredmény –1 lesz, ami nem helyes, mert –1/64 egész része 0.

```
11111111 11111111 11111111 11111111 – 1 2-cs komplemense
11111111 11111111 11111111 11111111 – 1 jobbra léptetve 6 bittel = –1
```

Általában a jobbra léptetés hibát eredményez, mert lefelé kerekít (a még negatívabb számok felé), ami hibás a negatív egészek aritmetikájában. A balra léptetés azonban helyesen adja a 2-vel való szorzást.

A forgatás műveletek hasznosan alkalmazhatók bitsorozatok szóba való be- és kipakolásakor. Ha egy szó minden bitjét ellenőrizni kell, akkor egyesével balra forgatva minden bit egyszer az előjelbit helyére kerül, amikor is tesztelhető, és végül visszakapjuk az eredeti szót. A forgatás műveletek tisztábbak, mint a léptetések, mert nem eredményeznek információvesztést: minden forgatáshoz van olyan másik forgatás, amely helyreállítja az eredeti operandust.

Bizonyos diadikus műveletek olyan gyakran fordulnak elő egy adott operandusszal, hogy az ISA külön monadikus utasítást vezet be ezek számára a gyorsabb végrehajtás céljából. Számítások inicializálásakor rendkívül gyakori, hogy egy regiszterbe vagy memóriaszóba 0-t kell tölteni. Ez természetesen az általános moz-

gató utasítás speciális esete. Hatékonyság miatt gyakran bevezetik a CLR utasítást, amely törli (0-ra állítja) egy cím tartalmát.

Egy szóhoz 1 hozzáadása szintén gyakori számláláskor. Az ADD utasítás monadikus megfelelője az INC utasítás, amely 1-et ad az operandushoz. A NEG (negáció) utasítás szintén jó példa. X negálása a $0 - X$ kiszámítását jelenti, ami diadikus kivonás lenne, de szintén gyakori, ezért külön monadikus utasítást alkalmaznak, ez a NEG. Fontos megjegyezni, hogy a NEG utasítás különbözik a logikai NOT utasítástól. A NEG művelet a szám **additív inverzét** adja (azt a számot, amelyet az eredetihez hozzáadva 0-t kapunk). A NOT művelet pedig a szó minden bitjét ellenkezőjére változtatja. A két művelet nagyon hasonló, sőt az olyan gépeken, amelyek 1-es komplemenst használnak, az eredmény megegyezik. (A 2-es komplemens aritmetikában a NEG műveletet úgy végzik, hogy először minden bitet negálnak, majd 1-et hozzáadnak.)

A diadikus és monadikus utasításokat gyakran együtt csoportosítják a használatuk szerint, és nem az operandusuk száma szerint. Az aritmetikai utasítások csoportja tartalmazza a negációt is. Egy másik csoportba sorolják a logikai utasításokat, idetartoznak a léptetések és forgatások, mert ezeket gyakran együtt használják.

5.5.4. Összehasonlító és feltételes elágazó utasítások

Majdnem minden program használja azt a lehetőséget, hogy adatokon összehasonlítást végezzen, és az eredmény függvényében az utasítások végrehajtásának sorrendjét megváltoztassa. Egyszerű példa erre a $\sqrt{x}$ függvény értékének kiszámítása. Ha x negatív, az eljárás hibaüzenetet küld, egyébként kiszámítja a négyzetgyököt. Az *sqrt* függvénynek először tesztelnie kell x-et, és elágazni attól függően, hogy x negatív-e.

Az elágazás megvalósításának általános módszere az, hogy feltételes elágazó utasítást vezetnek be, amely megadott feltétel teljesülését ellenőrzi, és ha a feltétel teljesül, akkor elágazás (ugrás) következik egy megadott memóriacímre. Néha az utasítás egy bitje jelzi, hogy a feltétel teljesülése avagy nemteljesülése esetén történjék elágazás. Gyakran az elágazás célcíme nem abszolút, hanem az aktuális utasítás helyéhez relatív.

A leggyakrabban a gép egy bizonyos bitjének 0 vagy nem 0 értéke jelenti a feltételt. Ha egy utasítás azt írja elő, hogy teszteljük egy szám előjelbitjét és ugorjunk a *CÍMKE* címre, ha a bit 1-es, akkor a *CÍMKE* című utasítás hajtódik végre, ha a szám negatív, és a feltételes elágazást követő utasítás, ha a szám 0 vagy pozitív.

Sok gép feltételeket kódoló biteket tartalmaz, ezek a bitek megadott feltételeket azonosítanak. Például, lehet túlcsordulásbit, amelyet minden aritmetikai utasítás 1-re állít be, ha a keletkezett eredmény nem helyes. Ezt a bitet tesztelve megtudhatjuk, hogy az előző aritmetikai művelet során túlcsordulás keletkezett-e, és ha igen, akkor hibakezelő eljárásra ugorhatunk.

Hasonlóan, sok gép tartalmaz átviteli feltételbitet, amelyet akkor állítanak be, ha a művelet során átvitel keletkezett a bal szélső bitről, például két negatív szám összeadásakor. Átvitel a bal szélső bitről éléggé normális, és nem tévesztendő ösz-

sze a túlcsordulással. Az átviteli bit tesztelése szükséges például a nagyobb pontosságú aritmetika megvalósításához (tehát amikor két vagy több szó ábrázol egy egész számot).

A 0 tesztelése ismétlések megvalósításánál és sok más esetben is fontos. Ha minden feltételes elágazó utasítás csak egy bitet vizsgálhatna, akkor annak tesztelését, hogy egy szó 0-e, csak úgy lehetne elvégezni, hogy a szó minden bitjét külön tesztelnénk, hogy egyik sem 1-es. Ennek elkerülése érdekében sok gép tartalmaz olyan feltételes elágazó utasítást, amely egy szót tesztel és elágazik, ha az 0. Természetesen ez a megoldás csak áthárítja a munkát a mikroarchitektúrára. A gyakorlatban a hardver általában tartalmaz egy olyan regisztert, amelynek minden bitjét OR kapcsolatba állítja (összvagyolja), és így jelzi egyetlen bittel, hogy valamelyik bit 1-es-e. A 4.1. ábra Z bitje normálisan úgy számítódna, hogy az ALU minden kimeneti bitjét összevagyolnánk, és aztán invertálnánk.

Rendezéseknél nagyon fontos két szó vagy karakter összehasonlítása abból a célból, hogy megtudjuk, egyenlők-e, és ha nem, melyik a kisebb. Ennek megvalósításához három cím kell: kettő az adatokra és egy az ugrási cím megadására. Azoknál a gépeknél, amelyeknek van háromcímes utasítása, nem jelent gondot a megoldás, de amelyeknek nincs, más módon kell eljárni.

Általánosan úgy oldják meg a problémát, hogy külön tesztelő utasítás van, ez tárolja a teszt eredményét, amelyet aztán a feltételes elágazó utasítás lekérdezhet. Mind a Pentium 4, mind az UltraSPARC III alkalmazza ezt a módszert.

Két szám összehasonlítása tartalmaz néhány különlegességet. Az összehasonlítás nem annyira egyszerű, mint például a kivonás. Ha egy nagyon nagy egész számot hasonlítunk egy nagyon nagy negatív számhoz, akkor a kivonás túlcsordulást eredményezne, mert a művelet eredménye nem reprezentálható. Mindazonáltal, az összehasonlítás utasításnak meg kell adnia az összehasonlítás helyes eredményét, túlcsordulás nem keletkezhet.

A számok összehasonlításával kapcsolatos másik különlegesség, hogy a szám vajon előjelesnek vagy előjel nélkülinek tekintődjék. Például a hárombites egész számok kétféleképpen is rendezhetők ennek megfelelően:

Előjel nélküli	Előjeles
000	100 (legkisebb)
001	101
010	110
011	111
100	000
101	001
110	010
111	011 (legnagyobb)

A bal oldali oszlop a pozitív egészeket tartalmazza 0-tól 7-ig. A jobb oldali pedig a 2-es komplemens kódokat –4-től +3-ig. A válasz arra a kérdésre, hogy „a 011 nagyobb-e, mint 100", attól függ, hogy a számokat előjelesnek vagy előjel nélkülinek tekintjük. A legtöbb ISA mindkét típust tudja kezelni.

5.5.5. Eljáráshívó utasítások

Utasítások olyan sorozatát nevezzük eljárásnak, amely meghatározott feladatot old meg, és a program különböző helyeiről aktivizálható (hívható). A **szubrutin** elnevezés is használatos, különösen, ha assembly nyelvről van szó. A Java nyelvben a **metódus** elnevezést használják. Ha az eljárás elvégezte munkáját, vissza kell térnie a hívását követő utasításra. Ezért a visszatérési címet valahogy át kell adni az eljárásnak, vagy valahol tárolni kell, hogy elő tudja venni, amikor eljön a visszatérés ideje.

A visszatérési címet elhelyezhetjük memóriában, regiszterben vagy veremben. A legrosszabb megoldás, ha rögzített memóriahelyre rakjuk. Ugyanis, ha a hívott eljárás egy másikat hív, az előző visszatérési cím elveszik.

Némileg javított megoldás az, amikor a visszatérési címet a hívó utasítás a hívott eljárás első szavába tölti. Ilyenkor az eljárás első végrehajtandó utasítása az ezt követő szó. Ekkor az eljárás az első szót használva indirekt ugrással vissza tud térni, vagy direkt módon, ha a hardver az első szóba az ugró utasítás műveleti kódját is beteszi. Így az eljárás másik eljárást is hívhat, mivel minden eljárásban van hely a visszatérési cím számára. Ez a séma is megbukik, ha az eljárás önmagát hívja, mivel az első visszatérési címet felülírja a másodikkal. Azt a képességet, amikor egy eljárás önmagát hívhatja, **rekurzió**nak nevezzük. A rekurzió rendkívül fontos mind elméletileg, mind a gyakorló programozók számára. A „kicsit javított" séma akkor is megbukik, ha az *A* eljárás hívja a *B* eljárást, az pedig a *C* eljárást, amely aztán ismét hívja az *A* eljárást (indirekt vagy margarétalánc rekurzió).

Nagyobb javítást érünk el, ha a visszatérési címet regiszterben tároljuk, és a hívott eljárás felelősségére bízzuk, hogy azt biztonságos helyen tárolja. Ha az eljárás rekurzív, akkor minden hívott példánynak más helyre kell tennie a visszatérési címet.

A visszatérési cím kezelésére a legjobb megoldás, ha verembe tesszük. Amikor az eljárás befejezte munkáját, kiveszi a veremből a visszatérési címet, és betölti az utasításszámlálóba. Ha ez az eljáráshívási séma alkalmazható, nincs gond a rekurzív eljárásokkal, a visszatérési cím automatikusan elmentődik, megakadályozva az előző visszatérési cím elvesztését. A 4.12. ábrán láttuk az IJVM visszatérési cím tárolási technikáját.

5.5.6. Ismétléses vezérlés

Gyakran előfordul, hogy utasítások egy sorozatát adott számszor meg kell ismételni, ezért néhány gép erre külön utasítást biztosít. Minden ilyen séma tartalmaz egy számlálót, amelynek tartalmát a ciklusmag minden egyes lefutásakor egy konstans értékkel növelik vagy csökkentik. A számláló értékét tesztelik is a ciklusmag minden egyes lefutásakor. Ha a megadott feltétel teljesül, az ismétlés véget ér.

Az egyik módszer szerint a számlálót a ciklusmagon kívül inicializálják, és azonnal elkezdődik a ciklusmag végrehajtása. A ciklusmag utolsó utasítása aktualizálja (növeli vagy csökkenti) a számláló értékét, és ha a befejezési feltétel nem teljesül, akkor ugrás következik a ciklusmag első utasítására. Egyébként az ismétlés véget

ér, és a ciklusmagot követő utasításra kerül a vezérlés. Ezt az ismétlést végfeltételes ismétlésnek nevezzük, és az 5.30. (a) ábrán szemléltetjük C nyelven. (Azért nem a Java nyelvet választottuk, mert annak nincs goto utasítása.)

```
        i = 1;
L1:     első utasítás
        .
        .
        .
        utolsó utasítás
        i = i + 1;
        if (i < n) goto L1;
```

(a)

```
        i = 1;
L1:     if (i > n) goto L2
        első utasítás
        .
        .
        .
        utolsó utasítás
        i = i + 1;
        goto L1;
L2:
```

(b)

5.30. ábra. *(a) Végfeltételes ismétlés. (b) Kezdőfeltételes ismétlés*

A végfeltételes ismétlés jellemzője, hogy a ciklusmagot legalább egyszer mindig végrehajtja, akkor is, ha *n* értéke negatív vagy 0. Példaként tekintsük azt a programot, amely egy vállalat dolgozóinak adatait kezeli. A program egy bizonyos helyén beolvassa egy adott dolgozó adatait. Beolvassa a dolgozó gyerekeinek *n* számát, és végrehajt egy ismétlést *n*-szer, gyermekenként egyszer beolvasva a gyermek nevét, születési dátumát és nemét azzal a céllal, hogy a gyermek születésnapjára ajándékot küldjön a vállalat. Ha a dolgozónak nincs gyereke, az *n* értéke 0, de a ciklus ekkor is végrehajtódik egyszer, és ajándékot küld, hibásan.

Az 5.30. (b) ábrán látható program helyesen működik akkor is, ha az *n* értéke kisebb vagy egyenlő 0. Vegyük észre, hogy a tesztelés különbözik a két esetben, tehát ha az ISA egyetlen utasítása végzi mind a növelést, mind a tesztet, akkor a tervezők választásra kényszerülnek.

Vizsgáljuk azt a kódot, amely az alábbi utasítás fordításaként keletkezhet.

```
for (i = 0; i < n; i++) { utasítások }
```

Ha a fordítónak nincs semmi információja az *n* értékéről, akkor az 5.30. (b) módszert kell választania, hogy helyesen kezelje az $n \leq 0$ esetet. Azonban, ha korábbi értékadás miatt tudja, hogy $n > 0$, akkor a jobb 5.30. (a) kódot generálhatja. A FORTRAN szabvány korábban megkövetelte, hogy a ciklusmag legalább egyszer végrehajtódik azért, hogy mindig az 5.30. (a) hatékonyabb kódot generálhassa a fordító. Ezt a hiányosságot 1977-ben kijavították, amikorra már a FORTRAN közösség is rájött, hogy nem szerencsés az a furcsa szemantika, amely időnként helytelen eredményt ad, még ha így ciklusonként egy elágazó utasítást meg is lehet takarítani. A C és a Java mindig helyesen működik.

5.5.7. Bemenet/kimenet

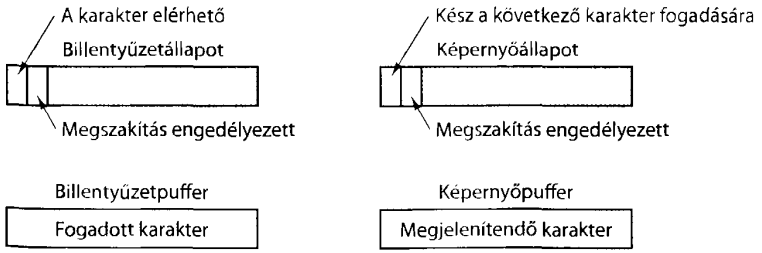
Nincs más olyan utasításcsoport, amely gépenként annyira különbözne, mint a B/K utasítások. A mai személyi számítógépek három különböző sémát használnak. Ezek a következők.

1. Programozott B/K tevékeny várakozással.
2. Megszakításvezérelt B/K.
3. DMA B/K.

A továbbiakban egyenként vizsgáljuk ezeket.

A legegyszerűbb B/K módszer a **programozott B/K**, amelyet általánosan használnak egyszerű mikropocesszorokban, például beágyazott rendszerekben, ahol gyorsan kell válaszolni a külső változásokra (valós idejű rendszerek). A CPU-nak általában egyetlen bemeneti és egyetlen kimeneti utasítása van. Mindegyik utasítás egy B/K eszközt választ ki. Egyetlen karakter kerül átvitelre a processzor rögzített regisztere és a kiválasztott B/K eszköz között. A processzornak végre kell hajtania egy utasítássort minden egyes beolvasandó és kiírandó karakterre.

Tekintsük a következő egyszerű példát erre a módszerre, amikor egy terminál négy 1 bájtos regiszterrel rendelkezik, amint azt az 5.31. ábra mutatja. Két regiszter használatos a bemenet (állapot és adat) és ugyancsak kettő a kimenet (állapot és adat) számára. Mindegyik regiszternek saját címe van. A memóriába leképezett B/K esetén mind a négy regiszter a memória címtartományában van, és közönséges utasításokkal olvashatók és írhatók. Különben speciális B/K utasítások, mondjuk, IN és OUT, vannak olvasásukra és írásukra. Mindkét esetben az adat- és állapotinformációnak a CPU és a regiszterek közötti átvitele valósítja meg a B/K műveletet.



5.31. ábra. *Egyszerű terminál eszközregiszterei*

A billentyűzetállapot-regiszter 2 használt és 6 nem használt bitet tartalmaz. A bal szélső (7.) bitet a hardver mindig 1-esre állítja, ha karakter érkezik. Ha a szoftver előzőleg a 6. bitet 1-re állította, akkor megszakítás generálódik, egyébként nem. (A megszakításokat hamarosan tanulmányozzuk.) Programozott B/K esetén a CPU normálisan végtelenített ciklusban ismételten olvassa a billentyűzetállapot regiszterét arra várva, hogy a 7. bit 1-es legyen. Amikor ez bekövetkezik, a szoftver

beolvassa a karaktert a billentyűzetpufferből. A beolvasás hatására az állapotregiszter „Karakter elérhető" (vagyis 7.) bitje 0-ra állítódik.

A kimenet hasonlóan működik. Egy karakternek a képernyőre írása végett a szoftver először kiolvassa a képernyőállapot regiszterét, hogy megtudja, a „Kész a következő karakter fogadására" bit 1-es-e. Ha nem, akkor ciklusban ismétli ezt mindaddig, amíg 1-es nem lesz. Ez jelzi, hogy az eszköz készen áll karakter fogadására. A szoftver azonnal kiírja a karaktert a képernyőpuffer regiszterébe, ahogy a terminál kész, és 0-ra állítja az állapotregiszter kész bitjét. Ha a karakter megjelenítése megtörtént, a vezérlő újból 1-re állítja a kész bitet.

Tekintsük az 5.32. ábrán látható Java-programot, amely a programozott B/K-re példa. Az eljárásnak két bemenő paramétere van: az első a kiírandó karaktereket tartalmazó tömb, a második pedig a karakterek száma, legfeljebb 1 KB. Az eljárás törzse egy ciklus, amely egyenként kiírja a karaktereket. Minden egyes karakterre először a CPU mindaddig várakozik, amíg az eszköz kész állapotba nem kerül, aztán kiírja a karaktert. Az *in* és *out* eljárások tipikusan assembly eljárások lehetnek. Az *in* eljárás az eszközállapot regiszterét olvassa, az *out* eljárás első paramétere az eszközpuffer regisztere, a második pedig a kiírandó karakter. A 128-cal való osztás (7 bittel jobbra léptetés) célja, hogy eldobja az alsó 7 bitet, ezzel a 0. pozícióra állítja a READY bitet.

```
public static void output_buffer(int buf[ ], int count) {

 // Adatblokk kiírása az eszközre
 int status, i, ready;

  for (i = 0; i < count; i++) {
   do {
     status = in(display_status_reg);      // az állapot lekérdezése
     ready = (status >> 7 ) & 0x01;        // a kész bit elkülönítése
   } while (ready != 1);
   out(display_buffer_reg, buf[i]);
  }
}
```

5.32. ábra. *Példa programozott B/K-re*

A programozott B/K elsődleges hátránya, hogy a CPU azzal van elfoglalva, hogy egy ciklusban arra várjon, mikor válik az eszköz átviteli állapotra késszé. Ezt **tevékeny várakozás**nak nevezzük. Ha a CPU-nak nincs más dolga (például egy mosógépben), akkor a tevékeny várakozás módszere elfogadható (habár még az egyszerű vezérlőknek is gyakran több eseményt kell párhuzamosan figyelniük). Azonban, ha más dolgot is kell végeznie, például másik programot futtatni, akkor a tevékeny várakozás módszere pazarló, és más B/K módszert kell keresni.

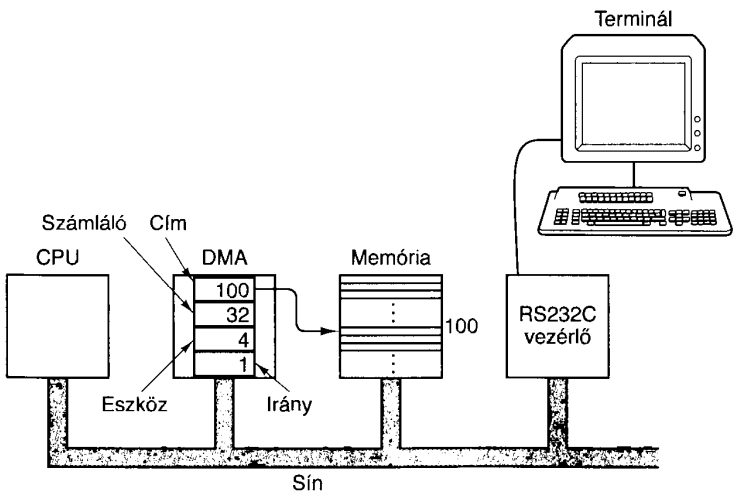
Úgy szabadulhatunk meg a tevékeny várakozástól, hogy a CPU-t megkérjük, indítsa el a B/K eszközt, és az eszközmegszakítással jelezzen, ha elvégezte az átvitelt. Az 5.31. ábrára tekintve láthatjuk, mit kell tenni. Beállítjuk az eszköz állapot-

regiszterének a „Megszakítás engedélyezett" bitjét 1-re, ezzel a szoftver megszakítási jelet igényel a hardvertől a B/K befejeződésekor. Ebben a fejezetben később, a vezérlési folyamatokkal foglalkozó részben részletesen tanulmányozzuk a megszakításokat.

Érdemes megjegyezni, hogy sok számítógép esetén a megszakítási jelet úgy generálják, hogy veszik a „Megszakítás engedélyezett" és a „Karakter elérhető" bitek logikai AND szorzatát. Ha a szoftver előbb állítja be a megszakítás engedélyezését (mielőtt a B/K elkezdődne), akkor azonnal jelentkezik a megszakítás, mivel a „Karakter elérhető" bit 1-es lesz. Tehát előbb el kell indítani az eszközt, és utána azonnal engedélyezni a megszakítást. Az állapotregiszterbe írva a készenlét bit nem módosul, mert csak olvasható.

Habár a megszakításvezérelt B/K hatalmas lépés előre a programozott B/K-hez képest, távol áll a tökéletestől. Az a gond, hogy minden egyes átviendő karakter megszakítást igényel, és a megszakítás feldolgozása költséges. Kell valamilyen módszer, amely megszabadít a megszakítások többségétől.

A megoldást a programozott B/K-hez való visszatérés jelenti, de rá kell venni valaki mást, hogy ő végezze el a munkát. (Sok probléma megoldása azon alapszik, hogy más végzi el a munkát.) Az 5.33. ábra mutatja a megoldás elrendezését. Itt egy új lapkát adtunk a rendszerhez, a **DMA- (Direct Memory Access, direkt memóriaelérés)** vezérlőt, amely közvetlenül eléri a sínt.



5.33. ábra. *DMA-vezérlőt tartalmazó rendszer*

A DMA lapka legalább négy olyan regisztert tartalmaz, amelyek a CPU-n futó szoftverrel betölthetők. Az első azt a kezdő memóriacímet tartalmazza, amelyiket írni vagy olvasni kell. A második az átviendő bájtok számát tartalmazza. A harmadik az átvitelre kiválasztott eszköz sorszámát vagy a B/K címet tartalmazza.

A negyedik regiszter az átvitel irányát tartalmazza, tehát hogy olvasást vagy írást kívánunk.

A memória 100-as címéről 32 bájtos blokk kiírása a terminálra (legyen ez a 4. eszköz) a következőképpen zajlik. A CPU először a 32-es, 100-as és 4-es számokat írja a DMA első három regiszterébe, az írás kódját (mondjuk 1) pedig a negyedik regiszterbe, amint az az 5.33. ábrán látható. Ezután az inicializálás után a DMA-vezérlő kérést küld a sínnek a 100-as memóriacím kiolvasására, ugyanúgy, mint amikor a CPU akar a memóriából olvasni. Miután megkapta az igényelt bájtot, a DMA-vezérlő B/K kérést kezdeményez a 4. eszköz felé a bájt kiírása végett. Miután mindkét művelet sikeresen befejeződött, a DMA-vezérlő növeli 1-gyel a memóriacímet és csökkenti 1-gyel a számláló értékét. Ha a számláló új értéke még mindig nagyobb, mint 0, akkor a memóriából a következő bájtot olvassa és írja ki.

Amikor a számláló értéke 0-vá válik, a DMA-vezérlő leállítja az átvitelt, és megszakítást kezdeményez a CPU felé. DMA-val a CPU-nak csak néhány regisztert kell inicializálnia. Ezután felszabadul más munkára mindaddig, amíg a teljes átvitel befejeződik, és akkor megszakítást kap a DMA-tól. Néhány DMA két, három vagy akár több regiszterkészlettel rendelkezik, így párhuzamosan több átvitelt is képes vezérelni.

Bár a DMA megszabadítja a CPU-t a B/K munka dandárjától, a CPU nem teljesen szabad. Ha a DMA nagy sebességű eszközt, például mágneslemezt vezérel, akkor sok sínciklusra van szüksége mind a memória, mind az eszköz elérése érdekében. E ciklusok alatt a CPU várakozásra kényszerül (a DMA magasabb prioritással igényel sínciklust, mert a B/K eszközök gyakran nem tolerálják a késleltetést). Amikor a DMA sínciklust vesz el a CPU-tól, azt **ciklusloparás**nak hívjuk. Mindazonáltal az a nyereség, amely abból származik, hogy nem kell bájtonként megszakítást kezelni, felülmúlja a ciklusloparás okozta veszteséget.

5.5.8. A Pentium 4 utasításai

Ebben és a következő két szakaszban a három példaként használt gép, a Pentium 4, az UltraSPARC III és a 8051 utasításrendszerét tekintjük át. Mindegyiknek van egy alapkészlete, amelyet a fordítóprogramok normálisan generálnak, és ezenfelül olyan utasítások, amelyeket ritkán használnak, vagy csak az operációs rendszer használja. Vizsgálatainkban az általános utasításokra koncentrálunk. Kezdjük a Pentium 4-gyel.

A Pentium 4 utasításrendszere 32 bites utasítások és a korábbi 8088-as életéből származó utasítások keveréke. Az 5.34. ábrán a fordítók és a programozók által manapság gyakran használt egész utasításoknak egy válogatását adjuk. Ez a lista távolról sem teljes, nem tartalmaz sem lebegőpontos, sem vezérlő utasításokat, sőt néhány egzotikus egész utasítást sem (mint a táblázatban keresés AL-ben adott bájt alapján). Mindazonáltal érzékelteti, hogy mit csinál a Pentium 4.

Sok Pentium 4 utasításnak egy vagy két operandusa van, ez lehet memória vagy regiszter. Például a két operandusos ADD utasítás, amely a forrást hozzáadja a céloperandushoz, vagy az INC utasítás, amely az operandust növeli eggyel (1 hozzá-

adásával). Néhány utasításnak több hasonló variánsa van. Például a léptető utasításoknak van balra és jobbra léptető változata, sőt, mindegyiknek van előjeles és előjel nélküli változata is. A legtöbb utasításnak különböző kódolása lehet, az operandus természetétől függően.

Az 5.34. ábrán az SRC mező információforrás, amely nem változik. Ezzel ellenben a DST mező céloperandus, amelyet az utasítás módosít. Létezik szabály arra, hogy mit lehet, és mit nem lehet csinálni a forrással, illetve a céloperandussal, amely hektikusan változik utasításról utasításra, de ezt mi most nem részletezzük. Sok utasításnak három változata van, 8, 16 és 32 bites operandussal. Ezeket különböző műveleti kódok és/vagy egy bit különböztetik meg az utasításban. Az 5.34. ábra a 32 bites utasításokra helyezi a hangsúlyt.

Kényelmi okból az utasításokat csoportokba soroltuk. Az első csoport az adatmozgató utasításokat tartalmazza, amelyek a regiszterek, memória és verem között mozgatják az adatokat. A második az előjel nélküli és előjeles aritmetikai utasítások csoportja. Szorzás és osztás esetén a 64 bites szorzatot, illetve az osztandót az EAX (alsó rész) és EDX (felső rész) regiszterpár tartalmazza.

A harmadik csoport a binárisan kódolt decimális (BCD) aritmetika, ahol minden bájt két négybites **falat**ra (**nibble**) van osztva. Minden falat egy decimális számjegyet tartalmaz (0–9). Az 1010–1111 bitkombinációk nem használatosak. Tehát egy 16 bites egész 0-tól 9999-ig terjedő számokat tartalmazhat. Bár ez a módszer tárolási szempontból nem hatékony, de szükségtelenné teszi a decimálisról binárisra való és a visszakódolást. Az ebbe a csoportba tartozó utasítások BCD számokon végeznek műveleteket. Ezeket intenzíven használják a COBOL-programok.

A logikai és léptető/forgató utasítások egy szó vagy bájt bitjeit manipulálják a legkülönbözőbb módon. Több kombinációjuk létezik.

A következő két csoport teszteléssel, összehasonlítással és elágazással foglalkozik. A tesztelés és összehasonlítás eredménye az EFLAGS regiszter bitjeiben tárolódik. A Jxx jelzésű utasítások feltételes elágazást valósítanak meg az előző összehasonlítás (tehát az EFLAGS bitjei) alapján.

A Pentium 4 több utasítást tartalmaz, amelyek karakterláncokon végeznek műveletet; ilyenek a betöltés, a tárolás, a másolás és a keresés. Ezeknek az utasításoknak lehet ún. prefix bájtja, a REP, amely az utasítás ismétlését eredményezi megadott feltétel teljesüléséig, például az ECX minden végrehajtáskor csökken 1-gyel mindaddig, amíg 0 nem lesz. Ezzel a módszerrel akármekkora adatblokk átmozgatható, összehasonlítható és így tovább. A következő csoport a feltételkódok kezelésével foglalkozik.

Az utolsó csoport olyan utasítások összevisszasága, melyek máshova nem fértek be. Idetartozik a konverzió, veremkeret-kezelés, CPU-leállítás és B/K.

A Pentium 4 számos **prefix** lehetőséget tartalmaz, egyet (REP) ezek közül már említettünk. Minden prefix egy speciális bájt, amely a legtöbb utasítás előtt szerepelhet, hasonlóan, mint az IJVM esetén a WIDE. A REP prefix hatása az, hogy az utasítás mindaddig ismétlődik, amíg az ECX regiszter tartalma 0-vá nem válik. A REPZ és a REPNZ addig ismétli az utasítást, amíg a feltételkód *Z* bitje 1-es, illetve 0. A LOCK lefoglalja a sínt az utasítás teljes végrehajtásáig, ami lehetővé teszi multiprocesszor esetén a szinkronizálást. Van olyan prefix, amely az utasítást 16 vagy

Mozgató utasítások

MOV DST, SRC	SRC megy DST-be
PUSH SRC	SRC a verembe
POP DST	Veremből kivesz DST-be
XCHG DS1, DS2	DS1 és DS2 tartalmát kicseréli
LEA DST, SRC	SRC effektív címe DST-be
CMOVcc DST, SRC	Feltételes mozgatás

Aritmetikai utasítások

ADD DST, SRC	DST + SRC
SUB DST, SRC	DST– SRC
MUL SRC	EAX * SRC (előjel nélküli)
IMUL SRC	EAX * SRC (előjeles)
DIV SRC	(EDX:EAX)/SRC (előjel nélküli)
IDIV SRC	(EDX:EAX)/SRC (előjeles)
ADC DST, SRC	DST + SRC + átvitelbit
SBB DST, SRC	DST – SRC – átvitelbit
INC DST	DST + 1
DEC DST	DST – 1
NEG DST	0 – DST

Binárisan kódolt decimális utasítások

DAA	Decimális igazítás összeadás után
DAS	Decimális igazítás kivonás után
AAA	ASCII igazítás összeadás után
AAS	ASCII igazítás kivonás után
AAM	ASCII igazítás szorzás után
AAD	ASCII igazítás osztás előtt

Logikai utasítások

AND DST, SRC	DST logikai és SRC
OR DST, SRC	DST logikai vagy SRC
XOR DST, SRC	DST kizáró vagy SRC
NOT DST	DST 1-es komplemense

Léptető/forgató utasítások

SAL/SAR DST, #	DST balra/jobbra léptetése # bittel
SHL/SHR DST, #	DST logikai balra/jobbra léptetése # bittel
ROL/ROR DST, #	DST balra/jobbra forgatása # bittel
RCL/RCR DST, #	DST és átvitel balra/jobbra forgatása # bittel

Teszt(elő) és összehasonlító utasítások

TST SRC1, SRC2	Logikai és, feltételbit beállítása
CMP SRC1, SRC2	Feltételbit beállítása SRC1–SRC2 alapján

Vezérlésátadó utasítások

JMP ADDR	Ugrás az ADDR címre
Jxx ADDR	Feltételes ugrás EFLAGS alapján
CALL ADDR	Az ADDR című eljárás hívása
RET	Visszatérés eljárásból
IRET	Visszatérés megszakításból
LOOPxx	Ismétlés, amíg a feltétel teljesül
INT ADDR	Szoftvermegszakítás kezdeményezése
INTO	Megszakítás túlcsordulás esetén

Karakterlánc utasítások

LODS	Karakterlánc betöltése
STOS	Karakterlánc tárolása
MOVS	Karakterlánc mozgatása
CMPS	Két karakterlánc összehasonlítása
SCAS	Keresés karakterláncban

Feltételkódok

STC	Átvitel beállítása EFLAGS-ben
CLC	Átvitelbit nullázása EFLAGS-ben
CMC	Átvitelbit komplementálása EFLAGS-ben
STD	Irány beállítása EFLAGS-ben
CLD	Irány nullázása EFLAGS-ben
STI	Megszakításbit beállítása EFLAGS-ben
CLI	Megszakításbit nullázása EFLAGS-ben
PUSHFD	EFLAGS verembe töltése
POPFD	Kiolvasás veremből az EFLAGS regiszterbe
LAHF	EFLAGS regiszter AH-ba töltése
SAHF	AH regiszter EFLAGS-be töltése

Egyéb utasítások

SWAP DST	DST kis/nagy endián váltása
CWQ	EAX kiterjesztése (EDX:EAX)-re
CWDE	16 bites AX kiterjesztése EAX-re
ENTER SIZE, LV	SIZE méretű veremkeret létesítése
LEAVE	ENTER létesítette veremkeret torlése
NOP	Üres utasítás
HLT	Megállás (Halt)
IN AL, PORT	Bájtbevitel PORT ról AL be
OUT PORT, AL	Bájtkivitel AL-ből PORT-ra
WAIT	Várakozás megszakításra

cc feltétel
SRC = forrás
DST = cél
# = léptetés/forgatás száma
LV = # lokális változók száma

5.34. ábra. *Válogatás a Pentium 4 egész számokra értelmezett utasításkészletéből*

32 bites módba kényszeríti, ami nemcsak azt jelenti, hogy meghatározza az operandus hosszát, hanem a címzési módot is teljesen átdefiniálja. Végül, a Pentium 4 bonyolult szegmentálási sémával rendelkezik, van kód, adat, verem és extra szegmens, amely a 8088 architektúrától származik. Prefix megadásával lehet elérni, hogy az utasítás egy meghatározott szegmenst használjon, de ezzel nem foglalkozunk (szerencsére).

5.5.9. Az UltraSPARC III utasításai

Az UltraSPARC III valamennyi olyan felhasználó módú egész aritmetikai utasítását tartalmazza az 5.35. ábra, amelyet fordítóprogram generálhat. A lebegőpontos utasítások hiányoznak, csakúgy, mint a vezérlő (például gyorsítótár-kezelő, rendszer-újraindító) és a nem a felhasználói címtartománnyal kapcsolatos utasítások, az elavult utasítások. A készlet meglepően kicsi, az UltraSPARC III valóban redukált utasításrendszerű gép.

A LOAD és STORE utasítások jelentése nyilvánvaló, 1, 2, 4 és 8 bájtos változatuk létezik. Ha 64 bitnél kisebb méretű a betöltendő, akkor vagy előjelesen, vagy előjel nélkül (0-val) kiegészítődik 64 bitre. Mindkét változatra van utasítás.

A következő csoport az aritmetikai utasításokat tartalmazza. A nevükben CC karaktereket tartalmazó utasítások beállítják az NZVC feltételkód bitjeit, a többi azonban nem. CISC gépeken a legtöbb utasítás beállítja a feltételkódot, a RISC gépeken azonban ez nem kívánatos, mert korlátozná a fordítóprogram abbeli szabadságát, hogy utasításokat lépjen át üres időszelet kitöltése érdekében. Ha az eredeti utasítási sorrend A ... B ... C, és A beállítja a feltételkódot, valamint B teszteli, akkor a fordító nem illesztheti be a C-t A és B közé, ha C is beállítja a feltételkódot. Éppen ezért sok utasításnak két változata van. A fordító általában azokat használja, amelyek nem állítják a feltételkódot, kivéve, ha arra később szükség van. Van szorzás, előjel nélküli és előjeles osztás művelet.

A címkézett aritmetika speciális önazonosító 30 bites számformátumot jelent. Ez olyan nyelvek esetén használható, mint a Smalltalk vagy a Prolog, ahol a változók típusa fordításkor nem adott, illetve futási időben változhat. Az ilyen számok esetén a fordító ADD utasítást generálhat, és a gép futáskor döntheti el, hogy egész vagy lebegőpontos összeadás műveletet kell végeznie.

A léptető csoport egy balra és két jobbra léptető utasítást tartalmaz, mindegyiknek van 32 és 64 bites változata. SLL esetén a teljes 64 bit részt vesz a léptetésben, mert ez még kompatibilis a korábbi szoftverrel. A léptetések nagyrészt bitmanipulációra használatosak. A legtöbb CISC gépnek rengeteg léptető és forgató utasítása van, többségük teljesen haszontalan. Hiányuk kevés fordítóprogram-készítőnek okoz álmatlan éjszakát.

A logikai utasítások csoportja hasonló az aritmetikaihoz. Van AND, OR, XOR, ANDN, ORN és XNORN utasítás. Az utóbbi három értéke megkérdőjelezhető, bár egyetlen ciklus alatt végrehajthatók, és lényegében nem kívánnak plusz hardvert, ezért készültek el. Időnként még a RISC-tervezők sem tudnak ellenállni a kísértésnek.

Betöltő utasítások

LDSB ADDR, DST	Előjeles bájt (8 bit) betöltése
LDUB ADDR, DST	Előjel nélküli bájt (8 bit) betöltése
LDSH ADDR, DST	Előjeles félszó (16 bit) betöltése
LDUH ADDR, DST	Előjel nélküli félszó (16 bit) betöltése
LDSW ADDR, DST	Előjeles szó (32 bit) betöltése
LDUW ADDR, DST	Előjel nélküli szó (32 bit) betöltése
LDX ADDR, DST	Duplaszó (64 bit) betöltése

Tároló utasítások

STB SRC, ADDR	Bájt (8 bit) tárolása
STH SRC, ADDR	Félszó (16 bit) tárolása
STW SRC, ADDR	Szó (32 bit) tárolása
STX SRC, ADDR	Duplaszó (64 bit) tárolása

Aritmetikai utasítások

ADD R1, S2, DST	Összeadás
ADDCC "	Összeadás és cc beállítása
ADDC "	Összeadás átvitellel
ADDCCC "	Összeadás átvitellel és cc beállítása
SUB R1, S2, DST	Kivonás
SUBCC "	Kivonás és cc beállítása
SUBC "	Kivonás átvitellel
SUBCCC "	Kivonás átvitellel és cc beállítása
MULX R1, S2, DST	Szorzás
SDIVX R1, S2, DST	Előjeles osztás
UDIVX R1, S2, DST	Előjel nélküli osztás
TADCC R1, S2, DST	Címkézett összeadás

Léptető/forgató utasítások

SLL R1, S2, DST	Logikai léptetés balra (32 bites)
SLLX R1, S2, DST	Kiterjesztett logikai léptetés balra (64 bites)
SRL R1, S2, DST	Logikai léptetés jobbra (32 bites)
SRLX R1, S2, DST	Kiterjesztett logikai léptetés jobbra (64 bites)
SRA R1, S2, DST	Aritmetikai léptetés jobbra (32 bites)
SRAX R1, S2, DST	Kiterjesztett aritmetikai léptetés jobbra (64 bites)

Logikai utasítások

AND R1, S2, DST	Logikai AND
ANDCC "	Logikai AND cc beállításával
ANDN "	Logikai NAND
ANDNCC "	Logikai NAND cc beállításával
OR R1, S2, DST	Logikai OR
ORCC "	Logikai OR cc beállításával
ORN "	Logikai NOR
ORNCC "	Logikai NOR cc beállításával
XOR R1, S2, DST	Logikai XOR
XORCC "	Logikai XOR cc beállításával
XNOR "	Logikai kizáró NOR
XNORCC "	Logikai kizáró NOR cc beállításával

Vezérlő utasítások

BPcc ADDR	Elágazás jövendöléssel
BPr SRC, ADDR	Elágazás regisztertartalom alapján
CALL ADDR	Eljáráshívás
RETURN ADDR	Visszatérés eljárásból
JMPL ADR, DST	Ugrás és kapcsolás
SAVE R1, S2, DST	A regiszterablak tárolása
RESTORE "	A regiszterablak visszaállítása
Tcc CC, TRAP #	Feltételes csapda
PREFETCH FCN	Előre betöltés memóriából
LDSTUB ADDR, R	Atomi betöltés/tárolás
MEMBAR MASK	Memóriasorompó

Egyéb utasítások

SETHI CON, DST	A 10–31. bitek beállítása
MOVcc CC, S2, DST	Feltételes mozgatás
MOVr R1, S2, DST	Mozgatás regisztertartalom alapján
NOP	Üres utasítás
POPC S1, DST	Populációszámláló
RDCCR V, DST	A feltételregiszter olvasása
WRCCR R1, S2, V	A feltételregiszter írása
RDPC V, DST	Az utasításszámláló olvasása

SRC = forrásregiszter
DST = célregiszter
R1 = forrásregiszter
S2 = forrás: regiszter vagy közvetlen
ADDR = memóriacím

TRAP# = csapdaszám
FCN = függvénykód
MASK = műveleti típus
CON = konstans
V = regiszterjelölő

CC = feltételkód-beállítás
R = célregiszter
cc = feltétel
r = LZ, LEZ, Z, NZ, GZ, GEZ

5.35. ábra. *Az UltraSPARC III alapvető egész utasításai*

A következő csoport tartalmazza a vezérlésátadó utasításokat. A BPcc utasítások olyan csoportját reprezentálja, amelyek különböző feltételek alapján történő ugrást végeznek, és az utasításban jelzik, hogy a fordító szerint létrejön-e az ugrás, vagy sem. A BPr regisztert tesztel, és a feltétel teljesülésekor ugrik.

Eljáráshívásokra két módszer létezik. A CALL utasítás az 5.15. ábrán látható 4. formát alkalmazza 30 bites PC-relatív eltolási értékkel. Ez az érték elegendő 2 GB távolság áthidalására mindkét irányban. A CALL utasítás az R15 regiszterbe rakja a visszatérési címet, ebből a regiszterből a hívás után R31 lesz.

Az eljáráshívás másik módját az 1a vagy 1b formájú JMPL utasítás szolgáltatja, amely lehetővé teszi, hogy a visszatérési címet bármelyik regiszterben elhelyezzük. Ez a forma hasznos, ha a célcímet futás közben számítjuk ki.

A SAVE és a RESTORE a regiszterablakon és a veremmutatón manipulál. Mindkettő csapdát eredményez, ha a következő (megelőző) ablak nem érhető el.

Az utasítások utolsó csoportja segédutasításokat tartalmaz. A SETHI utasításra azért van szükség, mert csak így tudunk 32 bites közvetlen adatot regiszterbe tölteni. Ezt úgy tehetjük, hogy a SETHI utasítással a 10–31. bitet állítjuk be, a maradékot pedig a következő utasításban adjuk meg közvetlen operandusként.

A populációszámláló utasítás rejtély. Ez az utasítás megszámlálja az egy szóban található 1-es biteket. Az a szóbeszéd járja, hogy nagyon jó robbantások szimulálására, és a Los Alamos National Laboratory (egy nagy felhasználó) is jobban szereti azokat a számítógépeket, amelyeknek van ilyen utasításuk.

Számos olyan szokásos CISC-utasítás, amely hiányzik a felsorolásból, könnyen szimulálható vagy G0 regiszterrel, vagy konstans operandussal (1b forma). Ezek közül mutat néhányat az 5.36. ábra. Ezeket tartalmazza az UltraSPARC III assembly, és a fordítók gyakran alkalmazzák. Közülük sok kihasználja, hogy a G0 hardveres 0, és a G0-ba töltés hatástalan.

Utasítás	Megvalósítás
MOV SRC,DST	OR SRC, G0, DST
CMP SRC1,SRC2	SUBCC SRC2, SRC1, G0
TST SRC	ORCC SRC, G0, G0
NOT DST	XNOR DST, G0, G0
NEG DST	SUB DST, G0, DST
INC DST	ADD 1, DST, DST (1 közvetlen operandus)
DEC DST	SUB 1, DST, DST (1 közvetlen operandus)
CLR DST	OR G0, G0, DST
NOP	SETHI 0, G0
RET	JMPL %I7+8,%G0

5.36. ábra. *Néhány szimulált UltraSPARC III utasítás*

5.5.10. A 8051 utasításai

A 8051-esnek egyszerű utasításrendszere van, amelynek első része az 5.37. ábrán látható. Minden sor tartalmazza a műveleti kód jelét, az utasítás tömör leírását és a forrás- vagy a céloperandus számára alkalmazható címzési módot, attól függően, hogy SRC avagy DST szerepel a műveletben. A forrásoperandus jele SRC, a céloperandusé pedig DST. Ahogy várható, sokféle MOV utasítás létezik az ACC (akkumulátor) és regiszter vagy memória közötti adatmozgatásra. Vannak verembe töltő és veremből kivevő utasítások. A verem mindig a 256 cím feletti memóriában van. A verem teteje dedikált regiszter által mutatott címen van. A verem mindig a külső memóriában van, mert a 8051-esnek csak 128, a 8052-esnek pedig csak 256 bájt belső memóriája van, ezért a címzés a 16 bites DPTR regiszteren keresztül történik. Az adatmozgató utasítások csoportját néhány olyan segédutasítás teszi teljessé, amelyek regiszterek részeinek felcserélését végzik.

A 8051-es egyszerű aritmetikai utasításokat tartalmaz összeadás, kivonás, szorzás és osztás elvégzésére. Az utóbbi utasításokban a regiszter rögzített. Növelés és csökkentés szintén lehetséges, és gyakran használatos is. Logikai és léptető utasítások szintén vannak.

A 8051-es további utasításait az 5.38. ábrán láthatjuk. Itt vannak a bitenkénti utasítások, például a

```
SETB 43
```

utasítás, amely 1-re állítja a 43. bitet, de nem változtatja a bájt többi bitjét. Ezt követően látjuk a vezérlésátadó utasításokat, többek között az ugró és szubrutinhívó, valamint két feltételes ugró utasítást, amelyek a forrást hasonlítják valamihez, majd a DJNZ ciklusszervező utasítás látható.

5.5.11. Az utasításrendszerek összehasonlítása

A három példának tekintett utasításrendszer nagyon különböző. A Pentium 4 egy hagyományos kétcímes, 32 bites CISC gép hosszú előtörténettel, különleges és szabálytalan címzési rendszerrel, valamint sok memóriára hivatkozó utasítással. Az UltraSPARC III egy modern, háromcímes, 64 bites RISC, töltő/tároló architektúra kevés címzési móddal, kompakt és hatékony utasításkészlettel. A 8051 architektúra egy kicsi beágyazott processzor, amelyet úgy terveztek, hogy egyetlen lapkán megvalósítható legyen.

Mindegyik gépnek megvan a maga célja. A Pentium 4 három tervezési elven alapszik:

1. Visszafelé kompatibilitás.
2. Visszafelé kompatibilitás.
3. Visszafelé kompatibilitás.

Utasítás	Leírás	ACC	Reg	Dir	@R	#	C	Bit
MOV	SRC megy ACC-be		X	X	X	X		
MOV	SRC regiszterbe töltése	X		X		X		
MOV	SRC memóriába töltése	X	X	X	X	X		
MOV	SRC indirekt töltése RAM-ba	X		X		X		
MOV	16 bites konstans DPTR-be töltése							
MOVC	Kód megy DPTR eltolással ACC-be							
MOVC	Kód megy PC eltolással ACC-be							
MOVX	Külső RAM bájt töltése ACC-be				X			
MOVX	Külső RAM bájt töltése ACC@DPTR-be							
MOVX	ACC-ből bájt külső RAM-ba töltése				X			
MOVX	ACC@DPTR-ből bájt külső RAM-ba töltése							
PUSH	SRC bájt verembe töltése			X				
POP	Bájt kivétele veremből DST-be			X				
XCH	ACC és DST felcserélése	X		X	X			
XCHD	ACC és DST alsó számjegyének felcserélése			X				
SWAP	DST fél bájtjainak felcserélése	X						
ADD	SRC+ACC=>ACC		X	X	X	X		
ADDC	SRC+ACC+átvitelbit =>ACC		X	X	X	X		
SUBB	ACC-SRC-átvitelbit=>ACC		X	X	X	X		
INC	DST növelése	X	X	X	X			
DEC	DST csökkentése	X	X	X	X			
INC	DPTR növelése							
MUL	Szorzás							
DIV	Osztás							
DA	DST decimális igazítása	X						
ANL	SRC AND ACC=>ACC		X	X	X	X		
ANL	ACC AND DST=>DST			X				
ANL	Közvetlen operandus AND DST=>DST			X				
ORL	SRC OR ACC=>ACC		X	X	X	X		
ORL	DST OR ACC=>DST			X				
ORL	Közvetlen operandus OR DST=>DST			X				
XRL	SRC XOR ACC=>ACC		X	X	X	X		
XRL	DST XOR ACC=>DST			X				
XRL	Közvetlen operandus XOR DST=>DST			X				
CLR	DST nullázása	X						
CPL	DST komplementálása	X						
RL	DST balra forgatása	X						
RLC	DST balra forgatása átvitelbiten keresztül	X						
RR	DST jobbra forgatása	X						
RRC	DST jobbra forgatása átvitelbiten keresztül	X						

5.37. ábra. *A 8051 utasításkészlete, 1. rész*

Utasítás	Leírás	ACC	Reg	Dir	@R	#	C	Bit
CLR	Bit nullázása						X	X
SETB	Bit 1-esre állítása						X	X
CPL	Bit komplementálása						X	X
ANL	SRC AND átvitelbit							X
ANL	SRC komplemense AND átvitelbit							X
ORL	SRC OR átvitelbit							X
ORL	SRC komplemense OR átvitelbit							X
MOV	SRC megy az átvitelbitbe							X
MOV	Átvitelbit megy SRC-be							X
JV	Relatív ugrás, ha átvitelbit=1							
JNC	Relatív ugrás, ha átvitelbit=0							
JB	Relatív ugrás, ha az adott bit 1-es							X
JNB	Relatív ugrás, ha az adott bit 0							X
JBC	R. ugrás, ha az adott bit 1-es és az átvitel 0							X
ACALL	Szubrutinhívás (11 bites cím)							
LCALL	Szubrutinhívás (16 bites cím)							
RET	Visszatérés szubrutinból							
RETI	Visszatérés megszakításból							
SJMP	Rövid relatív ugrás (8 bites cím)							
AJMP	Abszolút ugrás (11 bites cím)							
LJMP	Abszolút ugrás (16 bites cím)							
JMP	Indirekt ugrás DPR+ACC relatív							
JZ	Ugrás, ha ACC=0							
JNZ	Ugrás, ha ACC nem 0							
CJNE	Ugrás, ha SRC és ACC nem egyenlő			X		X		
CJNE	Ugrás, ha SRC ≠ közvetlen operandus		X		X			
DJNZ	DST csökkentése; ugrás, ha DST ≠ 0							
NOP	Üres utasítás							

5.38. ábra. *A 8051 utasításkészlete, 2. rész*

Mai tudásunk alapján senki nem tervezne ilyen szabálytalan gépet, ilyen kevés és különböző regiszterrel. Ez megnehezíti a fordítóprogram készítését. A regiszterek hiánya arra kényszeríti a fordítóprogramokat, hogy a változókat ki-be pakolják a memóriába, ami költséges üzlet, még akkor is, ha két- vagy háromszintű gyorsítótár van. Jó ajánlólevél az Intel tervezői számára, hogy a Pentium 4 az ISA-korlátok ellenére, ilyen gyors. De láttuk a 4. fejezetben, hogy a megvalósítás rendkívül bonyolult.

Az UltraSPARC III modern ISA-tervezést reprezentál. Teljesen 64 bites architektúra (128 bites sínnel). Sok regisztere van, utasításai a háromregiszteres műveleteket hangsúlyozzák, plusz vannak a LOAD és STORE utasítások. Minden utasítása azonos hosszúságú, bár az utasításformátumokat nem sikerült kézben tartani. De még így is kézenfekvő és hatékony megvalósításhoz vezet. Minden modern tervezés példának tekintheti, de kevesebb utasításformátummal.

A 8051-es egyszerű és eléggé szabályos utasításrendszerrel rendelkezik, viszonylag kevés utasítással és kevés címzési móddal. Kitüntetett jelentőségű tulajdonsága, hogy négy regiszterkészlete van a gyors megszakításfeldolgozásra, valamint a regiszterek memóriabeli elhelyezkedése és elérése, és a meglepően hatékony bitmanipuláló utasítások. Fő célja, hogy nagyon kevés tranzisztorral megvalósítható legyen, így sok felvihető egy klisére, és ez alacsony CPU-árat eredményez.

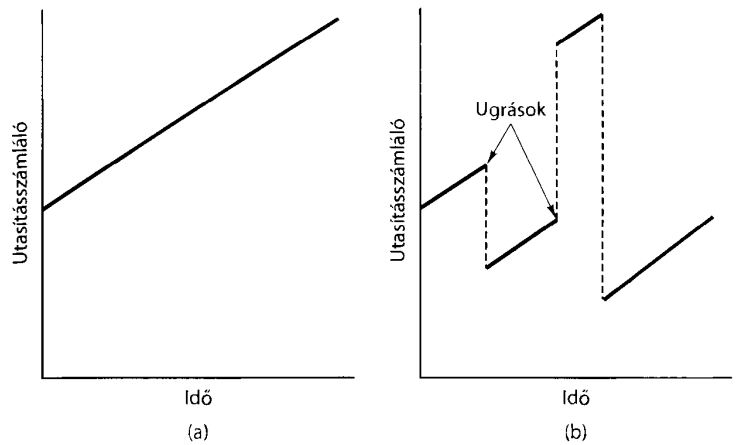
5.6. Vezérlési folyamat

A vezérlési folyamat azt a sorrendet jelenti, amelyben az utasítások végrehajtódnak dinamikusan, tehát a program végrehajtása során. Általában elágazás és eljáráshívás hiányában az egymás után végrehajtott utasítások egymás utáni memóriahelyekről töltődnek be. Eljáráshívás hatására a végrehajtási sorrend megváltozik, az éppen végrehajtás alatt lévő eljárás felfüggesztődik, és a hívott eljárás végrehajtása elkezdődik. A korutinok (társrutinok) hasonlók az eljárásokhoz, ezek is hasonlóan térítik el a végrehajtási sorrendet. A korutinok hasznosak a párhuzamos folyamatok szimulálásánál. A csapdák és a megszakítások speciális feltételek bekövetkezésekor térítik el a végrehajtási sorrendet. Mindezeket a témákat tárgyaljuk ebben az alfejezetben.

5.6.1. Szekvenciális vezérlés és elágazás

A legtöbb utasítás nem változtatja a végrehajtási sorrendet. Miután egy utasítás végrehajtódott, a memóriában őt követő utasítás olvasódik és hajtódik végre. Minden utasítás után az utasításszámláló értéke megnövelődik a végrehajtott utasítás hosszával. Az utasításszámláló értéke az idő függvényében hozzávetőleg lineárisan növekszik, az átlagos utasításhossz/átlagos utasítás-végrehajtási idő mértékével. Másképpen kifejezve, az a dinamikus sorrend, ahogy a processzor ténylegesen végrehajtja az utasításokat, megegyezik az utasítások programbeli sorrendjével; ezt mutatja az 5.39. (a) ábra. Ha a program tartalmaz elágazást, akkor ez az egyszerű kapcsolat az utasítások memóriabeli sorrendje és a végrehajtásuk sorrendje között nem áll fenn. Ha van elágazás, akkor az utasításszámláló értéke nem lineáris függvénye többé az időnek, amint azt az 5.39. (b) ábrán látjuk. Végeredményben nehézzé válik az utasítások sorrendjének vizualizálása a programlista alapján.

Ha a programozók számára nehézséget okoz nyomon követni, hogy a processzor milyen sorrendben fogja végrehajtani az utasításokat, akkor könnyen hibázhatnak. Ez az észrevétel vezetett Dijkstra (1968a) „GO TO Statements Considered Harmful" (A GO TO utasítás ártalmas) című ellentmondásos leveléhez, amelyben a goto utasítás mellőzését javasolta. Ez a levél váltotta ki a strukturált programozás forradalmának megszületését, amely a goto kiküszöbölésére jobban strukturált vezérlési szerkezeteket javasol, például a while ciklusokat. Természetesen az ilyen programokat is 2. szintű (utasításrendszer-architektúra szintű) programokra



5.39. ábra. *Az utasításszámláló értéke az idő függvényében (simított). (a) Elágazás nélkül. (b) Elágazással*

fordítják, amelyek sok ugrást tartalmaznak, hiszen az if, while és más magas szintű vezérlési szerkezetek megvalósítása ezen a szinten ugrásokkal történik.

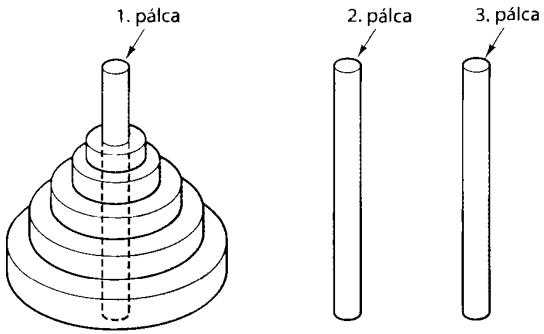
5.6.2. Eljárások

A programok stukturálásának legfontosabb technikáját az eljárások adják. Bizonyos szempontból az eljáráshívás ugyanúgy eltéríti a végrehajtási sorrendet, mint az ugrás, csak amikor befejezte feladatát az eljárás, a vezérlés visszatér a hívást követő utasításra.

Más szempontból azonban az eljárás törzse úgy tekinthető, mint egy magasabb szintű utasítás definíciója. Ebből a szempontból az eljáráshívás egy egyszerű utasításnak tekinthető, még akkor is, ha az eljárás meglehetősen bonyolult. Eljáráshívást tartalmazó program megértéséhez csak azt kell tudni, hogy *mit* csinál, azt nem, hogy az eljárás *hogyan* dolgozik.

Különösen érdekes az eljárások egyik speciális fajtája, a **rekurzív eljárás**, amely önmagát hívja vagy közvetlenül, vagy más eljárásokon keresztül. A rekurzív eljárások tanulmányozása betekintést nyújt az eljáráshívások megvalósításába és a lokális változók mibenlétébe. A továbbiakban a rekurzív eljárásra mutatunk egy példát.

A Hanoi tornyai egy ősi probléma, amely egyszerűen megoldható rekurzív eljárással. Hanoi egyik kolostorában van három aranypálca. Az egyikre 64 darab, közepén lyukas koncentrikus aranykorongot fűztek. Minden korong kicsit kisebb átmérőjű, mint az alatta lévő. A második és harmadik pálcán kezdetben nincs semmi. A szerzetesek szorgalmasan dolgoznak azon, hogy minden korongot a harmadik pálcára mozgassanak úgy, hogy közben egyszer sem rakhatnak nagyobb korongot nálánál kisebbre. Azt mondják, ha befejezik, eljön a világvége. Ha vala-



5.40. ábra. *A Hanoi tornyai probléma kezdeti állása 5 koronggal*

ki kézzelfogható gyakorlati tapasztalatokat akar szerezni, megteheti mindezt kevesebb műanyag koronggal is, de a világvége effektus bekövetkezéséhez 64 darab aranykorong kell. Az 5.40. ábra 5 korongra mutatja be a kezdeti állást.

n korongnak az 1. pálcáról a 3. pálcára való mozgatásának megoldása: Először mozgassuk át a felső *n* – 1 korongot a 2. pálcára, majd rakjuk át az 1. pálcán maradt egyetlen korongot a 3. pálcára, aztán a 2. pálcáról mozgassuk át az ott lévő *n* – 1 korongot a 3. pálcára (lásd 5.41. ábra).

A probléma megoldására olyan eljárás kell, amely *n* korongot átmozgat az *i*. pálcáról a *j*. pálcára. Ha ezt az eljárást meghívjuk a

```
towers(n, i, j)
```

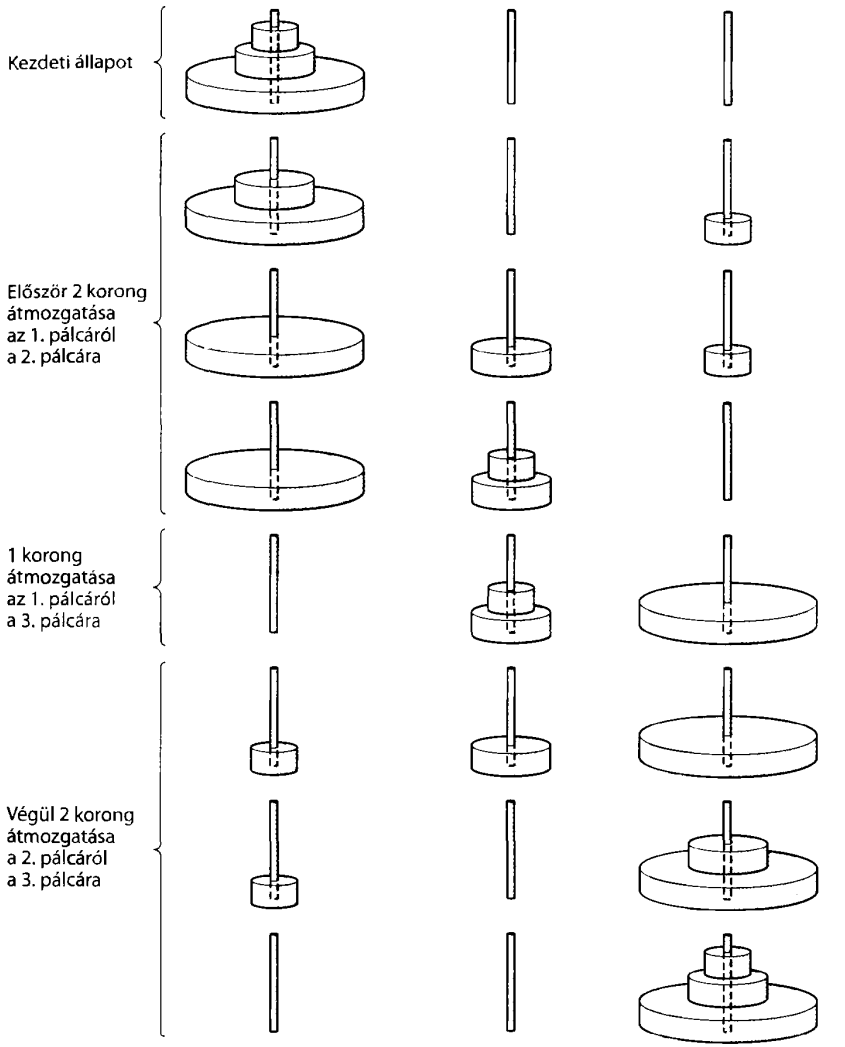
utasítással, az a megoldást írja ki. Az eljárás először ellenőrzi az $n = 1$ feltétel teljesülését. Ha teljesül, akkor a megoldás triviális, át kell rakni a korongot az *i*. pálcáról a *j*. pálcára. Ha $n \square 1$, akkor a megoldás három lépésből áll, mint már említettük, mindegyik egy rekurzív eljáráshívás. A teljes megoldás az 5.42. ábrán látható. A

```
towers(3, 1, 3)
```

hívás, amely az 5.41. ábrán látható probléma megoldását adja, három további hívást generál:

```
towers(2, 1, 2)
towers(1, 1, 3)
towers(2, 2, 3)
```

Az első és a harmadik további három hívást eredményez, összesen tehát hét hívás lesz.



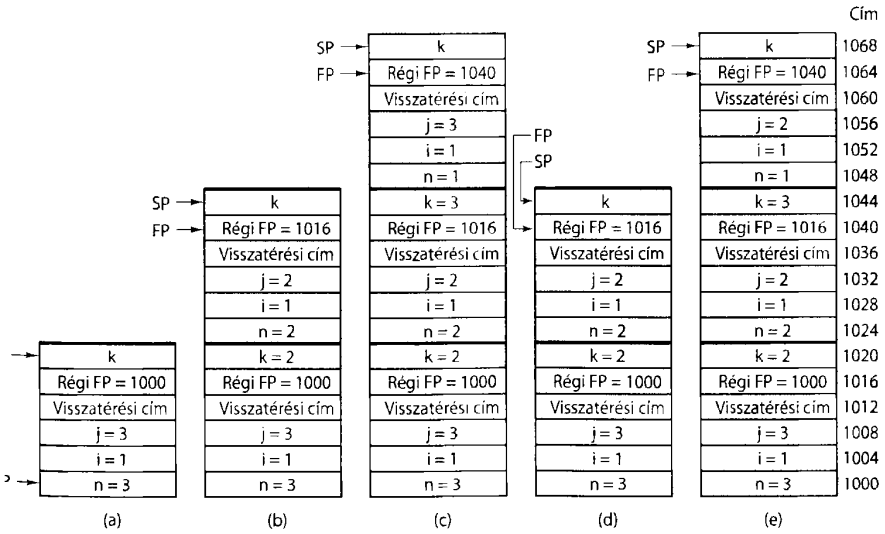
5.41. ábra. *A Hanoi tornyai probléma lépései 3 korongra*

Rekurzív eljárások megvalósításához verem kell, ahol az eljárás paramétereit és lokális változóit tároljuk minden egyes hívás esetén, mint azt az IJVM esetén láttuk. Minden egyes eljáráshíváskor a hívott eljárás számára új veremkeret létesül a verem tetején. A példánkban a verem felfelé növekszik, alacsonyabb címtől magasabb felé, mint az IJVM-ben.

```
public void towers(int n, int i, int j) {
 int k;

 if (n == 1)
  System.out.println("Korong mozgatása" + i + "-ről " + j "-re");
 else {
  k = 6 - i - j;
  towers(n - 1, i, k);
  towers(1, i, j);
  towers(n - 1, k, j);
 }
}
```

5.42. ábra. *Eljárás a Hanoi tornyai probléma megoldására*

A veremmutató mellett, amely mindig a verem tetejére mutat, gyakran kényelmes, ha van egy veremkeret-mutatónk is, az FP, amely a kereten belül fix helyre mutat. Ez a hely lehet a kapcsolómutató (OBJREF), mint az IJVM-ben, vagy a nulladik lokális változó. Az 5.43. ábra egy 32 bites szavas gép esetére mutatja a veremkeretet. Az első *towers* hívás a verembe rakja az *n*, *i* és *j* paramétereket, majd CALL utasítást hajt végre, amely a verembe teszi a visszatérési címet az 1012 címre. A hívott eljárásba való belépéskor a hívott eltárolja a régi FP értékét (1000) a verembe az 1016 címre, aztán növeli a veremmutatót, ezzel helyet foglal a lokális változók számára. Egy 32 bites lokális változó (*k*) esetén az SP értéke 4-gyel nő, és 1020 lesz. Az 5.43. (a) ábra mutatja ezt a helyzetet.



5.43. ábra. *A verem állapotai az 5.42. ábra programjának végrehajtása során*

Az első, amit a hívott eljárásnak tennie kell, az FP értékének elmentése (így az visszatölthető lesz kilépéskor), aztán az SP-t FP-be kell másolnia, és esetleg növelnie kell egy szó méretével, attól függően, hogy az FP hova mutasson. Példánkban az FP a régi FP-re mutat, de IJVM esetén a kapcsolómutatóra. Különböző gépek kissé különböző módon kezelik a veremkeretet, az FP-t néha a keret aljára, néha a tetejére, máskor pedig valahova középre teszik, mint az 5.43. ábrán. Ebből a szempontból megéri összehasonlítani az 5.43. és a 4.12. ábrát, amelyek a kapcsoló két különböző kezelését mutatják. De más módszerek is lehetségesek. Minden esetben a lényeg annak a biztosítása, hogy az eljárás visszatérését és a verem hívás előtti állapotának helyreállítását meg tudjuk oldani.

Eljárásprológusnak nevezzük azt a tevékenységet, amely a régi veremkeret-mutató elmentését, az új veremkeret-mutató megadását és a veremmutatónak a lokális változók befogadása miatti növelését tartalmazza. Az eljárásból való visszatéréskor a vermet ki kell takarítani, ezt nevezzük **eljárásepilógus**nak. Minden számítógép egyik legfontosabb jellemzője, hogy a prológust és az epilógust milyen röviden és gyorsan tudja elvégezni. Ha ez hosszú és lassú, akkor az eljáráshívások költségesek. Azok a programozók, akik számára fontos a hatékonyság, kerüljék a sok rövid eljárást, helyette inkább hosszú, monolitikus programot írjanak. A Pentium 4 ENTER és LEAVE utasításait azért tervezték, hogy a prológus, illetve epilógus nagy részét hatékonyan elvégezzék. Természetesen, ezek a veremkezelés egy bizonyos módszerére épülnek, így ha a fordítóprogram más módszert alkalmaz, akkor nem használhatók.

Térjünk vissza a Hanoi tornyai problémához. Minden egyes eljáráshívás új veremkeretet ad hozzá a veremhez, és minden visszatérés eltávolít egyet. Annak illusztrálására, hogy a verem használatával hogyan valósítjuk meg a rekurzív eljárásokat, kövessük nyomon a towers(3, 1, 3) eljáráshívást.

Az 5.43. (a) ábra azt az állapotot mutatja, amely közvetlenül a hívás (és a prológus) után keletkezik. Az eljárás először ellenőrzi, hogy teljesül-e az $n = 1$ feltétel, és mivel azt találja, hogy $n = 3$, így végrehajtja a

```
towers(2, 1, 2)
```

eljáráshívást. Ennek befejezése utáni veremállapotot mutatja az 5.43. (b) ábra. Az eljárás törzsének végrehajtása elölről kezdődik (a hívott eljárás mindig a legelején kezdődik). Ekkor sem teljesül az $n = 1$ feltétel, így ismét kiszámítja *k* értékét és a

```
towers(1, 1, 3)
```

hívást hajtja végre. A verem ekkori állapotát mutatja az 5.43. (c) ábra, és az utasításszámláló az eljárástörzs elejére mutat. Ekkor az $n = 1$ feltétel teljesül, így kiír egy sort. Ezután az eljárás visszatér, az aktuális veremkeret törlődik a veremből,

* Az eljáráshívás befejezésébe azt is beleértjük, hogy a hívott eljárás végrehajtotta a prológust. *(A lektor)*

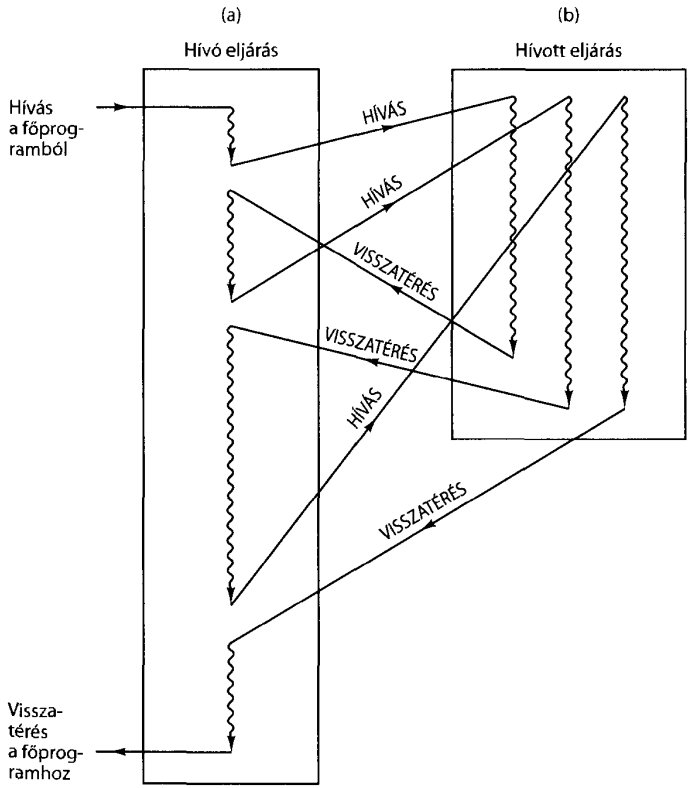
az FP és SP felveszi az értékét; ezt mutatja az 5.43. (d) ábra. Ezután folytatódik a visszatérési címen a végrehajtás, ami a

```
towers(1, 1, 2)
```

végrehajtását jelenti. Ez egy újabb veremkeretet ad a veremhez, az állapotot az 5.43. (e) ábrán látjuk. Egy újabb sor íródik ki, aztán a visszatéréskor a veremkeret törlődik a veremből. A végrehajtás ennek megfelelően folytatódik mindaddig, amíg az eredeti eljáráshívás végrehajtása be nem fejeződik, és az 5.43. (a) ábrán látható veremkeret nem törlődik a veremből. A rekurzió működésének jobb megértése végett javasoljuk, hogy papíron ceruzával szimulálja a

```
towers(3, 1, 3)
```

eljáráshívás teljes végrehajtását.



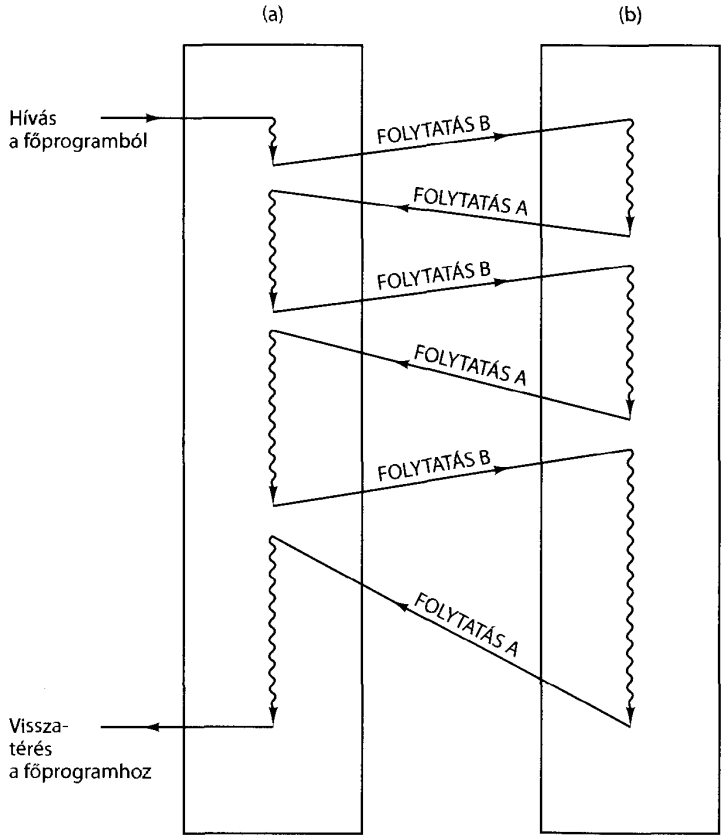
5.44. ábra. *A hívott eljárás végrehajtása mindig az első utasításával kezdődik*

5.6.3. Korutinok (társrutinok)

A szokásos végrehajtási sorozatot tekintve világos a különbség a hívó és a hívott eljárás között. Tekintsük az *A* eljárást, amely hívja a *B* eljárást az 5.44. ábrán mutatott módon.

A *B* eljárás számol egy ideig, aztán visszatér az *A* eljáráshoz. Első ránézésre úgy tűnhet, hogy a helyzet szimmetrikus, hisz sem az *A*, sem a *B* nem főprogram, mindkettő eljárás (az *A* eljárást a főprogram hívhatta, de ez nem lényeges). Továbbá, először a vezérlés az *A*-tól átadódik a *B*-hez – ez a hívás, majd a *B*-től vissza az *A*-hoz – ez a visszatérés.

Az aszimmetria abból adódik, hogy amikor a vezérlés *A*-tól *B*-hez kerül, akkor a végrehajtás *B* eljárástörzsének elején kezdődik, amikor pedig a visszatér *B*-től



5.45. ábra. *Korutin visszatérésekor a végrehajtás ott folytatódik, ahol előzőleg a vezérlés elhagyta, nem az elején*

A-hoz, akkor a végrehajtás nem *A* törzsének elején folytatódik, hanem a hívást követő utasítással.

Ha *A* később ismét hívja a *B*-t, akkor a végrehajtás ismét *B* elején kezdődik, nem pedig a korábbi visszatérést követő helyen. Ha *A* futása közben többször is hívja *B*-t, minden egyes hívás hatására *B* végrehajtása mindig az elején kezdődik, az *A* végrehajtása azonban sohasem kezdődik az elejéről.

Ez a különbség tükröződik abban a módszerben, ahogy a vezérlés átadódik *A* és *B* között. Ha *A* hívja *B*-t, akkor eljáráshívó utasítást hajt végre, amely a visszatérési címet (tehát a hívás helyét követő utasítás címét) eltárolja alkalmas helyen, például a verem tetején. Ezután a hívott eljárás címét tölti az utasításszámlálóba, így kezdeményezi a hívást. Amikor *B* visszatér, nem eljáráshívó utasítást hajt végre, hanem visszatérőt, amely egyszerűen kiveszi a veremből a címet, és azt az utasításszámlálóba tölti.

Néha hasznos, ha két eljárás, *A* és *B* egymást eljárásként hívja oly módon, mint azt az 5.45. ábra mutatja. Ha *B* visszatér *A*-hoz, akkor a végrehajtás a *B*-t hívó utasítás után folytatódik, mint fent látható. Ha a vezérlés *A*-tól *B*-hez kerül, akkor a végrehajtás (kivéve az első alkalmat) a legutolsó *B*-ből való visszatérést követő, vagyis az utolsó *A*-t hívó utasítással folytatódik. Az olyan két eljárást, amelyek ilyen módon működnek, **korutin**nak (**társrutin**) nevezzük.

Korutinokat általában párhuzamos feldolgozás szimulálására használnak olyan gépeken, amelyekben csak egy CPU van. Mindegyik korutin pszeudopárhuzamosan fut a másikkal, mintha saját processzora lenne. Ez a programozási stílus megkönnyíti bizonyos alkalmazások megvalósítását, továbbá hasznos olyan szoftver tesztelésénél, amelyet később többprocesszoros gépen futtatnak.

Sem a hagyományos eljáráshívó CALL, sem a visszatérő RETURN nem alkalmazható korutinoknál. Mivel az ugrás helyének címe a veremből jön, mint a visszatérés esetén, de a korutin maga is eltárolja a címet a későbbi visszatérés végett. Jó lenne, ha lenne olyan utasítás, amely kicserélné az utasításszámláló és a verem tetején lévő címet. Részletesen, ez az utasítás először kivenné a régi visszatérési címet a veremből egy regiszterbe, aztán betenné az utasításszámlálót a verembe, és végül az előbbi regiszter tartalmát az utasításszámlálóba másolná. Mivel mindig egy szót kivenne, egyet meg beletenne a verembe, így a veremmutató nem változna. Nemigen van azonban ilyen utasítás, többnyire több utasítással kell megvalósítani.

5.6.4. Csapdák

A **csapda** olyan automatikusan hívott eljárás, amelyet valamely, a program által előidézett fontos, de ritkán előforduló feltétel vált ki. Jó példa erre a túlcsordulás. Sok számítógépen, ha az aritmetikai művelet eredménye olyan nagy lenne, hogy nem lehet a gépen ábrázolni, akkor csapda keletkezik, ami azt jelenti, hogy a vezérlés egy meghatározott memóriacímre adódik, nem folytatódik normálisan. Azt az eljárást, amelyre ilyenkor a vezérlés adódik, **csapdakezelő**nek hívják, amely valamilyen megfelelő tevékenységet végez, például hibaüzenetet ír ki. Ha az aritmetikai művelet eredménye az ábrázolási tartományban marad, akkor nem keletkezik csapda.

A csapda alapvető jellemzője, hogy a program maga okoz valamilyen kivételes helyzetet, amelyet a hardver vagy a mikroprogram fedez fel. A túlcsordulás kezelésének alternatív módja, amikor a túlcsordulás egy 1 bites regisztert állít 1-re. A programozónak, ha kezelni akarja a túlcsordulást, minden aritmetikai utasítás után le kell kérdeznie ezt a bitet. Ez azonban lassú és tárpazarló megoldás. A csapda időt és memóriát takarít meg a programvezérelt megoldáshoz képest.

A csapda megvalósítható a mikroprogram (hardver) által végzett tényleges ellenőrzéssel. Ha a csapda bekövetkezett, akkor a csapdakezelő rutin címe töltődik az utasításszámlálóba. Ami csapda egy adott szinten, az programvezérelt lehet alacsonyabb szinten. Ha mikroprogrammal végeztetjük az ellenőrzést, a program által végzett teszthez képest még mindig időt takarítunk meg, mert könnyen átlapolható más tevékenységgel. Memóriát is megtakarítunk, mert csak egy helyen, a mikroprogram fő ciklusában kell elhelyezni.

Néhány kivételes helyzet, amely csapdával lekezelhető: lebegőpontos túlcsordulás, lebegőpontos alulcsordulás, egész túlcsordulás, védelem megsértése, definiálatlan műveleti kód, veremtúlcsordulás, nem létező B/K eszköz indítása, szóolvasási kísérlet páratlan memóriacímről, 0-val való osztási kísérlet.

5.6.5. Megszakítások

A **megszakítás** olyan eltérítése a vezérlési folyamatnak, amelyet nem a program okoz, hanem valami más, általában B/K. Például, a program utasíthatja a lemezegységet, hogy kezdje az adatátvitelt, és annak befejeztekor megszakítást küldjön. Ugyanúgy, mint a csapda, a megszakítás bekövetkeztekor is megáll a program végrehajtása, és a vezérlés a **megszakításkezelő**re adódik, amely elvégzi a kívánt tevékenységet. Ha ez befejeződött, a megszakításkezelő visszaadja a vezérlést a megszakított programnak. A megszakított processzusnak pontosan azt az állapotát kell helyreállítani, mint ami akkor volt, amikor a megszakítás történt.

Az alapvető különbség a csapdák és a megszakítások között a következő: a *csapdák* a programmal szinkronban vannak, a *megszakítások* pedig aszinkronban. Ha egy programot milliószor megismétlünk ugyanazzal a bemenettel, a csapda mindig pontosan ugyanott keletkezik, de a megszakítás nem, függhet például attól, hogy a gépkezelő mikor nyomja le a return gombot. A csapdák reprodukálhatóságának az az oka, hogy közvetlenül a program okozza, míg a megszakításokat nem, azokat csak közvetetten okozza a program.

Megszakítások tényleges működésének bemutatására tekintsünk egy általános példát: hogyan végzi a számítógép egy karaktersor kiírását terminálra. A rendszerszoftver először összegyűjti a kiíratandó karaktereket egy pufferben, beállítja, hogy egy *ptr* globális változó mutasson a puffer elejére, egy másik *count* globális változóba pedig a kiírandó karakterek számát írja. Ezután lekérdezi, hogy a terminál készen áll-e a kiírás fogadására, ha igen, akkor elindítja a B/K műveletet az első karakter kiíratásával (lásd 5.31. ábra). Miután a CPU elindította a B/K műveletet, felszabadul más program futtatására.

Bizonyos idő múlva a karakter a terminál képernyőjére íródik. Itt kezdődik a megszakítás. Leegyszerűsített formában a következő lépések hajtódnak végre.

Hardvertevékenység
1. Az eszközvezérlő beállít egy megszakítási vonalat a rendszersínen a megszakítási sorozat elindítására.
2. Amint a CPU a megszakítás kezelésére képes állapotba kerül, azonnal beállítja a megszakítást nyugtázó jelet a rendszersínen.
3. Amint az eszközvezérlő észleli, hogy a megszakítási kérelmét nyugtázta a CPU, saját azonosítására egy kis egész számot küld az adatvonalon. Ezt a számot **megszakítási vektor**nak nevezik.
4. A CPU kiolvassa a sínről megszakítási vektort, és időlegesen elmenti.
5. A CPU berakja a verembe az utasításszámlálót és a PSW-t.
6. Ezután a CPU beállítja az új utasításszámláló értékét arra a memóriacímre, amelyet a megszakítási vektor meghatároz. Ezt úgy kapja, hogy a megszakítási vektort egy táblázat indexének tekinti. Ha például az utasításszámláló 4 bájtos, és a megszakítási vektor n, akkor a cím $4n$ lesz. Az új utasításszámláló annak a megszakításkezelő rutinnak a kezdőcíme lesz, amely a megszakítást kérő eszközt szolgálja ki. Gyakran betölti vagy módosítja PSW-t (például azért, hogy újabb megszakítás bekövetkezését megakadályozza).

Szoftvertevékenység
7. A megszakításkezelő rutin első dolga, hogy elmentse a regisztereket, hogy később vissza tudja tölteni. A mentés történhet a verembe vagy rendszerterületre.
8. Az azonos típusú eszközök megszakítási vektora általában azonos, így a szerviz rutin nem tudja, melyik terminál okozta a megszakítást. A terminál számát eszközregiszterből olvassa ki.
9. A megszakítással kapcsolatos minden egyéb információt, mint például az állapotkódot most már be lehet olvasni.
10. Ha B/K hiba keletkezik, itt lehet lekezelni.
11. A *prt* és *count* globális változók értékét aktualizálja. Az előbbit növeli, hogy a következő bájtra mutasson, az utóbbit csökkenti eggyel, mert eggyel kevesebb átviendő bájt maradt. Ha a *count* nagyobb, mint 0, akkor még van átviendő karakter. Átmásolja az aktuális karaktert a kimeneti pufferbe.
12. Ha megkövetelt, akkor speciális kódot küld ki, hogy közölje az eszközzel vagy a megszakításvezérlővel, hogy a megszakítás feldolgozását elvégezte.
13. Visszatölti az elmentett regisztereket.
14. Végrehajtja a RETURN FROM INTERRUPT (VISSZATÉRÉS MEGSZAKÍTÁSBÓL) utasítást, amellyel a CPU visszakerül abba az állapotába, amelyben a megszakítás bekövetkezése előtt volt. A gép ott folytatja munkáját, ahol abbahagyta.

A megszakításokkal kapcsolatos kulcsfogalom az **átlátszóság**. Amikor megszakítás következik be, bizonyos műveletek végrehajtódnak, bizonyos kód végrehajtódik, de amikor ennek vége, a számítógép ugyanabba az állapotba kerül, mint amelyben a megszakítás előtt volt. Az ilyen módon működő megszakítási rutint

átlátszónak nevezzük. Az átlátszóság miatt sokkal egyszerűbb megérteni a megszakításokat.

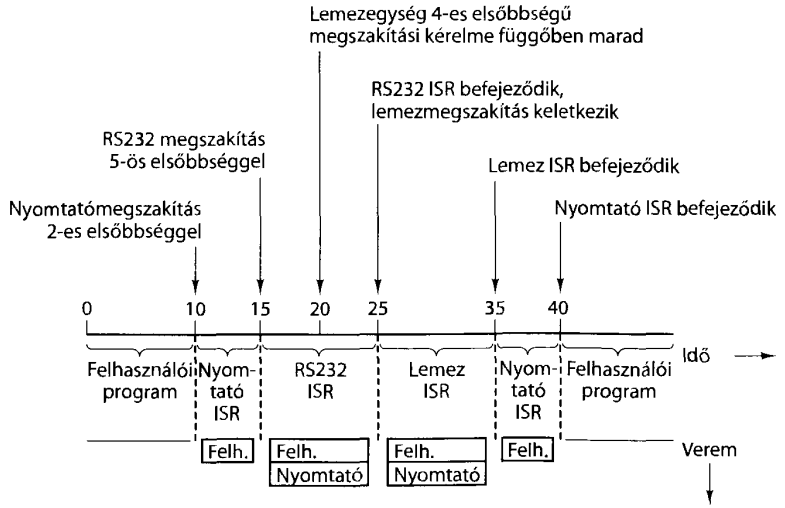
Ha a számítógépnek csak egy B/K eszköze van, akkor a megszakítás mindig az előbb leírt módon működik, és többet nem is tudunk róla mondani. Egy nagy számítógépnek azonban sok B/K eszköze lehet, amelyek egyidejűleg működhetnek, gyakran különböző felhasználók számára. Nem nulla a valószínűsége annak, hogy amíg egy megszakítás feldolgozása folyik, aközben egy másik B/K eszköz akar megszakítást kérni.

Két megoldási mód is kínálkozik. Az első szerint minden megszakítási rutin, még mielőtt bármi mást tenne, lehetetlenné tesz további megszakításokat. Ez a módszer egyszerűvé tesz mindent, mert a megszakításokat szigorúan szekvenciálisan dolgozzuk fel. De ez problémát okozhat azoknál az eszközöknél, amelyek nem tolerálják a késleltetést. Például egy 9600 bps kommunikációs vonalról 1042 μs-onként érkeznek a karakterek, akár kész az eszköz, akár nem. Ha az elsőt még nem dolgozta fel és közben a második megérkezik, az elveszhet.

Ha egy számítógépnek időkritikus B/K eszköze van, akkor jobb megközelítés az, ha minden eszközhöz elsőbbségi értéket rendelünk. Hasonlóan, a CPU-nak is lehet elsőbbségi értéke, amelyet a PSW-ben tárol. Ha egy n elsőbbségi számú eszköz okoz megszakítást, akkor a megszakításkezelő rutin is n elsőbbségi érték szerint fut.

Mindaddig, amíg egy n elsőbbségi értékű megszakításkezelő rutin fut, minden ennél kisebb elsőbbségi értékű megszakítást letilt, amíg a CPU vissza nem tér a felhasználói program futtatásához (0 elsőbbségi érték). Másrészről, magasabb elsőbbségű eszköz megszakítási kérelme késleltetés nélkül engedélyezett lesz.

Ha a megszakítási rutinok maguk is megszakíthatók, akkor az adminisztráció megbízhatóvá tétele érdekében a legjobb módszer, ha a megszakításo-



5.46. ábra. *Példa többszörös megszakításra*

kat átlátszóvá tesszük. Lássunk egy egyszerű példát többszörös megszakításra. Számítógépünknek három B/K eszköze van, egy nyomtató, egy mágneslemez-egység és egy RS232 soros vonal, rendre 2, 4 és 5 elsőbbségi értékkel. Kezdetben ($t = 0$) egy felhasználói program fut, amikor hirtelen a $t = 10$ időben nyomtató-megszakítás keletkezik. A nyomtató megszakításkezelő rutinjának (Interrupt Service Routine, ISR) futása elkezdődik, amint az 5.44. ábrán látható.

A $t = 15$ időben az RS232 vonal figyelmet kíván, és megszakítást generál. Mivel az RS232 elsőbbsége (5) nagyobb, mint a nyomtatóé (2), a megszakítás bekövetkezik. A gép állapota (most a nyomtató megszakításkezelő rutinja fut) verembe kerül, és az RS232 megszakításkezelő rutinja elindul.

Kis idővel később, a $t = 20$ időben egy lemezművelet befejeződött és szolgáltatást igényel. Azonban a lemez elsőbbsége (4) alacsonyabb, mint az éppen futóé (5), így a CPU-hardver nem nyugtázza a kérelmet, az függőben marad. A $t = 25$ időben az RS232 rutin befejeződik, visszatér abba az állapotba, amelyben az RS232 megszakításának bekövetkezése előtt volt, nevezetesen a nyomtató rutinjának 2-es elsőbbségű futtatásába. Amint a CPU a 2-es elsőbbségű állapotba vált, még mielőtt egyetlen utasítás végrehajtódna, a lemez 4-es elsőbbségű megszakítását elfogadja, és a lemezkezelő rutin elindul. Amikor ez befejeződik, a nyomtató rutin folytatódik. Végül, a $t = 40$ időben minden megszakításkezelő rutin befejeződik, és a felhasználói program ott folytatódik, ahol abbamaradt.

A 8088 óta az Intel CPU-nak két megszakítási szintje (elsőbbségi értéke) van: maszkolható és nem maszkolható. A nem maszkolható megszakítások általában csak a katasztrófaközeli állapotokra használatosak, mint a memóriaparitás-hiba. Minden B/K eszköz maszkolható szinten dolgozik.

Amikor egy B/K eszköz megszakítást kér, a CPU a megszakítás vektort egy 256 elemű táblázat indexének tekinti, ahol a megszakításkezelő rutin címe van. A táblázat elemei 8 bájtos szegmensleírók, és a táblázat bárhol lehet a memóriában, annak kezdőcímét egy globális regiszter tartalmazza.

Egyetlen használható megszakítási szint esetén sincs lehetőség arra, hogy egy magasabb elsőbbségű eszköz megszakítsa egy közepes elsőbbségű megszakításkezelő rutinját, és megvédje attól, hogy egy alacsonyabb elsőbbségű ugyanazt tegye vele. A probléma megoldása érdekében az Intel-processzorok külső megszakításvezérlőt használnak (például 8259A). Amikor az első megszakítás megjelenik, mondjuk n elsőbbségű, a CPU megszakítódik. Ha később egy magasabb elsőbbségű megszakítás keletkezik, a megszakításvezérlő másodszor is megszakít. Ha a második megszakítás alacsonyabb elsőbbségű, akkor függőben marad mindaddig, amíg az előző be nem fejeződik. Ahhoz, hogy ez a séma működjön, a megszakításvezérlőnek tudnia kell, hogy mikor fejeződik be egy megszakításkezelő futása. Ezért a CPU-nak parancsot kell küldenie, amikor az aktuális megszakítás teljesen feldolgozódott.

5.7. Részletes példa: Hanoi tornyai

Miután tanulmányoztuk három gép utasításrendszer-architektúráját, összerakva a részleteket, vizsgáljunk meg részletesebben mindhárom gép esetében egy példaprogramot. A példánk a Hanoi tornyai probléma. Az 5.42. ábrán a probléma Java-megoldását adtuk. A következő szakaszban assembly szintű megoldásokat fogunk adni.

Azonban egy kicsit csalunk. Nem a Java-verziót fogjuk átfordítani Pentium 4- és UltraSPARC III-verziókra, hanem a C-verziót, hogy elkerüljük a Java B/K problémáit. Az egyetlen különbség, hogy a Java *println* utasítása helyett a megfelelő C-utasítást használjuk, amely

```
printf("Korong mozgatása %d-ről %d-re\n", i, j);
```

Céljainkat tekintve a *printf* kiíratási formátuma lényegtelen (tulajdonképpen a karakterlánc íródik ki, ahol a %d azt jelenti, hogy a számokat decimális formában kell kiírni).

Azért vesszük a C-változatot, mert a Pentium 4 és UltraSPARC III esetén nincs meg a Java B/K könyvtár, míg a C igen. A különbség minimális, csak a nyomtató utasítás különbözik.

5.7.1. A Hanoi tornyai probléma megoldása Pentium 4 assembly nyelven

Az 5.47. ábra a Hanoi tornyai probléma C megoldásának egyik lehetséges átfordítását tartalmazza Pentium 4 assembly nyelvre. A program nagy része eléggé nyilvánvaló. Az EBP regisztert használjuk veremkeret-mutatónak. Az első két szó a kapcsoló, így az első aktuális paraméter, az n (vagy N, mivel a MASM kisnagybetű érzéketlen) helye EBP + 8, ezt követi i és j az EBP + 12 és EBP + 16 helyen. A k lokális változó helye EBP – 4.

Az eljárás azzal kezdődik, hogy új veremkeretet létesít a verem tetején. Ezt úgy csinálja, hogy ESP-t az EBP keretmutatóba másolja. Aztán n-et hasonlítja 1-hez, elugrik az else ágra, ha $n > 1$. A then ág három értéket rak a verembe: a formátum karakterlánc címét, i-t és j-t, majd meghívja a nyomtató eljárást.

Az aktuális paramétereket fordított sorrendben rakja a verembe, mert a C-eljárások hívási konvenciója ezt így kívánja. Ezért kell a formátum karakterlánc címét a verem tetejére tenni. Mivel a *printf*-nek változó számú paraméterei lehetnek, így az egyenes sorrend esetén nem tudná, hogy milyen mélyen van a veremben a formátum karakterlánc.

A call utasítás után hozzáad 12-t az ESP-hez, hogy kivegye a paramétereket a veremből. Természetesen ténylegesen nem töröl, de az ESP módosításával hozzáférhetetlenekké válnak a verem műveletek számára.

Az else ág az *L1* címkénél kezdődik, és világos, hogy mit csinál. Először kiszámítja a $6 - i - j$ értéket és k-ban tárolja. Mindegy, hogy mi az i és j értéke, a harma-

```
        .586                        ; Pentium-kódra fordítás (nem 8088-ra)
.MODEL  FLAT
PUBLIC          _towers             ; exportáljuk towers-t
EXTERN  _printf: NEAR               ; importáljuk printf-et
.CODE
_towers: PUSH   EBP                 ; EBP mentése (ez a keretmutató)
        MOV     EBP, ESP            ; új keretmutató beállítása ESP felett
        SUB     ESP, 4              ; helyfoglalás a k lokális változónak
        CMP     [EBP+8], 1          ; ha (n == 1)
        JNE     L1                  ; ugrás ha n ≠ 1
        MOV     EAX, [EBP+16]       ; printf("...", i, j)
        PUSH    EAX                 ; az i, j paraméterek és a formátum
        MOV     EAX, [EBP+12]       ; karakterlánc a verembe kerül
        PUSH    EAX                 ; fordított sorrendben, a C konvenció miatt
        PUSH    OFFSET FLAT:format; ; OFFSET FLAT a format karakterlánc címe
        CALL    _printf             ; printf hívása
        ADD     ESP, 12             ; a paraméterek eltávolítása a veremből
        JMP     Done                ; végeztünk
L1:     MOV     EAX, 6              ; k = 6 – i – j
        SUB     EAX, [EBP+12]       ; EAX = 6 – i
        SUB     EAX, [EBP+16]       ; EAX = 6 – i – j
        MOV     [EBP-4], EAX        ; k = EAX
        PUSH    EAX                 ; towers(n – 1, i, k) hívás kezdete
        MOV     EAX, [EBP+12]       ; EAX = i
        PUSH    EAX                 ; i-t a verembe
        MOV     EAX, [EBP+8]        ; EAX = n
        DEC     EAX                 ; EAX = n –1
        PUSH    EAX                 ; n –1 megy a verembe
        CALL    _towers             ; towers(n – 1, i, k) hívás
        ADD     ESP, 12             ; a paraméterek eltávolítása a veremből
        MOV     EAX, [EBP+16]       ; towers(1, i, j) hívás kezdete
        PUSH    EAX                 ; j-t a verembe
        MOV     EAX, [EBP+12]       ; EAX = i
        PUSH    EAX                 ; i-t a verembe
        PUSH    1                   ; 1-et a verembe
        CALL    _towers             ; towers(1, i, j) hívás
        ADD     ESP, 12             ; a paraméterek eltávolítása a veremből
        MOV     EAX, [EBP+16]       ; towers(n – 1, k, j) hívás kezdete
        PUSH    EAX                 ; j-t a verembe
        MOV     EAX, [EBP-4]        ; EAX = k
        PUSH    EAX                 ; k-t a verembe
        MOV     EAX, [EBP+8]        ; EAX = n
        DEC     EAX                 ; EAX = n – 1
        PUSH    EAX                 ; n – 1 megy a verembe
        CALL    _towers             ; towers(n – 1, k, j) hívás
        ADD     ESP, 12             ; a paraméterek eltávolítása a veremből
Done:   MOV     ESP, EBP            ; a lokális terület felszabadítása
        POP     EBP                 ; Régi EBP visszaállítása
        RET     0                   ; visszatérés a hívó eljárásba
.DATA
format  DB "Korong mozgatása %d-ről %d-re\n"; formátum karakterlánc
END
```

5.47. ábra. *A Hanoi tornyai megoldása Pentium 4 assemblyben*

dik pálca sorszáma mindig $6 - i - j$. Azért tároljuk ezt a k lokális változóban, hogy ne kelljen még egyszer kiszámítani.

Aztán az eljárás önmagát hívja háromszor, mindig különböző paraméterekkel. A hívások után a verem kiürítődik. Ez minden.

A rekurzív eljárások elsőre zavarosnak tűnhetnek, de ezen a szinten tekintve őket eléggé világos képet kapunk. Ami a lényeg, a paramétereket a verembe kell rakni, és az eljárás meghívja önmagát.

5.7.2. A Hanoi tornyai probléma megoldása UltraSPARC III assemblyben

Próbáljuk meg még egyszer, most az UltraSPARC III-mal. A program listáját az 5.48. ábra tartalmazza. Mivel az UltraSPARC III kódja különösen olvashatatlan, még az assembly is, sőt még sok gyakorlat után is, ezért vettük a bátorságot, és az elején definiáltunk néhány szimbólumot, hogy világosabb legyen. Ezért a programot assembly fordítás előtt a *cpp* C-előfeldolgozóval át kell alakítani. Kisbetűs írásmódot használtunk, mert az UltraSPARC ezt megkívánja (arra az esetre, ha az olvasó be akarja gépelni a programot, és futtatni akarja).

Algoritmikusan az UltraSPARC-változat azonos a Pentium 4-változattal. Mindkettő n tesztelésével kezdődik, és elágazik, ha $n > 1$. Az UltraSPARC-változat bonyolultsága az ISA néhány tulajdonságából ered.

Kezdésként az UltraSPARC III program a formátum karakterlánc címét át kell adja a *printf* rutinnak, de nem tudja a kimenő paraméter címét regiszterben megadni, mert nem lehet 32 bites konstanst egyetlen utasítással betölteni. Ehhez két utasítás kell, a SETHI és az OR.

A másik megjegyzendő, hogy nem kell a vermet aktualizálni a hívás után, mert a RESTORE utasítás módosítja az ablakregisztert az eljárás végén. Az a képesség, hogy a kimenő paramétert regiszterbe lehet tenni, és így nem kell a memóriához fordulni, nagy nyereség, ha a hívási lánc nem túl mély, de általában az egész regiszterablak mechanizmus valószínűleg nem éri meg a bonyolultságot.

Vegyük észre a *b Done* ugrás után álló NOP utasítást. Ez eltolás résben van. Ez az utasítás annak ellenére végrehajtódik, hogy ugró utasítás előzi meg. Az a gond, hogy az UltraSPARC III olyan hosszú csővezetékkel rendelkezik, hogy mire a hardver észreveszi, hogy ugrással van dolga, a következő utasítás már gyakorlatilag befejeződött. Üdvözöljük a RISC-programozás csodálatos világában!

Ez a furcsa tulajdonság az eljáráshívásokra is érvényes. Figyeljük meg a *towers* első hívását az else ágban. $n - 1$-et *Param*0-ba és i-t *Param*1-be teszi, de előbb elvégzi a *towers* eljárás hívását, még mielőtt az utolsó paramétert átadná. A Pentium 4-nél először átadjuk a paramétereket, és utána jön a hívás. Itt először átadunk néhány paramétert, aztán elvégezzük a hívást, és utána átadjuk az utolsó paramétert. Ismét ugyanaz a jelenség, amire a hardver észreveszi, hogy *CALL* utasítással van dolga, a hívást követő utasítás már olyan mélyen benne van a csőben, hogy végre kell hajtania. Akkor miért nem adjuk át az utolsó paramétert az eltolásrésben? Ha a hívott eljárás első utasításának kell, még akkor is rendelkezésére fog állni.

```
#define  N %i0                          /* N a 0. bemenő paraméter */
#define  I  %i1                         /* I az1 . bemenő paraméter */
#define  J  %i2                         /* J a 2. bemenő paraméter */
#define  K  %l0                         /* K a 0. lokális változó */
#define  Param0 %o0                     /* Param0 a 0. kimenő paraméter */
#define  Param1 %o1                     /* Param1 az 1. kimenő paraméter */
#define  Param2 %o2                     /* Param2 a 2. kimenő paraméter */
#define  Scratch %l1                    /* a cpp C által használt megjegyzéskonvenció */
         .proc   04
         .global towers

towers:  save %sp, -112, %sp
         cmp N, 1                       ! ha (n == 1)
         bne  else                      ! ha (n != 1), ugrás az else ágra

         sethi   %hi(format), Param0    ! printf(„ ...", i, j)
         or Param0, %lo(format), Param0 ! Param0 = a formátum kar.lánc címe
         mov I, Param1                  ! Param1 = i
         call  printf                   ! printf hívása j beállítása előtt
         move  J, Param2                ! eltolásrés a hívás után
         b done                         ! végeztünk
         nop                            ! eltolásrés kitöltése

else:    mov 6, K                       ! k = 6 - i - j számítás kezdete
         sub  K, J, K                   ! k = 6 - j
         sub  K, I, K                   ! k = 6 - i - j

         add  N, -1, Scratch            ! towers(n - 1, i, k) hívás kezdete
         mov Scratch, Param0            ! Scratch = n - 1
         mov I, Param1                  ! paraméter1 = i
         call  towers                   ! towers hívása par.2 (k) beállítása előtt
         mov K, Param2                  ! eltolásrés a hívás után

         mov 1, Param0                  ! towers(1, i, j) hívás kezdete
         mov I, Param1                  ! paraméter1 = i
         call  towers                   ! towers hívása par.2 (j) beállítása előtt
         mov J, Param2                  ! paraméter2 = j

         mov Scratch, Param0            ! towers(n - 1, k, j) hívás kezdete
         mov K, Param1                  ! paraméter1 = k
         call  towers                   ! towers hívása par.2 (j) beállítása előtt
         mov J, Param2                  ! paraméter2 = j

done:    ret                            ! visszatérés
         restore                        ! eltolásrés

format: .asciz "Korong mozgatása %d-ről %d-re\n"
```

5.48. ábra. *A Hanoi tornyai probléma megoldása UltraSPARC III assemblyben*

Végül, láthatjuk, hogy a *Done* címkénél található RET utasításnak is van eltolásrése. Ezt a RESTORE használja, amely növeli CWP-t, hogy visszaállítsa az ablakregisztert a hívó kívánta állapotba.

5.8. Az Intel IA-64 architektúra és az Itanium 2

Az Intel gyorsan abba az állapotba kerül, hogy kifacsart minden cseppet az IA-32 architektúrából és a Pentium 4 processzorból. Az új modellek még kihasználhatják a gyártási technológia fejlődését, ami kisebb tranzisztorokat (következésképpen nagyobb órajelsebességet) jelent. Azonban egyre nehezebb lesz új trükköket találni a megvalósítás gyorsabbá tételére, mivel az a korlátozás, amelyet az IA-32 ISA jelent, idővel egyre nagyobbnak látszik.

A valódi megoldást az IA-32-nek mint a fejlesztés fő vonalának a feladása és egy teljesen új ISA bevezetése jelentheti. Ez az, amire az Intel valójában törekszik. Valójában két új vonal terveivel rendelkezik. Az EMT-64 a Pentium 4 szélesebb változata, 64 bites regiszterekkel és 64 bites memória-címtartománnyal. Ez a processzor megoldja a címtartomány problémáját, de megmarad a Pentium 4 megvalósítási bonyolultsága.

A másik új architektúra, amelyet az Intel és a Hewlett Packard közösen fejlesztett az **IA-64**. Ez elejétől a végéig teljesen 64 bites architektúra, és nem egy létező 32 bites gép továbbfejlesztése. Továbbá, ez az architektúra több tekintetben is radikális eltávolodás a Pentium 4-től. Az elsődlegesen megcélzott piac a nagy teljesítményű kiszolgálók piaca, de végül elérheti az asztali gépeket is. Mindenesetre az architektúra olyan radikálisan különbözik az eddig tanultaktól, hogy már ezért is megéri, hogy megvizsgáljuk. Az IA-64 architektúra első implementációja az Itanium sorozat. A fejezet hátralévő részében az IA-64 architektúrát és megvalósítását, az Itanium 2 CPU-t tanulmányozzuk.

5.8.1. A Pentium 4 problémái

Mielőtt belemennénk IA-64 és az Itanium 2 részleteibe, hasznos lehet áttekinteni, hogy mi is a gond a Pentium 4-gyel, hogy lássuk, milyen problémákat kívánt az Intel megoldani az új architektúrával. A fő ok, ami az IA-32 bajait okozza az ősi ISA az összes rossz tulajdonságával a mai technológiában. Az IA-32 CISC ISA, változó utasításhosszal, számtalan formával, amelyet nehéz gyorsan dekódolni. A jelenlegi technológiák a RISC ISA-val működnek legjobban, amelyekben egy utasításhossz és rögzített műveleti kódhossz van, azaz könnyű dekódolni. Az IA-32 utasításait szét lehet tördelni RISC-szerű mikroutasításokká futási időben, de ez hardvert (lapkaterületet) és időt igényel, továbbá bonyolulttá teszi a tervezést. Ez az első csapás.

Az IA-32 kétcímes, memóriaorientált ISA. A legtöbb utasítás hivatkozik a memóriára, és a legtöbb programozó és fordítóprogram nem foglalkozik a memória-

hivatkozásokkal. A mai technológia előnyben részesíti a töltő/tároló architektúrákat, amelyek csak az operandusok betöltése/kiírása végett hivatkoznak a memóriára, egyébként háromregiszteres utasításokkal számolnak. És mivel a CPU órajelsebessége nagyobb mértékben növekszik, mint a memóriáké, idővel a probléma egyre súlyosabb lesz. Ez a második csapás.

Az IA–32-nek kicsi és szabálytalan regiszterkészlete van. Az hagyján, hogy ez megköti a fordítóprogramok kezét, de a kevés általános regiszter (négy vagy hat, attól függően, hogy az ESI-t és EDI-t hova soroljuk) miatt a részeredményeket a memóriában kell tárolni, ami fölösleges memóriahivatkozásokat generál akkor is, amikor erre logikusan nincs szükség. Ez a harmadik csapás. Az IA–32 kiütve.

Lássuk a második menetet. A kisszámú regiszter sok függőséget okoz, különösen WAR függőségeket, mert az eredményt tárolni kell, de nincs elég regiszter. A regiszterek hiányának leküzdése belső átnevezésre kényszerít – a legrosszabb, ami létezhet –, titkos regiszterekké az átrendező pufferen belül. A gyakori gyorsítótárhiány-blokkolás elkerülésére az utasításokat sorrenden kívül kell végrehajtani. Azonban az IA–32 szemantika pontos megszakításokat specifikál, ezért a sorrenden kívüli utasítások sorrendjét helyre kell állítani. Mindez nagyon bonyolult hardvert igényel. Negyedik csapás.

Ahhoz, hogy mindezt gyorsan el lehessen végezni, hosszú csővezetékre van szükség. Másrészt a hosszú cső azt eredményezi, hogy az utasításoknak befejezésük előtt több ciklussal előbb be kell lépniük a csőbe. Következésképpen nagyon pontos elágazási jövendölés kell ahhoz, hogy a megfelelő utasítás kerüljön a csőbe. A jövendölési tévedés a cső kiürítését kényszeríti ki, ez nagy költség, még csekély tévedési arány is jelentős hatékonyságcsökkenést okozhat. Ötödik csapás.

Hogy enyhítsék a jövendölési tévedés okozta problémát, a processzornak spekulatív végrehajtást kell végeznie minden felmerülő problémával, különösen ha a téves úton való memóriahivatkozások csapdát okoznak. Hatodik csapás.

Nem kívánjuk tovább folytatni a mérkőzést, de világos, hogy itt valóban probléma van. És még meg sem említettük azt a tényt, hogy a 32 bites címzés miatt az egyes programok a 4 GB memóriába vannak korlátozva, ami egyre növekvő problémája a nagy szervereknek.

Summa summarum, az IA–32 esete hasonlítható a Kopernikusz előtti égi mechanika állapotához. A csillagászatot uraló akkori elmélet szerint a Föld mozdulatlanul áll a világűrben, és a bolygók körpályán keringenek körülötte. Azonban az egyre pontosabb megfigyelések a modelltől való egyre több eltérés megállapítását eredményezték, ami végül oda vezetett, hogy a körpályákhoz újabb körpályákat vettek, amíg az egész modell a belső bonyolultságától össze nem omlott.

Az Intel ugyanezzel a gonddal küzd. A Pentium 4 tranzisztorainak nagy hányada arra szolgál, hogy szétbontsa a CISC-utasításokat, kiderítse, hogy mit lehet párhuzamosítani, végzi a konfliktusok feloldását, jövendöléseket tesz, kijavítja a téves jövendöléseket, és más könyvelést végez, és kevés marad arra, hogy azt csinálja, amit a felhasználó kért tőle. Az Intel számára kíméletlen a végkövetkeztetés: kukába az egésszel (IA–32), és kezdjük az egészet elölről tiszta lappal (IA–64). Az EMT–64 ad némi lehetőséget, de valójában csak elfedi a bonyolultságot.

5.8.2. Az IA–64 modell: explicit utasításszintű párhuzamosság

Az IA–64 alapötlete, hogy futási idő helyett a fordítási időre koncentráljunk. A Pentium 4 esetén a CPU végrehajtás során átrendezi az utasítások sorrendjét, átnevez regisztereket, funkcionális egységeket ütemez, és számos más dolgot is csinál annak érdekében, hogy biztosítsa a hardvererőforrások elfoglaltságát. Az IA–64 esetén a fordítóprogram előre elvégzi mindezt, és olyan programot eredményez, amely futtatható, nem a hardvernek kell zsonglőrködnie a végrehatás során. Például ahelyett, hogy azt mondanánk a fordítónak, hogy a gépnek 8 regisztere van, holott ténylegesen 128, és futási időben derítse ki a hardver, hogyan lehet a függőségeket elkerülni, az IA–64 modellben megmondjuk a fordítónak, hogy ténylegesen hány regisztere van a gépnek, tehát olyan programot tud generálni, amelynek nem kell regiszterkonfliktussal foglalkoznia. Hasonlóan, ebben a modellben a fordítóprogram nyilvántartja, hogy mely funkcionális egységek foglaltak, és így nem bocsát ki olyan utasítást, amely foglalt egységet igényelne. Azt a modellt, amely láthatóvá teszi a hardver párhuzamosítási lehetőségét, **EPIC**-nek (**Explicitly Parallel Instruction Computing**) nevezzük. Bizonyos értelemben az EPIC a RISC utódjának tekinthető.

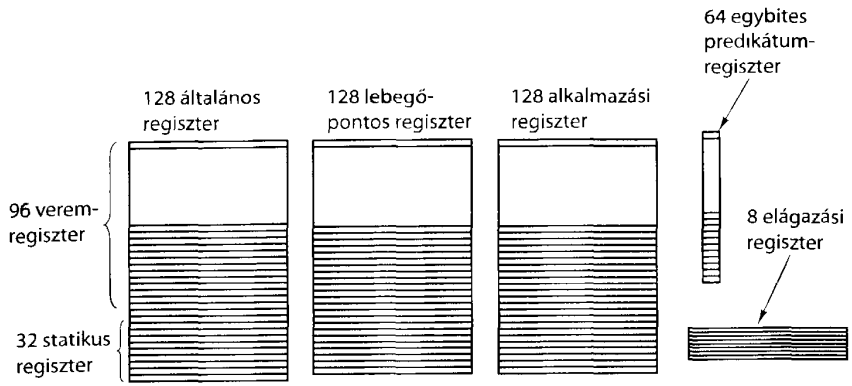
Az IA–64-nek számos olyan tulajdonsága van, amely a hatékonyságot növeli. Ezek közé tartozik a memóriahivatkozások csökkentése, utasításütemezés, feltételes elágazások csökkentése és előrejelzés. Ezeket a tulajdonságokat és az Itanium 2-beli megvalósításait fogjuk vizsgálni.

5.8.3. A memóriahivatkozások csökkentése

Az Itanium 2 memóriamodellje egyszerű. A memória maximálisan 2^{64} méretű lineáris tartomány. Az utasítások 1, 2, 4, 8, 16 vagy 10 bájt méretű mezők elérését teszik lehetővé, ahol az utolsó a 80 bites IEEE 754 szabvány szerinti lebegőpontos számok kezelésére szolgál. A memóriamezőknek nem kell a természetes határra igazítottnak lenniük, de ennek hiánya a hatékonyságot csökkenti. A memória lehet kis endián vagy nagy endián, amit az operációs rendszer által betölthető regiszter egy bitje határoz meg.

A memóriaelérés hatalmas útszűkületet képez a modern számítógépeknél, mert a CPU sokkal gyorsabb, mint a memória. A memóriahivatkozások gyorsításának egyik módja a nagy 1. szintű, lapkára integrált gyorsítótár, és a még nagyobb, lapkához közeli 2. szintű gyorsítótár. Minden modern tervezés tartalmazza ezt a két elemet. A gyorsítótár használatát túl lehet lépni, más módszer is van a memóriahivatkozások gyorsítására, és az IA–64 használ is néhányat.

A legjobb módja a memóriahivatkozások felgyorsításának, ha elkerüljük. Az Itanium 2 által megvalósított IA–64 modellben 128 általános célú 64 bites regiszter van. Az első 32 statikus, a maradék 96 pedig regisztervermet alkot, nagyon hasonlóan az UltraSPARC III regiszterablak technikájához. Azonban az UltraSPARC III-tól eltérően a program által látható regiszterek száma változhat,



5.49. ábra. *Az Itanium 2 regiszterei*

eljárásonként különböző lehet. Tehát minden eljárás eléri a 32 statikus regisztert és még néhány (változó számú) dinamikusan lefoglalt regisztert.

Eljárás hívásakor a regiszterverem mutatója növelődik, tehát a bemeneti paraméterek láthatók lesznek regiszterekben, de lokális változók számára nem foglalódik regiszter. Az eljárás maga dönti el, hogy hány regiszterre van szüksége, és ennek megfelelően növeli a regiszterverem mutatóját, hogy lefoglalja a regisztereket. Ezeket a regisztereket nem kell elmenteni belépéskor vagy visszaállítani befejezéskor, de ha az eljárás módosítani akarja a statikus regisztereket, akkor azokat mentenie kell és vissza kell állítania. Az eljárások igényeihez igazított, változó számú regiszterek használatának az az előnye, hogy nem pazaroljuk a kevés regisztert, és az eljáráshívások mélyebbre juthatnak anélkül, hogy a regisztereket memóriába kellene kirakni.

Az Itanium 2-nek 128 db IEEE 745 formátumú lebegőpontos regisztere is van. Ezek nem regiszterverem módon működnek. A regiszterek nagy száma lehetővé teszi, hogy lebegőpontos számítások során a részeredményeket is regiszterben tároljuk, ezzel elkerülhető, hogy a részeredményeket memóriában kelljen tárolni.

Van továbbá 64 egybites predikátumregisztere, nyolc elágazási regisztere és 128 speciális célú alkalmazási regisztere, ezek a különböző célokra használhatók, például az operációs rendszer és az alkalmazási program közötti paraméterátadásra. Az Itanium 2 regisztereinek áttekintése az 5.49. ábrán látható.

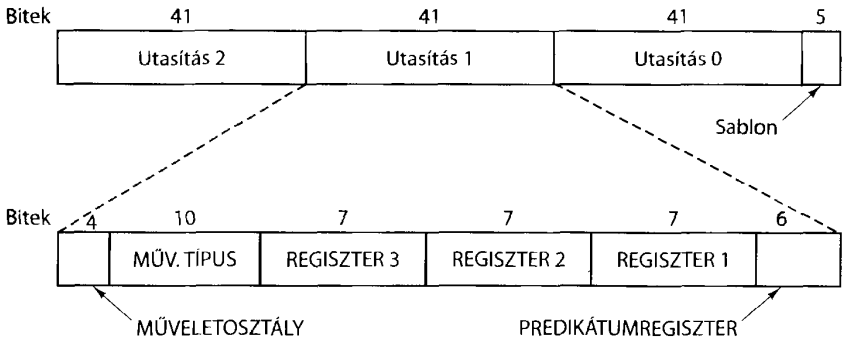
5.8.4. Utasításütemezés

A Pentium 4 egyik fő problémája, hogy bonyolult az utasításokat úgy ütemezni a különböző funkcionális egységek között, hogy elkerüljük a függőségeket. Különlegesen összetett mechanizmusok kellenek ahhoz, hogy futási időben tudjuk kezelni a fellépő igényeket, és ezek megvalósítása a lapka nagy részét elfoglalja. Az IA–64 és az Itanium 2 azzal kerüli el ezeket a problémákat, hogy a fordítóprogramra bízza a megoldást. Az alapötlet, hogy a program utasításcsoportok

sorozata. Bizonyos határokon belül az egy csoportba tartozó utasítások konfliktusmentesek, nem igényelnek több funkcionális egységet, mint amennyi a gépnek van, nem tartalmaznak RAW és WAW függőségeket, és csak korlátozott számú WAR függőség lehet. Az a látszat, hogy az egymást követő csoportok utasításait szigorúan szekvenciálisan kell végrehajtani, a következő csoport végrehajtása csak akkor indulhat el, ha a megelőző teljesen befejeződött. A CPU azonban azonnal elkezdheti a következő csoport végrehajtását, amint azt biztonságosnak érzi.

Ennek a szabálynak a következménye, hogy a CPU az egy csoportba tartozó utasításokat tetszés szerinti sorrendben ütemezheti, nem kell tartania a konfliktusoktól. Ha egy utasításcsoport megsérti ezt a követelményt, akkor definiálatlan lesz a hatása. A fordítóprogram feladata, hogy olyan utasítássorozatot generáljon a forrásprogramból, amely teljesíti a követelményeket. Gyors fordításkor, a program fejlesztése során, a fordítóprogram minden utasítást külön csoportba rakhat, ez egyszerű, de nem eredményez hatékony programot. Amikor eljön az ideje a végső termék előállításának, akkor a fordítóprogram sok időt tölthet az optimalizálással.

Az utasítások 128 bites **kötegekbe** vannak csoportosítva, ezt mutatja az 5.50. ábra felső része. Minden köteg három 41 bites utasítást és egy 5 bites sablont tartalmaz. Az utasításcsoportoknak nem kell teljes kötegekből állniuk; egy csoport kezdődhet és végződhet köteg belsejében is.



5.50. ábra. *Az IA–64 köteg három utasítást tartalmaz*

Több mint 100 utasításforma létezik. Tipikusnak tekinthető az ALU utasítás, amit az 5.50. ábra alsó része mutat. Ilyen az ADD, amely két regiszter tartalmának összegét képezi egy harmadik regiszterben. Az első mező a műveletosztály, amely többnyire az utasítás osztályát határozza meg, most például fixpontos ALU művelet. A második mező, a művelettípus, az alkalmazandó utasítást adja meg mint például ADD vagy SUB. Ezt követi a három regisztermező. Végül a predikátumregiszter áll, amelyet a következő szakaszban ismertetünk.

A köteg sablonja elsősorban azt mondja meg, hogy melyik funkcionális egységet igényli, és az utasításcsoport határát, ha kell. A fő funkcionális egységek: fixpontos ALU, fixpontos nem ALU utasítás, memóriaművelet, lebegőpontos művelet, el-

ágazás és egyéb. Természetesen 6 funkcionális egység és 3 utasítás esetén 216 kombináció lehetséges, plusz másik 216 a 0. utasítás utáni utasításcsoport marker után, további 216 az 1. utasítás utáni utasításcsoport-marker után, és még másik 216 a 2. utasítás utáni utasításcsoport-marker után. A sablon számára azonban csak 5 bit van, így erősen korlátozott a megengedett kombinációk száma. Másrészről, három lebegőpontos utasítás egy kötegben nem lehet, még akkor sem, ha valahogy meg tudnánk adni, mert három lebegőpontos utasítást nem végezhet párhuzamosan a CPU. Azok a megengedett kombinációk, amelyek ténylegesen alkalmazhatók.

5.8.5. Feltételes elágazások csökkentése: predikáció

Az IA–64 másik fontos jellemzője az a módszer, amellyel a feltételes elágazásokat kezeli. Ha mód lenne arra, hogy a legtöbbjüktől megszabaduljunk, a CPU sokkal egyszerűbb és gyorsabb lenne. Első ránézésre lehetetlen megszabadulni tőlük, mert a programok tele vannak if utasításokkal. Azonban, az IA–64 olyan technikát használ, az ún. **predikációt**, amely használatával nagymértékben csökkenthető a számuk (August és társai, 1998; Hwu, 1998). Most röviden ismertetjük a módszert.

A jelenlegi számítógépek minden utasítása feltétel nélküli abban az értelemben, hogy amikor a CPU eléri, akkor feltétel nélkül végrehajtja. Nincs helye belső kérdésnek: „Tenni vagy nem tenni?" Ezzel ellentétben a predikációs architektúrákban az utasítások feltételt tartalmaznak (predikátum), amely megmondja, hogy végre kell-e hajtani az utasítást, vagy sem. Ez a paradigma, amely elmozdulást jelent a feltétel nélküli utasításoktól a predikációs utasítások felé, lehetővé teszi, hogy (sok) feltételes elágazástól megszabaduljunk. Ahelyett hogy két, feltétel nélküli utasítássorozat között kellene választani, valamennyi utasítást egyetlen predikációs utasítássorozatba fésülnek össze, különböző predikátumot alkalmazva különböző utasításokban.

A predikáció működésének bemutatását kezdjük egy egyszerű példával, amelyet az 5.51. ábra mutat. Ez a **feltételes végrehajtás** (feltételes utasítás), a predikáció előd-je. Az 5.51. (a) ábra az if utasítást mutatja, a (b) ábrán pedig ennek három utasítássá való lefordítása látható: összehasonlítás, feltételes elágazás és mozgató utasítás.

Az 5.51. (c) ábrán megszabadultunk a feltételes elágazástól, bevezetve egy új utasítást, a feltételes mozgatást. Ez megvizsgálja az R1 regiszter tartalmát, ha 0, akkor R3-at átmásolja R2-be, ha nem 0, akkor nem tesz semmit.

Ha egyszer van olyan utasításunk, amely adatot másol, ha egy meghatározott regiszter 0, akkor csak egy lépésre van az az utasítás, amely akkor másol, ha valamely regiszter nem 0, mondjuk a CMOVN utasítás. Képzeljünk el egy if utasítást, amely több értékadó utasítást tartalmaz a then ágban és ugyancsak több értékadó utasítást az else ágban is. Az egész összetett utasítás átalakítható olyan kóddá, amely beállít egy regisztert 0-ra, ha a feltétel nem teljesül, egy másikat pedig 1-re, ha a feltétel teljesül. A regiszterek beállítása után a then ág utasításait CMOVN utasítások, az else ág utasításait pedig CMOVZ utasítások sorozatává alakíthatjuk.

```
if (R1 == 0)          CMP R1,0          CMOVZ R2,R3,R1
  R2 = R3;            BNE L1
                      MOV R2,R3
                L1:
(a)                   (b)               (c)
```

5.51. ábra. *(a) if utasítás. (b) Általános assembly kód az (a) esetre. (c) Feltételes utasítás*

Mindezeket az utasításokat, a regiszterek beállítását, a CMOVN és CMOVZ utasításokat egyetlen feltételes elágazást nem tartalmazó blokkba foglalhatjuk. Sőt, az utasítások sorrendje még át is alakítható akár a fordító által (beleértve azt is, hogy az értékadó utasítást a tesztelő elé teszi), akár futási időben. Az egyetlen, amit biztosítani kell, hogy a feltételt akkor már ismerni kell, amikor az utasítás (közel a cső végéhez) a tényleges végrehajtásba kezd. Erre mutat egy egyszerű példát az 5.52. ábra.

```
if (R1 == 0) {        CMP R1,0          CMOVZ R2,R3,R1
  R2 = R3;            BNE L1            CMOVZ R4,R5,R1
  R4 = R5;            MOV R2,R3         CMOVN R6,R7,R1
} else {              MOV R4,R5         CMOVN R8,R9,R1
  R6 = R7;            BR L2
  R8 = R9;      L1:   MOV R6,R7
}                     MOV R8,R9
              L2:
  (a)                 (b)               (c)
```

5.52. ábra. *(a) if utasítás. (b) Általános assembly kód az (a) esetre. (c) Feltételes végrehajtás*

Habár csak nagyon egyszerű példát mutattunk a feltételes utasításra (valójában a Pentium 4-ből véve), az IA–64 minden utasítása predikátumos. Ez azt jelenti, hogy minden utasítás végrehajtása feltételessé tehető. Az extra 6 bites mező az utasításban a 64 darab 1 bites predikátumregiszterből egy előre kiválasztottra hivatkozik. Így az if utasítást olyan kóddá fordítják, amely egy predikátumregisztert 1-re állít, ha a feltétel teljesül, és 0-ra, ha nem teljesül. Egyidejűleg és automatikusan beállítja egy másik predikátumregiszter tartalmát a feltétel inverze szerint. Predikációt használva az if utasítás then ága és else ága egy-egy gépi utasítássorozatba foglalódik, az előző a predikátumot, az utóbbi pedig annak inverzét alkalmazza.

Az 5.53. ábra mutatja a feltételes elágazás kiküszöbölésének az alapötletét predikáció alkalmazásával. A CMPEQ utasítás összehasonlít két regisztert, és beállítja a P4 predikátumregiszter értékét 1-re, ha egyenlők, és 0-ra, ha nem egyenlők. Beállítja a regiszter párját is, mondjuk, a P5 regisztert a feltétel inverzére. Ezután a then ág és az else ág következik egymás után, mindegyik feltételes valamelyik regiszterre (csúcsos zárójelek között vannak megadva). Itt tetszőleges kód megadható, feltéve, hogy minden utasítás megfelelően van predikálva.

Az IA–64 ezt az elvet a végsőkig viszi azáltal, hogy minden összehasonlító utasítás beállíthat predikátumregisztert, és minden más utasítás lehet predikációs valamely predikátumregiszterre hivatkozva. A predikációs utasítások sorban bekerülhetnek a végrehajtási csőbe, nincs elakadás, nincs probléma. Ez az, amiért nagyon hasznos a predikáció.

```
if (R1 == R2)                CMP R1,R2          CMPEQ R1,R2,P4
   R3 = R4 + R5;             BNE L1             <P4>ADD R3,R4,R5
else                         MOV R3,R4          <P5>SUB R6,R4,R5
   R6 = R4 - R5              ADD R3,R5
                             BR L2
                        L1:  MOV R6,R4
                             SUB R6,R5
                        L2:
   (a)                       (b)                (c)
```

5.53. ábra. *(a) if utasítás. (b) Általános assembly kód az (a) esetre. (c) Predikációs végrehajtás*

Az IA–64 esetén a predikáció valójában úgy működik, hogy minden utasítás ténylegesen végrehajtódik. A végrehajtási cső legvége felé, amikor aktuálissá válik az utasítás befejezése, elvégződik a teszt, hogy a predikátum igaz-e. Ha igaz, akkor az utasítás normálisan fejeződik be, az eredmény bekerül a célregiszterbe. Ha a predikátum hamis, akkor nincs visszaírás, az utasításnak nincs hatása. A predikációt részletesen tanulmányozza Dulong könyve (Dulong, 1998).

5.8.6. Spekulatív betöltés

Az IA–64 egy másik tulajdonsága, amely felgyorsítja a végrehajtást, a spekulatív betöltés. Ha egy LOAD utasítás spekulatív, és sikertelen a végrehajtása, akkor ahelyett hogy megszakítást okozna, beállít egy, a regiszterhez tartozó bitet, jelezve, hogy annak tartalma érvénytelen. Ez az oka a mérgezésbit bevezetésének, amelyet a 4. fejezetben tárgyaltunk. Ha később a mérgezett regisztert akarjuk használni, akkor a megszakítás bekövetkezik, egyébként nem lesz megszakítás.

A spekulatív betöltésnek az a normális használata, hogy a fordító előrébb hozza a LOAD utasításokat, mint ahogy azokra szükség lenne. Mivel előbb elkezdődnek, azelőtt be is fejeződhetnek, ahogy az eredményre szükség lenne. Ahol a fordító a betöltés eredményét használni szeretné, kiad egy CHECK utasítást. Ha az érték a regiszterben van, akkor üres utasításként működik, és a végrehajtás azonnal folytatódik. Ha az érték nincs ott, akkor a felfüggesztett megszakítás bekövetkezik.

Összefoglalva, az IA–64 architektúrát megvalósító gép gyorsaságát több tényező eredményezi. Alapvetően egy modern végrehajtási cső, töltő/tároló architektúra és háromcímes RISC-motor. Ezenfelül az IA–64 modell az explicit párhuzamosításra is, amely megköveteli a fordítótól, hogy kiderítse, mely utasítások végezhetők párhuzamosan konfliktus nélkül, és ezeket egy kötegbe csoportosítsa. Ezáltal a CPU vakon ütemezheti az egy kötegben lévő utasításokat, nem kell hosszasan gondolkodnia. Továbbá a predikáció lehetővé teszi, hogy az if utasítás mindkét ágában lévő utasításokat egyetlen sorozattá olvasszuk, kiküszöbölve a feltételes elágazást. Végül a spekulatív betöltés lehetővé teszi, hogy operandusokat előre betöltsünk, nem szenvedve büntetést, ha később kiderül, hogy nincs rá szükség.

Az Itanium 2-ről és architektúrájáról bővebben lásd (McNairy és Solts, 2003; Rusu és társai 2004).

5.9. Összefoglalás

Az utasításrendszer-architektúra szintje az, amelyet a legtöbb ember a „gépi nyelv" szintjének tekint. Ezen a szinten a számítógépnek néhány tíz megabájtnyi bájt vagy szószervezésű memóriája van, és vannak utasításai, például a MOVE, ADD és BEQ.

A legtöbb modern számítógép memóriája bájtok sorozataként van szervezve, 4-8 bájtos szavakba csoportosítva. Általában 8–32 regiszterük van, mindegyik mérete egy szó. Néhány gépen (például Pentium 4) a memóriában lévő szavakra hivatkozások nem korlátozódnak szóhatárra, míg más gépeknél igen (például UltraSPARC III).

Az utasításoknak általában egy, kettő vagy három operandusuk van, amelyek címzése lehet közvetlen, direkt, regiszteres, indexelt vagy más módú. Néhány gép sok és bonyolult címzési módot tartalmaz. Általában létezik utasításadatok mozgatására, monadikus és diadikus műveletekre, beleértve az aritmetikai-logikai műveleteket, van elágazás, eljáráshívás, ismétlés és néha B/K utasítás. A tipikus utasítás egy szót betölt a memóriából egy regiszterbe (vagy fordítva), összeadás, kivonás, szorzás vagy osztás műveletet végez két regiszter vagy egy regiszter és egy memóriaszó tartalmán, vagy összehasonlítja azokat. Nem szokatlan, ha a számítógép 200-nál is több utasítást tartalmaz. CISC gépeknek gyakran még ennél is több utasítása van.

A vezérlési folyamatot a 2. szinten olyan vezérlési alapműveletek valósítják meg, mint a feltételes elágazás, eljáráshívás, korutinhívás, csapda és megszakítás. Az elágazás azt jelenti, hogy egy utasítássorozat véget ér, és egy másik utasítássorozat végrehajtása elkezdődik. Az eljárás egy absztrakciós mechanizmust jelent, amely lehetővé teszi, hogy a program egyik részletét elkülönítsük, mint olyan egységet, amelyet a program több helyéről lehet hívni. A korutin lehetővé teszi két vezérlési szál szimultán működését. A csapdák arra használatosak, hogy kivételes eseményeket kezeljenek, mint például az aritmetikai túlcsordulás. A megszakítások biztosítják, hogy a B/K a fő számítással párhuzamosan menjen, azonnal jelezve a CPU-nak a B/K művelet befejeződését.

A Hanoi tornyai probléma egy érdekes kis probléma, elegáns rekurzív megoldással, amelyet részletesen vizsgáltunk.

Végül az IA–64 architektúra az EPIC modellt alkalmazza számításokra azzal a céllal, hogy kihasználja a programok párhuzamosíthatóságát. Predikációt és spekulatív betöltést is alkalmaz a sebesség növelésére. Mindent egybevetve, az IA–64 jelentős fejlődést reprezentál a Pentium 4-hez képest, bár a hangsúlyt a fordító által végzendő párhuzamosításra helyezi.

5.10. Feladatok

1. Egy kis endián gépen egy szó tartalma a 3 numerikus érték. Ezt a szót bájtról bájtra átmásoltuk egy nagy endián gépre, tehát a 0. bájtot a 0. bájtba és így tovább. Mi a numerikus értéke az adott szónak a nagy endián gépen?

2. A Pentium 4 utasítások hossza lehet akár páros, akár páratlan szám. Az UltraSPARC III minden utasítása szó egész számú többszöröse, tehát hossza páros szám. Mondjon egy érvet amellett, hogy a Pentium 4 sémája előnyös.
3. Tervezzen olyan műveleti kód kiterjesztést, amely 36 bites utasításhossz esetén lehetővé teszi a következő utasítások kódolását:
   7 db utasítás két 15 bites címmel és egy 3 bites regiszterszámmal
   500 db utasítás egy 15 bites címmel és egy 3 bites regiszterszámmal
   40 db utasítás cím és regiszter nélkül
4. Tegyük fel, hogy számítógépünknek 16 bites utasításai vannak 6 bites címrészszel. Az utasítások egy része kétcímes, a többi egycímes. Ha n db kétcímes utasítás van, mennyi az egycímesek maximális száma?
5. Van olyan műveletikód-kiterjesztés, amely lehetővé teszi az alábbi utasítások kódolását 12 biten? A regisztercím 3 bites.
   4 db utasítás 3 regiszterrel
   255 utasítás egy regiszterrel
   16 utasítás regiszter nélkül
6. Adottak az alábbi memóriatartalmak és egy egycímes, akkumulátoros gép. Milyen értékeket töltenek a felsorolt utasítások az akkumulátorba?
   20. szó értéke: 40
   30. szó értéke: 50
   40. szó értéke: 60
   50. szó értéke: 70
   a) LOAD IMMEDIATE 20
   b) LOAD DIRECT 20
   c) LOAD INDIRECT 20
   d) LOAD IMMEDIATE 30
   e) LOAD DIRECT 30
   f) LOAD INDIRECT 30
7. Hasonlítsa össze a 0, 1, 2, 3 címes gépeket azáltal, hogy programot ír mindegyik gépre az alábbi kifejezés kiértékelésére:
   $X = (A + B \times C)/(D - E \times F)$
   A következő utasítások használhatók az egyes gépeken:

0 címes	1 címes	2 címes	3 címes
PUSH M	LOAD M	MOV (X = Y)	MOV (X = Y)
POP M	STORE M	ADD (X = X + Y)	ADD (X = Y + Z)
ADD	ADD M	SUB (X = X – Y)	SUB (X = Y – Z)
SUB	SUB M	MUL (X = X × Y)	MUL (X = Y × Z)
MUL	MUL M	DIV (X = X/Y)	DIV (X = Y/Z)
DIV	DIV M		

   M 16 bites cím, X, Y, Z vagy 16 bites cím vagy 4 bites regisztercím lehet. A 0 címes gép veremmel dolgozik, az 1 címes akkumulátort használ, a másik két gépnek 16 db regisztere van, és minden memória- és regiszterkombináció alkalmazható a címzésben. A SUB X,Y utasítás Y-t vonja ki X-ből, a SUB X,Y,Z Z-t vonja ki Y-ból, és az eredmény X-be kerül. 8 bites műveleti kódot és olyan utasításhosszt feltételezve, amely 4 többszöröse, minimálisan hány bit kell az egyes gépek esetén X kiszámításához?

8. Adjon olyan címzési módszert, amely lehetővé teszi, hogy 64 darab tetszőleges cím, nem feltétlenül összefüggő tartományban, megadható legyen 6 bittel!
9. Mondjon az önmódosító kód ellen egy olyan érvet, amelyet a könyvben nem említettünk.
10. Konvertálja át fordított lengyel jelölésre az alábbi infix formulákat:
    a) A + B + C + D – E
    b) (A – B) × (C + D) + E
    c) (A × B) + (C × D) + E
    d) (A – B) × (((C – D × E)/F)/G) × H
11. Az alábbi, fordított lengyel jelölésben adott formulapárok közül melyek ekvivalensek matematikai értelemben?
    a) A B + C + és A B C + +
    b) A B – C – és A B C – –
    c) A B × C + és A B C + ×
12. Konvertálja infix formára az alábbi fordított lengyel jelölésű formulákat:
    a) A B – C + D ×
    b) A B/C D/+
    c) A B C D E + × ×/
    d) A B C D E × F/+ G – H/× +
13. Adjon három olyan formulát fordított lengyel jelölésben, amelyet zárójelek nélkül nem lehet infix formára konvertálni.
14. Konvertálja az alábbi infix logikai formulákat fordított lengyel jelölésre:
    a) (A AND B) OR C
    b) (A OR B) AND (A OR C)
    c) (A AND B) OR (C AND D)
15. Konvertálja az alábbi formulát fordított lengyel jelölésre, és adja meg a formulát kiértékelő IJVM-kódot:
    (5 × 2 + 7) – (4 / 2 + 1)
16. Az alábbi assembly nyelvű Pentium 4 utasítás regisztert tölt a memóriából:
    MOV REG, ADDR
    Ugyanez az utasítás UltraSPARC III-ban:
    LOAD ADDR, REG
    Miért különbözik a két utasításban az operandusok sorrendje?
17. Hány regisztere van annak a gépnek, amelynek utasításformáit az 5.25. ábra tartalmazza?
18. Az 5.25. ábrán a 23. bit szolgál az 1. és 2. utasításforma megkülönböztetésére. Nincs bit a 3. forma megkülönböztetésére. Honnan tudja a hardver, hogy a 3. formát kell használnia?
19. A programozásban gyakori, hogy a programnak meg kell határoznia, hogy adott X változó az $A - B$ intervallumhoz képest hol helyezkedik el. Ha lenne háromcímes utasítás az A, B és X operandusokkal, akkor hány feltételbitet kellene ennek az utasításnak beállítania?

20. A Pentium 4-nek van olyan feltételkódja, amelyet akkor állít be, ha a 3. bitről van átvitel. Mire jó ez?

21. Az UltraSPARC III-nek nincs olyan utasítása, amely 32 bites konstanst töltene regiszterbe. Helyette a SETHI és ADD utasítások használatosak. Létezik egynél több módszer 32 bites konstans betöltésére? Indokolja válaszát.

22. A barátja azzal a briliáns ötlettel állít be önhöz, hogy feltalálta a műveleti kód nélküli utasítást. Orvoshoz küldené, vagy elhinné neki?

23. A 8051-nek nincs 8 bitnél hosszabb eltolási értéke. Azt jelenti ez, hogy nem lehet 255-nél nagyobb memóriacímet elérni? Ha mégis, hogyan?

24. Az alábbi formájú tesztek nagyon gyakoriak a programozásban:

```
if (k == 0) ...
if (a > b) ...
if (k < 5) ...
```

Tervezzen olyan utasítást, amely hatékonyan megvalósítja ezeket a teszteket. Milyen mezők szerepelnek az utasításában?

25. Az 1001 0101 1100 0011 16 bites bináris számra mutassa meg az alábbi műveletek hatását:

a) jobbra léptetés 4 bittel, 0 kitöltéssel
b) előjeles jobbra léptetés 4 bittel
c) balra léptetés 4 bittel
d) balra forgatás 4 bittel
e) jobbra forgatás 4 bittel

26. Hogyan tudna nullázni egy memóriaszót olyan gépen, amelynek nincs CLR utasítása?

27. Számítsa ki az (*A* AND *B*) OR *C* logikai kifejezés értékét, ha

A = 1101 0000 1010 0011
B = 1111 1111 0000 1111
C = 0000 0000 0010 0000

28. Adjon olyan módszert, amellyel az *A* és *B* változók tartalmát megcseréli úgy, hogy nem használ harmadik változót vagy regisztert. *Tipp:* gondoljon az XOR kizáró vagy műveletre.

29. Tegyük fel, hogy számítógépünkön van olyan utasítás, amely regiszterből számot tölt regiszterbe, balra léptet és összead, és ezt a három utasítást kevesebb idő alatt hajtja végre, mint a szorzást. Milyen feltételek mellett hasznos ilyen módon számolni a „konstans × változó" értékét?

30. A különböző gépeknek eltérő az utasításssűrűsége (azon bájtok száma, amely adott számítás elvégzéséhez szükséges). Az alábbi három Java nyelvű kódrészletet fordítsa le Pentium 4, UltraSPARC III és IJVM assemblyre. Számítsa ki, hány bájt kell az egyes nyelvek esetében. Tegyük fel, hogy *i* és *j* lokális változó a memóriában, de egyébként optimális feltételekkel számolhat.

a) `i = 3;`
b) `i = j;`
c) `i = j - 1;`

31. A könyvben a ciklusszervező utasítást a for ciklusokra vettük. Tervezzen olyan utasítást, amely a legtöbb while (kezdőfeltételes) ismétlés megvalósítására kedvező.

32. Tegyük fel, hogy a szerzetesek Hanoiban 1 pere alatt tudnak egy korongot átrakni (nem kell sietniük, mert az ő végzettségükkel nehezen találnak munkát Hanoiban). Mennyi ideig tart nekik a teljes 64 korongos probléma megoldása? Fejezze ki az eredményt években.

33. A B/K eszközök miért küldik ki a sínre a megszakítási vektort? Lehetséges lenne a memóriában táblázatban tárolni?

34. A számítógép DMA-t használ mágneslemez olvasására. A lemezegység 64 szektort tartalmaz sávonként, a szektor 512 bájtos. A lemez forgási sebessége 16 ms. A sín 16 bit széles, átviteli sebessége pedig 500 ns. Az átlagos CPU-utasítás két sínciklust igényel. Mennyire lassítja le a DMA a CPU-t?

35. Miért rendelnek a megszakításkezelő rutinokhoz elsőbbségi értéket, amikor a közönséges eljárásokhoz meg nem?

36. Az IA–64 architektúra szokatlanul sok (128) regisztert tartalmaz. Összefüggésben van ez a predikációval? Ha igen, hogyan? Ha nem, mi az oka a sok regiszternek?

37. A könyvben a spekulatív LOAD utasítást vizsgáltuk. Azonban nem említettünk spekulatív STORE utasítást. Miért nem? Ezek alapjában véve hasonlók lehetnek a LOAD-dal, avagy más oka van annak, hogy nem említettük?

38. Ha két lokális számítógép-hálózatot kell összekötni, egy számítógépet, az ún. hidat (bridge) illesztik a két hálózat közé. Bármelyik hálózatban kibocsátott csomag megszakítást eredményez a híd gépen, hogy a gép döntsön a továbbításról. Tegyük fel, hogy 250 μs-ig (mikromásodpercig) tart egy csomag esetén a megszakítás kezelése, és a csomag megvizsgálása, de a továbbítást DMA hardver végzi, megszabadítva a CPU-t ettől. Ha minden csomag 1 KB méretű, mi a maximális adatátviteli sebesség a hálózatok között, amelyet a híd még kezelni tud adatvesztés nélkül?

39. Az 5.43. ábrán a keretmutató a nulladik lokális változóra mutat. Milyen információra van szükség az eljárásból való visszatérés megvalósításához?

40. Írjon olyan assembly nyelvű rutint, amely előjeles bináris számot ASCII kódúvá konvertál.

41. Írjon olyan assembly nyelvű rutint, amely infix formulát fordított lengyel jelölésre konvertál.

42. A Hanoi tornyai nem az egyetlen kedvenc példa rekurzív eljárásra. Egy másik népszerű feladat az $n!$, ahol $0! = 1$ és $n! = n(n-1)!$. Írjon rekurzív eljárást kedvenc assembly nyelvében $n!$ értékének kiszámítására.

43. Ha nincs meggyőződve arról, hogy a rekurzió nélkülözhetetlen, próbálja meg megoldani a Hanoi tornyai problémát rekurzió nélkül, nem használva vermet a rekurzió szimulálására. Vigyázat, valószínű, hogy nem találja meg a megoldást ezen az úton.

6. Az operációs rendszer gép szintje

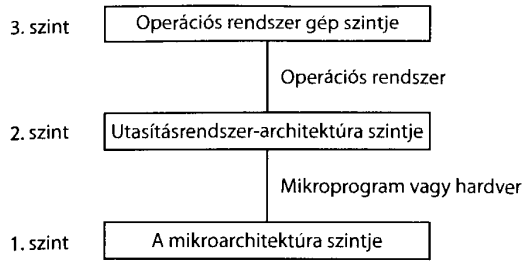
Könyvünk arról szól, hogy a modern számítógépek egymásra épülő szintek sorozataként foghatók fel, s minden szint további könnyítést nyújt az alatta lévőkhöz képest. Már láttuk a digitális logikai, a mikroarchitektúra és az utasításrendszer-architektúra szintjét. Itt az ideje, hogy egy szinttel feljebb, az operációs rendszer birodalmába lépjünk.

Az **operációs rendszer (operating system)** a programozó szempontjából olyan program, amely az ISA-szint fölött és alatt számos új utasítást és lehetőséget kínál. Az operációs rendszert általában szoftveres úton valósítják meg, de a mikroprogramokéhoz hasonló hardveres realizációnak sincs elméleti akadálya. Az általa megvalósított szintet röviden **OSM (Operating System Machine Level, operációs rendszer gép szintje)** néven fogjuk emlegetni. Ezt mutatja a 6.1. ábra.

Bár mind az ISA, mind az OSM absztrakt szintek abban az értelemben, hogy nem igazi hardveres szintek, van egy lényeges különbség közöttük. Az OSM-szintű utasítások az alkalmazói programozók rendelkezésére álló teljes utasításkészletet jelentik. Tartalmazzák majdnem az összes ISA-szintű utasítást, valamint az operációs rendszer által hozzáadott új utasításokat. Ezeket az új utasításokat **rendszerhívások**nak **(system calls)** szokás nevezni. Ténylegesen minden rendszerhívás az operációs rendszer valamely előre definiált szolgáltatását hívja meg. Tipikus rendszerhívás például az adatok olvasása fájlból. A rendszerhívások nevét kisbetűs Helvetica betűtípussal fogjuk írni.

Az OSM szint mindig interpretált. Ha egy felhasználói program OSM utasítást (például fájlból történő adatbeolvasást) végez, az operációs rendszer ezt az utasítást lépésenként hajtja végre, ugyanúgy, mint ahogy egy ADD utasítás is a mikroprogram lépésenkénti végrehajtását jelenti. Amikor azonban ISA-szintű utasítást hajt végre a program, ezt az operációs rendszer közreműködése nélkül, közvetlenül az alatta lévő mikroarchitektúra szintje végzi el.

Könyvünk csak igen rövid betekintést tud nyújtani az operációs rendszerek témájába, ezért figyelmünket három fontos területre összpontosítjuk. Az első a sok operációs rendszer által nyújtott virtuális memória technika, amellyel úgy tűnik, mintha a gépben a valóságosnál nagyobb memória lenne. A második az előző fejezetben vizsgált B/K utasításoknál magasabb szintű fogalom, a fájlszintű B/K.



6.1. ábra. *Az operációs rendszer gép szintjének elhelyezkedése*

A harmadik téma a párhuzamos feldolgozás: több processzus végrehajtása, kommunikációja és szinkronizációja.

A **processzus (process)** fontos fogalom, ezért a fejezet további részében részletesebben is tárgyaljuk. Egyelőre egy processzus úgy fogható fel, mint egy futó program és ennek összes állapotinformációja (memória, regiszterek, utasításszámláló, B/K állapot stb.). Az alapelvek általános vizsgálata után azt mutatjuk meg, hogyan alkalmazhatók két példaként választott gépünk operációs rendszerére, a Pentium 4-re (Windows XP) és az UltraSPARC III-ra (UNIX). Mivel a 8051-est rendszerint beágyazott rendszerekben használják, nincs is teljes operációs rendszere.

6.1. Virtuális memória

Az első számítógépek idejében a memória kicsi és drága volt. Az IBM 650, amely akkoriban (az 1950-es évek végén) a legjobb tudományos számítógép volt, csak 2000 szavas memóriát tartalmazott. Az első ALGOL fordítóprogramok egyikét olyan gépre készítették, amelynek csak 1024 szóból álló memóriája volt. Egy korai időosztásos rendszer egészen jól futott azon a PDP–1 gépen, amely az operációs rendszer és a felhasználói programok számára együttesen csupán 4096 18 bites szóból álló memóriával rendelkezett. A programozók akkoriban rengeteg időt töltöttek el programjaik „bepréselésével" a csöppnyi memóriába. Gyakran csupán azért kellett lassúbb algoritmusokat használni, mert a jobb algoritmusok túl nagyok voltak; programjaik nem fértek el a gép memóriájában.

A probléma hagyományos megoldása másodlagos memória, például lemez igénybevétele volt. A programozó olyan **átfedések**nek **(overlays)** nevezett kisebb részekre osztotta fel programját, amelyek külön-külön már elfértek a memóriában. A program futásakor először az első rész töltődött be és futott egy darabig. Amikor befejezte működését, beolvasta és elindította a következő részt és így tovább. A programozónak kellett gondoskodni a program átfedésekre darabolásáról, az egyes részek elhelyezéséről a másodlagos memóriában, a memória és a lemez közti mozgatásukról, és általában az átfedéses rendszer kezeléséről. A gép ehhez semmi segítséget nem adott.

Ez a megoldás sok éven át általánosan elterjedt volt, bár az átfedések kezelése igen sok munkával járt. Egy manchesteri angol kutatócsoport 1961-ben olyan módszert javasolt az átfedések automatikus kezelésére, melynél a programozónak nem is kell tudnia az egész rendszerről (Fotheringham, 1961). Ez a módszer, amely ma **virtuális memória (virtual memory)** néven ismert, azzal a nyilvánvaló előnnyel járt, hogy megszabadította a programozót egy halom unalmas adminisztrációtól. Először az 1960-as években alkalmazták több gépen, főleg a számítógépes rendszerek tervezését célzó kutatási projektekkel kapcsolatban. Az 1970-es évek elejére a virtuális memória már a legtöbb gépen elérhető volt. Ma már a csupán egyetlen lapkából álló gépeknek (így a Pentium 4-nek és az UltraSPARC IIInak) is fejlett virtuálismemória-kezelő rendszere van. Ezeket e fejezetben később fogjuk vizsgálni.

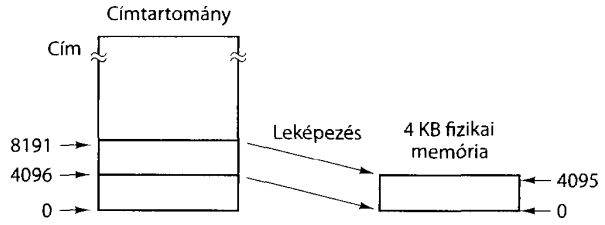
6.1.1. Lapozás

A manchesteri csoport által kifejlesztett új ötlet a címtartomány és a memóriarekeszek fogalmának szétválasztása volt. Példaként vegyünk egy akkoriban tipikusnak számító gépet, melynek, mondjuk, 4096 szavas memóriája, utasításaiban pedig 16 bites címrésze volt. A gépen futó programok 65 536 szót tudtak megcímezni, mivel 65 536 (2^{16}) 16 bites cím létezik, amelyek mindegyike a memória különböző szavainak felel meg. Figyeljük meg, hogy a megcímezhető szavak száma a címben lévő bitek számától függ, és semmi köze a ténylegesen rendelkezésünkre álló szavak számához. Ennek a gépnek a **címtartomány**a **(address space)** tehát a 0, 1, 2, ..., 65 535 számokból áll, mivel ez a lehetséges címek halmaza. A gépnek persze könnyen lehet ennél a 65 535-nél kevesebb szóból álló memóriája.

A virtuális memória felfedezése előtt meg kellett volna különböztetni a 4096 alatti és az ennél nagyobb címeket. Bár ritkán fejtették ki ilyen részletességgel, ezt a két részt a hasznos, illetve a haszontalan címtartománynak tekintették (a 4096 fölötti címek azért voltak haszontalanok, mert nem tartozott hozzájuk valódi memória). Azért nem különböztették meg a címtartományt és a memóriacímeket, mert a hardver egy-egy értelmű megfeleltetést hozott létre közöttük.

A címtartomány és a memóriacímek elkülönítésének elve a következő. Bármely időpillanatban közvetlenül 4096 szó érhető el a memóriából, de ezek nem feltétlenül a 0 és 4095 közti memóriacímeknek felelnek meg. „Megmondjuk" a gépnek, hogy mostantól kezdve például a 4096 címre való hivatkozás esetén a memória 0 című szavát kell használni. Ha a 4097 címre hivatkozunk, ez jelentse az 1 memóriacímet, a 8191 a memória 4095 címén található szót stb. Más szóval a 6.2. ábrán látható leképezést definiáljuk a címtartományból a valódi memóriacímek halmazára.

Ha a címtartományt a tényleges memóriarekeszekre az ábrán megadott séma szerint képezzük le, azt mondhatjuk, hogy egy 4 KB-os, virtuális memória nélküli gép esetén fix leképezésünk van a 0 és 4095 közti címek, valamint a memória 4096 szava között. Ezek után feltehetjük azt az érdekes kérdést, hogy mi történik akkor, ha a program végrehajtása során 8192 és 12287 közti címekre ágazik el. Virtuális memória nélküli gépen ez futási hibát okozna, melynek hatására „Nem létező memóriacímre való hivatkozás" vagy hasonló durva hibaüzenet után befejeződne a program végrehajtása. Virtuális memóriával rendelkező gépen a következő lépések hajtódnának végre:



6.2. ábra. *A 4096 és 8191 közti virtuális címekhez a 0 és 4095 közti memóriacímeket hozzárendelő leképezés*

1. A memória tartalmának lemezre mentése.
2. A 8192 és 12287 közti szavak megkeresése a lemezen.
3. A 8192 és 12287 közti szavak betöltése a memóriába.
4. A memóriatérkép megváltoztatása: a 8192 és a 12287 közti címek leképezése a 0 és 4095 közti memóriarekeszekre.
5. A végrehajtás folytatása – mintha mi sem történt volna.

Az átfedések automatikus kezelésének ezt a technikáját **lapozás**nak **(paging)**, a lemezről beolvasott programrészeket pedig **lapok**nak **(pages)** nevezzük.

A címtartományhoz tartozó címeknek a valódi memóriacímekre való leképezését kifinomultabban is kezelhetjük. A nyomaték kedvéért azokat a címeket, amelyekre a program hivatkozhat, **virtuális címtartomány**nak **(virtual address space)**, míg a tényleges, „bedrótozott" (fizikai) memóriacímeket **fizikai címtartomány**nak **(physical address space)** fogjuk nevezni. A **memóriatérkép (memory map)** vagy **laptábla (page map)** az egyes virtuális címeknek megfelelő fizikai címeket határozza meg. Feltesszük, hogy a lemezen van elég hely a teljes virtuális címtartomány (vagy legalábbis a használatban lévő része) számára.

A programok úgy írhatók meg, mintha a teljes virtuális címtartomány számára elegendő memória lenne a gépben, még ha a valóságban nem is ez a helyzet. A program betölthet tetszőleges virtuális címről, írhat oda, a végrehajtás elágazhat a virtuális címtartomány tetszőleges helyén található utasításra, tekintet nélkül arra, hogy nincs is elég fizikai memória. Valójában a programozónak nem is kell tudnia a virtuális memória létezéséről. Egyszerűen úgy látja, mintha a gépben rengeteg memória lenne.

Igen fontos ennek szembeállítása a később ismertetendő szegmentálással, ahol a programozónak tekintettel kell lennie a szegmensek létezésére. Még egyszer hangsúlyozzuk: a lapozás azt az illúziót nyújtja a programozónak, mintha a virtuális címtartománnyal megegyező méretű nagy, folytonos, lineáris memória lenne. A valóságban a rendelkezésre álló fizikai memória lehet nagyobb is, kisebb is

a virtuális címtartománynál. A programok nem tudják érzékelni, hogy lapozással szimulált nagy memóriával dolgoznak (kivéve, ha időmérő teszteket futtatnak). Valahányszor hivatkozunk egy címre, úgy tűnik, mintha az adatot vagy az utasítást tartalmazó valódi szavakkal dolgoznánk. A lapozási mechanizmust **transzparens**nek hívjuk, hiszen a programozó úgy írhat kódot, mintha nem is létezne a lapozás.

Végül is egyáltalán nem újdonság az az ötlet, hogy a programozó anélkül használhasson valamely nem létező sajátosságot, hogy törődnie kellene a működési módjával. Az ISA-szintű utasításkészlet gyakran tartalmaz MUL utasítást, holott csak az alatta lévő mikroarchitektúrában van szorzóegység. A mikrokód szintje teszi lehetővé azt az illúziót, hogy a gép tud szorozni. Az operációs rendszer által szolgáltatott virtuális gép hasonlóan tud olyan illúziót nyújtani, hogy minden virtuális cím mögött valódi memória van, még ha nem is ez az igazság. Csak az operációs rendszerek íróinak (és az ezeket tanulmányozó hallgatóknak) kell tudnia, hogyan tartható fönt ez az illúzió.

6.1.2. A lapozás megvalósítása

A virtuális memória egyik alapkövetelménye az egész program és az összes adat tárolására szolgáló lemezegység. Egyszerűbb úgy elképzelni, hogy a program lemezen lévő változata az eredeti, és a memóriába időközönként betöltött darabjai a másolatok, mint fordítva. Természetesen fontos az eredeti frissítése; ha változott a memóriában lévő másolat, ennek tükröződnie kell az eredeti példányon is.

A virtuális címtartományt azonos méretű lapokra szokás felosztani. A ma szokásos lapméretek 512 és 64 KB bájt közé esnek, bár alkalmanként használnak akár 4 MB-os lapokat is. A lapméret mindig 2 valamely hatványa. A fizikai címtartományt hasonlóan osztják fel. A darabok mérete a lapmérettel megegyező, így mindegyikük pontosan egy lap tárolására alkalmas. Ezeket a memóriadarabokat, melyekbe lapokat töltünk be, **lapkeretek**nek **(page frames)** fogjuk nevezni. A 6.2. ábrán a memória csak egy lapkeretet tartalmazott. A gyakorlatban megvalósított rendszerekben általában több ezer lapkeret van.

A 6.3. (a) ábra a virtuális címtartomány első 64 KB-os részének egyik lehetséges felosztását mutatja 4 KB-os lapokra. (Megjegyezzük, hogy 64 KB, illetve 4 KB címről beszéltünk. Egy cím mindig egy cella címe, és a cella lehet bájt, de ugyanígy lehet szó is olyan gépeken, ahol az egymás utáni szavak egymásra következő címen találhatók.) A 6.3. ábrán látható virtuális memóriát annyi elemű laptáblával lehetne megvalósítani, ahány lapból a virtuális címtartomány áll. Az egyszerűség kedvéért csak az első 16 elemet tüntettük fel. Ha a program a virtuális címtartomány első 64 KB-os részébe eső szóra próbál hivatkozni, hogy onnan utasításokat vagy adatokat töltsön be, illetve írjon vissza, először egy 0 és 65 532 közé eső virtuális címet kell generálnia. Eközben alkalmazhat indexelést, indirekt címzést és minden más szokásos módszert.

A 6.3. (b) ábrán 8 darab 4 KB méretű lapkeretet tartalmazó fizikai memóriát látunk. A 32 KB-os memóriakorlát származhat abból, hogy (1) nincs is több a gépben (egy mikrosütőbe vagy mosógépbe beépített processzornak talán elég is ennyi); vagy (2) a többi memória más programoknak van kiosztva.

Lap — Virtuális címek

15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

A fizikai memória alsó 32 KB-os része

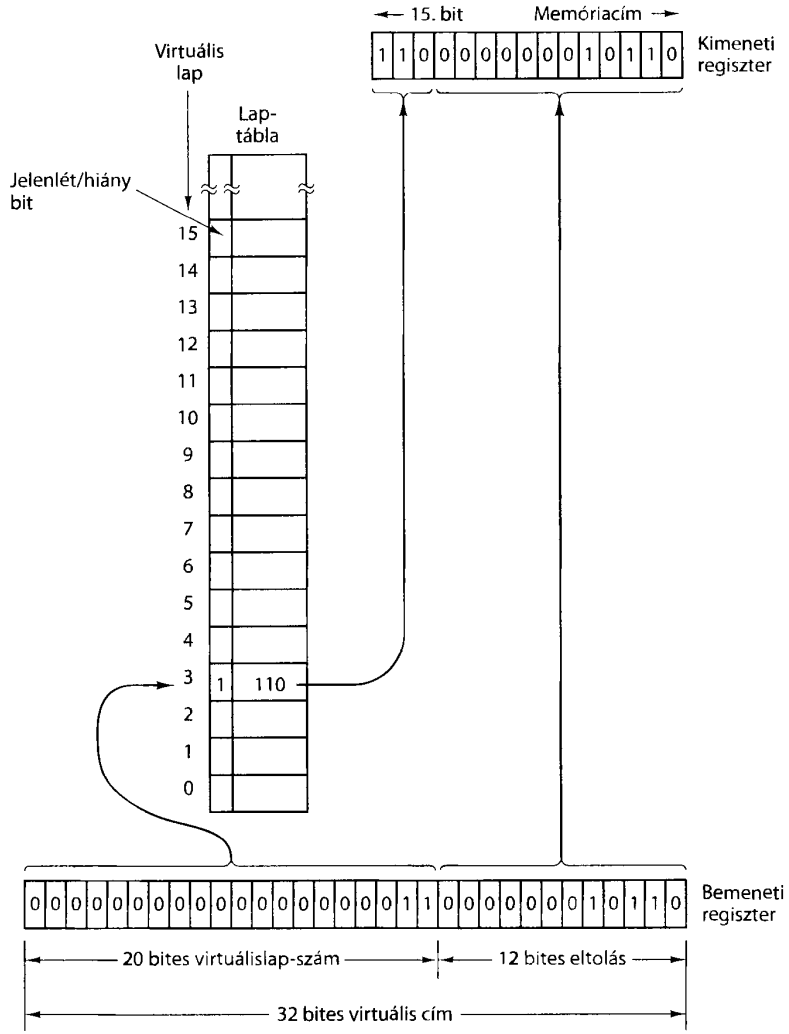
Lap-keret	Fizikai címek
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

6.3. ábra. *(a) A virtuális címtartomány első 64 KB-os része, melyet 16 darab 4 KB méretű lapra osztottunk fel. (b) A 32 KB-os fizikai memória nyolc darab szintén 4 KB-os lapkerettel*

Vizsgáljuk meg, hogyan képezhető le valamely 32 bites virtuális cím a fizikai memória egy címére. Elvégre a memória csak ilyen címekkel tud dolgozni, tehát ezt kell valahogy megadnunk. Minden virtuális memóriával ellátott számítógép tartalmaz a virtuálisról fizikai címre való leképezést megvalósító eszközt. Ennek neve **MMU (Memory Management Unit, memóriakezelő egység)**. Elhelyezkedhet magán a CPU lapkán vagy külön lapkán, mely utóbbi szorosan együttműködik a CPU-val. Az általunk példaként felhozott MMU a 32 bites virtuális címeket 15 bites fizikai címekre képezi le; ehhez 32 bites bemeneti és 15 bites kimeneti regiszterre van szüksége.

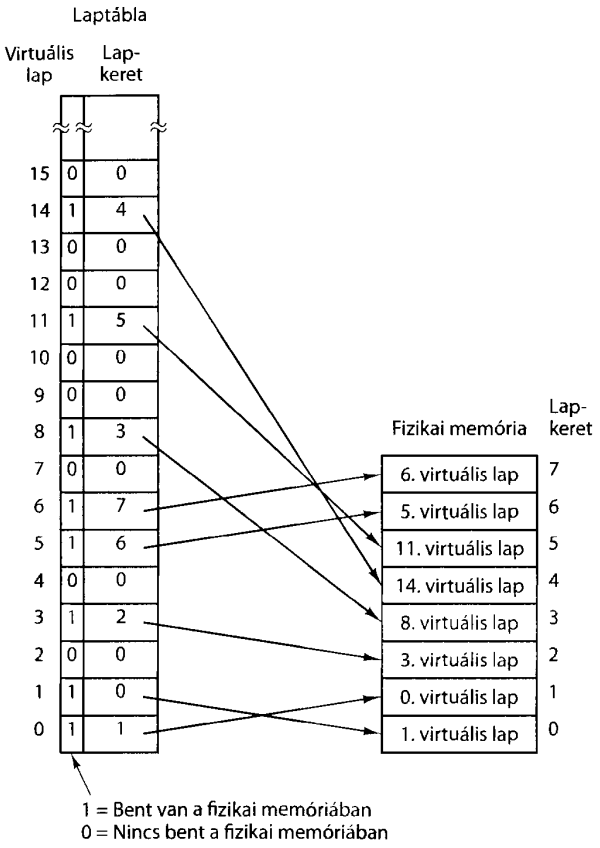
Az MMU működésének megértéséhez tekintsük a 6.4. ábrán látható példát. A megkapott 32 bites virtuális címet az MMU 20 bites virtuálislap-számra és a lapon belüli 12 bites eltolásra (offset) bontja fel (a 20 bit a 4 KB-os lapméretből adódik). A virtuálislap-számot indexként használva keresi ki a laptáblából a megfelelő bejegyzést. A 6.4. ábrán a virtuálislap-szám 3, így a laptábla 3. elemét veszi.



6.4. ábra. *A tényleges memóriacím meghatározása a virtuális cím alapján*

Először azt vizsgálja meg az MMU, hogy a hivatkozott lap nincs-e jelenleg a memóriában. Elvégre a 2^{20} virtuális lap és a mindössze nyolc lapkeret miatt nem lehet az összes lap egyszerre a memóriában. Az MMU a laptábla-bejegyzésben található **jelenlét/hiány bit (present/absent bit)** segítségével végzi az ellenőrzést. Példánkban a bit értéke 1, vagyis a lap most is a memóriában van.

A következő lépésben a kiválasztott bejegyzésben szereplő lapkeretértéket (esetünkben 6-ot) átmásolja a 15 bites kimeneti regiszter felső 3 bitjére. Azért van



6.5. ábra. *Az első 16 virtuális lap egy lehetséges leképezése a 8 lapkeretből álló memóriára*

szükség 3 bitre, mert a fizikai memória a példánkban 8 lapkeretből áll. Mint az ábrából is látható, ezzel párhuzamosan a virtuális cím alsó 12 bitje (a lapon belüli eltolás) átmásolódik a kimeneti regiszter alsó 12 bitjére. Ezt a 15 bites címet küldi tovább a gyorsítótárhoz vagy a memóriához.

A 6.5. ábra a virtuális lapok fizikai lapkeretekre való egyik lehetséges leképezését mutatja. A 0. virtuális lap az 1. lapkeretben van. Az 1. virtuális lap a 0. lapkeretben található. A 2. virtuális lap nincs a memóriában. A 3. virtuális lap a 2. lapkeretben van. A 4. virtuális lap nincs a memóriában. Az 5. virtuális lap a 6. lapkeretben található stb.

6.1.3. A kérésre lapozás és a munkahalmaz modell

Az előző fejtegetések során feltettük, hogy a hivatkozott virtuális lap a memóriában van. Ez a feltevés azonban nem mindig teljesül, mert a memóriában nincs elég hely az összes virtuális lap számára. Az olyan hivatkozás, amely nem a memóriában lévő lapon található címre vonatkozik, **laphiányt (page fault)** okoz. Laphiány felléptekor az operációs rendszernek be kell olvasnia lemezről a kért lapot, be kell írnia új fizikai helyét a laptáblába, és meg kell ismételnie a hibát okozó utasítást.

Virtuális memóriával rendelkező gépen akkor is elindíthatunk egy programot, ha egyetlen része sincs a memóriában. Csupán úgy kell beállítani a laptáblát, hogy jelezze, a programhoz tartozó lapok közül egyik sincs a memóriában, mind a másodlagos tárolón helyezkedik el. Amikor a CPU megpróbálja az első utasítást betölteni, rögtön laphiány lép fel, melynek hatására az első utasítást tartalmazó lap betöltődik a memóriába, és ez az információ bekerül a laptáblába. Ezután elkezdődhet az első utasítás végrehajtása.

Ha az első utasítás 2 címet tartalmaz, s ezek a már betöltöttől különböző két további lapon találhatók, két újabb laphiány lép fel, és két további lap töltődik be mielőtt az utasítás végrehajtódhatna. A következő utasítás további laphiányokat okozhat stb.

A virtuális memória használatának ezt a módszerét **kérésre lapozás**nak **(demand paging)** nevezzük, a jól ismert gyereketetési algoritmus analógiájára, melynél ha sír a baba, megetetjük – szemben a precíz napirend szerinti etetéssel. A kérésre lapozásnál a lapok nem előzetesen, hanem csak akkor töltődnek be a memóriába, ha ténylegesen kérik.

Csak a program elindulásakor lényeges, hogy a kérésre lapozást használjuk-e. Ha már futott valameddig, a szükséges lapok általában bent lesznek a memóriában. Ha a gépen időosztást használunk, és a processzusok körülbelül 100 msec futás után cserélődnek, akkor minden program rengetegszer lesz elindítva a futása során. Mivel minden programhoz egyedi memóriatérkép tartozik, s programváltáskor ez is változik, időosztásos rendszerekben kritikussá válhat a helyzet.

Az alternatív megközelítés azon a megfigyelésen alapul, hogy a legtöbb program nem egyenletesen hivatkozik a címtartományára, hanem néhány lap körül „sűrűsödnek össze" a hivatkozások. Ezt a fogalmat **lokalitás elv**nek **(locality principle)** nevezzük Egy memóriahivatkozás utasítást tölthet be, adatokat olvashat vagy tárolhat. Bármely *t* időpillanatban tekinthetjük a legutóbbi *k* memóriahivatkozásban szereplő lapok halmazát. Denning (1968) nyomán ezt **munkahalmaz (working set)** néven fogjuk emlegetni.

Mivel a munkahalmaz időben lassan változik, a legutóbbi megállításkori halmaz alapján elfogadható becslés adható arra, hogy a program újabb indításakor mely lapokra lesz szükség. Ezek a lapok már a program indítása előtt betölthetők (feltéve, hogy befférnek a memóriába).

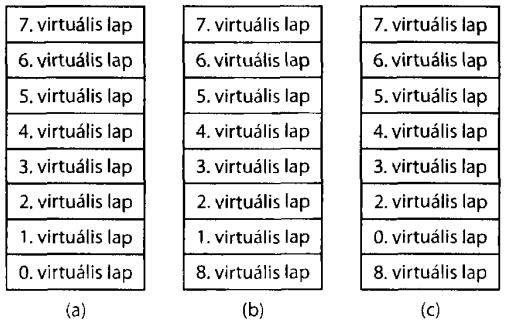
6.1.4. Lapcserélő eljárások

Ideális esetben a program által gyakran és aktívan használt lapok halmaza, a munkahalmaz, a memóriában tartható lenne, s ezzel csökkenthetnénk a laphiányok számát. Ám a programozók csak ritkán tudják, hogy mely lapok tartoznak a munkahalmazhoz, ezt az operációs rendszernek kell (dinamikusan) felfedeznie. Ha a program olyan lapra hivatkozik, amely nincs a memóriában, a szükséges lapot be kell tölteni lemezről. Legtöbbször azonban először „helyet kell csinálni", bizonyos lapokat vissza kell írni a lemezre. Szükségünk van tehát az eltávolítandó lapokat megadó algoritmusra.

Valószínűleg nem túl jó ötlet az eltávolítandó lap véletlenszerű kiválasztása. Ha a laphiányt okozó utasítást tartalmazó lapra esne a választás, a következő utasítás betöltésének megkísérlése azonnal újabb laphiányt eredményezne. A legtöbb operációs rendszer megpróbálja megjósolni, hogy melyek a memóriában lévő „leghaszontalanabb" lapok abban az értelemben, hogy eltávolításuk a lehető legkisebb mértékben zavarná a futó programot. Ennek egyik módja az lehet, hogy minden lapra megjósolja a várható legközelebbi hivatkozás időpontját, s azt a lapot távolítja el, amelynél ez a legtávolabbi jövőbeli időpont. Más szóval ahelyett, hogy egy nemsokára szükséges lapot mentenénk ki, próbáljunk olyat választani, amely még sokáig nem kell.

Az egyik népszerű algoritmus a legrégebben nem használt lapot veszi, mivel nagy a valószínűsége, hogy ez nincs benne a munkahalmazban. A módszer neve **LRU (Least Recently Used, legrégebben használt)** algoritmus. Általában jól használható, bár vannak olyan patologikus esetek, mint az alább ismertetendő példa, amikor csúnyán kudarcot vall.

Képzeljük el azt a programot, amely egy 9 lapon elhelyezkedő hosszú ciklust hajt végre olyan gépen, amelynek memóriájában csak 8 lap számára van hely. A 6.6. (a) ábra a memória helyzetét mutatja, miután a program futása a 7. laphoz ért. Végül olyan utasításhoz érünk, amely a 8. lapról próbál utasítást betölteni. Ez laphiányt eredményez. El kell dönteni, hogy melyik lapot távolítsuk el. Az LRU



6.6. ábra. *Az LRU algoritmus csődje*

algoritmus a 0. virtuális lapot választja, mivel ezt használtuk legrégebben. A 0. virtuális lap elmozdítása és a 8. virtuális lap betöltése után a 6.6. (b) ábrán látható helyzet alakul ki.

A 8. virtuális lapon található utasítások végrehajtása után a program visszatér a ciklus elejére, a 0. virtuális lapra. Ez a lépés újabb laphiányt okoz. Az előbb kirakott 0. virtuális lapot vissza kell hozni a memóriába. Az LRU algoritmus ezelőtt az 1. lapot írja ki a lemezre, s így a 6.6. (c) ábrán látható helyzet áll elő. A program kis ideig még a 0. lapon folytatódik. Azután az 1. lapról próbál utasítást betölteni, amiből laphiány lesz. Vissza kell hozni az 1. lapot, és kikerül a 2.

Ugye az már nyilvánvaló, hogy az LRU algoritmus minden alkalommal a legrosszabb lehetőséget választja? (Hasonló körülmények között más algoritmusok is hibázhatnak.) Ha azonban a rendelkezésre álló memória mérete nagyobb a munkahalmazénál, az LRU várhatóan minimalizálni fogja a laphiányok számát.

A másik lapcserélő algoritmus a **FIFO (First-In First-Out, először be, először ki)**. A FIFO módszer a legrégebben betöltött lapot távolítja el, függetlenül attól, hogy mikor hivatkoztak rá utoljára. Minden lapkerethez egy számláló tartozik, melyet kezdetben 0-ra állítunk. Az éppen a memóriában tartózkodó lapokhoz tartozó számlálók értékét minden laphiány után eggyel növeljük, az újonnan betöltött lapét pedig 0-val inicializáljuk. Ha lapot kell eltávolítani, azt választjuk, amelynek számlálója a legnagyobb értéken áll. Mivel ennek a számlálója a legnagyobb, ez a lap „élte túl" a legtöbb laphiányt. Ez azt jelenti, hogy korábban lett betöltve a memóriában lévő összes többi lapnál, és így (remélhetőleg) komoly esélye van annak, hogy nem is lesz rá szükség a továbbiakban.

Ha a munkahalmaz nagyobb a rendelkezésünkre álló lapkeretek számánál, a laphiányok gyakoriak lesznek, s egyetlen algoritmus sem adhat jó eredményt, hacsak nincsen jóstehetséggel megáldva. Azt a jelenséget, amikor egy program gyakran, szinte folyamatosan laphiányokat generál, **vergődés**nek **(thrashing)** nevezzük. Talán fölösleges megemlítenünk, hogy nemkívánatos jelenségről van szó. Az olyan programokkal nincsen sok gond, amelyek nagy virtuális címtartományt használnak, de munkahalmazuk kicsi, időben lassan változó, és mindig elfér a rendelkezésre álló memóriában. Ez a megállapítás még akkor is igaz, ha a program futása során több százszor annyi virtuális memóriaszót használ fel, mint a gép memóriájában lévő szavak száma.

Ha egy eltávolítandó lapon nem módosítottunk semmit a beolvasás óta (ami nagyon valószínű, ha nem adatokat, hanem programkódot tartalmaz), szükségtelen visszaírni a lemezre, hiszen ott már létezik egy pontos másolata. Ha a beolvasás óta módosítottuk, a lemezen lévő példány már nem pontos, ezért ki kell írni a mostanit.

Amennyiben meg tudjuk mondani valahogyan, hogy a beolvasás óta a lap változatlan maradt (tiszta, clean) vagy pedig írtunk rá (szennyezett, dirty), elkerülhető a tiszta lapok újraírása. Ezzel sok időt spórolunk meg. Sok gépen az MMU-ban minden laphoz tartozik egy bit, amelyet betöltéskor 0-ra állít, és a hardver vagy a mikroprogram 1-re módosít, ha a lap szennyezetté válik (ráírtunk valamit). Az operációs rendszer ennek a bitnek a vizsgálatával el tudja dönteni, hogy a lap tiszta vagy szennyezett, tehát vissza kell-e másolni a lemezre lapcsere esetén.

6.1.5. Lapméret és elaprózódás

Ha véletlenül úgy adódik, hogy a felhasználó programja és adatai valahány lapon pontosan elférnek, a memóriába való betöltésükkor nem pocsékolunk el helyet. Az ellenkező esetben az utolsó lapon valamennyi hely kihasználatlanul marad. Ha egy 4096 bájtos lapméretű gépen például a program és az adatok összesen 26 000 bájtot igényelnek, akkor megtelik az első 6 lap ($6 \times 4096 = 24\,576$ bájt), és az utolsó lapra marad még $26\,000 - 24\,576 = 1424$ bájt. Mivel laponként 4096 bájt számára van hely, elpocsékoltunk 2672 bájtot. Ahányszor csak betöltjük a hetedik lapot a memóriába, ezek a bájtok teljesen haszontalanul foglalnak le helyet. Az elvesztegetett bájtokkal kapcsolatos fenti problémára **belső elaprózódás (internal fragmentation)** néven fogunk hivatkozni (a kihasználatlan hely valamelyik lap belsejében található).

A lapméretet n bájtosra választva a belső elaprózódás miatt a program utolsó lapján elveszített hely átlagosan $n/2$ bájt lesz, ami azt sugallja, hogy kicsi lapméret használatával minimalizálható a veszteség. A kis lapméret azonban rengeteg lapot és nagy laptáblát eredményez. Ha a laptáblát hardveres úton valósítjuk meg, a nagyméretű laptábla tárolásához több regiszter kell, ez növeli a gép árát. Továbbá minden program elindításakor vagy megállításakor több időre lesz szükség ezen regiszterek betöltéséhez és elmentéséhez.

Ráadásul kis lapméretnél a lemez sávszélességének kihasználtsága sem megfelelő. Mivel a tényleges átvitel kezdete előtt körülbelül 10 ms-ot kell várakozni a keresési és a forgási késleltetés miatt, a nagyobb mennyiségű átvitel hatékonyabb a kisebbnél. 10 MB/s átviteli sebesség esetén 1 KB helyett 8 KB átmásolása csak 0,7 ms-mal igényel több időt.

A kis lapoknak viszont megvan az az előnye, hogy a virtuális címtartomány elkülönült helyein elhelyezkedő sok kis lapból álló munkahalmaz esetén kisebb a vergődés esélye. Példaként tekintsünk egy 10 000 × 10 000-es A mátrixot, melynek $A[1,1]$, $A[2,1]$, $A[3,1]$ stb. elemeit egymás utáni 8 bájtos szavakban tároljuk. Ez az oszlopfolytonos tárolás azt jelenti, hogy az első sor $A[1,1]$, $A[1,2]$, $A[1,3]$ stb. elemei egymástól 80 000 bájt távolságra helyezkednek el. Ha egy program sokat számolna a sor összes elemével, 10 000 olyan régióval dolgozna, melyek mindegyikét 79 992 bájt választja el a rákövetkezőtől. 8 KB lapméretnél a felhasznált lapoknak összesen 80 MB tárolóhely kellene.

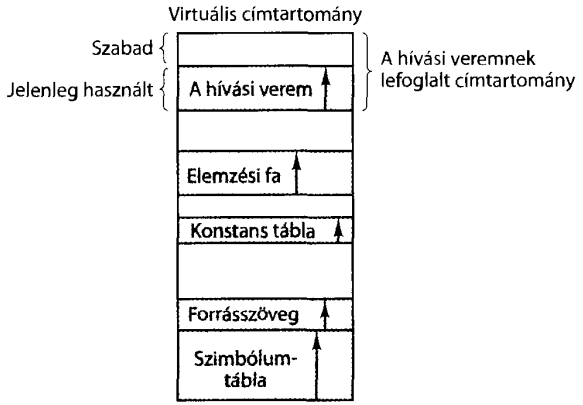
1 KB lapmérettel csak 10 MB RAM-ra lenne szükség. Ha a rendelkezésre álló memória 32 MB, akkor 8 KB lapmérettel vergődne a program, 1 KB-tal azonban nem. Mindent figyelembe véve, a fejlődés irányát a nagyobb lapméretek használata jellemzi.

6.1.6. Szegmentálás

Az eddig vizsgált virtuális memória lineáris (egydimenziós) volt; a címek 0-tól kezdve valamely maximális címig sorjáztak. Sok esetben előnyösebb, ha egyetlen virtuális címtartomány helyett kettő vagy több külön címtartományunk van. A fordítóprogramoknak például a fordítás során felépített számos táblázatuk lehet:

1. A változók nevét és attribútumait tartalmazó szimbólumtábla.
2. A listázáshoz megőrzött forráskód.
3. Az összes felhasznált egész és lebegőpontos konstanst tartalmazó tábla.
4. A program szintaktikus elemzésekor létrehozott elemzési fa.
5. A fordítóprogramon belüli eljáráshívásokhoz tartozó verem.

Az első négy tábla mérete folyamatosan nő a fordítás előrehaladtával. Az utolsóé előre nem látható módon nő és csökken menet közben. Egydimenziós memóriában az öt tábla számára a 6.7. ábrán látható módon foglalnánk le folytonos szeleteket a virtuális címtartományból.



6.7. ábra. *Az egydimenziós címtartományban a növekvő táblák egymásba ütközhetnek*

Vizsgáljuk meg, mi történik, ha egy program kivételesen sok változót tartalmaz. A szimbólumtábla számára lefoglalt memóriarész akkor is betelhet, ha a többi táblában még rengeteg hely van. A fordítóprogram egyszerűen kiírhatja, hogy a túl sok változó miatt nem tudja folytatni munkáját, de ez nem tűnik túl sportszerű megoldásnak, hiszen a többi táblában még maradt kihasználatlan hely.

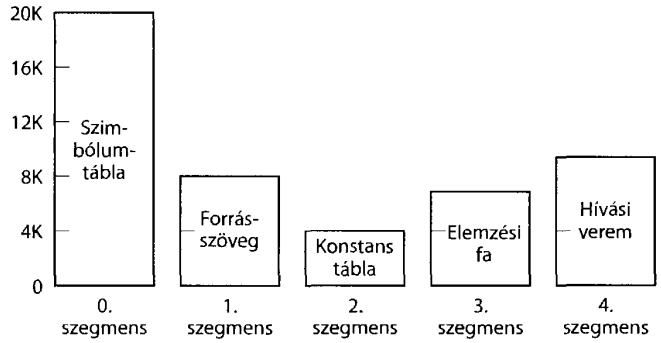
A másik lehetőség az lenne, hogy a fordítóprogram Robin Hood módjára csökkentené a sok üres hellyel rendelkező táblák méretét, s a szabad területet a helyszűkében lévőknek adná át. Ez az átrendezés kivitelezhető, de akárcsak az átfedések kezelésének megszervezése, legalábbis kellemetlen, a legrosszabb esetben pedig jó sok fáradságos, nem kifizetődő munkát jelent.

Valójában olyan módszerre lenne szükség, amely mentesíti a programozót az összehúzódó és kinyúló táblák kezelésének kényszerétől, mint ahogy a virtuális memória megszüntette a program átfedésekre osztásának a gondját.

Az egyik nyilvánvaló megoldás sok teljesen független címtartomány, ún. **szegmensek (segments)** bevezetése. Minden szegmens 0-tól valamely maximális értékig terjedő címek lineáris sorozata. Az egyes szegmensek hossza 0-tól a megengedett maximumig bármekkora lehet. A különböző szegmensek hossza eltérő lehet, rendszerint ez is a helyzet. A szegmensek hossza még változhat is a végrehajtás során. Például valahányszor a verembe beteszünk valamit, a veremszegmens mérete nőhet, ha pedig kiveszünk belőle, csökkenhet.

Mivel minden szegmens külön címtartományt alkot, a különböző szegmensek egymástól függetlenül nőhetnek vagy zsugorodhatnak. Amikor az egyik szegmensben elhelyezkedő veremnek a növekedéshez nagyobb címtartományra van szüksége, nyugodtan megkaphatja, mert semmi más nincs a címtartományában, amivel összeütközésbe kerülhetne. A szegmensek természetesen betelhetnek, de általában olyan nagyok szoktak lenni, hogy ez csak ritkán fordulhat elő. Ilyen szegmentált vagy kétdimenziós memória esetén a programok két részből álló címmel adják meg a tényleges címet. Az első rész a szegmens száma, a második a szegmensen belüli cím. A 6.8. ábra a korábban említett fordítóprogram tábláinak szegmentált memóriába való elhelyezését illusztrálja.

Hangsúlyozzuk, hogy a szegmens olyan logikai egység, amely a programozó számára is látható, s amelyet ő is egyetlen logikai egységként kezel. A szegmens tartalmazhat eljárást, tömböt, vermet vagy skaláris változókból álló gyűjteményt, de rendszerint nincs benne különböző típusokból álló keverék.



6.8. ábra. *Szegmentált memóriában minden tábla a többiektől függetlenül nőhet vagy zsugorodhat*

A változó méretű adatstruktúrák egyszerűbb kezelése mellett a szegmentált memóriának más előnyei is vannak. Ha az egyes eljárások 0 kezdőcímtől más-más szegmensekben helyezkednek el, a külön lefordított eljárások összeszerkesztése is sokkal könnyebb. A programot alkotó eljárások fordítása és összeszerkesztése

után az *n*. szegmensben lévő eljárás hívása a 0. szó (a belépési pont) megcímzése az (*n*, 0) kétrészes címmel történik.

Ha az *n*. szegmensben található eljárást a későbbiekben módosítjuk és újrafordítjuk, egyetlen másikon sem kell változtatnunk, mivel a kezdőcímek nem változtak (még akkor sem, ha az új változat hosszabb a réginél). Egydimenziós memóriába szorosan egymás után helyeznék el ezeket az eljárásokat, hogy ne maradjon ki köztük címtartomány. Következésképpen valamelyik eljárás méretének megváltoztatása befolyásolhatja más, hozzá nem kapcsolódó eljárások kezdőcímét is. Emiatt viszont módosítani kell minden olyan eljárást, amely az áthelyezetteket hívja, hogy ezekbe is bekerüljön az új kezdőcím. Mindez elég drága mulatság lehet több száz eljárásból álló programok esetén.

A különböző programok közti kód- és adatmegosztást is elősegíti a szegmentálás. Ha a gépen több program fut párhuzamosan (akár valódi, akár szimulált párhuzamos feldolgozással), és ezek mind használnak bizonyos közös könyvtári eljárásokat, a memória pocsékolása lenne, ha mindegyik program saját másolatot kapna belőlük. Minden eljárást külön szegmensbe helyezve könnyen megoszthatjuk őket, s így nem kell belőlük egynél több fizikai másolatot a memóriában tartani. Ezzel memóriát spórolunk meg.

Mivel minden szegmens a programozó által is látható logikai egységet képez (ami lehet eljárás, tömb vagy verem), a szegmensek védelme eltérő lehet. Az eljárásszegmensek lehetnek csak végrehajthatók, megtiltva ezzel olvasásukat vagy felülírásukat. A lebegőpontos tömböket specifikálhatjuk írhatónak és olvashatónak, de nem végrehajthatónak. Így észrevehető, ha a program végrehajtása rájuk próbálna futni. Az efféle védelem gyakran segítheti a programozási hibák megtalálását.

Próbáljuk meg átgondolni, hogy miért van értelme a védelemnek a szegmentált memóriában és miért nincsen az egydimenziós (lineáris) lapozott memóriában. Szegmentált memória esetén a felhasználó tudja, hogy mi van az egyes szegmensekben. Normális esetekben például nem volna ugyanabban a szegmensben egy eljárás is meg egy verem is, hanem csak vagy az egyik, vagy a másik. Mivel minden szegmens csak egyetlen típusú objektumot tartalmaz, a védelme ennek megfelelő lehet. A lapozást és a szegmentálást a 6.9. ábrán hasonlítjuk össze.

A lapok tartalma bizonyos értelemben véletlenszerű. A programozó még a lapozás tényét sem veszi észre. Bár ki lehetne egészíteni a laptábla bejegyzéseit né-

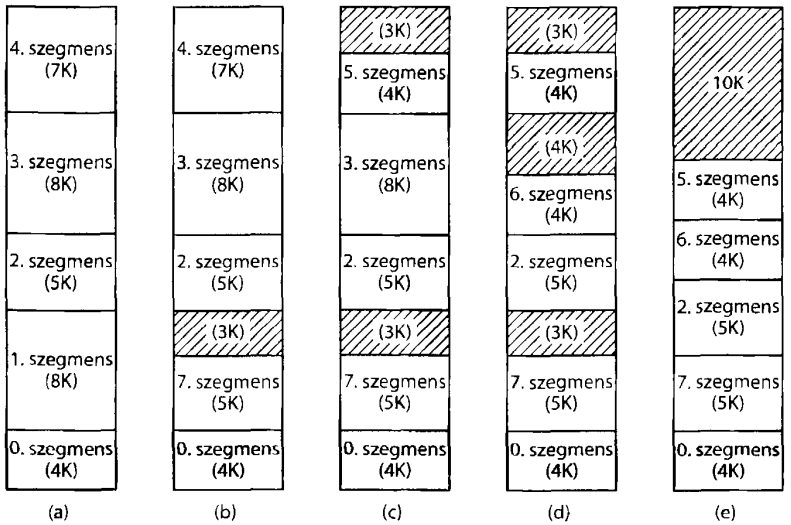
Szempontok	Lapozás	Szegmentálás
Tudnia kell-e róla a programozónak?	Nem	Igen
Hány lineáris címtartomány létezik?	1	Sok
Meghaladhatja-e a virtuális címtartomány nagysága a fizikai memória méretét?	Igen	Igen
Könnyen kezelhetők-e a változó méretű táblák?	Nem	Igen
Milyen céllal dolgozták ki ezt a technikát?	Nagy memória szimulálására	Több címtartomány biztosítására

6.9. ábra. *A lapozás és a szegmentálás összehasonlítása*

hány olyan bittel, amelyek a megengedett hozzáférést határoznák meg, ennek kihasználásához a programozónak nyilván kellene tartania, hogy címtartományán belül hová esnek a laphatárok. Az ötlet szépséghibája az, hogy a lapozást pontosan az effajta adminisztrációs feladatok kiküszöbölésére találták ki. A szegmentált memória használóinak úgy tűnik, hogy az összes szegmens állandóan a memóriában van, tehát ezek megcímezhetők anélkül, hogy átfedésekként való adminisztrációjukkal kellene foglalkozni.

6.1.7. A szegmentálás megvalósítása

A szegmentálás kétféle módon valósítható meg: cseréléssel (swapping) vagy lapozással. Az első módszernél minden időpillanatban szegmensek bizonyos halmaza van a memóriában. Ha olyan szegmensre történik hivatkozás, amely pillanatnyilag nincs bent, akkor az betöltődik. Ha nincs elég hely számára, először egy vagy több szegmenst lemezre kell írni (kivéve, ha ezekből már van a lemezen tiszta példány; ekkor a memóriában található másolat eldobható). A szegmensek cserélése bizonyos értelemben hasonló a kérésre lapozáshoz: a szegmensek szükség szerint mozognak a lemez és a memória között. A szegmentálás megvalósítása azonban lényegesen eltér a lapozásétól: míg a lapméret rögzített, a szegmensek mérete változó. A 6.10. (a) ábra példaként kezdetben hat szegmenst tartalmazó memóriát mutat. Nézzük meg, mi történik, ha eltávolítjuk az 1. szegmenst, s helyére a nála kisebb 7. szegmenst hozzuk be. A 6.10. (b) ábrán látható memóriakonfigurációt



6.10. ábra. *(a)–(d) A külső elaprózódás kialakulása. (e) Az elaprózódás megszüntetése összepréseléssel*

kapjuk. A 7. és a 2. szegmens között kihasználatlan terület, „lyuk" van. Ezután a 6.10. (c) ábra szerint a 4. szegmenst az 5.-kel, majd a 6.10. (d) ábrán látható módon a 3. szegmenst a 6. szegmenssel helyettesítjük. Miután egy ideig fut a rendszer, a memória szegmenseket és „lyukakat" tartalmazó részekre osztható. A jelenség neve **külső elaprózódás (external fragmentation)**, mivel az elpocsékolt hely a szegmenseken kívül, a köztük lévő lyukakban van. A külső elaprózódást néha **lyukacsosodás (checkerboarding)** néven is emlegetik.

Nézzük meg, mi történne, ha a memóriának a 6.10. (d) ábrán látható elaprózódása után a program a 3. szegmensre hivatkozna. A lyukakban lévő hely összesen 10 KB, ami bőven elég lenne a 3. szegmensnek, de mivel ez a hely kicsiny, haszontalan darabokra van szétszabdalva, a 3. szegmens egyszerűen nem tölthető be. Először el kell távolítani valamelyik szegmenst a memóriából.

A külső elaprózódás elkerülésének egyik lehetséges módszere a következő. Valahányszor lyuk keletkezik, a lyuk mögötti szegmenseket toljuk el a 0 memóriacím felé úgy, hogy ezt a lyukat megszüntessük. Ekkor persze a memória végén keletkezik egy nagyobb lyuk. Esetleg várhatunk is addig, míg a külső elaprózódás komolyabb szintet nem ér el (például a teljes memória adott százaléka fölé nem kerül a lyukak összege), és csak ekkor kezdjük el a memória „összepréselését". Ennek az összepréselésnek az a célja, hogy a sok kis haszontalan lyukat egy olyan nagy lyukba gyűjtsük össze, amelyben már elhelyezhető egy vagy több szegmens. Az összepréselés nyilvánvaló hátránya, hogy végrehajtásához bizonyos időre van szükség. Ezért a minden új lyuk keletkezése utáni összepréselés rendszerint túl sok időt igényelne.

Ha a memória összepréseléséhez szükséges idő elfogadhatatlanul nagy, akkor olyan algoritmusra van szükség, amely el tudja dönteni, hogy az adott szegmenst melyik lyukba helyezzük el. A lyukakkal való gazdálkodáshoz az összes lyuk címét és méretét tartalmazó listára van szükségünk.

Egy népszerű algoritmus, a **legjobb illesztés (best fit)**, azt a legkisebb lyukat választja, amelybe még belefér a kért szegmens. Emögött az az ötlet áll, hogy a lyukak és a szegmensek összepárosításánál lehetőleg ne „törjünk le" darabokat olyan nagy lyukakból, amelyeket fel lehetne használni a későbbi nagy szegmensekhez.

A másik kedvelt eljárás, az **első illesztés (first fit)**, körbemegy a listán, és azt az első lyukat választja, amely elég nagy a szegmens tárolására. Ehhez nyilvánvalóan kevesebb idő kell, mint az egész lista átnézéséhez a legjobb illesztés megkeresésénél. Meglepő, hogy az általános hatékonyság szempontjából is jobb az első illesztés algoritmusa a legjobb illesztésnél. Az utóbbi hajlamos arra, hogy rengeteg kicsiny, teljesen haszontalan lyukat hozzon létre (Knuth, 1997).

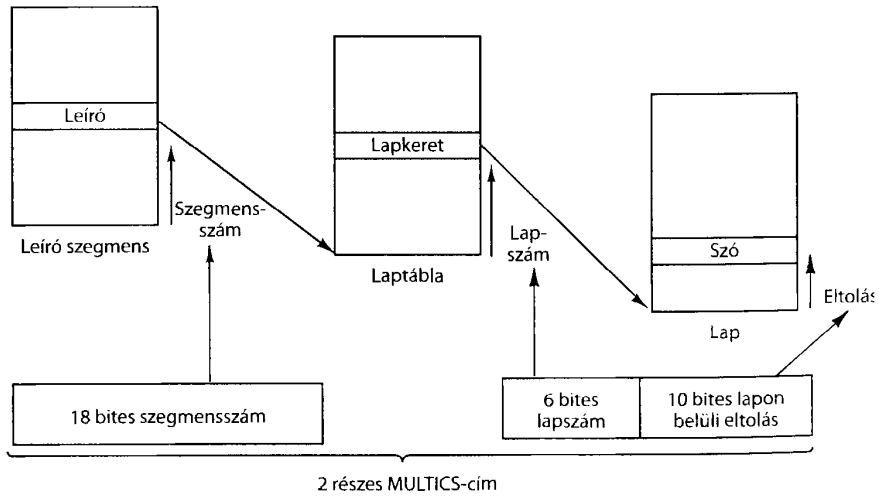
Mindkét algoritmusnál csökken a lyukak átlagos nagysága. Valahányszor egy szegmenst nála nagyobb lyukba töltünk be (majdnem mindig ez történik, ritka a pontos egyezés), a lyuk két részre bomlik. Az egyiket a szegmens foglalja el, a másikból lesz az új lyuk, amely mindig kisebb a réginél. Ha nincs a fentieket ellensúlyozó olyan folyamat, amely a kis lyukakból újra nagyokat hoz létre, végül mind a legjobb illesztés, mind az első illeszkedés apró, haszontalan lyukakkal tölti meg a memóriát.

Ilyen kompenzációs eljárás lehet a következő: amikor olyan szegmenst távolítunk el a memóriából, amelynek mindkét közvetlen szomszédja lyuk, az egy-

más utáni lyukak egy nagy lyukká olvaszthatók össze. Ha a 6.10. (d) ábráról az 5. szegmenst távolítanánk el, a két környező lyuk és a szegmens által használt 4 KB egyetlen 11 KB méretű lyukká olvadna össze.

A szakasz elején azt állítottuk, hogy a szegmentálás kétféleképpen, cseréléssel vagy lapozással valósítható meg. Az eddigi fejtegetések során a cserélésre összpontosítottunk. Ennél a sémánál teljes szegmensek ingáznak igény szerint a lemez és a memória között. A szegmentálás másik megvalósításánál a szegmenseket fix méretű lapokra osztjuk, és ezeket lapozzuk a kérések szerint. Lehetséges, hogy valamelyik szegmens lapjainak egy része a memóriában, a többi meg a lemezen van, ha ezt a megoldást választjuk. A lapozáskor minden szegmenshez külön laptábla kell. Mivel minden szegmens csupán egy lineáris címtartomány, az összes eddig megismert lapozási technika alkalmazható a szegmensekre. Az egyetlen új sajátosság az, hogy minden szegmens saját laptáblát kap.

Az eredetileg az M. I. T., a Bell Labs és a General Electric együttműködésével létrehozott korai operációs rendszer, a **MULTICS (MULtiplexed Information and Computing Service)**, is lapozással kombinált szegmentálást használt (Corbató és Vyssotsky; 1965; Organick, 1972). A MULTICS címei két részből álltak, a szegmensszámból és a szegmensen belüli címből. Minden processzushoz tartozott egy leíró szegmens, amely minden szegmenshez tartalmazott egy leírót. Amikor a hardver egy virtuális címet kapott, a 6.11. ábrán látható módon a szegmensszámot a leíró szegmens indexezésére használva kereste meg benne a megcímzendő szegmens leíróját. Ez a leíró a szegmens laptáblájára mutatott, ami lehetővé tette a szegmensek szokásos módon történő lapozását. A rendszer teljesítményét 16 rekeszes, hardverrel megvalósított **asszociatív memória** javította, amelyben a legutóbb használt szegmens/lap kombinációkat tárolták, így ezeket gyorsan visz-



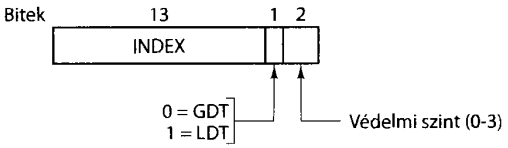
6.11. ábra. *A kétrészes MULTICS-cím átalakítása memóriacímmé*

sza lehetett keresni. Bár a MULTICS már rég a múlté, de a szelleme tovább él. A 386-os óta az összes Intel-processzor virtuális memóriakezelését lényegében róla mintázták.

6.1.8. A Pentium 4 virtuális memóriája

A Pentium 4-nek olyan kifinomult virtuálismemória-rendszere van, amely egyaránt támogatja a kérésre lapozást, a tiszta szegmentálást és a lapozással kombinált szegmentálást. A Pentium 4 virtuális memóriájának legfontosabb része két tábla, az **LDT (Local Descriptor Table, lokális leírótábla)** és a **GDT (Global Descriptor Table, globális leírótábla)**. Minden programnak saját LDT-je van, de a gépen futó összes program egyetlen megosztott GDT-t használ. Míg a GDT rendszer szegmenseket ír le, ideértve magát az operációs rendszert is, addig az LDT a programra nézve lokális szegmenseket ad meg (kód, adat, verem stb.).

A Pentium 4-en futó program egy szegmens eléréséhez először a szegmensre vonatkozó szelektort tölti be valamelyik szegmensregiszterébe az 5. fejezetben leírt módon. A végrehajtás során CS-ben van a kódszegmens szelektora, DS-ben az adatszegmensé stb. Minden szelektor 16 bites (lásd 6.12. ábra).



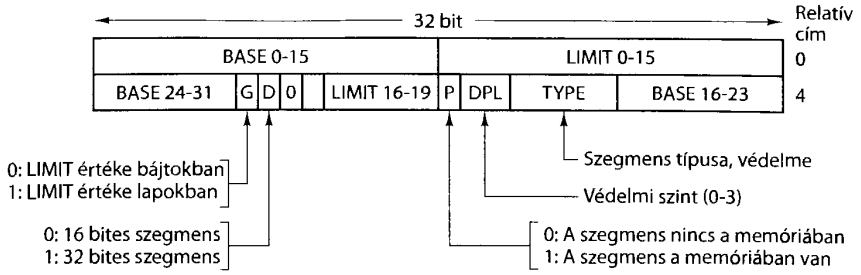
6.12. ábra. *A Pentium 4 szelektora*

A szelektor egyik bitje azt mondja meg, hogy a szegmens lokális vagy globális (tehát hogy az LDT-ben vagy a GDT-ben található meg). Tizenhárom további bit adja meg az LDT-n vagy GDT-n belüli bejegyzés sorszámát, ezek a táblák tehát legfeljebb 8 K (2^{13}) bejegyzést tárolhatnak. A védelemre vonatkozó további 2 bitet később ismertetjük.

A 0. leíró érvénytelen, használata csapdát (trap) okoz. Szegmensregiszterbe töltése nem okoz problémát; ezzel jelezhetjük, hogy a regiszter pillanatnyilag nem használható. Ha mégis hivatkozunk rá, csapdát okoz.

Amikor a szelektor betöltődik valamelyik szegmensregiszterbe, az LDT-ből vagy a GDT-ből a megfelelő leíró is bekerül az MMU belső regisztereibe, hogy gyorsabban el lehessen érni. Mint a 6.13. ábrán látható, a 8 bájtos leíró tartalmazza a szegmens báziscímét (base address), méretét és egyéb információkat.

A szelektor formátumát ügyesen választották, hogy könnyű legyen a leírót megtalálni. Először a szelektor 2. bitje alapján vesszük az LDT vagy a GDT táblát. Ezután a szelektor az MMU egyik munkaregiszterébe másolódik át, és a 3 legalacsonyabb helyértékű bitje törlődik, ami gyakorlatilag a 13 bites szelektorszám nyolccal való szorzását jelenti. Végül hozzáadódik az LDT vagy a GDT tábla bel-



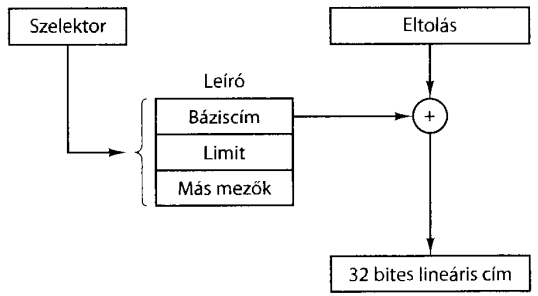
6.13. ábra. *A Pentium 4 kódszegmensének leírója (az adatszegmenseké ettől kicsit eltérő)*

ső MMU-regiszterekben tárolt címe, amivel közvetlenül a leíróra mutató pointert kapunk. Ha a szelektor például a GDT tábla 9. bejegyzésére hivatkozik, akkor az a GDT + 72. címen található.

Kövessük végig a (szelektor, eltolás) pár fizikai címmé alakításának lépéseit. Mihelyt megtudja a hardver, hogy melyik szegmensregisztert kell használni, belső regisztereiben rögtön megtalálja a szelektorhoz tartozó teljes leírót. Ha nem létező szegmensről van szó (a szelektor 0), vagy a szegmens pillanatnyilag nincs a memóriában (P értéke 0), csapda keletkezik. Az első esetben programozási hiba van, a másodikat az operációs rendszernek kell kezelnie.

Ezután azt vizsgálja, hogy az eltolás (offset) a szegmens határain belül van-e; ha nem, ez ismét csapdát okoz. Logikailag 32 bit kellene a szegmens méretének megadásához, de a leíróban erre a célra csak 20 bit van, ezért más eljárást használtak. Ha a leíró G bitje 0, akkor a LIMIT mező a szegmens pontos méretét adja meg (legfeljebb 1 MB). Ha ez a bit 1, akkor a LIMIT mező bájtok helyett lapokban tartalmazza a méretet. A Pentium 4 lapmérete mindig legalább 4 KB, így a 20 bit egészen 2^{32} méretű szegmensekig elegendő.

Ha szegmens a memóriában van, és az eltolás is a megengedett határok közé esik, akkor a Pentium 4 a leíró 32 bites BASE mezőjét az eltoláshoz hozzáadva a 6.14. ábrán látható **lineáris címet (linear address)** alakítja ki. A BASE mező a le-



6.14. ábra. *A (szelektor, eltolás) pár lineáris címmé alakítása*

íróban szétszórt három részből áll, hogy kompatibilis maradjon a 80286-tal, ahol a BASE mező csak 24 bites volt. A BASE mező teszi lehetővé, hogy a szegmens a 32 bites lineáris címtartomány tetszőleges címén kezdődhessen.

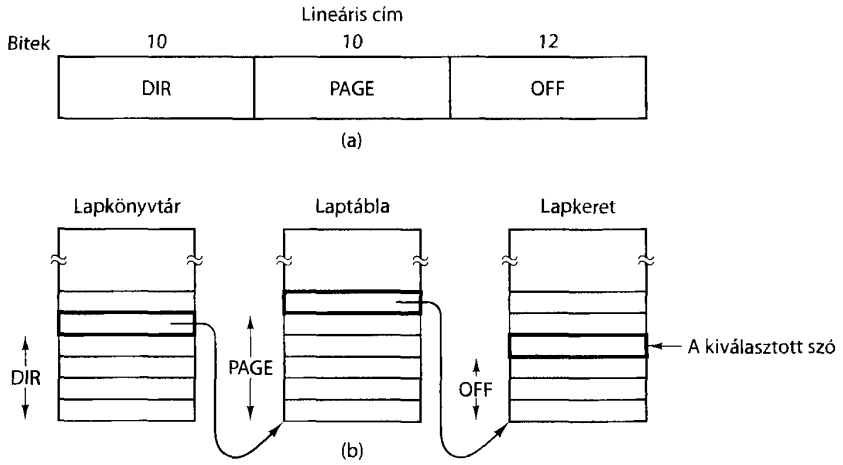
Ha a lapozás tiltott (egy globális vezérlőregiszter egy bitjével), akkor a lineáris címet fizikai címként értelmezi, és rögtön továbbítja a memória felé íráshoz vagy olvasáshoz. Így lapozás nélküli tiszta szegmentálással állunk szemben, a szegmensek báziscíme leírójukban található. A szegmensek átfedhetik egymást, ennek valószínűleg csupán az a magyarázata, hogy a diszjunktság ellenőrzése túl bonyolult lenne és sokáig tartana.

Ha viszont engedélyezett a lapozás, akkor a lineáris címet virtuális címként értelmezi, és laptáblák használatával képezi le a fizikai címre, ahogy azt a korábbi példákban láttuk. Az egyetlen bonyodalom, hogy a 32 bites virtuális címek és a 4 KB-os lapméret miatt egy szegmens egymillió lapot is tartalmazhat, így kétszintű leképezést használunk, hogy csökkentsük a kis szegmensek laptáblájának méretét.

Minden futó programhoz tartozik egy **lapkönyvtár (page directory)**, amely 1024 darab 32 bites bejegyzést tartalmaz, erre a lapkönyvtárra egy globális regiszter mutat. A könyvtár minden bejegyzése szintén 1024 darab 32 bites bejegyzést tartalmazó laptáblákra mutat. A laptáblák bejegyzései lapkeretekre mutatnak. Ezt a sémát láthatjuk a 6.15. ábrán.

A 6.15. (a) ábrán látható cím három mezőre van osztva: DIR, PAGE és OFF. Először a DIR mezőt használjuk fel a laptábla címének kikeresésére a lapkönyvtárból. Ezután a PAGE mező értéke alapján keressük ki a laptáblából a lapkeret fizikai címét. Végül az OFF (a lapon belüli eltolás) mező értékét a lapkeret címéhez hozzáadva megkapjuk a megcímezett bájt vagy szó fizikai címét.

A laptábla bejegyzései 32 bitesek, ebből 20 bit tartalmazza a lapkeret számát. A fennmaradó részt az operációs rendszer számára hasznos értékekkel tölti fel a



6.15. ábra. *Lineáris cím leképezése fizikai címre*

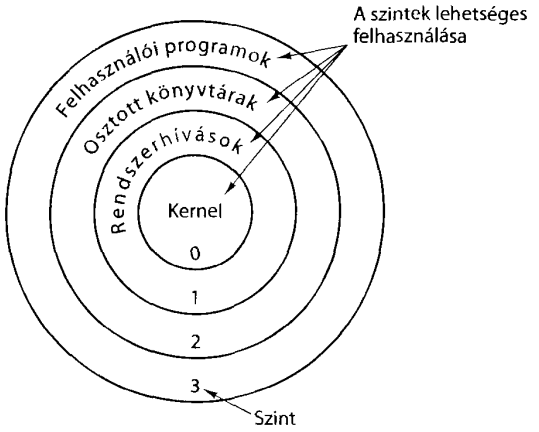
hardver (elérési, szennyezettségi, védelmi stb. bitek). Minden laptábla 1024 bejegyzést tartalmaz, amelyek 4 KB-os lapokra mutatnak, így egy laptábla 4 MB memóriát tud kezelni. A 4 MB-nál rövidebb szegmensek lapkönyvtárában csak egy bejegyzés lesz, amely a szegmens egyetlen laptáblájára mutat. Emiatt a rövid szegmensek többletköltsége csak 2 lap, szemben az egyszintű laptábla esetén szükséges ezernyi lappal.

Az ismételt memóriahivatkozások gyorsítására a Pentium 4 MMU egysége speciális hardverrel támogatja a legutóbb használt DIR-PAGE kombinációk gyors visszakeresését és a megfelelő lapkeret fizikai címére való leképezést. Csak akkor hajtódnak végre a 6.15. ábrán látható lépések, ha a keresett kombinációt mostanában még nem használtuk.

Kicsit utánagondolva gyorsan rájöhetünk, hogy lapozás használatakor nincs értelme a leírók BASE mezőjét nullától különböző értékre állítani. A BASE érték hatása csupán annyi, hogy a laptábla kezdete helyett egy kis eltolással, a tábla közepéből vett bejegyzéssel dolgozunk. A BASE mező használatának valódi oka a tiszta (nem lapozásos) szegmentálás támogatása a régi 80286-tal való kompatibilitás miatt, mivel az nem ismerte a lapozást.

Érdemes megemlíteni, hogy könnyen kielégíthető az olyan programok igénye is, amelyek nem használnak szegmentálást, hanem megelégszenek egyetlen lapokra osztott 32 bites címtartománnyal. Minden szegmensregiszterbe ugyanazt a szelektort kell betölteni, s ennek leírójában a BASE értékét 0-ra, a LIMIT-et maximálisra kell beállítani. Az eltolás maga lesz a lineáris cím, egyetlen címtartománnyal – ez lényegében a hagyományos lapozásnak felel meg.

Ezzel befejeztük a Pentium 4 virtuális memóriájának tárgyalását. Érdemes néhány szót ejteni védelmi rendszeréről, hiszen ez a téma szorosan kapcsolódik a virtuális memóriához. A Pentium 4 négy védelmi szintet támogat, a 0. a legerősebb, a 3. a leggyengébb. Ezeket a szinteket mutatja a 6.16. ábra. A program futá-



6.16. ábra. *A Pentium 4 védelmi rendszere*

sának minden pillanatában adott szinten van, ezt a **PSW (Program Status Word, programállapotszó)** nevű regiszter 2 bites mezője tartalmazza. A rendszer összes szegmenséhez is hozzá van rendelve egy szintszám.

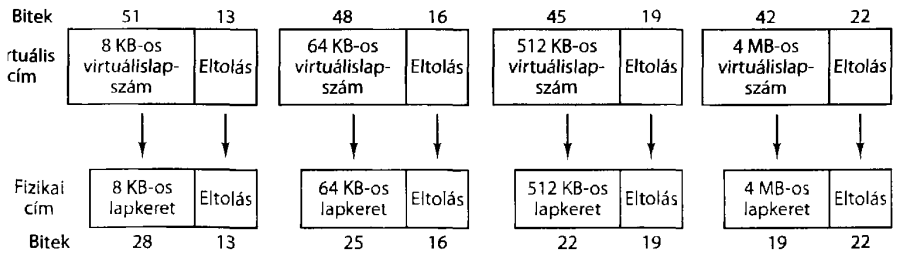
Amíg a program csak a saját szintjén lévő szegmenseket használja, minden simán megy. Megengedett a magasabb szinteken található adatokhoz való hozzáférés. Az alacsonyabb szinteken található adatok elérése illegális, az ilyen kísérletek csapdát okoznak. Más szinteken (akár alacsonyabbakon, akár magasabbakon) lévő eljárások hívása megengedett, de csak alaposan ellenőrzött módon. Egy szintközi hívásnál a CALL utasításnak a cím helyett szelektort kell tartalmaznia. Ez a szelektor egy **híváskapu (call gate)** nevű leírót jelöl ki, amely a hívandó eljárás címét adja meg. Emiatt csak a hivatalos belépési pontok használhatók; nem lehet egy másik szinten lévő tetszőleges kódszegmens közepébe ugrani.

A 6.16. ábra ennek a mechanizmusnak egyik javasolt felhasználását mutatja. A 0. szinten az operációs rendszer magja (kernel) található. Ez kezeli a B/K-t, a memóriát és más kritikus dolgokat. Az 1. szinten van a rendszerhívás-kezelő. A felhasználói programok rendszerhívások végrehajtása céljából meghívhatnak itteni eljárásokat, de az eljárásoknak csak egy meghatározott és védett részhalmaza használható. A 2. szint könyvtári eljárásokat tartalmaz, melyeket esetleg több program megosztva használ. A felhasználói programok csak hívhatják, de nem módosíthatják ezeket az eljárásokat. Végül a felhasználói programok a legkisebb védettségű 3. szinten futnak. A Pentium 4 memóriakezeléséhez hasonlóan ez a védelmi rendszer is nagyon hasonlít a MULTICS-nál alkalmazottra.

A csapdák és a megszakítások a híváskapukhoz hasonló mechanizmusokat használnak. Abszolút címek helyett ezek is leírókra hivatkoznak, és ezek a leírók mutatnak a végrehajtandó eljárásokra. A 6.13. ábrán látható TYPE mezővel különböztethetők meg a kód- és adatszegmensek, valamint a különféle kapuk.

6.1.9. Az UltraSPARC III virtuális memóriája

Az UltraSPARC III 64 bites virtuális címeken alapuló lapszervezésű virtuális memóriát támogató 64 bites gép. Tervezési és költségszempontok miatt azonban a programok nem használhatják a teljes 64 bites virtuális címtartományt. Csak



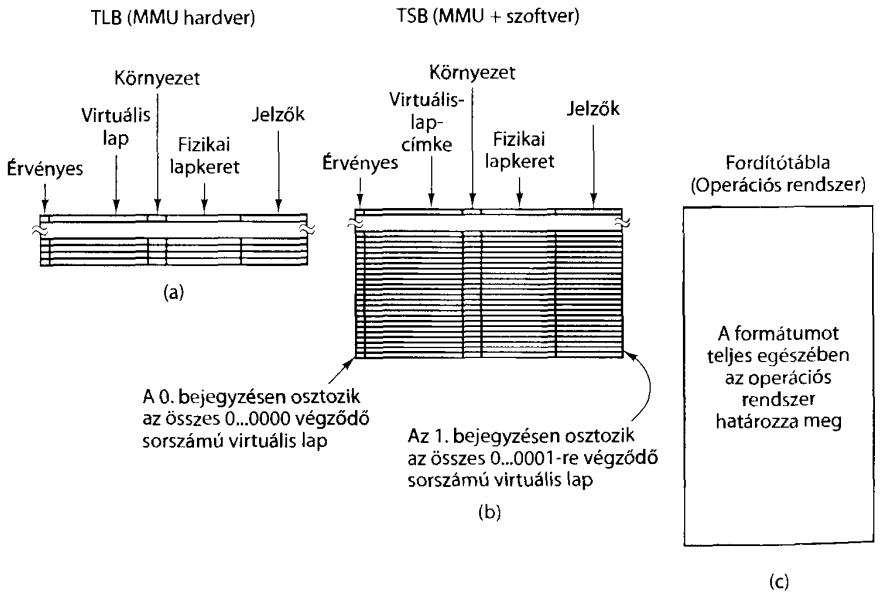
6.17. ábra. *A virtuális címek fizikai címekre való leképezése az UltraSPARC-on*

44 bit érhető el, emiatt a programok nem lehetnek $1{,}8 \times 10^{13}$ bájtnál hosszabbak. A megengedett virtuális memória két darab, egyenként 2^{43} bájt hosszúságú zónára van felosztva. Az egyik a virtuális címtartomány legalján, a másik a legtetején helyezkedik el. A köztük lévő résbe eső virtuális címek nem érhetők el, használatuk laphibát okoz.

Az UltraSPARC III-ba maximum 2^{41} bájt fizikai memória építhető be, ami nagyjából 2200 GB-nak felel meg. Ez elég is a legtöbb hétköznapi alkalmazáshoz. A négy támogatott lapméret: 8 KB, 64 KB, 512 KB és 4 MB. A négyféle lapméretthez tartozó leképezéseket mutatja a 6.17. ábra.

A rendkívül nagy virtuális címtartomány miatt a Pentium 4-hez hasonló direkt laptábla nem volna praktikus. Az UltraSPARC MMU-ja ehelyett egészen más megközelítést használ. Tartalmaz egy **TLB (Translation Lookaside Buffer, lapkezelő segédpuffer)** nevű hardvertáblát a virtuális lapok fizikai lapkeretekre való leképezésére. A 8 KB-os lapméret esetén 2^{31}, vagyis több mint 2 milliárd virtuális lapunk lehet. Mindegyik nyilván nem képezhető le.

A TLB ehelyett csak a legutóbb használt virtuális lapok sorszámát tárolja. A kód- és az adatlapokat külön kezeli, mindkét kategóriából a legutóbbi 64 lapszámot őrzi meg. Minden TLB-bejegyzés egy virtuális lapsorszámból és a neki megfelelő fizikai lapkeret sorszámából áll. Amikor az MMU egy processzusszámot vagy másképp **környezetet (context)**, és egy ebbe a környezetbe tartozó virtuális címet kap, speciális áramkörök segítségével összehasonlítja az adott környezethez tartozó összes TLB-bejegyzéssel. Ha valahol egyezést talál, a megfelelő



6.18. ábra. *Az UltraSPARC virtuális címek fordítására használt adatstruktúrái. (a) TLB. (b) TSB. (c) Fordítótábla*

TLB-bejegyzésben szereplő lapkeretsorszámot a virtuális címben szereplő eltolással (offset) összekombinálva képezi a 41 bites fizikai címet és néhány egyéb jelzőbitet (védelem stb.). A TLB-t mutatja be a 6.18. (a) ábra.

Ha nem talál egyezést, **TLB-hiány (TLB miss)** keletkezik, amelyet csapdáz az operációs rendszer. A hiány kezelése az operációs rendszeren múlik. Figyeljük meg, hogy a TLB-hiány teljesen más, mint a laphiány. TLB-hiány még akkor is előfordulhat, ha a hivatkozott lap a memóriában van. A kért virtuális lapra vonatkozó új TLB-bejegyzés beírásakor az operációs rendszer elméletileg bárhogyan eljárhatna. A hardver azonban némi segítséget is felajánl ennek a kritikus tevékenységnek minél gyorsabb elvégzéséhez – már ha a szoftver hajlandó vele együttműködni.

Elsősorban azt várjuk el, hogy az operációs rendszer készítsen a gyakran használt TLB-bejegyzésekről egy szoftveres gyorsítótárat a **TSB (Translation Storage Buffer)** nevű táblázatban. Ez a táblázat úgy épül fel, mint egy direkt leképezésű gyorsítótár. A 16 bájtos TSB-bejegyzések egy-egy virtuális lapra hivatkoznak és néhány egyéb jelzőbit mellett tartalmaznak egy érvényességi bitet, a környezetet, a virtuális címet és a fizikai lapkeret sorszámát. Ha a gyorsítótárba mondjuk 8192 bejegyzés fér el, az összes olyan virtuális lap, amely címének legalacsonyabb helyi értékű 13 bitje 0000000000000, a TSB 0. helyére pályázik. A 0000000000001-re végződő című virtuális lapok ugyanígy a TSB 1. helyéért versenyeznek a 6.18. (b) ábrán látható módon. A TSB méretét a szoftver határozza meg; az MMU-val való kommunikációja olyan speciális regisztereken keresztül zajlik, amelyekhez csak az operációs rendszer férhet hozzá.

TLB-hiány esetén az operációs rendszer megvizsgálja, hogy a megfelelő TSB-bejegyzés a szükséges lapot tartalmazza-e. Az MMU kellemes szolgáltatása a bejegyzés címének automatikus kiszámítása és az operációs rendszer által elérhető belső MMU regiszterbe helyezése. TSB-találat esetén valamelyik TLB-bejegyzés törlődik, és helyére bemásolódik a kért TSB-bejegyzés. A kitörlendő TLB-bejegyzés kiválasztását a hardver 1 bites LRU algoritmusa támogatja.

Ha a TSB-keresés sikertelen volt, a virtuális lap címe nincs a „gyorsítótárban", az operációs rendszer egy másik táblázatban keresi a lapra vonatkozó információt. A végső megoldásként használt tábla neve **fordítótábla (translation table)**. Mivel ehhez már semmilyen hardveres segítség sincs, a tábla formátumát az operációs rendszer szabadon döntheti el. Választhat például tördeléses kódolást: a virtuálislap-számot valamely p prímmel elosztva a maradékot használja egy olyan pointertábla indexezésére, amelynek elemei az azonos p-hez tartozó virtuálislap-számokból álló listákra mutatnak. Figyeljük meg, hogy ezek a bejegyzések nem maguk a lapok, hanem a TSB-nek megfelelő elemek. A fordítótáblában végzett keresés eredményeként elképzelhető az is, hogy a memóriában találjuk a kérdéses lapot, ekkor frissíteni kell a szoftveres gyorsítótár TSB-bejegyzését. Ha az derül ki, hogy a lap nincs a memóriában, a laphiány esetén szokásos tevékenységek kezdődnek el.

Érdekes összehasonlítani a Pentium 4 és az UltraSPARC III lapozási sémáját. A Pentium 4 támogatja a tiszta szegmentálást, a tiszta lapozást és a lapozott szegmenseket. Az UltraSPARC III-on csak lapozás van. A Pentium 4 is használ hardver segítségét a laptábla bejárásához, TLB-hiány esetén újra kell tölteni TLB-t. Az UltraSPARC TLB-hiány előfordulásakor csupán átadja a vezérlést az operációs rendszernek.

Ezen különbségek fő oka abban rejlik, hogy a Pentium 4 csupán 32 bites szegmenseket használ, az ilyen kicsi szegmensek (melyeknek csak 1 millió lapja van) hagyományos laptáblákkal kezelhetők. Elméletileg a Pentium 4-nek is gondjai lennének, ha egy program szegmensek ezreit használná, de mivel sem a Windows, sem a UNIX nem támogatja processzusonként egynél több szegmens használatát, ez a probléma nem jöhet elő. Ezzel szemben az UltraSPARC III igazi 64 bites gép, akár 2 milliárd lap is lehetséges, tehát a hagyományos laptáblák nem alkalmazhatók. A jövőben minden gépnek 64 bites virtuális címtartománya lesz, ezért az UltraSPARC-nál alkalmazotthoz hasonló sémák válnak irányadóvá.

6.1.10. Virtuális memória és gyorsítótár

A (kérésre lapozásos) virtuális memória és a gyorsítótár első pillantásra elég távoli fogalmaknak tűnnek, holott koncepcionálisan igenis hasonlók. Virtuális memória használatakor az egész programot lemezen tartjuk, és fix méretű lapokra daraboljuk fel. A lapoknak csak egy részhalmaza van a memóriában. Ha a program többnyire a memóriában található lapokat használja, kevés laphiány lesz, és a program gyorsan fut. Gyorsítótár esetén az egész programot a memóriában tartjuk, és fix méretű gyorsítótárblokkokra osztjuk. Ha a program többnyire a gyorsítótárban található blokkokat használja, kevés gyorsítótárhiány (cache miss) lép fel, ezért a program futása gyors lesz. Fogalmilag két hasonló jelenségről van szó, amelyek a hierarchia különböző szintjein hatnak.

Természetesen vannak eltérések is a virtuális memória és a gyorsítótár között. Egyrészt a gyorsítótárhiányokat a hardver, míg a laphiányokat az operációs rendszer kezeli. Továbbá a gyorsítótár blokkjainak tipikus mérete jóval kisebb, mint a lapméret (64 bájt, illetve 8 KB). Ezenkívül a virtuális lapok és a lapkeretek közötti leképezés is más: a laptáblákat a virtuális címek legnagyobb helyértékű bitjei szerint szervezik, a gyorsítótárak meg a memóriacímek legkisebb helyértékei szerint indexeznek. Mindazonáltal fontos, hogy megértsük, ezek csak implementációs eltérések, az alapötlet nagyon hasonló.

6.2. Virtuális B/K utasítások

Az ISA-szintű utasításkészlet teljesen más, mint a mikroarchitektúra utasításai. A két szinten egészen más utasítások hajthatók végre, az utasítások formátuma is eltérő. Inkább csak véletlennek tekinthető a két szint néhány megegyező utasításának létezése.

Ezzel szemben az OSM-szint utasításkészlete tartalmazza a legtöbb ISA-utasítást, csak kisszámú, ámde fontos új utasítást vezet be, és egy pár potenciálisan veszélyeset hagy el. A bevitel/kivitel az egyik olyan terület, ahol a két szint lényegesen eltér. A különbség oka egyszerű: az a felhasználó, aki valódi ISA-szintű utasításokat tudna végrehajtani, a rendszerben bárhol tárolt bizalmas adatokat el tud-

ná olvasni, írni tudna mások termináljára, s általában sok kellemetlenséget tudna okozni magának, mivel veszélyeztetné az egész rendszer biztonságát. Továbbá nincs olyan normális, épelméjű programozó, aki maga kívánná az ISA-szintű B/K műveleteket kezelni, hiszen az rendkívül fárasztó és bonyolult lenne. Be kellene állítani a hardvereszközökhöz tartozó regiszterek megfelelő bitjeit és mezőit, megvárni a műveletek befejeződését, majd ellenőrizni, hogy mi is történt valójában. Az utóbbira vonatkozó példaként a merevlemezekhez tartozó eszközregiszterek sok más mellett tipikusan a következő hibákat érzékelő biteket tartalmazzák:

1. A vezérlő hibásan pozicionálta az olvasófejet.
2. Pufferként nem létező memóriát adtunk meg.
3. A lemez B/K az előző művelet befejezése előtt kezdődött el.
4. Olvasásidőzítési hiba.
5. Nem létező lemezt címeztünk meg.
6. Nem létező cilindert címeztünk meg.
7. Nem létező szektort címeztünk meg.
8. Olvasáskor ellenőrző összeg hiba lépett fel.
9. Az írás utáni ellenőrzés hibát jelzett.

A fenti hibák valamelyikének előfordulása az eszközregiszter megfelelő bitjét állítja be. Kevés olyan felhasználó van, aki ezeknek a hibáknak és még egy csomó egyéb állapotinformációnak a kezelésével óhajtana bajlódni.

6.2.1. Fájlok

A virtuális B/K megszervezésének egyik módja a **fájl (file)** nevű absztrakció használata. Legegyszerűbb formájában egy B/K eszközre írt bájtsorozatot jelent. Ha lemezhez hasonló tárolóeszközről van szó, a fájl később vissza is olvasható, más esetekben, például egy nyomtatónál ez természetesen nem lehetséges. A lemezen sok fájl lehet, ezek mindegyike valamely meghatározott fajtájú adatot tárol, például képet, számolótáblát vagy egy könyv valamelyik fejezetének szövegét. Különböző fájlok hossza és egyéb tulajdonságai eltérők lehetnek. Az absztrakt fájlfogalom teszi lehetővé a virtuális B/K egyszerű megszervezését.

Mint korábban leírtuk, az operációs rendszer szempontjából a fájl általában csak egy bájtsorozat. További struktúrája a felhasználói programoktól függ. A B/K a fájlok megnyitását, olvasását, írását és lezárását végző rendszerhívásokkal történik. A fájlokat olvasás előtt meg kell nyitni. A fájl megnyitásának folyamata teszi lehetővé, hogy az operációs rendszer megkeresse a fájlt a lemezen, és a memóriába töltse az eléréséhez szükséges információt.

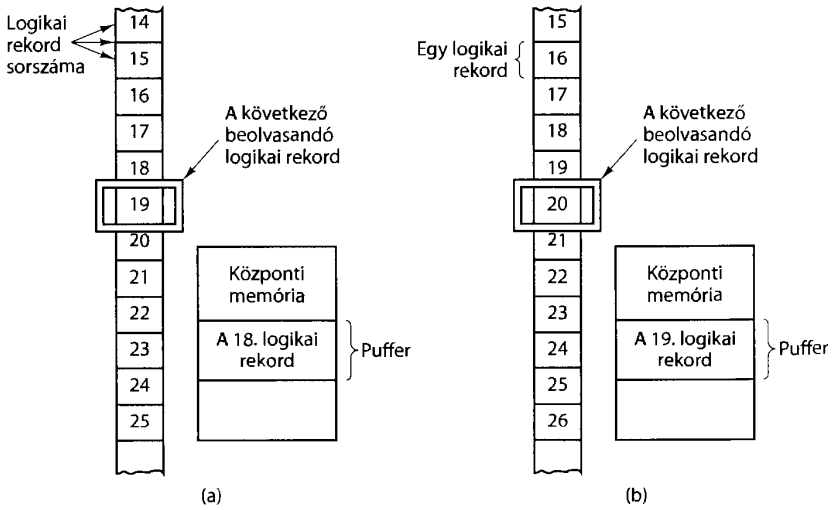
Megnyitás után már olvashatók a fájlok. A read rendszerhíváshoz minimálisan a következő paraméterek szükségesek:

1. Annak jelzése, hogy melyik megnyitott fájlból akarunk olvasni.
2. A beolvasott adatok tárolására szolgáló memóriapufferre mutató pointer.
3. Az olvasandó bájtok száma.

A read hívás a kért adatokat a pufferbe helyezi. Általában a ténylegesen beolvasott bájtok számát adja vissza, amely lehet kevesebb is a kérésben szereplő értéknél (1000 bájt hosszúságú fájlból nem lehet 2000 bájtot olvasni). Minden megnyitott fájlhoz tartozik egy pointer, amely a fájl legközelebb kiolvasható bájtjára mutat. Értéke a read végrehajtása után a beolvasott bájtok számának megfelelően változik, így az egymás utáni read hívások egymás utáni adatblokkokat olvasnak a fájlból. Ezt a pointert legtöbbször tetszőleges adott értékre be lehet állítani, vagyis a programok véletlenszerűen is elérhetik a fájl bármely részét. Az olvasás befejezése után a program lezárhatja a fájlt, ezzel tudatva az operációs rendszerrel, hogy a továbbiakban már nem fogja használni, tehát a rendszer táblázataiban felszabadítható a fájlról szóló információk tárolására lefoglalt hely.

A nagygépes operációs rendszereknek már bonyolultabb elképzelése van a fájlokról. Itt a fájl jól definiált struktúrával rendelkező **logikai rekordok (logical records)** sorozata. Ilyen logikai rekord lehet például az az ötelemű adatstruktúra, amely két karaktersorozatból (Név, Témavezető), két egészből (Intézet, Szobaszám), valamint egy logikai értékből (Nő/Férfi) áll. Bizonyos operációs rendszerek megkülönböztetik a csupa azonos szerkezetű rekordból álló fájlokat azoktól, amelyek különböző rekordtípusok keverékét tartalmazzák.

A legegyszerűbb virtuális bemeneti utasítás a megadott fájl következő rekordját olvassa be és helyezi el a memória meghatározott helyére. Ezt mutatja a 6.19. ábra. A művelet végrehajtásához a virtuális utasításnak meg kell mondani, hogy melyik fájlból olvasson, és a beolvasott rekordot hová tegye a memóriában. Gyakran arra is lehetőség van, hogy a fájlban elfoglalt helye vagy kulcsa alapján adjuk meg az olvasandó rekordot.



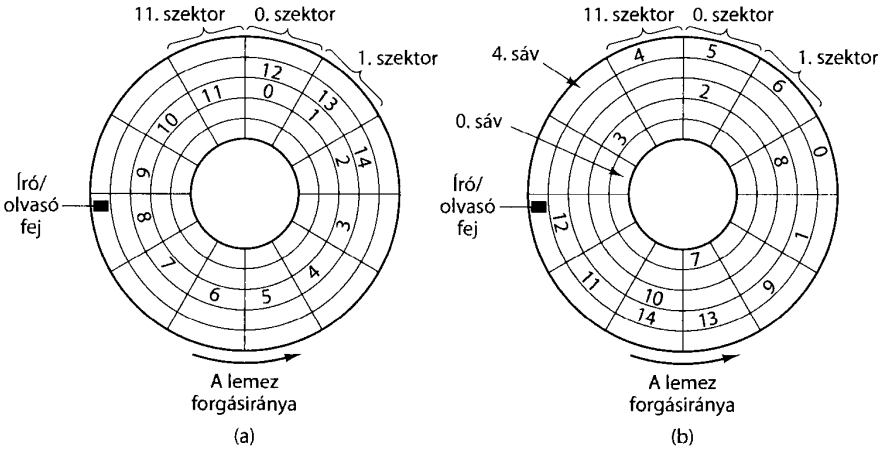
6.19. ábra. *Logikai rekordokból álló fájl olvasása. (a) A 19. rekord beolvasása előtt. (b) A 19. rekord beolvasása után*

A legegyszerűbb virtuális kimeneti utasítás egy rekordot ír ki a memóriából fájlba. Az egymás utáni szekvenciális write utasítások a fájlban egymás után sorakozó logikai rekordokat eredményeznek.

6.2.2. A virtuális B/K utasítások megvalósítása

A virtuális B/K utasítások implementációjának megértéséhez először a fájlok felépítését és tárolását kell megvizsgálnunk. Minden fájlrendszernél alapkérdés a helyfoglalás megoldása. A helyfoglalás alapegysége lehet a lemez egy szektora, de sokszor inkább egymás utáni szektorokból álló blokkokat vesznek.

A fájlrendszer megvalósításának másik alapvető sajátossága, hogy a fájlok egymás utáni helyfoglalási egységekben tárolódnak-e. A 6.20. ábrán olyan egyszerű lemezegységet látunk, amelynek egyetlen felülete 5 sávban található 12-12 szektort tartalmaz. A 6.20. (a) ábra helyfoglalási sémájánál az alapegység a szektor, és a fájl egymást követő szektorokban tárolódik. CD-ROM-ok esetében általános ez a megoldás. A 6.20. (b) ábra olyan sémát mutat, ahol az alapegység szintén a szektor, de egy-egy fájlnak nem feltétlenül egymás utáni szektorokat foglalunk le. Lemezeken általában ez a séma használatos.



6.20. ábra. *Lemezes helyfoglalási stratégiák. (a) Egymást követő szektorokban elhelyezkedő fájl. (b) Nem egymás utáni szektorokban található fájl*

Az operációs rendszer egészen másként tekinti a fájlokat, mint a felhasználói programozó. A programozó szempontjából a fájl bájtok vagy logikai rekordok lineáris sorozata. Az operációs rendszer lemezen elhelyezkedő helyfoglalási egységek rendezett, bár nem feltétlenül egymást követő kollekciójának tekint minden fájlt.

Ha az operációs rendszertől egy fájl *n*. bájtját vagy rekordját kérjük, valamilyen módon meg kell találnia a lemezen. Amennyiben egymás utáni szektorokat foglalt le a fájlnak, a kívánt bájt vagy rekord helyzetének kiszámításához elegendő a fájl kezdetének pozícióját ismernie.

Ha nem sorban foglalt le helyet, tetszőleges bájt vagy logikai rekord pozícióját nem lehet pusztán a fájl kezdőpozíciója alapján meghatározni. Ehhez egy **fájlindex**nek **(file index)** nevezett táblázatra van szükség, amely a helyfoglalási egységeket és tényleges lemezcímeiket tartalmazza. A fájlindex lehet lemezblokkok címeiből álló lista (a UNIX ezt használja) vagy logikai rekordok listája lemezcímükkel és a hozzá tartozó eltolással. Néha minden logikai rekordnak külön **kulcsa (key)** van, így sorszáma helyett kulcsával is hivatkozhatnak rá a programok. Ebben az esetben az utóbbi módszert kell alkalmaznunk: a bejegyzésben nem csak a rekordlemezen elfoglalt helye, hanem a kulcsa is szerepel. A nagygépeken ez a szervezési mód általános.

A fájl helyfoglalási egységeit úgy is megkereshetjük, hogy a fájlt láncolt listaként építjük fel. Minden helyfoglalási egység tartalmazza a rákövetkező címét. A séma hatékony megvalósításának egyik lehetséges útja: az összes követő címből álló táblát mindig a memóriában tartjuk. 64 KB helyfoglalási egységet tartalmazó lemez esetén például az operációs rendszer olyan 64 KB elemű táblát tartana a memóriában, ahol minden elem a rákövetkező indexét tartalmazná. Ha mondjuk a 4., 52. és 19. helyfoglalási egységekben helyezkedne el a fájl, akkor a tábla 4. eleme 52, az 52. eleme 19 és a 19. eleme valamely speciális érték (például 0 vagy –1) lenne a fájl végének jelzésére. Az MS-DOS, a Windows 95 és a Windows 98 fájlrendszerei így működnek. Ezt a fájlrendszert a Windows XP is támogatja, de van a UNIX-hoz hasonlóan működő saját fájlrendszere is.

Idáig párhuzamosan vizsgáltuk az egymás után lefoglalt egységekből álló fájlokat a nem egymás utáni egységekben tároltakkal, de még nem mondtuk meg, miért használják mindkét fajtájukat. Az első típus előnye, hogy könnyebb a blokkok adminisztrációja, viszont ha a maximális fájlméret előre nem ismert, csak ritkán alkalmazható. Ha a fájl a *j*. szektoron kezdődik és az ezutáni szektorokon terjeszkedhet, beleütközhet a *k*. szektoron kezdődő másik fájlba, s így már nem marad helye a további növekedéshez. Amennyiben nem egymást követő szektorokat foglalunk le, ez a helyzet nem okoz gondot, hiszen az első fájl további blokkjait a lemezen bárhol elhelyezhetjük. Ha a lemezen sok növekvő méretű fájl van, és egyikük végső nagysága sem ismert előzetesen, akkor szinte lehetetlen soros helyfoglalással tárolni őket. A meglévő fájlok átmozgatása néha megoldható, de igen költséges.

Ha viszont előre tudjuk a fájlméretek maximumát, mint például CD írásakor, a felíró program a fájlmérettel megegyező hosszúságú szektorsorozatot tud lefoglalni. Ha tehát 1200, 700, 2000 és 900 szektoros fájlokat akarunk a CD-re írni, akkor (a tartalomjegyzéket mellőzve) egyszerűen a 0., 1200., 1900. és a 3900. szektoron kezdheti ezeket. A kezdőszektor ismeretében bármelyik fájl bármelyik részét könnyű megtalálni.

Az operációs rendszer csak úgy tud helyet lefoglalni az új fájlok részére, ha nyomon követi, hogy melyek a szabad blokkok, és melyeket használt már fel fájlok tárolására. A CD-ROM esetén csupán egyetlenegyszer, előzetesen kell elvégeznie ezeket a kalkulációkat. Nem így a lemezes egységeknél, ahol fájlok keletkeznek

Sáv	Szektor	A lyukban található szektorok száma
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

	Szektor											
Sáv	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

6.21. ábra. *A szabad szektorok nyilvántartásának két módszere. (a) Szabad listával. (b) Bittérképpel*

és szűnnek meg állandóan. Az egyik az összes lyukat tartalmazó lista használata. A **lyuk (hole)** itt tetszőleges számú folytonosan elhelyezkedő szabad helyfoglalási egységet jelent. A listát **szabad listának (free list)** szokás nevezni. A 6.20. (b) ábra lemezéhez tartozó szabad listát tünteti fel a 6.21. (a) ábra.

A másik módszernél helyfoglalási egységenként 1 bitet tartalmazó bittérképpel dolgozunk a 6.21. (b) ábra szerint. A 0 bit a szabad, az 1 pedig a már felhasznált helyfoglalási egységeket jelöli.

Az első megoldás előnye, hogy könnyű vele adott hosszúságú lyukat találni. Hátránya, hogy a lista változó méretű.

A lista hosszának fájlok megszűnésével és újak létrehozásával kapcsolatos ingadozása nemkívánatos jelenség. A bittérkép előnye az állandó méret, továbbá az, hogy valamelyik helyfoglalási egység állapotának szabadról foglaltra állítása egyetlen bit megváltoztatását jelenti. Adott méretű blokkot találni azonban már sokkal nehezebb. Mindkét módszer megkívánja a helyfoglalási lista vagy a bittérkép frissítését, ha a fájl számára újabb egységeket foglalunk le vagy felszabadítunk valamennyit.

Mielőtt befejeznénk a fájlrendszer megvalósításának tárgyalását, érdemes néhány szót szólni a helyfoglalási egység méretének választásáról. Itt több tényező is kulcsszerepet játszik. Először is a lemezhozzáférések idejét döntően a keresési és a forgási késleltetés szabja meg. Ha már 10 ms-ot vártunk egy helyfoglalási egységre, sokkal jobb nagyjából 1 ms alatt 8 KB-ot beolvasni, mint 0,125 ms alatt 1 KB-ot, hiszen a 8 KB nyolc 1 KB-os egységben történő beolvasása nyolcszori keresést igényel. Az átvitel hatékonysága tehát a nagy helyfoglalási egységek mellett szól.

A nagyobb egységeket támogatja az is, hogy kis egységeket választva sok lesz belőlük. A sok egység pedig a memóriában tárolandó nagy fájlindexet vagy láncolt listás táblát jelent. Az MS-DOS-nak azért kellett több szektornyi helyfoglalási egységekkel dolgozni, mert a lemezcímeket 16 bites számokként tárolta. Amikor a lemezek mérete meghaladta a 64 KB szektort, ezeket csak nagyobb helyfoglalási egységek használatával lehetett megcímezni, hogy a helyfoglalási egységek száma

ne lépje túl a 64 KB-os határt. A Windows 95 első kiadása ugyanezzel a problémával küszködött, csak a következő kiadástól alkalmaztak 32 bites számokat. A Windows 98 mindkét méretet támogatta.

A kisebb helyfoglalási egységek használatát indokolhatná az a tény, hogy kevés fájl tölt ki pontosan valahány egységet. Ezért majdnem minden fájlnál helyet veszítünk az utolsó egységben. Ha a fájlméret sokkal nagyobb a helyfoglalási egység méreténél, átlagosan fél egységnyi helyet pocsékolunk el. Minél nagyobb az egységek mérete, annál több az eltékozolt hely. Egy 2 GB-os MS-DOS vagy Windows 95 Rel. 1 partíción például a helyfoglalási egység 32 KB volt, tehát egy 100 karakteres fájllal 32 668 bájt lemezterületet tékozoltunk el. A hatékony tárolás kis egységeket kíván. Mindent összevetve manapság a hatékony átvitel a döntő tényező, emiatt egyre nagyobb blokkméretek használatosak.

6.2.3. Könyvtárkezelő utasítások

A régi szép időkben az emberek programjaikat és adataikat dolgozószobájuk szekrényfiókjaiban lyukkártyán őrizték. Ahogy szaporodtak az adatok és nőttek a programok, egyre kellemetlenebbé vált ez a megoldás. Végül megszületett az ötlet, hogy az iratszekrények helyett az adatok és a programok alternatív tároló-eszközeként használjuk a számítógép másodlagos memóriáját (például a lemezegységet). A számítógép szempontjából megkülönböztetünk **on-line** (emberi beavatkozás nélkül közvetlenül elérhető) és **off-line** (csak emberi közreműködéssel hozzáférhető) információt. Az utóbbira példa lehet a megfelelő CD-ROM behelyezése az olvasóba.

Az on-line információt fájlokban tároljuk, a programok a korábban vizsgált B/K utasításokkal férhetnek hozzájuk. De további utasításokra is szükség van az on-line információ nyilvántartásához, megfelelő egységekbe rendezéséhez és az illetéktelen használattól való megvédéséhez.

Az operációs rendszerek az on-line fájlokat általában **könyvtárakba (directories)** csoportosítják. A 6.22. ábra egy könyvtárszervezési példát mutat. Az operációs rendszerek minimálisan a következő funkciókat ellátó rendszerhívásokat tartalmazzák:

1. Fájl létrehozása és könyvtárba helyezése.
2. Fájl törlése könyvtárból.
3. Fájl átnevezése.
4. A fájl védelmi állapotának megváltoztatása.

Különböző védelmi sémák használatosak. Az egyik lehetőség az lehet, hogy minden fájlhoz egy titkos jelszót rendel a tulajdonosa. Ha egy program megpróbál a fájlhoz hozzáférni, meg kell adnia a jelszót, s ennek helyességét ellenőrzi az operációs rendszer a hozzáférés engedélyezése előtt. A másik védelmi módszernél minden fájl tulajdonosa explicit módon felsorolja egy listában, hogy mely felhasználók programjai érhetik el a fájlt.

0. fájl	Fájlnév:	Rubber-ducky	
1. fájl	Hossz:	1840	
2. fájl	Típus:	Anatiadae dataram	
3. fájl	Létrehozás dátuma:	1066. március 16.	
4. fájl	Utolsó hozzáférés:	1492. szeptember 1.	
5. fájl	Utolsó módosítás:	1776. július 4.	
6. fájl	Összes hozzáférés:	144	
7. fájl	0. blokk:	4. sáv	6. szektor
8. fájl	1. blokk:	19. sáv	9. szektor
9. fájl	2. blokk:	11. sáv	2. szektor
10. fájl	3. blokk:	77. sáv	0. szektor

6.22. ábra. *Felhasználói könyvtár és tipikus könyvtári bejegyzés tartalma*

Minden modern operációs rendszer megengedi több felhasználói könyvtár használatát. A tipikus megoldás az, hogy minden könyvtár maga is fájl, s mint ilyen, besorolható egy másik könyvtárba. Ezzel könyvtárfákat kapunk. A több projekten dolgozó programozók számára különösen hasznos ez a lehetőség. Az azonos projektekhez kapcsolódó fájlokat egy-egy könyvtárba gyűjthetik össze. Míg az adott projekten dolgoznak, nem zavarják őket oda nem tartozó fájlok. A könyvtárak arra is kényelmes eszközt nyújtanak, hogy fájlokat osszunk meg a csoportunkhoz tartozókkal.

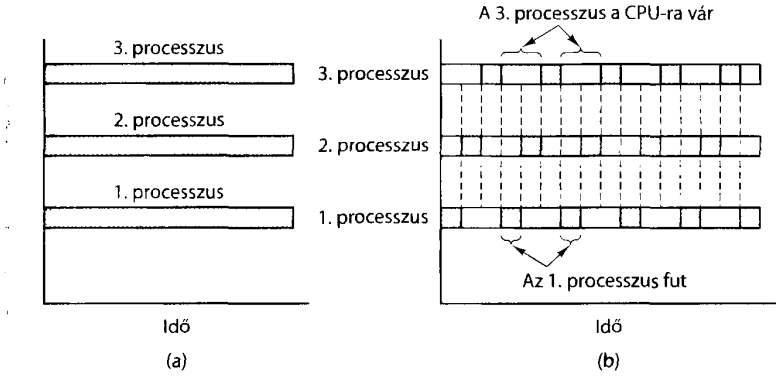
6.3. A párhuzamos feldolgozás virtuális utasításai

Bizonyos számításokat egy processzus helyett két vagy több párhuzamosan (például különböző processzorokon) futó, együttműködő processzusként lehet legkényelmesebben programozni. Máskor meg a teljes végrehajtáshoz szükséges idő csökkentése miatt daraboljuk fel a feladatot párhuzamosan végrehajtható részekre. Processzusok párhuzamos végrehajtásához bizonyos virtuális utasításokra van szükség. Ezekről lesz szó a következő szakaszokban.

A párhuzamos feldolgozás iránti jelenlegi érdeklődést a fizika törvényei is motiválják. Az Einstein-féle speciális relativitáselmélet szerint elektromos jeleket nem lehet a fénysebességnél gyorsabban továbbítani, ami vákuumban közel 1 láb/ns sebességet jelent, rézdrótban vagy optikai szálban ennél kicsit kevesebbet. Ez a korlát a számítógépek felépítésére vonatkozó fontos következményeket eredményez. Ha például a CPU a tőle 1 láb távolságra lévő memóriától kér adatokat, legalább 1 ns-ig tart, míg a kérés megérkezik a memóriához, s legalább még egy nsec telik el a válasz megérkezéséig. A nanoszekundumnál kisebb válaszidejű számítógépek tehát csak nagyon kicsik lehetnek. A számítógépek gyorsításának másik módja a

több CPU-t tartalmazó gépek építése. Ha a gép 1000 darab 1 ns-os CPU-t tartalmaz, számítási teljesítménye megegyezik az 1 CPU-s 0,001 ns ciklusidejűével, de az előbbit valószínűleg sokkal könnyebb és olcsóbb megépíteni.

A több CPU-s gépeken az együttműködő processzusok mindegyikéhez saját CPU-t rendelhetünk, ezzel biztosítva egyidejű végrehajtásukat. Ha csak egy processzorunk van, a párhuzamos feldolgozást úgy szimulálhatjuk, hogy a processzorral felváltva futtatjuk egy-egy kis ideig a processzusokat. Más szóval a processzor megosztható több processzus között.



6.23. ábra. *(a) Több CPU-s valódi párhuzamos feldolgozás. (b) Szimulált párhuzamos feldolgozás az egyetlen CPU három processzus közti megosztásával*

A 6.23. ábra a több processzorral végzett valódi párhuzamos feldolgozás és az egy fizikai processzorral szimulált párhuzamosság közti különbséget mutatja. Még a szimulált párhuzamos feldolgozás esetén is célszerű úgy felfogni, mintha minden processzusnak saját dedikált virtuális processzora lenne. Így a szimulált esetben is ugyanazokkal a kommunikációs problémákkal kell szembenéznünk, mint amelyek a valódi párhuzamos feldolgozásnál előfordulnak.

6.3.1. Processzusok létrehozása

A végrehajtandó programok mindig valamely processzus részeként futnak. Ez a processzus az összes többihez hasonlóan, állapotával és azzal a címtartománnyal jellemezhető, melyen keresztül a program és adatai elérhetők. Az állapot minimálisan az utasításszámlálót, a programállapotszót, a veremmutatót és az általános regisztereket tartalmazza.

A legtöbb modern operációs rendszer megengedi processzusok dinamikus létrehozását és befejezését. Új processzust létrehozó rendszerhívásra van szükség ahhoz, hogy ezt a sajátosságot teljesen ki tudjuk használni a párhuzamos feldolgozás elérésére. A rendszerhívás vagy csak egyszerű másolatot készít a hívóról, vagy

esetleg azt is megengedi, hogy a hívó határozza meg az új processzus kezdeti állapotát, beleértve a futtatandó programot, annak adatait és kezdőcímét.

Néha a létrehozó „szülő" processzus részben vagy teljesen ellenőrzése alatt tartja a létrehozott „gyerek" processzust. Ezt a célt szolgálják azok a virtuális utasítások, melyekkel a szülő a gyerek processzusok megállítását, újraindítását, megvizsgálását és befejezését érheti el. Máskor meg a szülőnek sokkal kevesebb befolyása van a gyerek processzusokra: ha létrejött az új processzus, a szülő már nem tudja erőszakkal megállítani, újraindítani, megvizsgálni vagy befejezni. A két processzus egymástól függetlenül fut tovább.

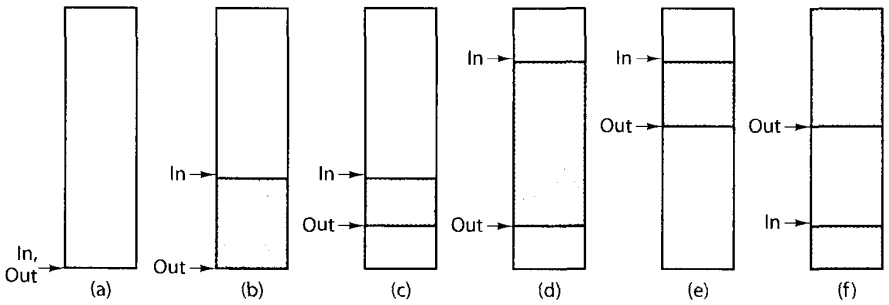
6.3.2. Versenyhelyzetek

A párhuzamosan futó processzusok feladataikat sokszor csak úgy képesek megoldani, ha kommunikálnak és működésüket szinkronizálják. Ebben a szakaszban processzusok szinkronizációját vizsgáljuk, az ezzel kapcsolatos bonyodalmakat egy példa kapcsán részletezzük. A következő szakasz ismerteti a problémák egyik lehetséges megoldását.

Tekintsünk két független processzust, melyek a memóriában található közös puffer segítségével kommunikálnak. Az egyszerűség kedvéért az 1. processzust **termelő**nek **(producer)**, a 2. processzust pedig **fogyasztó**nak **(consumer)** fogjuk nevezni. A termelő prímszámokat keres, és egyesével a pufferbe helyezi. A fogyasztó szintén egyesével kiveszi és kinyomtatja a pufferben található számokat.

A két processzus párhuzamosan, de eltérő sebességgel fut. Ha a termelő észreveszi, hogy a puffer betelt, „elalszik", vagyis a fogyasztótól érkező jelzésig (signal) felfüggeszti működését. Miután kivett egy prímet a pufferből, a fogyasztó jelzést küld a termelőnek – felébreszti –, hogy újraindítsa. Hasonló a helyzet, ha a fogyasztó veszi észre, hogy kiürült a puffer; ő is elalszik. A termelő az üres pufferbe betett első szám után ébreszti fel az alvó fogyasztót.

A példában körkörös puffert használunk a processzusok közti kommunikációra. Az *in* és az *out* mutatókat a következő módon használjuk: *in* a következő szabad szóra (ide teszi be a termelő a következő prímet), *out* meg a fogyasztó által leg-



6.24. ábra. *Körkörös puffer használata*

közelebb kivehető számra mutat. Ha *in* = *out*, a puffer üres, ez látható a 6.24. (a) ábrán. Miután a termelő néhány új prímet generált, a 6.24. (b) ábrának megfelelő helyzet áll elő. A 6.24. (c) ábrán azt látjuk, hogy a fogyasztó az előbbi számok közül néhányat már kinyomtatott. A 6.24. (d)–(f) ábrák a processzusok további tevékenységének hatását mutatják. A puffer teteje logikailag az aljánál folytatódik, vagyis körkörös. Ha egy hirtelen beérkező nagy adag input miatt a puffer „átfordul", és *in* csak egy szóval van *out* mögött (például *in* = 52, *out* = 53), a puffer (majdnem) betelt. Az utolsó szót nem használjuk, hiszen egyébként nem tudnánk eldönteni, hogy az *in* = *out* eset üres vagy teli puffert jelent.

A 6.25. ábrán a termelő-fogyasztó probléma Java nyelvű egyszerű megvalósítása látható. A megoldás három osztályt használ: *m*, *producer* és *consumer*. Az *m* (main) osztály tartalmazza a konstansok definícióját, az *in* és az *out* mutatókat és a 100 prím tárolására alkalmas puffert, melyet *buffer*[0]-tól *buffer*[99]-ig indexezünk.

A program Java **szálak**at **(threads)** használ a párhuzamos processzusok szimulálására. A megoldásban egy *producer* és egy *consumer* osztály szerepel, ezekből hozunk létre egy *p*, illetve egy *c* nevű példányt. Mindkét osztályt a *Thread* nevű alaposztályból származtatjuk, amely tartalmazza a *run* metódust. A *run* metódus tartalmazza a szálak kódját. Ha meghívjuk a *Thread* osztályból származtatott valamelyik objektum *start* eljárását, új szál indul el.

A szálak a processzusokhoz hasonlítanak, azzal a kivétellel, hogy az ugyanazon Java-programhoz tartozó összes szál ugyanabban a címtartományban fut. Ez a tulajdonság kényelmessé teszi a megosztott pufferek alkalmazását. Ha a gépben kettő vagy több CPU van, az ütemező minden szálat másik CPU-n futtathat, ami valódi párhuzamosságot tesz lehetővé. Ha csak egy CPU van, a szálak ezen időosztással hajtódnak végre. Bár a Java csak a párhuzamos szálakat, és nem az igazi párhuzamos processzusokat támogatja, a továbbiakban is termelő és fogyasztó processzusokról beszélünk, mivel valójában ezek érdekelnek bennünket.

A *next* segédfüggvény az *in* és az *out* egyszerű növelését segíti. Így nem kell minden alkalommal azt vizsgálni, hogy „átfordult-e" már valamelyik értéke. Ha a *next*-nek átadott paraméter 98, vagy annál kisebb, a következő nála nagyobb egészet adja vissza. Amikor a paraméter 99, a puffer végéhez értünk, ezért a függvény 0-t ad vissza.

Szükségünk van valamire, amivel mindkét processzus „elaltatható", ha nem tudja folytatni a működését. A Java tervezői is gondoltak erre, már a nyelv legelső verziója tartalmazta a *Thread* osztály *suspend* (felfüggeszt, megállít) és *resume* (felébreszt) metódusait. Ezeket használtuk a 6.25. ábrán.

Most érünk a termelő és a fogyasztó kódjához. Először a P1 utasítással új prímet generál a termelő. Figyeljük meg az *m.MAX_PRIME* konstans használatát. Az *m.* prefix jelzi, hogy az *m* osztályban definiált *MAX_PRIME*-ról van szó. Ugyanezért kell használnunk *m*-et az *in*, az *out*, a *buffer* és a *next* mellett is.

A P2 utasításban a termelő az nézi meg, hogy *in* az *out* mögött van-e. Ha igen (például *in* = 62 és *out* = 63), akkor a puffer megtelt, ezért P2-ben a *suspend*-et hívja és elalszik. Ha van még hely a pufferben, akkor új prímet rak bele (P3), és megnöveli *in* értékét (P4). Ha *in* új értéke eggyel nagyobb *out*-nál (például *in* = 17 és *out* = 16), akkor az inkrementálás előtt *in* és *out* értéke azonos volt. Ebből arra

```
public class m {
   final public static int BUF_SIZE = 100;                    // a buffer 0-tól 99-ig változik
   final public static long MAX_PRIME = 100000000000000L; // itt álljunk meg
   public static int in = 0, out = 0;                         // adatpointerek
   public static long buffer [ ] = new long[BUF_SIZE];        // a prímeket itt tároljuk
   public static producer p;                                  // a termelő neve
   public static consumer c;                                  // a fogyasztó neve

   public static void main(String args[ ]) {                  // a main osztály
      p = new producer( );                                    // a termelő létrehozása
      c = new consumer( );                                    // a fogyasztó létrehozása
      p.start( );                                             // a termelő elindítása
      c.start( );                                             // a fogyasztó elindítása
   }
   // Ez a segédfüggvény cirkulárisan növeli meg in és out értékét
   public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return (0);}
}

class producer extends Thread {                               // a termelő osztály
   public void run( ) {                                       // a termelő kódja
      long prime = 2;                                         // segédváltozó

      while (prime < m.MAX_PRIME) {
         prime= next_prime(prime);                            // P1 utasítás
         if (m.next(m.in) == m.out) suspend( );               // P2 utasítás
         m.buffer[m.in] = prime;                              // P3 utasítás
         m.in = m.next(m.in);                                 // P4 utasítás
         if (m.next(m.out) == m.in) m.c.resume( );            // P5 utasítás
      }
   }

   private long next_prime(long prime){ ... }                 // a következő prímet kiszámító
}                                                             // függvény


class consumer extends Thread {                               // a fogyasztó osztály
   public void run( ) {                                       // a fogyasztó kódja
      long emirp = 2;                                         // segédváltozó

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );                       // C1 utasítás
         emirp = m.buffer[m.out];                             // C2 utasítás
         m.out == m.next(m.out);                              // C3 utasítás
         if (m.out == m.next(m.next(m.in))) m.p.resume( );    // C4 utasítás
         System.out.println(emirp);                           // C5 utasítás
      }
   }
}
```

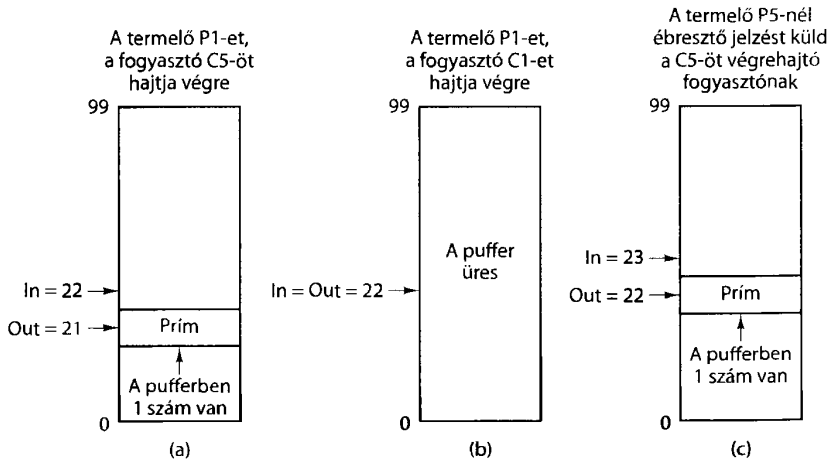
6.25. ábra. *Párhuzamos feldolgozás végzetes versenyhelyzettel*

következtet, hogy a puffer üres volt, s emiatt a fogyasztó még most is alszik. Ezért a *resume* hívásával felébreszti (P5). Végül elkezdi keresni a következő prímet.

A fogyasztó programja hasonló felépítésű. Először azt teszteli, hogy üres-e a puffer (C1). Ha igen, a fogyasztónak semmi dolga, ezért nyugovóra tér. Ha a puffer nem üres, kiveszi a következő kiírandó prímet (C2) és megnöveli *out*-ot (C3). Ha a C4-hez érve *out* két pozícióval előbbre mutat, mint *in*, akkor a növelés előtt pontosan egy hellyel járt előtte. Mivel az *in* = *out* – 1 érték a „teli puffer" feltételnek felel meg, a termelő bizonyára alszik,* tehát a fogyasztó felébreszti a *resume* hívással. Befejezésül a C5 utasítás kiírja a számot, s ezzel újra kezdődik a ciklus.

Sajnos ez a konstrukció egy végzetes hibát tartalmaz, melyet a 6.26. ábra tár fel. Ne feledjük, hogy a két processzus aszinkron módon, különböző, sőt esetleg időben változó sebességgel fut. Tekintsük azt a 6.26. (a) esetet, amikor a pufferben már csak egy szám maradt a 21. helyen, *in* = 22 és *out* = 21. A termelő a P1 utasítással újabb prímet keres, a fogyasztó a 20. helyen talált szám kiírásával van elfoglalva (C5). Miután befejezte a kiírást, a fogyasztó végrehajtja a C1 tesztet, C2-vel kiveszi a pufferből az utolsó számot, majd megnöveli *out*-ot. Ebben a pillanatban mind *in*, mind *out* értéke 22. A fogyasztó kiírja a számot, ezután C1-gyel előveszi *in*-t és *out*-ot a memóriából, hogy összehasonlítsa őket [lásd 6.26. (b) ábra].

Ugyanebben a pillanatban, amikor a fogyasztó már elővette a két értéket, de még nem hasonlította össze őket, a termelő megtalálja a következő prímet. P3-nál beteszi a pufferbe, P4-nél megnöveli *in*-t. Most *in* = 23 és *out* = 22. A P5-nél a program észreveszi, hogy *in* = *next*(*out*). Más szóval *in* eggyel nagyobb *out*-nál, ami azt jelzi, hogy most 1 elem van a pufferben. Ebből (helytelenül) arra követ-



6.26. ábra. *A termelő és a fogyasztó hibás kommunikációs mechanizmusa*

* Esetleg dolgozhat P1-en. *(A lektor)*

keztet a termelő, hogy a fogyasztó biztosan alszik, ezért a 6.26. (c)-nek megfelelően ébresztő jelzést küld neki (meghívja a *resume* eljárást). Természetesen a fogyasztó még mindig ébren van, tehát elvész az ébresztő jelzés. A termelő nekilát a következő prím keresésének.

Közben a fogyasztó is tovább dolgozik. Már elővette *in*-t és *out*-ot a memóriából, mielőtt a termelő az utolsó számot a pufferbe helyezte volna. Mivel mindkettő értéke 22, elalszik. Amikor a termelő meghatározza a következő prímet (P1), látja, hogy beteheti a pufferbe (P2), beteszi a pufferbe (P3), ellenőrzi a két mutatót (P4), és úgy találja, hogy *in* = 24 és *out* = 22. Ebből arra következtet, hogy a pufferben két szám van (ami igaz is), és a fogyasztó nem alszik (ez viszont nem igaz). A termelő folytatja a ciklust, végül a puffer megtelik, ekkor ő is elalszik. Mivel mindkét processzus alvó állapotba került, a továbbiakban így is maradnak.

A gondot az okozta, hogy a termelő éppen a két időpont között (amikor a fogyasztó betöltötte *in*-t és *out*-ot, illetve amikor elaludt) vette észre, hogy *in* = *out* + 1, ezért gondolta azt, hogy a fogyasztó alszik (ami még nem volt igaz), és küldött el egy olyan ébresztő jelzést, amely elveszett, hiszen a fogyasztó még ébren volt. Ez a probléma **versenyhelyzet (race condition)** néven ismert, hiszen az algoritmus sikeressége azon múlik, hogy *out* megnövelése után ki nyeri az *in* és *out* értékének tesztelésééért vívott „versenyt".

A versenyhelyzetekkel kapcsolatos problémák jól ismertek. Ezek valóban annyira súlyosak, hogy néhány évvel a Java kibocsátása után a Sun megváltoztatta a *Thread* osztályt, nem támogatottnak minősítve a *suspend* és a *resume* hívását, mivel igen gyakran versenyhelyzeteket eredményeztek. Helyettük a nyelv sajátosságaitól függő megoldást ajánlottak, de mivel mi az operációs rendszereket vizsgáljuk, ezért inkább egy olyan másik megoldással foglalkozunk, amelyet sok operációs rendszer támogat, beleértve a UNIX-ot és a Windows XP-t is.

6.3.3. Processzusok szinkronizációja szemaforokkal

A versenyhelyzet legalább két módon oldható fel. Az egyiknél minden processzushoz hozzárendelünk egy „ébresztésre várakozó állapotbitet". Ha egy még futó processzus kap ébresztő jelzést, beállítjuk ezt a bitet. Amennyiben a processzus úgy alszik el, hogy ez a bit be van állítva, akkor rögtön újra is indul, a várakozó bit pedig törlődik. Ez a bit megőrzi a fölösleges ébresztő jelzést a jövőbeli felhasználáshoz.

Igaz, hogy két processzus esetén ezzel a módszerrel megkerülhető a versenyhelyzet, de az általános esetben, n kommunikáló processzusnál már nem alkalmazható, mert ekkor néha már akár $n - 1$ ébresztő jelzést is meg kellene őrizni. Megtehetjük, hogy minden processzusban $n - 1$ bitet tartunk fenn erre a célra, de ez elég ügyetlen megoldás.

Párhuzamos processzusok szinkronizálására Dijkstra (1968b) ajánlott egy általánosabb módszert. A memóriában valahol **szemafor**nak (**semaphore**) nevezett nem negatív, egész értékű változókat tárolunk. Az operációs rendszer két rendszerhívása, az up és a down kezeli a szemaforokat. Az up 1-gyel növeli, a down 1-gyel csökkenti a szemafor értékét.

Ha pozitív értékű szemaforon hajtjuk végre a down műveletet, a szemafor értéke 1-gyel csökken, és a hívó processzus tovább fut. A 0 értékű szemaforon nem hajtható végre a down, a hívó processzus várakozik („elalszik"), és alva is marad, míg egy másik processzus up hívást nem hajt végre ugyanazon a szemaforon. Az alvó processzusokat általában egy sorba rendezzük, hogy nyomon követhetők legyenek.

Az up utasítás azt ellenőrzi, hogy a szemafor 0-e. Ha igen, és egy másik alvó processzus tartozik hozzá, akkor a szemafor 1-gyel nő. Az alvó processzus ezután be tudja fejezni a down műveletet, amely megakasztotta, újra 0-ra tudja állítani a szemafort és mindkét processzus futása folytatódhat. A nem nulla szemaforon végrehajtott up utasítás egyszerűen 1-gyel növeli értékét. A szemafor lényegében olyan számláló, amely a későbbi felhasználásig tárolja az ébresztő jelzéseket, hogy ne vesszenek el. A szemafor utasítások fontos tulajdonsága, hogy ha egy processzus valamelyik szemaforon utasítás végrehajtását kezdeményezte, akkor azt a szemafort egyetlen másik processzus sem érheti el mindaddig, míg az első processzus vagy be nem fejezte a műveletet, vagy alvó állapotba nem került, mivel 0 állásnál próbált down műveletet végrehajtani. A 6.27. ábra az up és a down rendszerhívások lényeges tulajdonságait összegzi.

Utasítás	Szemafor = 0	Szemafor > 0
Up	Szemafor = Szemafor + 1; ha egy másik processzus várakozott a szemafornál a down művelet befejezésére, most befejezheti a down-t és futhat tovább	Szemafor = Szemafor + 1
Down	A processzus várakozik, míg egy másik processzus up-ot nem hajt végre	Szemafor = Szemafor – 1

6.27. ábra. *A szemafor műveletek hatása*

Mint korábban említettük, a Java nyelvi sajátosságokon alapuló megoldást kínál a versenyhelyzetek kezelésére, mi viszont az operációs rendszereket vizsgáljuk. Ezért meg kell oldanunk a szemaforok kezelését Javában, bár sem a nyelv, sem standard osztályai nem tartalmaznak ilyesmit. Feltesszük, hogy megírtunk két natív eljárást (*up* és *down*), melyek az up és a down rendszerhívásokat hajtják végre. Java-programjainkban a szemaforok használatát ezen eljárások közönséges egész paraméterekkel való hívásával programozhatjuk.

A 6.28. ábra mutatja a versenyhelyzet kiküszöbölését szemaforok használatával. Az *m* osztályhoz két szemafort adtunk. Az *available* nevű kezdőértéke 100 (a pufferméret), a *filled*-é pedig 0. A termelő és a fogyasztó a végrehajtást a korábbiakhoz hasonlóan kezdi el a 6.28. ábrán látható P1, illetve a C1 utasítással. A *filled*-en végrehajtott *down* hívás azonnal megállítja a fogyasztót. Az első prím megtalálása után a termelő az *available* paraméterrel hívja *down*-t, ami *available*-t 99-re állítja. P5-nél *up*-ot hívja a *filled* paraméterrel, amitől *filled* értéke 1-re változik. Ezzel felszabadítja a fogyasztót, aki be tudja fejezni a félbemaradt *down* hívást. Most *filled* értéke 0, és fut mindkét processzus.

Vizsgáljuk meg újra a versenyhelyzetet. Legyen valamely időpontban *in* = 22, *out* = 21, a termelő hajtsa végre P1-et, a fogyasztó C5-öt. A fogyasztó az utasítás

befejeződése után C1-gyel folytatja, ahol *down*-t hívja a *filled* argumentummal. A szemafor értéke a hívás előtt 1, utána 0. Ezután kiveszi az utolsó számot a pufferből, és 100-ra növeli *available*-t. Még mielőtt *down*-t hívná a fogyasztó, a termelő megtalálja a következő prímet, és gyors egymásutánban végrehajtja P2, P3 és P4 utasításokat.

Ebben a pillanatban *filled* 0. A termelő éppen növelné, a fogyasztó meg éppen a *down*-t hívná. Ha először a fogyasztó utasítása hajtódik végre, akkor leáll, míg a termelő az *up* hívással újra el nem engedi. Ha viszont a termelő hívása jön először, akkor 1-re állítja a szemafort, s így a fogyasztónak nem is kell várakoznia. Egyik esetben sem veszett el ébresztési jelzés. A szemaforok bevezetésével pontosan ez volt a fő célunk.

A szemafor műveletek lényeges tulajdonsága *oszthatatlanságuk*. Miután egy művelet végrehajtása megkezdődött, a szemafort egyetlen másik processzus sem használhatja, amíg az első processzus vagy be nem fejezte a műveletet, vagy el nem aludt. A szemaforok alkalmazásával ébresztő jel nem veszhet el. Ezzel szemben a 6.25. ábra if utasítása nem oszthatatlan. A feltétel kiértékelése és a kiválasztott utasítás végrehajtása között egy másik processzus küldhet ébresztő jelzést.

Valójában az *up* és a *down* eljárások által végrehajtott up és down rendszerhívások oszthatatlanná nyilvánításával meg is szüntettük a processzusok szinkronizációs problémáit. Ahhoz, hogy ezek a műveletek tényleg oszthatatlanok legyenek, az operációs rendszernek meg kell tiltania, hogy két vagy több processzus egyszerre használja ugyanazt a szemafort. Minimálisan azt kell garantálnia, hogy egy up vagy down rendszerhívás kiadása után semmilyen felhasználói kód ne futhasson, míg a hívás be nem fejeződött. Egy processzoros rendszereken a szemaforokat néha a szemafor műveletek alatti megszakítások letiltásával valósítják meg. Több processzoros rendszereken ez a trükk nem működik.

Tetszőleges számú processzusra alkalmazható a szemaforokon alapuló szinkronizációs technika. A down rendszerhívás végrehajtására ugyanannál a szemafornál több processzus is várakozhat. Ha egy másik processzus végre up hívást hajt végre ezzel a szemaforral, az egyik várakozó processzus befejezheti a down hívást, és tovább futhat. A szemafor értéke 0 marad, a többi processzus tovább várakozik.

Próbáljuk egy analógiával megvilágítani a szemaforok jellegzetességeit. Képzeljük el, hogy egy kiránduláson a 20 részt vevő röplabdacsapattal 10 pályán 10 mérkőzést (processzust) szervezünk. A labdákat egy nagy kosárban tartjuk (ez a szemafor). Sajnos összesen csak 7 labdánk van. Bármely időpillanatban 0 és 7 közti számú labda található a kosárban (a szemafor értéke 0 és 7 közötti szám). Az up-nak az felel meg, hogy beteszünk egy labdát a kosárba, hiszen ez növeli a szemafor értékét. Labda kivétele a kosárból egy down végrehajtását jelenti, mivel ez csökkenti az értéket.

A piknik kezdetén minden pályáról egy játékost a kosárhoz küldenek labdáért. Közülük hétnek sikerül labdát szerezni (befejezni a down műveletet), hárman várakozni kényszerülnek (vagyis nem tudják befejezni a down-t). Az ő meccseik átmenetileg elhalasztódnak. Előbb-utóbb a többi mérkőzés közül befejeződik valamelyik, ekkor visszatesznek egy labdát a kosárba (up művelet). A három várakozó játékos közül az egyik labdához jut (befejezi a down-t), s egy mérkőzés folytatód-


```
public class m {
   final public static int BUF_SIZE = 100;                    // a buffer 0-tól 99-ig változik
   final public static long MAX_PRIME = 100000000000L         // itt álljunk meg
   public static int in = 0, out = 0;                         // adatpointerek
   public static long buffer[ ] = new long[BUF_SIZE]:         // a prímeket itt tároljuk
   public static producer p;                                  // a termelő neve
   public static consumer c;                                  // a fogyasztó neve
   public static int filled = 0, available = 100;             // szemaforok

   public static void main(String args[ ]) {                  // a main osztály
      p = new producer( );                                    // a termelő létrehozása
      c = new consumer( );                                    // a fogyasztó létrehozása
      p.start( );                                             // a termelő elindítása
      c.start( );                                             // a fogyasztó elindítása
   }
   // Ez a segédfüggvény cirkulárisan növeli meg in és out értékét
   public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return(0);}
}

class producer extends Thread {                               // a termelő osztály
   native void up(int s); native void down(int s);            // a szemaforok metódusai
   public void run( ) {                                       // a termő kódja
      long prime = 2;                                         // segédváltozó

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // P1 utasítás
         down(m.available);                                   // P2 utasítás
         m.buffer[m.in] = prime;                              // P3 utasítás
         m.in = m.next(m.in);                                 // P4 utasítás
         up(m.filled);                                        // P5 utasítás
      }
   }

private long next_prime(long prime){ ... }                    // a következő prímet kiszámító
}                                                             // függvény

class consumer extends Thread {                               // a fogyasztó osztály
   native void up(int s); native void down(int s);            // a szemaforok metódusai
   public void run( ) {                                       // a fogyasztó kódja
      long emirp = 2;                                         // segédváltozó

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                                      // C1 utasítás
         emirp = m.buffer[m.out];                             // C2 utasítás
         m.out = m.next(m.out);                               // C3 utasítás
         up(m.available);                                     // C4 utasítás
         System.out.println(emirp);                           // C5 utasítás
      }
   }
}
```

6.28. ábra. *Szemaforokkal irányított párhuzamos feldolgozás*

hat. A másik kettőre majd csak akkor kerülhet sor, ha további két labdát hoznak vissza a kosárba. Amikor még két labdát visszajuttatnak, az utolsó két mérkőzés is elindulhat.

6.4. Példák operációs rendszerekre

Ebben a részben tovább vizsgáljuk a példaként használt rendszereket, a Pentium 4-et és az UltraSPARC III-at. Mindkét processzornál megnézünk egy-egy operációs rendszert. A Pentium 4-nél ez a Windows XP lesz (a továbbiakban röviden XP-ként emlegetjük), az UltraSPARC III-on a UNIX-ot használjuk. Mivel a UNIX egyszerűbb és sok tekintetben elegánsabb is, ezzel kezdjük. Egyébként a UNIX-ot tervezték és valósították meg előbb, és nagy hatással volt az XP-re. Emiatt is célszerűbb az említett sorrend, mint a fordítottja.

6.4.1. Bevezetés

Ebben a szakaszban röviden ismertetjük a két operációs rendszert, a UNIX-ot és az XP-t. Figyelmünket a történeti vonatkozásokra, a rendszerek felépítésére és a rendszerhívásokra összpontosítjuk.

UNIX

A UNIX-ot az 1970-es évek elején a Bell Labs-ben fejlesztették ki. Az első változatot Ken Thompson írta meg assembly nyelven egy PDP–7 minigépre. Ezt nemsokára követte a PDP–11-re készült változat, amely már a Dennis Ritchie által kitalált és megvalósított, C-nek hívott új nyelven készült el. Határkő volt Ritchie és kollégája, Thompson 1974-es UNIX-ről szóló cikke (Ritchie és Thompson, 1974). A benne ismertetett eredményeikért később megkapták az ACM tekintélyes Turing díját (Ritchie, 1984 és Thompson, 1984). A cikk megjelenése sok egyetemet arra ösztönzött, hogy UNIX-másolatokat kérjen a Bell Labs-tól. Mivel a Bell Labs anya vállalata, az AT&T, abban az időben a monopólium ellenes törvények hatálya alatt állt, s emiatt nem foglalkozhatott számítógépes üzlettel, nem ellenezte, hogy szerény összegekért UNIX-licenceket kapjanak az egyetemek.

A történelmet formáló nagy véletlenek egyike, hogy akkoriban a PDP–11 volt az egyetemek informatikai részlegeinek legkedveltebb számítógépe, s ráadásul a vele szállított operációs rendszereket mind a hallgatók, mind az oktatók szörnyen pocséknak tartották. A UNIX gyorsan benyomult ebbe az űrbe, nem utolsósorban azért, mert teljes forráskóddal adták. Így aztán állandóan lehetett bütykölgetni.

A korán UNIX-ot szerző egyetemek egyike volt a University of California (Berkeley). Mivel rendelkezésükre állt a teljes forráskód, Berkeley-ben lényeges módosításokat hajtottak végre a rendszeren. Az első változtatás a VAX minigépre való portolás,* majd a lapozott virtuális memória bevezetése, a fájlnevek hosszának 14-ről 255 karakterre növelése és a TCP/IP hálózati protokoll beépítése volt, melyet ma már az egész internet használ (nem kis részben azért, mert benne volt a Berkeley UNIX-ban).

Amíg Berkeley-ben ezeken a változtatásokon dolgoztak, az AT&T maga is tovább fejlesztette a UNIX-ot. Ennek eredménye lett 1982-ben a System III, azután 1984-ben a System V. Az 1980-as évek végére széleskörűen elterjedt két inkompatibilis UNIX-változat: a Berkeley UNIX és a System V. Emellett az is megosztotta a UNIX-világot, hogy nem léteztek szabványos bináris formátumok, ami aláásta a UNIX üzleti sikerét. A szoftvergyártók nem tudtak olyan UNIX-programcsomagokat készíteni, amelyek minden UNIX-rendszeren futottak volna, holott az MS-DOS-nál ez már rutinszerűen működött. Hosszas civódások után az IEEE Szabványügyi Bizottsága kidolgozta a **POSIX (Portable Operating System-IX)** szabványt, amely az IEEE szabvány sorszáma alapján P1003-ként is ismeretes. Később nemzetközi (ISO) szabvánnyá vált.

A POSIX szabvány sok részből áll, melyek a UNIX különböző részeit ölelik fel. Az első rész, P1003.1, a rendszerhívásokat definiálja. A második, P1003.2 az alapvető segédprogramokat írja le stb. A P1003.1 szabvány mintegy 60 olyan rendszerhívást definiál, amelyet minden, a szabványnak megfelelő operációs rendszer támogatni köteles. Ezek a fájlokat író és olvasó, az új processzusokat létrehozó stb. alapvető rendszerhívások. Ma már csaknem minden UNIX-rendszer támogatja a P1003.1 rendszerhívásokat. De sok UNIX-rendszerben vannak további kiegészítő hívások is, különösen a System V-ben és/vagy a Berkeley UNIX-ban definiáltak. Ezek tipikusan még vagy 200-zal bővítik a rendszerhívások halmazát.

A könyv szerzője 1987-ben közzétette a MINIX-nek nevezett kis UNIX-változat forráskódját egyetemi használatra (Tanenbaum, 1987). Helsinkiben az egyetemen MINIX-et tanuló, és azt otthoni PC-jén futtató diákok egyike volt Linus Torvalds. A MINIX alapos megismerése után Torvalds úgy döntött, hogy ír egy saját MINIX-klónt, melyet Linuxnak nevez el. Ez egészen népszerűvé vált. A MINIX és a Linux összeillenek, és rájuk is igaz majdnem minden, amit ebben a fejezetben a UNIX-ról elmondunk. Bár mindezek a UNIX-változatok belülről erősen különbözők, a fejezet nagy részében a rendszerhívásoknak megfelelő felületet (interfészt) vizsgáljuk, amely mindegyikben megtalálható.

Az UltraSPARC III **Solaris**nak nevezett operációs rendszere a System V-on alapul. Támogatja a Berkeley UNIX sok rendszerhívását is.

A Solaris rendszerhívásainak funkció szerinti vázlatos osztályozása a 6.29. ábrán látható. A fájl- és a könyvtárkezelő rendszerhívások alkotják a legnagyobb és a legfontosabb kategóriát. Többségüket a P1003.1 szabvány definiálja. A többiek viszonylag nagy hányada a System V-ből származik.

A hálózatkezelés az a terület, amelyet a System V helyett inkább a Berkeley UNIX keretében fejlesztettek ki. Itt vezették be a **socket** (kb. foglalat, csatlako-

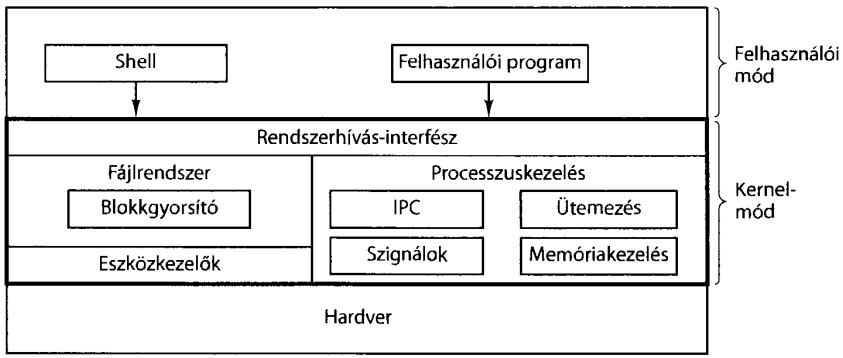
* portolás: egy programnak az adott típusú gépen futtatható változatának létrehozása *(A lektor)*

Kategória	Néhány példa
Fájlkezelés	Fájlmegnyitás, -olvasás, -írás, -zárás és -zárolás
Könyvtárkezelés	Könyvtárak létrehozása és törlése, fájlok mozgatása
Processzuskezelés	Processzusindítás, -leállítás, -nyomkövetés, szignálküldés
Memóriakezelés	A memória megosztása a processzusok között, lapok védelme
Paraméterlekérdezés/-beállítás	Felhasználó-, csoport- és processzusazonosító-lekérdezés; prioritásbeállítás
Dátum- és időkezelés	Fájl elérési idejének beállítása, intervallumos időzítők használata, időzített végrehajtás
Hálózatkezelés	Kapcsolat létrehozása/elfogadása, üzenetküldés/-fogadás
Egyebek	Könyvelés (accounting), lemezkvóta-kezelés, rendszer-újraindítás

6.29. ábra. *A UNIX-rendszerhívások vázlatos osztályozása*

zási pont) fogalmát, amely a hálózati kapcsolat végpontját jelenti. Modelljéül a négylyukú fali telefoncsatlakozó szolgált. A UNIX-processzusok socketeket hozhatnak létre, kapcsolódhatnak socketekhez vagy távoli gépen lévő sockettel is kapcsolatot teremthetnek. Ezen a kapcsolaton át kétirányú adatcserét hajthatnak végre, általában a TCP/IP protokollt felhasználva. Mivel ez a hálózati technológia stabil, kiérlelt és a UNIX évtizedek óta tartalmazza, az internet szervereinek jelentős része UNIX-ot futtat.

Nehéz részletesebben beszélni az operációs rendszer szerkezetéről, hiszen sok UNIX-implementáció létezik, és valamilyen értelemben mindegyikük eltér az összes többitől. A 6.30. ábra azonban legtöbbjükre alkalmazható. Legalul az eszközkezelők rétege található, amely a csupasz hardvert rejti el a fájlrendszer elől. Eredetileg minden eszközkezelőt az összes többitől független egységként írtak meg. Ez az elrendezés sok fölösleges erőfeszítést kívánt, hiszen sokuknak kell foglalkozni a vezérlési folyamattal, hibakezeléssel, prioritásokkal, az adatok és a vezérlőjelek elkülönítésével stb. Ez a megfigyelés indította Dennis Ritchie-et a **streams (folyamok)** nevű keretrendszer kidolgozására, amely moduláris eszközkezelő-készítést tesz lehetővé. A stream használatával a felhasználói processzus és a hardvereszköz között olyan kétirányú kapcsolat hozható létre, amely mentén egy vagy több modul illeszthető be. A felhasználó processzusa adatokat helyez el a streambe, amelyeket azután az egyes modulok transzformálnak és feldolgoznak, míg csak el nem érik a hardvert. A bejövő adatok a fordított feldolgozáson esnek át.



6.30. ábra. *A tipikus UNIX-rendszer felépítése*

Az eszközkezelők felett a fájlrendszer helyezkedik el. Ez a fájlnevekkel, a könyvtárakkal, a lemezblokkok foglalásával, védelemmel és még sok mással foglalkozik. A fájlrendszer része a **blokkgyorsító (block cache)**, amely a lemezről legutóbb beolvasott blokkokat tárolja, hátha a közeljövőben újra szükség lesz rájuk. Az évek során változatos fájlrendszereket használtak, például a Berkeley UNIX Fast File System (FFS) nevű rendszerét (McKusick és társai, 1984) és különféle naplózó fájlrendszereket (Rosenblum és Ousterhout, 1991; Seltzer és társai, 1993).

A UNIX-rendszer következő alkotórésze a processzusokat kezelő alrendszer. Sok egyéb funkciója mellett kezeli az **IPC**-t **(Inter Process Communication, processzusok közti kommunikáció)**. Az IPC teszi lehetővé a processzusok közti kommunikációt és szinkronizációt, s ezzel a versenyhelyzetek elkerülését. Sokféle mechanizmust használhatunk. A processzusokat kezelő kód végzi a prioritásokon alapuló processzusütemezést is. Az (aszinkron) szoftvermegszakításoknak megfelelő jelzések (signals) kezelése szintén itt történik. Végül a memória használatát is innen irányítja a rendszer. A legtöbb UNIX-rendszer támogatja a kérésre lapozott virtuális memóriát, esetleg még az olyan extra sajátosságokat is, mint a több processzus által megosztva használható közös címtartomány-régiók.

A UNIX születése óta megpróbált kis rendszer maradni, ezzel is támogatva a megbízhatóságot és a hatékonyságot. A UNIX első változatai 80 ASCII karakterből álló 24 vagy 25 sort megjelenítő terminálokat használtak. A felhasználói felületet biztosító, felhasználói szinten futó program, a **parancsértelmező (shell)** parancssoros elérést tett lehetővé. Mivel a shell nem tartozott a kernelhez, könnyű volt újabb shelleket adni a UNIX-hoz. Idővel egyre több és egyre okosabb shell programot készítettek.

Később, amikor megjelentek a grafikus terminálok, az M. I. T.-n egy **X Window System** nevű ablakozó rendszert fejlesztettek ki a UNIX-hoz. Még később az X Window System tetejére egy **Motif** nevű komplett **GUI (Graphical User Interface, grafikus felhasználói felület)** került. A kis kernelt kívánó UNIX-filozófiával összhangban az X Window System és a Motif kódjának döntő hányada felhasználói szinten, a kernelen kívül fut.

Windows XP

Mikor 1981-ben megjelent az eredeti IBM PC, az MS-DOS 1.0-nak nevezett 16 bites, valós módú, egyfelhasználós, parancssoros operációs rendszerrel szállították. Az operációs rendszer 8 KB memóriarezidens kódból állt. Két évvel később megjelent a 24 KB-os, lényegesen nagyobb tudású MS-DOS 2.0. Ez már tartalmazott egy parancsértelmezőt (shellt), amely számos a UNIX-tól kölcsönzött sajátosságot mutatott. Amikor 1984-ben az IBM kiadta a 286-ra épülő PC/AT-t,

ebben már a 36 KB-os MS-DOS 3.0 volt. Az évek során az MS-DOS újabb sajátosságokra tett szert, de még mindig parancssoros rendszer volt.

Az Apple Macintosh sikerén fölbuzdulva a Microsoft eldöntötte, hogy az MS-DOS-t grafikus felhasználói felülettel látja el, melyet **Windows**nak fog hívni. A Windows első három változata, a Windows 3.x sorozat még nem volt igazi operációs rendszer, csak grafikus felhasználói felület az MS-DOS fölötti; az utóbbi irányította a gépet. A programok közös címtartományban futottak, és bármelyikükben előforduló hiba térdre kényszerítette az egész rendszert.

A Windows 95 1995-ös kiadása még mindig nem jelentette az MS-DOS-tól való megszabadulást, csak egy új 7.0 verziót. A Windows 95 és az MS-DOS 7.0 együttesen már tartalmazta egy teljes operációs rendszer legtöbb funkcióját, a virtuális memóriát, a processzusok kezelését és a multiprogramozást. A Windows 95 azonban nem volt teljesen 32 bites program. A 32 bites mellett sok régi 16 bites kódrészletet is tartalmazott, és még mindig az MS-DOS fájlrendszerét használta, annak majdnem minden korlátjával. A fájlrendszerrel kapcsolatos lényeges változás csupán az MS-DOS-ban megengedett 8 + 3-as nevek helyett a hosszú fájlnevek bevezetése és 65 536-nál több lemezblokk kezelésének képessége volt.

Még a Windows 98 1998-as kiadásában is ott volt a 16 bites kódot futtató (most 7.1-es verziónak nevezett) MS-DOS. Bár kicsit több funkció került át az MS-DOS-ból a Windowsos részbe, és a nagyobb lemezek kezelésére alkalmas elrendezés vált szabványossá, a mélyebb részleteket tekintve a Windows 98 nem sokban különbözött a Windows 95-től. A legnagyobb eltérés a felhasználói felületben volt, amely szorosabban integrálta a munkaasztalt, az internetet és a tv-t. Pontosan ez az integráció keltette fel az amerikai Igazságügyi Minisztérium figyelmét, s emiatt perelte be a Microsoftot illegális monopólium kiépítésének vádjával. A Windows 98-at a rövid életű Windows Millenium Edition (Windows ME) követte, amely egy némileg följavított Windows 98 volt.

Az előző fejlesztésekkel párhuzamosan a Microsoft egy teljesen új 32 bites operációs rendszer elkészítésébe kezdett. Az új rendszer neve **Windows New Technology** vagy röviden **Windows NT** volt. Kezdetben az összes Intel-alapú PC-s operációs rendszer helyett ezt ajánlották, de nem volt túl nagy sikere. Ezután inkább a felső piaci szegmensek réseit célozták meg, ahová sikerült benyomulniuk. Az NT második változatát Windows 2000-nek nevezték el. Ez eléggé elterjedt még az asztali számítógépek piacán is. A Windows 2000 után az XP következett, de ekkor csak viszonylag jelentéktelen változtatások voltak. Az XP lényegében egy kicsit följavított Windows 2000.

Az XP-t két változatban, szerverként és munkaállomásként árulják. A két változat ugyanabból a forráskódból származik és majdnem teljesen azonos. A szerverváltozatot lokális hálózatok fájl- és nyomtatószervereit futtató gépekbe szánják. Részletesebben kidolgozott rendszeradminisztrációs lehetőségei vannak, mint az egyetlen felhasználót kiszolgáló asztali gépként működő kliens változatnak. A szervernek létezik nagy hálózatokat kiszolgáló (Enterprise) verziója is. A különböző változatokat eltérően hangolják, várható környezetüknek megfelelően optimalizálják. Ezektől az apró eltérésektől eltekintve az összes változat lényegében azonos. Valójában még a különböző változatok végrehajtható fájljai is majdnem

mind megegyeznek. Az XP maga a registrynek nevezett belső adatstruktúrában tárolt speciális változó értékéből tudja, hogy melyik verziót kell futtatnia. A licenc tiltja ennek megváltoztatását, hogy a felhasználók ne tudják az (olcsó) klienst a (sokkal drágább) szerver vagy enterprise változattá konvertálni. A továbbiakban nem teszünk különbséget a fönt említett verziók között.

Az MS-DOS és az előző Windows-változatok mind egyfelhasználós rendszerek voltak. Az XP ezzel szemben támogatja a multiprogramozást, tehát egyszerre több felhasználó is dolgozhat ugyanazon a gépen. Egy hálózati szerveren például a hálózatról bejelentkezett több felhasználó is lehet egyszerre, s mindegyikük hozzáférhet saját fájljaihoz.

Az XP valódi 32 bites multiprogramozásos operációs rendszer. Több felhasználói processzust támogat. Ezek mindegyike 32 bites kérésre lapozott virtuális memóriában fut. A rendszer maga is teljes egészében 32 bites kódból áll.

A Windows 95-höz képest az NT egyik rendkívüli újítása a moduláris felépítés volt. Kernelmódban futó, viszonylag kis kernelből, és felhasználói módban futó számos szerver processzusból állt. A felhasználói processzusok a kliens-szerver modell szerint kapcsolódtak a szerver processzusokhoz: a kliens elküldte kívánságát a szerverhez, a szerver elvégezte a munkát, és az eredményt egy második üzenetben visszaküldte a kliensnek. Ez a moduláris felépítés megkönnyítette a rendszer portolását az Intel vonal mellett más gépekre (DEC Alpha, IBM PowerPC, SGI MIPS). Hatékonysági megfontolások miatt azonban az NT 4.0-val kezdődően a rendszer nagy része visszakerült a kernelbe.

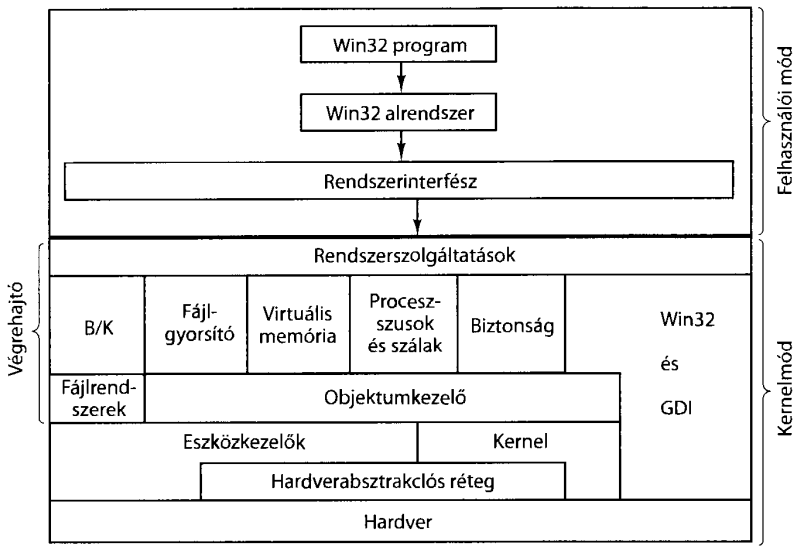
Még sok időt eltölthetnénk az XP belső felépítésének és rendszerhívásainak vizsgálatával. Mivel elsősorban a különféle operációs rendszerek által megvalósított virtuális gépek (vagyis a rendszerhívások) érdekelnek bennünket, röviden összefoglaljuk a rendszer felépítését, s azután rátérünk a rendszerhívások interfészére.

Az XP felépítését a 6.31. ábra mutatja. Rétegekbe rendezett számos modult láthatunk, együttes működésük valósítja meg az operációs rendszer funkcióit. Minden modulnak saját feladata, és a többi modul felé jól definiált interfésze van. Majdnem minden modult C-ben írtak, bár a grafikus eszközök interfészének egy része C++, a legalsó rétegek bizonyos darabjai pedig assembly nyelven készültek.

Legalul a vékony **hardverabsztrakciós réteg (hardware abstraction layer)** található. Feladata olyan absztrakt hardvereszközök definiálása az operációs rendszer többi része számára, amelyek már mentesek az igazi hardverre jellemző egyéni sajátosságoktól és csúf részletektől. A modellezett eszközök között előfordulnak lapkán kívüli gyorsítótárak, időzítők, B/K sínek, megszakítás- és DMA-vezérlők. Mivel ezeket az operációs rendszer többi része csak „idealizált formában" látja, könnyebb az XP-t más hardverplatformokra átvinni, hiszen a legtöbb szükséges módosítás egy helyre koncentrálódik.

A hardverabsztrakciós réteg fölött a **kernel**t és az eszközkezelőket tartalmazó réteg következik. A kernel és az összes eszközkezelő szükség esetén közvetlenül is hozzáférhet a hardverhez, mivel hardverfüggő kódot tartalmaznak.

A kernel az elemi kernelobjektumokat támogatja (megszakítás, csapda, kivételkezelés, processzusok ütemezése és szinkronizációja, több processzor szinkronizá-



6.31. ábra. *A Windows XP felépítése*

ciója, időkezelés). A réteg célja az operációs rendszer maradékának teljes hardverfüggetlenségét biztosítani, s ezzel könnyen hordozhatóvá (portolhatóvá) tenni. A kernel állandóan a memóriában található. Nem függeszthető fel (preemptible), bár néha át tudja adni a vezérlést a B/K megszakítások kiszolgálásához.

Az **eszközmeghajtók (device driver)** egy vagy több B/K eszközt irányíthatnak, bár végezhetnek nem eszközspecifikus tevékenységeket is (adatfolyam kódolása, kernelobjektumok elérhetőségének biztosítása). Mivel a felhasználók is telepíthetnek (installálhatnak) új eszközkezelőket, módjukban áll a kernel megváltoztatása és a rendszer tönkretétele. Az eszközmeghajtókat emiatt nagyon gondosan kell megírni.

A kernel és az eszközmeghajtók fölött van az operációs rendszer **végrehajtónak (executive)** nevezett felső része. A végrehajtó architektúrafüggetlen és viszonylag csekély erőfeszítéssel hordozható új gépekre. Három rétegből áll.

A legalsó réteg a fájlrendszereket és az objektumkezelőt tartalmazza. A **fájlrendszerek (file systems)** a fájlok és könyvtárak használatát támogatják. Az **objektumkezelő (object manager)** a kernel által ismert objektumokat kezeli. Idetartoznak a processzusok, a szálak (threads) – vagyis a közös címtartományú könnyűsúlyú processzusok –, a fájlok, a könyvtárak, a szemaforok, a B/K eszközök, az időzítők és még sok más. Az objektumkezelő egy névteret is kezel, melyben a létrehozott új objektumok helyezhetők el, hogy később hivatkozni lehessen rájuk.

Amint a 6.31. ábrán látható, a következő szint hat fő részből áll. A **B/K kezelő (I/O manager)** az általános B/K szolgáltatásokat és a B/K eszközök kezelését végzi. Felhasználja a fájlrendszer szolgáltatásait, az viszont az eszközmeghajtókra és az objektumkezelő szolgáltatásaira támaszkodik.

A **fájlgyorsító-kezelő (file cache manager)** a fájlok blokkjainak kezelésével foglalkozik, és annak eldöntésével, hogy melyik blokkokat tartsa a memóriában a jövőbeli felhasználás végett, továbbá a virtuálismemória-kezelőt segíti. A memóriára leképezett fájlok kezelésével is foglalkozik. Az XP konfigurálható több fájlrendszerrel is, ilyenkor minden fájlrendszert a fájlgyorsító-kezelő felügyel. Amikor egy blokkra van szükség, a gyorsítókezelőtől kell kérni. Ha a blokk nincs a memóriájában, a megfelelő fájlrendszertől kéri el. Mivel a fájlok processzusok címtartományába is leképezhetők, a szükséges konzisztencia biztosítása miatt a gyorsítókezelőnek együtt kell működnie a memóriakezelővel.

A **virtuálismemória-kezelő (virtual memory manager)** valósítja meg az XP kérésre lapozásos virtuálismemória-architektúráját. Elvégzi a virtuális lapok fizikai lapkeretekre való leképezését. Érvényesíti azokat a védelmi szabályokat, melyek (speciális helyzetektől eltekintve) minden processzus számára csak a saját címtartományához tartozó lapok elérését engedélyezik. Kezeli a virtuális memóriához kapcsolódó bizonyos rendszerhívásokat is.

A **processzus- és szálkezelő (process and thread manager)** a processzusokat és a szálakat felügyeli, ideértve létrehozásukat és megszüntetésüket. Ezen belül inkább a kezelésükre szolgáló mechanizmusokkal, és nem a használatukra vonatkozó elvekkel foglalkozik.

Az XP **biztonságiutalás-kezelője (security reference manager)** érvényesíti az XP részletesen kidolgozott biztonsági mechanizmusát, amely megfelel az USA Védelmi Minisztériumának az ún. Orange Bookban a C2 szintre kirótt feltételeinek. Az Orange Bookban lefektetett számos szabályt kell teljesíteniük a neki megfelelő rendszereknek. Ezek a szabályok a bejelentkezéskori megfelelő azonosítástól kezdve a hozzáférés ellenőrzésén át a virtuális lapok újrafelhasználás előtt 0-val való feltöltéséig terjednek.

A **grafikus eszköz interfész (graphics device interface, GDI)** végzi a nyomtatók és a monitor számára a képek kezelését. Olyan rendszerhívásokat biztosít, melyekkel a felhasználói programok eszközfüggetlen módon írhatnak a képernyőre vagy a nyomtatóra. Tartalmazza az ablakkezelőt és a hardvereszköz-kezelőket is. Az XP előtti NT 4.0 változatban felhasználói szinten futott, de olyan kiábrándító teljesítménnyel, hogy a sebesség megnövelése érdekében a Microsoft beépítette a kernelbe. A Win32 modul kezeli a rendszerhívások nagy részét is. Eredetileg ez is felhasználói módban futott, de hatékonysági okokból áttették a kernelbe.

A végrehajtó felett a **rendszerszolgáltatások (system services)** nevű vékony réteg helyezkedik el. Az a feladata, hogy megfelelő interfészt biztosítson a végrehajtónak. A valódi XP-rendszerhívásokat fogadja, és a végrehajtó más részeit hívja ezek végrehajtatásához.

A kernelen kívül vannak a felhasználói programok és a **környezeti alrendszer (environmental subsystem)**. Ez arra való, hogy a felhasználói programok lehetőleg ne hajtsanak végre közvetlen rendszerhívásokat (bár technikailag megtehetik). A környezeti alrendszer ehelyett a felhasználói programokban használható függvényhíváshalmazt biztosít a felhasználói programok számára. Eredetileg há-

rom környezeti alrendszer létezett: a Win32 (az NT-, Windows 2000, XP-, sőt még a Windows 95/98 programoknak), a POSIX (a portolt UNIX-programoknak) és az OS/2 (a portolt OS/2 programoknak). Ezek közül csak a Win32 támogatott. Létezik azonban egy új Services for UNIX modul, amely szerény UNIX-támogatást biztosít.

A Windows-alkalmazások a Win32 függvényeket használják, és a Win32 alrendszerrel kommunikálnak a rendszerhívások végrehajtásához. A **Win32 alrendszer (Win32 subsystem)** a Win32 függvényhívásokat (lásd alább) fogadja, és a **rendszerinterfész (system interface)** könyvtári modult (valójában egy DLL fájlt, lásd 7. fejezet) használja a szükséges valódi XP-rendszerhívások végrehajtásához.

Miután röviden áttekintettük az XP felépítését, térjünk át fő témánkra, az általa nyújtott szolgáltatásokra. Ez az interfész adja a programozó számára a rendszerhez való elsődleges kapcsolódási lehetőséget. A Microsoft sajnálatos módon soha nem publikálta az XP-rendszerhívások teljes listáját, és kiadásonként változtat is rajtuk. Ilyen körülmények között közvetlen rendszerhívásokat végrehajtó programok írása szinte lehetetlen.

Ehelyett a Microsoft bevezette és közzétette a **Win32 API (Application Programming Interface, felhasználói programozói interfész)** nevű hívások halmazát. Ezek olyan könyvtári eljárások, amelyek vagy rendszerhívásokat vesznek igénybe, vagy – bizonyos esetekben – magában a felhasználói módban futó könyvtári eljárásban végzik el feladatukat, vagy a Win32 alrendszerben. A stabilitást elősegítendő a Win32 API-hívások nem változnak kiadásonként. De léteznek XP API-rendszerhívások is, ezek viszont módosulhatnak az XP különböző kiadásaiban. Bár a Win32 hívások nem mind XP-rendszerhívások is egyben, jobb, ha a továbbiakban inkább ezekre koncentrálunk, mivel a Win32 API-hívások jól dokumentáltak és kevésbé változnak, mint a valódi XP-rendszerhívások. Mikor a Windowst 64 bites gépekre vitték át, a Microsoft megváltoztatta a Win32 elnevezést, hogy mind a 32, mind a 64 bites változatot lefedje, de a mi céljainkhoz elegendő, ha a 32 bites verziót tekintjük.

A Win32 API filozófiája teljesen eltér a UNIX-tól. Az utóbbiban az összes rendszerhívás publikus, és olyan minimális interfészt alkot, amelyből bármelyik hívást kihagyva csökkenne az operációs rendszer funkcionalitása. A Win32 filozófia: legyen az interfész részletes, mindenre kiterjedő, sokszor ugyanazt a dolgot három vagy négy módon is el lehessen érni. Legyen sok olyan függvény, amelyet nem rendszerhívással kellene megvalósítani (és nem is azzal valósul meg), például a teljes fájlokat másoló API-hívás.

Sok Win32 API-hívás valamilyen kernelobjektumot hoz létre: fájlt, processzust, szálakat, csövet stb. A kernelobjektumokat létrehozó minden hívás eredményként egy **kezelőt (handle)** ad vissza a hívónak. A továbbiakban ez a kezelő használható az objektumokon végzett műveleteknél. A kezelők az általuk hivatkozott objektumot létrehozó processzusra nézve specifikusak. Közvetlenül nem adhatók át és nem használhatók más processzusokban (mint ahogy a UNIX-fájlleírókat sem lehet másik processzusnak átadni, és abban felhasználni). Bizonyos körülmények között azonban a kezelőkről másolatok készíthetők, ezek védett módon átadhatók további processzusoknak, ellenőrzött hozzáférést megengedve számukra más processzusok objektumaihoz. Minden objektumhoz tartozhat egy **biztonságleíró (security descriptor)**, amely részletesen megmondja, hogy az adott objektumon ki és milyen típusú műveleteket végezhet.

Szokás az XP-t néha objektumorientáltnak nevezni, mivel a kernelobjektumok elérése és módosítása csak metódusaik (a kezelőiken végrehajtott API-függvényhívások) segítségével történhet. Másrészt viszont hiányzik az objektumorientált rendszerek sok alapvető tulajdonsága, például az öröklődés vagy a polimorfizmus.

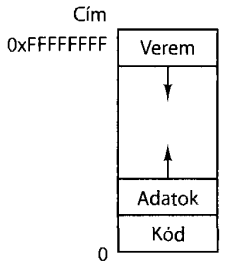
A Win32 API – kevés kivétellel – elérhető volt a Windows 95/98-on (továbbá a fogyasztói elektronikus eszközök operációs rendszereként használt Windows CE-n). A Windows 95/98 például nem tartalmazott biztonsági alrendszert, ezért a biztonsággal kapcsolatos API-hívások a Windows 95/98-on csupán egy visszatérési hibakódot produkáltak. Továbbá az XP fájlrendszere a Unicode karakterkészletet használja, amely szintén nem állt rendelkezésre a Windows 95/98-ban. Bizonyos API-függvényhívásokat másként kell paraméterezni. Az XP-n például a grafikus függvényekben használt képernyő koordináták 32 bites számok; a Windows 95/98 csak ezek alsó 16 bitjét használta (a Windows 3.1-gyel való visszafelé kompatibilitás miatt). A Win32 API több operációs rendszeren való elérhetősége megkönnyíti a programok átvitelét, de még világosabban mutatja, hogy ez a felület le van választva a valódi rendszerhívásokról.

6.4.2. Példák virtuális memória kezelésére

Ebben a szakaszban a UNIX és az XP virtuális memóriáját egyaránt vizsgáljuk. A programozó szempontjából többnyire hasonlóan viselkednek.

A UNIX virtuális memóriája

A UNIX memóriamodellje egyszerű. Minden processzusnak három szegmense van: a kód-, az adat- és a veremszegmens (6.32. ábra). Ha a gépen csak egyetlen lineáris címtartomány van, a kódot általában a memória aljához közel helyezik el, ezt az adatok követik. A vermet a memória tetején helyezik el. A kód mérete állan-



6.32. ábra. *UNIX-processzus címtartománya*

dó, az adaté és a veremé viszont egymással ellentétes irányban növekedhet. Ezt a modellt, melyet a Solaris is használ, szinte minden gépen könnyű megvalósítani.

Továbbá, ha a gép támogatja a lapozást, az egész címtartomány lapozható anélkül, hogy a felhasználói programok észrevennék. Csak azt érzékelik, hogy a gépben lévő fizikai memóriánál nagyobb méretű programok is futtathatók. Azok a UNIX-rendszerek, amelyek nem használnak lapozást, általában teljes processzusokat cserélnek a memória és a lemez között. Így képesek tetszőlegesen sok processzust időosztással futtatni.

A Berkeley UNIX-ra vonatkozóan a föntiek (kérésre lapozott virtuális memória) lényegében mindent leírnak. A System V (és a Solaris) azonban olyan további lehetőségeket is kínál, melyekkel a felhasználók kifinomultabb virtuálismemória-kezelést valósíthatnak meg. A legfontosabb, hogy a processzusok fájlokat (részben vagy egészben) leképezhetnek virtuális címtartományuk egy részére. Ha például egy 12 KB méretű fájlt a 144 K virtuális címtől kezdődően képezünk le, akkor a 144 KB címről beolvasott szó a fájl első szavát adja. Így a fájl B/K rendszerhívások nélkül végezhető el. A fájlok mérete nagyobb is lehet a virtuális címtartománynál, ezen segít az a lehetőség, hogy a teljes fájl helyett annak egy darabja is leképezhető. A leképezés végrehajtása úgy történik, hogy először megnyitjuk a fájlt. Ekkor egy *fd* fájlleírót (file descriptor) kapunk vissza, amely a leképezendő fájlt azonosítja. Ezután a processzus végrehajtja a

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset)
```

hívást, amely a fájlból a *file_offset*-től kezdődően *length* bájtot képez le a virtuális címtartomány *virtual_address* kezdőcímétől kezdve. Alternatív megoldásként a *flags* paraméter beállításával a rendszertől kérhetjük a virtuális cím meghatározását, ennek értékét a *paddr*-ban kapjuk meg. A *protection* paraméter az olvasás, az írás és a végrehajtás engedélyezésének bármely kombinációját tartalmazhatja. A leképezés később megszüntethető az unmap rendszerhívással.

Több processzus egy időben is leképezheti ugyanazt a fájlt. A megosztásra két lehetőségünk van. Az elsőnél minden lap közös, tehát ha az egyik processzus ír valamit, azt az összes többi is látja. Ez a processzusok közti nagy sávszélességű kommunikációt tesz lehetővé. A másik módszer csak addig osztja meg a lapokat, míg egyik processzus sem módosít rajtuk. Mihelyt azonban írni próbál bármelyik processzus valamelyik lapra, laphiány lép fel, és az operációs rendszer egy privát másolatot készít az illető processzusnak a módosítandó lapról. Ez a **copy on write (íráskori másolás)** néven ismert séma akkor használatos, ha mindegyik processzusban azt az illúziót próbáljuk kelteni, hogy egyedül ő képezte le az adott fájlt.

A Windows XP virtuális memóriája

Az XP-ben minden felhasználói processzusnak saját virtuális címtartománya van. A virtuális címek 32 bitesek, ezért minden processzus 4 GB méretű címtartományt használhat. Az alsó 2 GB a processzus kódjának és adatainak elhelyezésére szolgál, a felső 2 GB a kernel memóriájához való (korlátozott) hozzáférést tesz lehetővé, de a Windows Server változataiban 3 GB-ot kaphatnak a felhasználók és 1 GB-ot a kernel. A virtuális címtartomány kérésre lapozott, a lapméret rögzített (a Pentium 4-en 4 KB-os).

Minden virtuális lapnak 3 állapota lehet: szabad, foglalt vagy egyeztetett. A **szabad lap (free page)** pillanatnyilag nincs használatban, a rá való hivatkozás laphibát okoz. A processzus elindulásakor minden hozzá tartozó lap szabad, míg csak a program és a kiinduló adatok le nincsenek képezve a virtuális címtartományba. Mihelyst kódot vagy adatokat képeztünk le egy lapra, a lapot **egyeztetett**nek **(committed)** nevezzük. Az egyeztetett lapokra való hivatkozások a virtuális memóriát kezelő hardver segítségével képeződnek le, és akkor sikeresek, ha az illető lap a memóriában van. Az ellenkező esetben laphiány keletkezik, az operációs rendszer megkeresi és betölti a lemezről a kért lapot. A virtuális lap lehet **foglalt (reserved)** is, ami azt jelenti, hogy semmi sem képezhető le rá, ameddig a foglaltságot explicit módon meg nem szüntetjük. A szabad, foglalt és egyeztetett attribútumok mellett a lapoknak más tulajdonságai is vannak, például írható, olvasható vagy végrehajtható. A memória alsó és felső 64 KB-os része mindig szabad. Ezzel segítik a mutatóhibák felderítését (a nem inicializált mutatók értéke sokszor 0 vagy –1).

Minden egyeztetett lapnak egy „árnyéklap" felel meg a lemezen. Itt tároljuk, ha éppen nincs a memóriában. A szabad és a foglalt lapoknak nincs ilyen árnyéklapja, ezért a rájuk való hivatkozás laphibát okoz (a rendszer nem tud mit betölteni a lemezről). A lemezen található árnyéklapok egy vagy több lapozófájlba vannak összegyűjtve. Az operációs rendszer tartja számon, hogy melyik virtuális lap melyik lapozófájl melyik részére van leképezve. A (csak végrehajtható) programszöveg árnyéklapjait a végrehajtható bináris fájl tartalmazza. Az adatokat tartalmazó lapokhoz speciális lapozófájlokat használ.

Az XP a System V-hoz hasonlóan megengedi fájlok közvetlen leképezését a virtuális címtartomány régióira (vagyis egymás utáni lapok sorozatára). Miután egy fájlt leképeztünk a címtartományba, közönséges memóriahivatkozásokkal írhatjuk vagy olvashatjuk.

A memóriába leképezett fájlok megvalósítása ugyanolyan, mint a többi egyeztetett lapé, csak az árnyéklapok – a lapozófájlok helyett – lehetnek magukban az eredeti fájlokban. Ez azt eredményezheti, hogy a memóriában lévő verzió eltérhet a lemezen találhatótól (például, ha nemrég írtunk a virtuális címtartományba). A leképezés megszüntetésekor vagy explicit „flush" művelet hatására azonban frissítődik a lemezen található változat.

Az XP szándékosan megengedi, hogy egyszerre kettő vagy több processzus képezze le címtartományába ugyanazt a fájlt, esetleg különböző virtuális címekre is. Ilyenkor a memória szavainak írása és olvasása nagy sávszélességű kommunikációt tesz lehetővé az egyes processzusok között, mivel nincsen szükség másolásra. Az egyes processzusok hozzáférési jogosultságai eltérők lehetnek. Mivel a memóriába képezett fájlt használó minden processzus ugyanazokon a lapokon osztozik, az egyikük által végrehajtott változtatásokat a többiek azonnal látják, még ha a lemezen található változat frissítése nem is történt meg.

API-függvény	Jelentése
VirtualAlloc	Régió lefoglalása vagy egyeztetése
VirtualFree	Régió felszabadítása vagy egyeztetés megszüntetése
VirtualProtect	Régió olvasási/írási/végrehajtási védelmének megváltoztatása
VirtualQuery	Régió státusának lekérdezése
VirtualLock	Régió memóriarezidenssé tétele (kilapozás megtiltása)
VirtualUnlock	Régió szokásos lapozhatóságának beállítása
CreateFileMapping	Fájlleképezés-objektum létrehozása opcionális névadással
MapViewOfFile	Fájl (egy részének) leképezése a címtartományba
UnmapViewOfFile	Leképezett fájl eltávolítása a címtartományból
OpenFileMapping	Korábban létrehozott fájlleképezés-objektum megnyitása

6.33. ábra. *A Windows XP legfontosabb virtuális memóriát kezelő API-függvényei*

A Win32 API számos olyan függvényt tartalmaz, melyek a processzusok számára közvetlen memóriakezelést tesznek lehetővé. A legfontosabb ilyen függvények a 6.33. ábrán láthatók. Mindegyikük a virtuális címtér egy vagy több lapjából álló folytonos memóriarégiókkal dolgozik.

Az első négy API-függvény jelentése magától értetődő. A következő két függvénnyel egy processzus néhány lapot a memóriához „drótozhat", hogy azok soha ne lapozódjanak ki, illetve megszüntetheti ezt a tulajdonságot. Például valós idejű programnak lehet erre szüksége. Csak a rendszeradminisztrátor által futtatott programok rögzíthetnek le lapokat a memóriában. Az operációs rendszer által emelt korlát még az ilyen processzusok túlzott mohóságának is gátat vet. Bár a 6.33. ábrán nem tüntettük föl, az XP-nek vannak olyan API-függvényei is, melyekkel egy processzus hozzáférhet egy másik processzus virtuális memóriájához (például, amelyikhez kezelővel rendelkezik), és ezáltal vezérelheti.

Az utolsó négy API-függvény a memóriába leképezett fájlok kezelésére szolgál. A leképezéshez először a CreateFileMapping hívásával egy „fájlleképezés-objektumot" kell létrehozni. A függvényhívás visszaadja az objektum kezelőjét és (opcionálisan) nevet is ad neki a fájlrendszerben, hogy más processzus is használhassa. A következő két függvény fájlt képez le, illetve leképezést szüntet meg. Az utolsóval egy processzus egy másik processzus által már leképezett fájl leképezését tudja megvalósítani. Ezzel két vagy több processzus megoszthatja címtartományának régióit.

Ezek az alapvető API-függvények, rájuk épül a memóriakezelő rendszer további része. Vannak például olyan API-függvények is, amelyek egy vagy több halomba (heap) adatstruktúrákat hoznak létre vagy szüntetnek meg. A halmokat dinamikusan létrehozott és felszabadított adatstruktúrák tárolására használják. A halmokban nem végeznek szemétgyűjtést, tehát a felhasználói szoftver feladata a többé már nem használt memóriablokkok felszabadítása (szemétgyűjtésen a többé már nem használt adatstruktúráknak az operációs rendszer általi automatikus eltávolítását értjük). Az XP-ben a halmok kezelése hasonlít a UNIX-rendszerekből ismerős *malloc* függvény működéséhez; kivéve azt az eltérést, hogy több egymástól függetlenül kezelt halom is létezhet.

6.4.3. Példák virtuális B/K műveletekre

Minden operációs rendszer lényegét a felhasználói programoknak nyújtott szolgáltatások jelentik. Ezek leginkább a fájlok írásához és olvasásához hasonló B/K szolgáltatások. Mind a UNIX, mind az XP B/K szolgáltatások széles választékát kínálja a felhasználói programoknak. A legtöbb UNIX-rendszerhíváshoz található vele ekvivalens XP-rendszerhívás. A fordítottja már nem igaz: az XP-nek sokkal több rendszerhívása van, ezek lényegesen bonyolultabbak, mint UNIX-os megfelelőik.

Virtuális B/K a UNIX-ban

A UNIX-rendszer népszerűsége nagyrészt egyszerűségére vezethető vissza, ami viszont fájlrendszerének felépítéséből adódik. A közönséges fájlok 8 bites bájtok lineáris sorozatából állnak, melyek a 0. bájttal kezdődnek és legfeljebb $2^{32} - 1$ bájttal folytatódnak. Az operációs rendszer maga nem ír elő semmilyen rekordstruktúrát a fájlokra, bár az ASCII szövegfájlokat sok felhasználói program soremelésssel lezárt sorok sorozatának tekinti.

Minden megnyitott fájlhoz tartozik egy mutató, amely a fájl következőként olvasandó vagy írandó bájtjára mutat. A read és a write rendszerhívás az adatok olvasását, illetve írását a mutató által jelzett fájlpozíciónál kezdi. A művelet befejezése után mindkettő az átvitt bájtok számának megfelelő mennyiséggel állítja előbbre a mutatót. De lehetőségünk van fájlok véletlen elérésére is, ha a fájlmutatót közvetlenül állítjuk be a megfelelő értékre.

A közönséges fájlokon túl a UNIX támogatja a B/K eszközök elérésére szolgáló speciális fájlokat is. A tipikus megoldás, hogy minden B/K eszközhöz tartozik egy vagy több speciális fájl. A speciális fájlokon végrehajtott olvasási és írási művletekkel tudnak a programok az adott B/K eszközről olvasni vagy ráírni. Így kezelik a lemezeket, nyomtatókat, terminálokat és még sok más eszközt.

A fájlrendszerrel kapcsolatos legfontosabb UNIX-rendszerhívásokat a 6.34. ábrán soroltuk fel. A creat hívás (*e* nélkül!) új fájl létrehozására használható.

Rendszerhívás	Jelentése
creat(name, mode)	Fájl létrehozása, *mode* adja meg a védelmi módot
unlink(name)	Fájl törlése (feltéve, hogy csak 1 rá vonatkozó kötés volt)
open(name, mode)	Fájl létrehozása vagy megnyitása, és fájlleíró visszaadása
close(fd)	Fájl lezárása
read(fd, buffer, count)	*count* bájt beolvasása a *buffer*be
write(fd, buffer, count)	*count* bájt kiírása a *buffer*ből
lseek(fd, offset, w)	A fájlmutató beállítása *offset* és *w* alapján
stat(name, buffer)	A fájlra vonatkozó információk lekérdezése
chmod(name. mode)	A fájl védelmi módjának megváltoztatása
fcntl(fd, cmd, ...)	Különböző műveletek (például zárolás) végzése a fájlon

6.34. ábra. *A fájlrendszerre vonatkozó legfontosabb UNIX-rendszerhívások*

Szigorúan véve már nem is lenne rá szükség, mivel az open is létre tud hozni új fájlokat. Az unlink hívás törli a fájlt, feltéve, hogy csak egy könyvtárban fordult elő.

Az open-nel meglévő fájlokat nyithatunk meg (vagy újakat hozhatunk létre). A mode opció a megnyitás módját határozza meg (olvasásra, írásra stb.). A hívás egy kis egész számot, ún. **fájlleírót (file descriptor)** ad vissza. A későbbi hívások ezzel azonosíthatják a fájlt. Amikor már nincs szükségünk a fájlra, a close hívásával szabadítjuk fel a fájlleírót.

A read és a write végzi a fájlokon a tényleges B/K-t. Mindkettőhöz meg kell adni a használandó fájlra vonatkozó fájlleírót, a ki- vagy bemenő adatokat tartalmazó puffert és az átviendő bájtok számát. Az lseek-et a véletlen fájleléréshez szükséges fájlmutató pozicionálására használjuk.

A stat a fájlra vonatkozó információt ad vissza: méretét, utolsó elérésének időpontját, tulajdonosát és még sok mást. A chmod a fájlvédelem módját változtatja, például a tulajdonosától különböző felhasználóknak megtiltva vagy megengedve a fájl olvasását. Végül az fcntl különféle műveleteket hajt végre a fájlon, például a zárolást vagy annak feloldását.

A 6.35. ábra a legfontosabb fájl B/K hívások működését illusztrálja. Ez csak minimális kód, nem tartalmazza a szükséges hibaellenőrzéseket sem. A ciklus kezdete előtt a program megnyitja a létező *data* nevű fájlt és létrehozza a *newf* nevű új fájlt. A hívások az *infd*, illetve az *outfd* nevű fájlleírókat adják vissza. A két hívás második paramétere a védelmi biteket adja meg, melyek szerint a fájlokat olvasni, illetve írni akarjuk. Mindkét hívás fájlleírót ad vissza. Ha sikertelen az open vagy a creat, negatív fájlleírót kapunk, ami a hívás sikertelenségét mutatja.

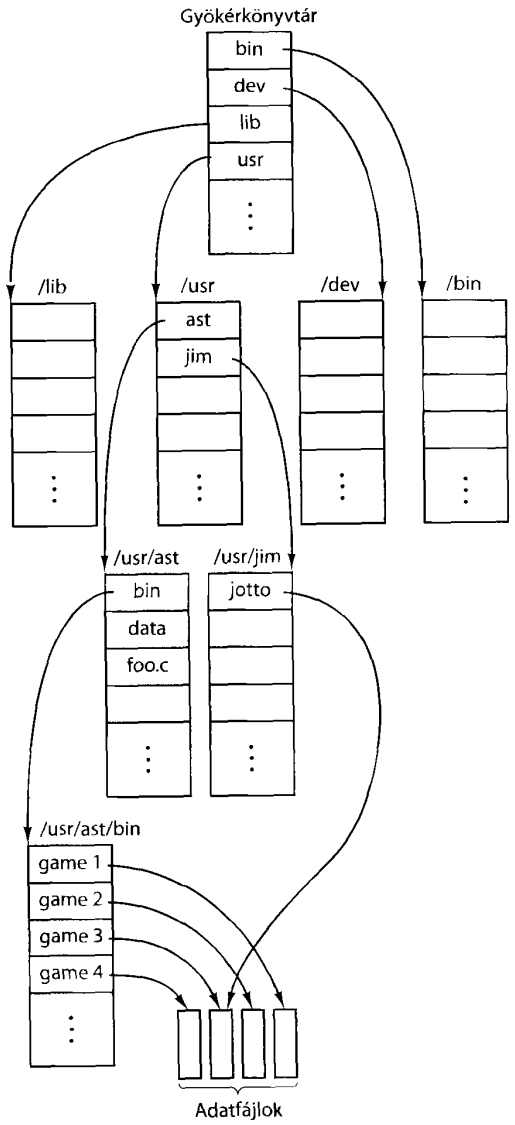
```
// Fájlleírók megnyitása
infd = open("data", 0);
outfd = creat("newf", ProtectionBits) ;

// Másolási ciklus
do {
   count = read(infd, buffer, bytes);
   if (count > 0) write(outfd, buffer, count);
} while (count > 0);

// A fájlok lezárása
close(infd);
close(outfd);
```

6.35. ábra. *Fájlok másolása UNIX-rendszerhívásokkal. A programrészlet C nyelvű, mivel a Java elrejti az alacsony szintű rendszerhívásokat, mi meg éppen azokat akarjuk megmutatni*

A read hívásnak három paramétere van: a fájlleíró, a puffer és a bájtok száma. A hívás a kívánt számú bájtot próbálja beolvasni a megadott fájlból. A ténylegesen beolvasott bájtok számát a *count* változóban adja vissza. Ha a fájl túl rövid volt, *count* értéke lehet *bytes*-nál kisebb is. A write hívás az újonnan beolvasott bájtokat

az output fájlba helyezi. A ciklus addig folytatódik, amíg az input fájl végére nem érünk, amikor is a ciklust befejezzük, és mindkét fájlt lezárjuk.

A UNIX fájlleírói kicsi (általában 20 alatti) egész számok. A 0, 1 és 2 speciális fájlleírók rendre a **szabványos bemeneti csatornának (standard input)**, a **szabvá-**



6.36. ábra. *Egy tipikus UNIX-könyvtárrendszer része*

nyos kimeneti csatornának (standard output) és a **szabványos hibacsatornának (standard error)** felelnek meg. Normális esetekben ezek a billentyűzetet, a képernyőt és újfent a képernyőt jelentik, de a felhasználó fájlokba is átirányíthatja. Sok UNIX-program bemenetét a standard bemenetről olvassa, a feldolgozott kimenetet pedig a standard kimenetre írja ki. Az ilyen programokat gyakran **szűrők**nek **(filters)** nevezik.

A fájlrendszerhez szorosan kapcsolódik a könyvtárrendszer. Minden felhasználónak lehet több könyvtára, ezek tartalmazhatnak további fájlokat és alkönyvtárakat. A UNIX-rendszereket általában a 6.36. ábrán látható módon egy **gyökérkönyvtár (root directory)** nevű főkönyvtárral és a benne elhelyezkedő következő alkönyvtárakkal konfigurálják: *bin* (a gyakran végrehajtott programoknak), *dev* (a speciális B/K eszközfájloknak), *lib* (a könyvtáraknak) és *usr* (a felhasználói könyvtáraknak). Az ábrán az *usr* könyvtár két alkönyvtárat tartalmaz *ast* és *jim* számára. Az *ast* könyvtárban két fájl, *data* és *foo.c*, továbbá a négy játékot tartalmazó *bin* alkönyvtár található.

A fájlok megnevezhetők a gyökérkönyvtártól hozzájuk vezető **útvonalnév**vel **(path)**. Az útvonalnév a gyökértől a fájlig vezető út mentén található összes könyvtár listájából áll, melyben a könyvtárakat pervonalak (/) választják el. A gyökérnél kezdődő útvonalak neve **abszolút útvonalnév (absolute path)**. A *game2* fájl abszolút útvonalneve például */usr/ast/bin/game2*.

Minden futó programhoz minden időpillanatban egy meghatározott **munkakönyvtár (working directory)** tartozik. Az útvonalnevek lehetnek a munkakönyvtárra nézve relatívak, ekkor az abszolút útvonalnevektől való megkülönböztetés végett nem a / jellel kezdjük őket. Az ilyen útvonalneveket **relatív útvonalnevek**nek **(relative path)** hívjuk. Ha a munkakönyvtár */usr/ast*, akkor *game3* a *bin/game3* útvonalnévvel érhető el. A link rendszerhívással a felhasználók más fájlokra mutató **kötés**eket **(link)** hozhatnak létre. Az előző példánál maradva a */usr/ast/bin/game3* és a */usr/jim/jotto* útvonalnévvel ugyanazt a fájlt érjük el. Tilosak a könyvtárakra vonatkozó kötések, nehogy ciklusokat kapjunk a könyvtárrendszerben. A create és a többi rendszerhívás argumentumaként abszolút vagy relatív útvonalnevek szerepelhetnek.

A UNIX legfontosabb könyvtárkezelő rendszerhívásait a 6.37. ábrán láthatjuk. A mkdir új könyvtárat hoz létre, az rmdir létező (üres) könyvtárat töröl. A kö-

Rendszerhívás	Jelentése
mkdir(name, mode)	Új könyvtár létrehozása
rmdir(name, mode)	Üres könyvtár törlése
opendir(name)	Könyvtár megnyitása olvasásra
readdir(dirpointer)	Könyvtár következő bejegyzésének olvasása
closedir(dirpointer)	Könyvtár bezárása
chdir(name)	A munkakönyvtár *name*-re változtatása
link(name1, name2)	A *name1*-re mutató *name2* könyvtári bejegyzés létrehozása
unlink(name)	*name* eltávolítása könyvtárából

6.37. ábra. *A legfontosabb könyvtárkezelő UNIX-rendszerhívások*

vetkező három hívás könyvtári bejegyzések olvasására szolgál. Az első megnyitja a könyvtárat, a második bejegyzéseket olvas belőle, a harmadik pedig bezárja. A chdir a munkakönyvtárat változtatja meg.

A link új könyvtári bejegyzést készít, amely egy létező fájlra mutat. A */usr/jim/jotto* bejegyzés készülhetett volna például a

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

hívással, vagy egy relatív útvonalneveket tartalmazó ekvivalens hívással attól függően, hogy éppen mi volt a hívást végrehajtó program munkakönyvtára. Az unlink kitöröl egy könyvtári bejegyzést. Ha a fájlra csak egy kötés hivatkozott, maga a fájl is törlődik. Ha kettő vagy több hivatkozása volt, akkor megmarad. Nem számít, hogy az eltávolított kötés az eredeti vagy későbbi másolat. Ha egyszer létrehozunk egy kötést, az eredetivel egyenjogú, tőle megkülönböztethetetlen példányt kapunk. Az

```
unlink("/usr/ast/bin/game3")
```

hívás után a *game3* csak a */usr/jim/jotto* útvonalnévvel érhető el. A link és az unlink ebben az értelemben arra is felhasználható, hogy fájlokat „mozgassunk" egyik könyvtárból a másikba.

Minden fájlhoz (és könyvtárhoz, hiszen ezek is fájlok) tartozik egy bittérkép, amely azt mondja meg, hogy ki férhet hozzá fájlhoz. A jogosultságok három RWX mezőben állnak, az első a tulajdonos olvasási (Read), írási (Write) és végrehajtási (eXecute) jogait szabályozza, a második a tulajdonossal azonos csoportba tartozó többi felhasználóét, a harmadik pedig mindenki másét. Az RWX R-X --X mezők tehát azt jelentik, hogy a tulajdonos írhatja, olvashatja és végre is hajthatja a fájlt (ami ezek szerint nyilván végrehajtható program, különben nem lenne beállítva az X jog), míg a vele azonos csoportban lévők csak olvashatják vagy végrehajthatják, az idegenek pedig csupán végrehajthatják. Ilyen beállítások mellett idegenek is használhatják a programot, de ellopni (lemásolni) nem tudják, mivel nincsen rá olvasási joguk. A felhasználók csoportokba sorolását a **rendszeradminisztrátor (superuser)** végzi. A rendszeradminisztrátor áthághatja ezeket a védelmi mechanizmusokat, minden fájlt olvashat, írhat vagy végrehajthat.

Vizsgáljuk meg röviden a fájlok és a könyvtárak megvalósítását UNIX-ban. A téma részletesebb tárgyalását tartalmazza (Vahalia, 1996). Minden fájlhoz (és könyvtárhoz, mivel a könyvtár is fájl), tartozik egy **i-csomópont**nak **(i-node)** nevezett, 64 bájtos információs blokk. Az i-csomópont megmondja a fájl tulajdonosát, a jogosultságait, hogy hol találhatók a hozzá tartozó adatok stb. Az egyes lemezeken található fájlok i-csomópontjai sorszámuk szerint rendezve a lemez elején találhatók, vagy – cilindercsoportokra osztott (particionált) lemezek esetén – az egyes cilindercsoportok elején. Tehát adott sorszámú i-csomópontot a UNIX-rendszer könnyen meg tud találni lemezcímének kiszámításával.

A könyvtári bejegyzések két részből állnak, egy fájlnévből és egy i-csomópont számából. Amikor a program az

```
open("foo.c", 0)
```

hívást hajtja végre, a rendszer a munkakönyvtárban megkeresi a „foo.c" fájlnevet, hogy meghatározza a hozzá tartozó i-csomópont sorszámát. Ha megtalálta, be tudja olvasni az i-csomópontot, amely mindent elmond a fájlról.

Hosszabb elérési útvonalnevet megadva az előző lépések ismétlődnek néhányszor, míg a teljes útvonalnevet végig nem nézi a rendszer. A */usr/ast/data*-hoz tartozó i-csomópont sorszám megkeresése például úgy zajlik, hogy először a gyökérkönyvtárban keres a rendszer egy *usr* bejegyzést. Miután megtalálta az *usr* i-csomópontját, tudja olvasni a fájlt (a UNIX-ban a könyvtár is fájl). Ebben a fájlban *ast* bejegyzést keres, vagyis a */usr/ast* fájlhoz tartozó i-csomópont sorszámát határozza meg. A */usr/ast* elolvasásával a rendszer meg tudja találni a *data* bejegyzést, ezzel pedig a */usr/ast/data* i-csomópont sorszámát. Ha adott a fájl i-csomópontjának sorszáma, az i-csomópontból mindent megtudhat az adott fájlról.

Az i-csomópontok elrendezése, tartalma és formátuma rendszerről rendszerre változik (különösen, ha hálózatos rendszerről van szó), de az alábbi bejegyzések általában mindig megtalálhatók az i-csomópontokban:

1. A fájl típusa, a 9 RWX jogosultsági bit és néhány más bit.
2. A fájlra hivatkozó kötések száma (vagyis a rá vonatkozó könyvtári bejegyzések száma).
3. A tulajdonos személye.
4. A tulajdonos csoportja.
5. A fájl bájtokban mért hossza.
6. 13 lemezcím.
7. A fájl utolsó olvasásának ideje.
8. A fájl utolsó írásának ideje.
9. Az i-csomópont utolsó módosításának ideje.

A fájl típusánál megkülönböztetünk közönséges fájlokat, könyvtárakat és két speciális fájltípust, melyek a blokkstruktúrájú, illetve a strukturálatlan B/K eszközöknek felelnek meg. A kötések számát és a tulajdonos azonosítását már tárgyaltuk. A fájl hossza 32 bites egész, a fájl legnagyobb sorszámú, már értékkel rendelkező bájtját adja meg. Megengedett dolog egy új fájl létrehozása, az lseek-kel az 1 000 000-ra pozicionálása, és oda 1 bájt beírása. Ezzel 1 000 001 bájt hosszúságú fájlt kapunk. A fájl összes „hiányzó" bájtjának *nem* kell tárterületet lefoglalni.

Az első tíz lemezcím adatblokkokra mutat. 1024 bájtos blokkméret esetén legfeljebb 10 240 bájt méretű fájlok kezelhetők ilyen módon. A 11. cím egy **indirekt blokk**nak nevezett, 256 lemezcímet tartalmazó blokkra mutat. Ezzel a 10 240 + 256 × 1024 = 272 384 méret alatti fájlokat tudjuk kezelni. A még nagyobb fájlokhoz felhasználhatjuk a 12. címet, amely 256 indirekt blokk címét tartalmazó blokkra mutat. Ezzel már 272 384 + 256 × 256 × 1024 = 67 381 248 bájtos fájlokkal is elboldogulunk. Ha ez a **kétszeresen indirekt blokkok**at használó séma sem elegendő, a 13. cím olyan **háromszorosan indirekt blokk**ra mutat, amely 256 kétszeresen indirekt blokk címét adja meg. Az összes direkt, egyszeresen, kétszeresen

és háromszorosan indirekt címet felhasználva legfeljebb 16 843 018 blokk címezhető meg, amiből a fájlok maximális méretére a 17 247 250 432 bájtos elméleti felső korlát adódik. Mivel a fájlmutatók csak 32 bitesek lehetnek, a gyakorlati felső korlát valójában 4 294 967 295 bájt. A szabad lemezblokkokat láncolt listára fűzi a rendszer. Ha új blokkra van szükség, a soron következőt veszi le. Emiatt a fájlok blokkjai véletlenszerűen szóródnak szét a lemezen.

A lemez B/K hatékonyságának fokozására a fájl i-csomópontja a megnyitás után bemásolódik a memóriában tárolt táblázatba, és a gyorsabb elérhetőség miatt ott is marad, amíg a fájl meg van nyitva. Emellett a rendszer a memóriában tárolja a mostanában hivatkozott lemezblokkok gyűjteményét is. Mivel a legtöbb fájlt szekvenciálisan olvassuk, gyakran előfordul, hogy a fájlra való hivatkozás ugyanazt a lemez blokkot kéri, mint az előző. Ezt az effektust felerősítendő, a rendszer *előre* megpróbálja beolvasni a következő blokkot, még mielőtt hivatkoznánk rá, ezzel is gyorsítva a feldolgozást. Mindez az optimalizáció a felhasználók elől rejtve zajlik. Ha a felhasználó read hívást ad ki, a program várakozik, míg a kért adatokat meg nem kapja a pufferben.

Ezzel a háttér-információval fölfegyverkezve most már megnézhetjük, hogyan is zajlik a fájl B/K. Az open hatására a rendszer a megadott útvonalnév menti könyvtárakban keres. Ha a keresés sikeres volt, az i-csomópontot belső táblázatában helyezi el. A read és a write hívások esetén a rendszernek az aktuális fájlpozícióból ki kell számítania a blokkszámot. Az első 10 blokk címe mindig a memóriában van (az i-csomópontban), a magasabb sorszámú blokkok elérése először egy vagy több indirekt blokk beolvasását igényli. Az lseek csak a fájlmutató aktuális értékét változtatja anélkül, hogy bármilyen tényleges B/K-t végezne.

Most már könnyű a link-et és az unlink-et is megérteni. A link első argumentuma alapján megkeresi a megfelelő i-csomópont sorszámot. Azután a második argumentum számára új könyvtári bejegyzést készít, amelyben az első fájl i-csomópontjának számát helyezi el. Végül eggyel megnöveli az i-csomópontban a hivatkozások számát. Az unlink eltávolítja a könyvtári bejegyzést és csökkenti az i-csomópontban tárolt hivatkozási számot. Ha ez a szám 0, a fájlt is kitörli, annak minden blokkja visszakerül a szabad listára.

Virtuális B/K a Windows XP-n

Az XP több fájlrendszert is támogat, közülük a legfontosabb az **NTFS (NT File System)** és a **FAT (File Allocation Table)** fájlrendszer. Az első új, speciálisan az NT-re kifejlesztett fájlrendszer, a második a régi MS-DOS fájlrendszer (csupán hosszabb fájlnevekkel kiegészített formában), amelyet a Windows 95/98-on is használtak. Mivel a FAT fájlrendszer alapjaiban idejétmúlt, az NTFS-t fogjuk tanulmányozni a továbbiakban.

Az NTFS-fájlnevek hosszúsága legfeljebb 255 karakter lehet. A fájlnevek Unicode kódolásúak, ez lehetővé teszi, hogy a nem latin ábécét használó országok (például Japán, India vagy Izrael) felhasználói is anyanyelvükön adják meg a fájlneveket. (Valójában az XP belül mindenütt Unicode-ot használ. A Windows 2000-

rel kezdődő verziókból egyetlen bináris változat készül, amely mégis használható minden országban, és mindig a helyi nyelvet használja, mivel az összes menüt, hibaüzenetet stb. országfüggő konfigurációs fájlokban tárolják.) Az NTFS támogatja a kis- és a nagybetűk megkülönböztetését (tehát *foo* és *FOO* mást jelent). Sajnos a Win32 API a fájlnevek esetében nem teljesen, könyvtárnevek esetén meg egyáltalán nem támogatja a megkülönböztetést. Ezért a Win32-t használó programok esetén elveszítjük ezt az előnyt.

A UNIX-hoz hasonlóan a fájl bájtok sorozata, bár a maximális hossz $2^{64}-1$ is lehet. Léteznek fájlmutatók is, mint a UNIX-ban, de nem 32, hanem 64 bitesek, hogy a maximális méretű fájlokat is kezelni tudják. A fájl- és a könyvtárkezelő Win32 API-függvényhívások nagyjából hasonlítanak a UNIX-os megfelelőikre, de legtöbbjüknek több paramétere van, és más biztonsági modellen alapulnak. A fájl megnyitása (fájl)kezelőt ad vissza, ezt használjuk a fájlból való olvasásra vagy írásra. A UNIX-tól eltérően a kezelők nem kis egész számok, és általában nincsen előre definiált, a 0, 1 és 2-höz kapcsolódó standard bemenet, kimenet és hiba. Ezeket nekünk kell beállítani (kivéve a konzolmódot, amelyben már előre megnyitva rendelkezésünkre állnak). A Win32 API legfontosabb fájlkezelő függvényeit a 6.38. ábrán soroltuk fel.

API-függvény	UNIX	Jelentése
CreateFile	open	Fájl létrehozása vagy létező fájl megnyitása, kezelőt ad vissza
DeleteFile	unlink	Létező fájl törlése
CloseHandle	close	Fájl lezárása
ReadFile	read	Adatok olvasása fájlból
WriteFile	write	Adatok írása fájlba
SetFilePointer	lseek	A fájlmutató beállítása
GetFileAttributes	stat	A fájl tulajdonságainak lekérdezése
LockFile	fcntl	A fájl bizonyos részeinek zárolása a kölcsönös kizárás biztosításához
UnlockFile	fcntl	A fájl korábban zárolt régiójának felszabadítása

6.38. ábra. *A Win32 API fájl B/K-hoz kapcsolódó legfontosabb függvényei. A második oszlop a hozzájuk leginkább hasonló UNIX-rendszerhívást tartalmazza*

Vizsgáljuk meg röviden az egyes hívásokat. A CreateFile új fájl létrehozására használható, kezelőt ad vissza. Ezt az API-függvényt alkalmazzuk létező fájlok megnyitására is, mivel nincs open API-függvény. Az XP API-függvényhívások paramétereit azért nem soroltuk fel, mert túl sok van belőlük. Például a CreateFile a következő hét paraméterrel hívandó:

1. A létrehozandó vagy megnyitandó fájl nevére hivatkozó mutató.
2. Jelzők (flags), melyek azt mutatják, hogy a fájl olvasható, írható vagy mindkettő.
3. Jelzők, melyek azt mutatják, hogy a fájl megnyitható-e egyszerre több processzus által.
4. A biztonságleíróra hivatkozó mutató.
5. Jelzők, melyek azt mutatják, hogy mi a teendő, ha a fájl létezik/nem létezik.

6. Az archiválásra, tömörítésre stb. vonatkozó jelzők.
7. Annak a fájlnak kezelője, amelynek attribútumait „klónozni" akarjuk a mostani fájl létrehozásakor.

A 6.38. ábrán látható következő hat API-függvény eléggé hasonlít a megfelelő UNIX-rendszerhívásokra. Az utolsó kettővel fájlok régiói zárolhatók, illetve szabadíthatók fel. Ezekkel valósítható meg a processzusok garantált kölcsönös kizáráson alapuló hozzáférése.

Az API-függvények segítségével a 6.35. ábrán látható UNIX-változattal analóg fájlmásoló eljárás írható. Ezt mutatja a 6.39. ábra (a hibaellenőrzéseket most is elhagytuk). Úgy próbáltuk megtervezni, hogy a 6.35. ábrán látható szerkezetet utánozza. A gyakorlatban nincs szükség ilyen programra, hiszen létezik a CopyFile API-függvény (ami könyvtári függvényként az itteni programhoz hasonló dolgokat hajt végre).

Az XP a UNIX-hoz hasonló hierarchikus felépítésű fájlrendszert támogat. A komponensek közti elválasztójel azonban nem a /, hanem a \ (egy MS-DOS őskövület). Létezik a munkakönyvtár fogalma, az elérési útvonalnevek lehetnek abszolútak vagy relatívak. Lényeges eltérés azonban, hogy a UNIX megengedi a különböző lemezeken és gépeken található fájlrendszerek felcsatolását egyetlen könyvtárfába, és ezzel elrejti a lemez szerkezetét az összes program elől. Az NT 4.0-nak nincs meg ez a tulajdonsága, ezért az abszolút elérési útvonalneveknek a logikai lemezegység betűjelével kell kezdődni, például így: *C:\windows\system\foo.dll*. A Windows 2000 verziótól kezdve használható a fájlrendszerek UNIX-stílusú felcsatolása is.

A 6.40. ábrán ismét a legközelebbi UNIX-megfelelőjükkel együtt soroltuk fel a legfontosabb könyvtárkezelő API-függvényeket. A függvények remélhetőleg maguktól értetődők.

```
// Fájlok megnyitása olvasásra és írásra
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
   FILE_ATTRIBUTE_NORMAL, NULL);

// A fájl másolása
do {
   s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
   if (s >0 && count > 0) WriteFile(outhandle, buffer, count, &count, NULL);
} while (s > 0 && count > 0);

// A fájlok lezárása
CloseHandle(inhandle);
CloseHandle(outhandle);
```

6.39. ábra. *A Windows XP API-függvényeivel megírt fájlmásoló programrészlet. A C nyelvet használtuk, mivel a Java elrejti az alacsony szintű rendszerhívásokat, mi meg éppen azokat akarjuk megmutatni*

API-függvény	UNIX	Jelentése
CreateDirectory	mkdir	Új könyvtár létrehozása
RemoveDirectory	rmdir	Üres könyvtár törlése
FindFirstFile	opendir	Inicializálás könyvtárelemek olvasásának megkezdéséhez
FindNextFile	readdir	A következő könyvtári bejegyzés olvasása
MoveFile		Fájl mozgatása egyik könyvtárból a másikba
SetCurrentDirectory	chdir	A munkakönyvtár megváltoztatása

6.40. ábra. *A legfontosabb könyvtárkezelő Win32 API-függvények. A második oszlopban a hozzájuk legközelebb álló UNIX-rendszerhívás áll (ha van ilyen)*

Az XP-nek a legtöbb UNIX-nál lényegesen bonyolultabb biztonsági mechanizmusa van. Bár több száz, a biztonsággal kapcsolatos API-függvény létezik, az alapelvek megérthetők a következő rövid leírásból. Amikor a felhasználó bejelentkezik, kezdeti processzusa az operációs rendszertől egy **hozzáférést vezérlő jelet (access token)** kap. A vezérlő jel tartalmazza a felhasználó SID-jét (**Security ID, biztonsági azonosító**), ez írja le, hogy melyik biztonsági csoportoknak tagja, esetleges speciális privilégiumait és még sok mást. A hozzáférést vezérlő jel célja az összes biztonsági információ egyetlen, könnyen megtalálható helyen való koncentrálása. A processzus által létrehozott összes további processzus ugyanezt a vezérlő jelet örökli.

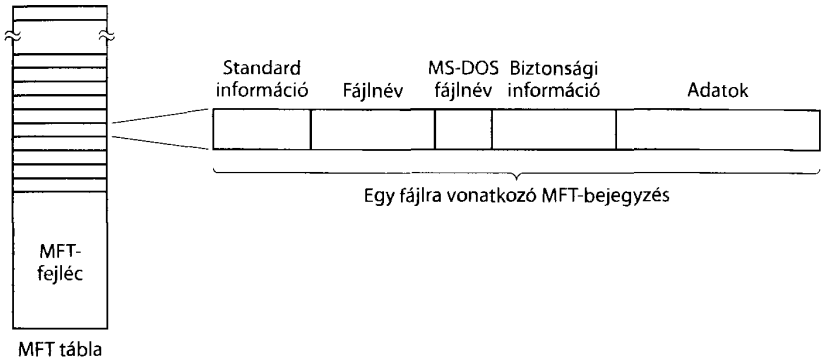
Az objektumok létrehozásánál megadható paraméterek egyike a **biztonságleíró (security descriptor)**. A leíró **ACL (Access Control List, elérést vezérlő lista)** nevű bejegyzések listájából áll. Minden bejegyzés az objektumon végezhető bizonyos műveletek halmazát tiltja vagy engedélyezi egyes csoportok vagy SID-ek számára. Például egy fájlnak lehet olyan biztonságleírója, amely szerint Elinor egyáltalán nem férhet hozzá a fájlhoz, Ken olvashatja, Linda olvashatja és írhatja, az XYZ csoport összes tagja pedig olvashatja, de mást nem tehet vele.

Amikor egy processzus az objektum megnyitásakor kapott kezelő segítségével valamilyen műveletet próbál végrehajtani az objektumon, a biztonságkezelő megkapja a processzus hozzáférést vezérlő jelét, és sorban végigmegy az ACL lista elemein. Mihelyst olyan bejegyzést talál, amely egyezik a hívó SID-jével, vagy a hívó valamelyik csoportjával, az ebben talált hozzáférések lesznek meghatározók. Ezért szokás a hozzáféréseket tiltó bejegyzéseket az engedélyező bejegyzések előtt elhelyezni a listában. Így az a felhasználó, akinek kifejezetten megtiltottuk a hozzáférést, a hátsó ajtón sem „osonhat be", még akkor sem, ha olyan csoport tagja, amelynek engedélyezett a hozzáférés. A biztonságleíró az objektumhoz való hozzáférések ellenőrzéséhez használható információt is tartalmaz.

Vizsgáljuk meg röviden, hogyan valósították meg a könyvtárakat és fájlokat az XP-ben. Minden lemezt önálló, statikus kötetekre (volume) osztottak fel. Ezek a UNIX-partíciók megfelelői. Minden kötetben vannak bittérképek, fájlok, könyvtárak és a benne foglalt információ kezeléséhez szükséges egyéb adatstruktúrák. A kötet **klaszterek (clusters)** lineáris sorozatából áll. A klaszterek mérete a kötet nagyságától függően rögzített 512 bájt és 64 KB közötti érték. A kötet kezdetétől számított, 64 biten ábrázolt eltolás segítségével lehet a klaszterekre hivatkozni.

A kötet legfontosabb adatstruktúrája a mesterfájltábla (MFT) (**Master File Table, MFT**), amely a kötet minden fájljáról és könyvtáráról tartalmaz egy bejegyzést. Ezek a UNIX i-csomópontjaival analóg szerepet játszanak. Az MFT maga is fájl, s mint ilyen, a köteten belül bárhol elhelyezhető. Ezzel megszabadulhatunk az olyan UNIX-ra jellemző problémáktól, mint az i-csomópontok táblájának közepén keletkező lemezhibák.

Az MFT a 6.41. ábrán látható. A kötetről szóló információkat tartalmazó fejléccel kezdődik: a gyökérkönyvtár(ra mutató pointer), a boot fájl, a hibás blokkokat tartalmazó fájl, a szabad lista adminisztrációja stb. Ezután a fájlonkénti, illetve könyvtárankénti bejegyzések következnek. Méretük 1 KB (ha a klaszterméret nem nagyobb vagy egyenlő, mint 2 KB). A bejegyzés tartalmazza a fájlra, illetve a könyvtárra vonatkozó összes metaadatot (adminisztratív információt). Több formátum is megengedett, egyikük a 6.41. ábrán látható.



6.41. ábra. *A Windows XP MFT táblája*

A standard információ mezőben a POSIX által megkövetelt időadatok, a kemény kötések (hard links) száma, a csak-olvasható és az archív bitek stb. állnak. Ez fix hosszúságú, és mindig van. A fájlnév változó hosszúságú, maximum 255 Unicode karakterből állhat. A fájloknak lehet MS-DOS neve is, hogy a régi 16 bites programok is el tudják érni. Ez a név a jól ismert 8 + 3 szabály szerint képezhető. Ha az igazi fájlnév szintén eleget tesz ennek a megszorításnak, nem használnak másodlagos MS-DOS fájlnevet.

Ezután a biztonsági információ következik. Az NT 4.0-ig terjedő verziókban itt maga a biztonságleíró állt. A Windows 2000-től kezdődően az összes biztonsági információt egyetlen fájlban gyűjtötték össze, a biztonsági mező egyszerűen ennek a fájlnak a megfelelő részére mutató pointer. Kis fájlok esetén a fájlhoz tartozó adatok maguk is az MFT-bejegyzésben találhatók. Az ötlet neve **közvetlen fájl (immediate file)**: lásd (Mullender és Tanenbaum, 1984). Valamivel nagyobb fájloknál az adatmezőben már az adatokat tartalmazó klaszterekre mutató pointerek állnak (vagy még inkább a fájl hossza és egyetlen klasztersorszám, amely az adatokat tartalmazó összefüggő klasztersorozat kezdetét adja meg, s tetszőleges mennyiségű

adatot tároló fájlokra alkalmazható). Ha egyetlen MFT-bejegyzésben nem tárolható az összes megkívánt információ, egy vagy több következő bejegyzés láncolt listaként hozzákapcsolható.

A maximális fájlméret 2^{64} bájt. Próbáljuk meg elképzelni, mekkora is lehet egy ilyen hosszúságú fájl. Ha a fájl tartalmát binárisan leírnánk, és minden 0 vagy 1 jegy 1 mm helyet foglalna el, akkor ez a 2^{64} mm-es lista 15 fényév hosszúságú lenne, jóval tovább nyúlna, mint a naprendszer, elérne az Alfa Centauriig, meg vissza.

Az NTFS fájlrendszernek még sok más érdekes tulajdonsága van, például adattömörítés, hibajavítás atomi tranzakciók felhasználásával. További információkat tartalmaz (Russovich és Solomon, 2005).

6.4.4. Példák processzusok kezelésére

Mind a UNIX, mind az XP megengedi, hogy egy feladatot (pszeudo)párhuzamosan futó, egymással kommunikáló processzusokra bontsunk fel a korábban tárgyalt termelő-fogyasztó példához hasonló stílusban. Ebben a szakaszban azt vizsgáljuk, hogy a két rendszer hogyan kezeli a processzusokat. Mindkettő támogatja az egyes processzusokon belül a szálakkal megvalósított párhuzamosságot is, ezzel szintén foglalkozunk majd.

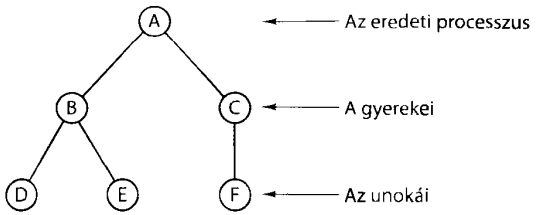
A UNIX processzuskezelése

A UNIX-processzus a fork rendszerhívás végrehajtásával bármikor létrehozhat egy olyan alprocesszust, amely saját maga pontos másolata. Az eredeti processzus neve **szülő (parent)**, az újé **gyerek (child)**. Közvetlenül a fork végrehajtása után a két processzus ugyanolyan, még ugyanazokat a közös fájlleírókat is használja. Ezután mindegyik elindul a saját útján, a másiktól függetlenül azt tesz, amit kíván.

A gyerek processzus sokszor zsonglőrködik még egy kicsit a fájlleírókkal, majd exec rendszerhívást hajt végre. Hatására az exec hívás paraméteréül megadott végrehajtható fájl tartalmának megfelelő program- és adatszegmenssel helyettesítődik az eredeti program- és adatszegmens. Amikor például a felhasználó az *xyz* parancsot gépeli be a terminálon, a parancsértelmező (shell) fork hívással létrehoz egy gyerek processzust, amely exec-et hajt végre az *xyz* program futtatásához.

A két processzus párhuzamosan fut (akár volt exec, akár nem), hacsak a szülő nem dönt úgy, hogy megvárja a gyerek processzus befejeződését, és futását csak azután folytatja. Ha várakozni akar, akkor wait vagy waitpid hívást hajt végre, amely megállítja, míg a gyerek be nem fejeződik az exit végrehajtásával. Ezután a szülő fut tovább.

A processzusok tetszőleges számú fork-ot hajthatnak végre, így „processzusfák" jönnek létre. A 6.42. ábrán például az *A* processzus két fork-ot hajtott végre, ezzel két gyereket, *B*-t és *C*-t hozott létre. Ezután *B* is kétszer hívta a fork-ot, *C* meg egyszer, s kialakult a végső, hat processzusból álló fa.



6.42. ábra. *UNIX-processzusfa*

A UNIX-processzusok a csőnek (pipe) nevezett struktúra segítségével kommunikálhatnak egymással. A cső egyfajta puffer, amelybe az egyik processzus „adatfolyamot" ír be, a másik processzus meg kiveszi ezeket az adatokat. A csőből mindig csak a beírásnak megfelelő sorrendben vehetünk ki bájtokat. Véletlen elérés nem lehetséges. A csövek nem őrzik meg az üzenetek határait, így ha az egyik processzus négy 128 bájtos írást hajt végre, a másik meg 512 bájtot olvas, az olvasó processzus egyszerre kiveszi az összes adatot, és semmi sem jelzi, hogy ezeket több művelet írta be.

A System V és a Solaris az **üzenetsorok (message queues)** révén másfajta kommunikációt is megenged. A processzusok új üzenetsort hozhatnak létre, vagy megnyithatnak meglévőt msgget használatával. Adott üzenetsorba a processzusok msgsnd hívásával üzeneteket küldhetnek, és msgrecv-tel üzenetet kérhetnek. Az így elküldött üzenetek több szempontból is eltérnek a csövekbe betöltöttektől. Először is, míg a cső csak egy bájtfolyam, itt megmaradnak az üzeneteket elválasztó határok. Másodszor, az üzeneteknek prioritása van, így a sürgősek a kevésbé fontosak elé ugorhatnak. Harmadszor, az üzeneteknek típusa van, és szükség esetén msgrecv-ben előírhatunk egy bizonyos típust.

Újabb kommunikációs mechanizmust jelent két vagy több processzus címtartományából bizonyos régió megosztott használata. A UNIX ezt a megosztott memóriát úgy kezeli, hogy ugyanazokat a lapokat az összes processzus virtuális címtartományába leképezi. Emiatt a többi processzus azonnal észreveszi, ha valamelyik processzus ír a megosztott régióba. Ezzel a mechanizmussal igen nagy sávszélességű kommunikációs útvonalat kapunk. A megosztott memóriával kapcsolatos rendszerhívások neve ilyesmi lehet: shmat, shmop stb.

A System V és a Solaris másik jellegzetessége a szemaforok használata. Ezek lényegében a termelő-fogyasztó példából megismert módon működnek.

Az összes UNIX-rendszer képes egyetlen processzuson belül több vezérlési szál kezelésére. Ezek a vezérlési szálak, vagy csak röviden **szálak (threads)** olyan könnyűsúlyú processzusokra hasonlítanak, melyeknek közös a címtartománya és minden, ami ahhoz kapcsolódik: fájlleírók, környezeti változók, külső időzítők. De minden szálnak saját utasításszámlálója, regiszterei és verme van. Ha egy szál blokkolódik (vagyis átmenetileg várakozni kényszerül, amíg a B/K befejeződik, vagy valamely egyéb esemény bekövetkezik), az ugyanabban a processzusban lévő többi szál még futhat. Ha egy processzuson belül két szál fut, és az egyik termelőként, a másik pedig fogyasztóként működik, hasonló, de nem pontosan ugyan-

olyan helyzet áll elő, mintha két egyszálas processzusunk lenne, melyek megosztva használják a puffert tartalmazó memóriarégiót. A különbségek onnan erednek, hogy az utóbbi esetben minden processzusnak saját fájlleírói stb. vannak, míg az elsőnél mindezek közösek. Korábbi termelő-fogyasztó példánk kapcsán már találkoztunk a Java-szálak használatával. A Java-futtatórendszer sokszor egy-egy operációsrendszer-szintű szálat használ saját szálai megvalósításához, de nem kell föltétlenül így dolgoznia.

A szálak hasznosságára vonatkozó példaként tekintsünk egy World Wide Web szervert. Az ilyen szerverek a memóriában található gyorsítóban helyezhetik el a gyakran használt weboldalakat. Ha a kérés a gyorsítóban lévő oldalra történt, akkor azonnal visszaküldhető a weboldal. Egyébként lemezről kell betölteni. Sajnos sokat kell várni a lemezre (kb. 20 ms a tipikus), ezalatt a processzus blokkolódik, nem tud újabb kéréseket kiszolgálni, még olyanokat sem, amelyek a gyorsítóban található lapokra vonatkoznak.

A megoldás a többszálú szerverprocesszus, melynek szálai mind a közös gyorsítót használják. Amikor valamelyik szál blokkolódik, a többiek még tudják fogadni az új kéréseket. Szálak nélkül csak több szerverprocesszussal tudnánk kikerülni a blokkolódást, de ekkor valószínűleg több példány kellene a gyorsítóból is, amivel értékes memóriát pocsékolnánk el. A szálakra vonatkozó UNIX-szabvány neve **pthreads**, és szabvány (P1003.1C) definiálta. Ez a szálak szinkronizálására és kezelésére vonatkozó hívásokat tartalmaz. A szabvány nem definiálja, hogy a szálak kezelése a kernel feladata, vagy pedig teljesen felhasználói módban megvalósítva fussanak. A leggyakrabban használt szálhívásokat a 6.43. ábrán soroltuk fel.

Szálhívás	Jelentése
pthread_create	Új szál létrehozása a hívó címtartományában
pthread_exit	A hívó szál befejezése
pthread_join	Várakozás szál befejeződésére
pthread_mutex_init	Új mutex létrehozása
pthread_mutex_destroy	Mutex megszüntetése
pthread_mutex_lock	Mutex zárolása
pthread_mutex_unlock	Mutex felszabadítása
Pthread_cond_init	Feltételváltozó létrehozása
pthread_cond_destroy	Feltételváltozó megszüntetése
pthread_cond_wait	Feltételváltozóra várakozás
pthread_cond_signal	A feltételváltozóra várakozó egyik szál elindítása

6.43. ábra. *A legfontosabb POSIX szálhívások*

Vizsgáljuk most meg röviden a 6.43. ábrán látható szálhívásokat. Az első hívás, pthread_create, egy új szálat hoz létre. A sikeres befejezés után a hívó címtartományában eggyel több szál fut, mint annak előtte. A szálak munkájuk elvégzése után a pthread_exit hívásával fejezhetik be működésüket. Másik szál befejezésére a pthread_join hívásával várakozhat egy szál. Ha az illető szál már befejeződött, a pthread_join is azonnal visszatér. Egyébként blokkolja a hívó szálat.

A szálak a **mutex** nevű zárolás segítségével szinkronizálhatják futásukat. A mutex tipikus feladata valamely erőforrás, például két szál által közösen használt puffer védelme. Elvárjuk, hogy a szálak az erőforráshoz való fordulás előtt zárolják, munkájuk befejezése után meg szabadítsák fel a mutexet. Ezzel biztosítjuk, hogy egyszerre csak egy szál érhesse el a megosztott erőforrást. Elkerülhető a versenyhelyzet, ha minden szál tiszteletben tartja ezt a protokollt. A mutexek olyanok, mint a bináris (0 vagy 1 értéket felvevő) szemaforok. A „mutex" név onnan származik, hogy valamely erőforráshoz kölcsönös kizáráson („mutual exclusion") alapuló hozzáférést biztosítunk segítségükkel.

Mutexek a pthread_mutex_init és a pthread_mutex_destroy hívásokkal hozhatók létre, illetve szüntethetők meg. A mutex két lehetséges állapota: zárolt (locked) és szabad (unlocked). Ha egy szál szabad mutexet próbál meg zárolni a pthread_lock hívással, beállítódik a zárolás, és tovább fut a hívó szál. Amennyiben zárolt mutexre vonatkozott a hívás, a hívó szál blokkolódik. A zárolást kérő szál munkájának befejezése után elvárjuk, hogy a mutexet szabadítsa fel a pthread_mutex_unlock hívással.

A mutexeket rövid távú zárolásra, például egy megosztott változó védelmére szánták. Nem valók hosszú távú szinkronizációra, mondjuk egy szalagegység felszabadulásának vezérlésére. Ilyen célokra a **feltételváltozók (condition variables)** alkalmasak. Létrehozásuk és megszüntetésük a pthread_cond_init, illetve a pthread_cond_destroy hívásokkal végezhető el.

A feltételváltozó használatakor egy szál várakozik, egy másik meg a változóra vonatkozó jelzéseket küld. Például, ha az egyik szál észreveszi, hogy a szalagegység, amelyet használni szeretne, éppen foglalt, akkor pthread_cond_wait hívást hajt végre azzal a feltételes változóval, amelyet minden szál egyezményesen a szalagegységhez rendelt. Amikor a szalagegységet használó szál munkáját befejezte (lehet, hogy csak órák múlva), a pthread_cond_signal segítségével engedélyezi a változóra várakozó szálak közül pontosan egynek a folytatást. Ha nem volt várakozó szál, a jelzés elvész. A feltételváltozók másképpen számolnak, mint a szemaforok. Szálakra, mutexekre és feltételváltozókra még néhány további műveletet is definiáltak.

A Windows XP processzuskezelése

Az XP több processzust támogat, ezek kommunikálhatnak és szinkronizálhatják futásukat. Minden processzus tartalmaz legalább egy szálat, amelyben viszont legalább egy **fonál (fiber)** van. A processzusok, a szálak és a fonalak együttesen nagyon általános eszközkészletet nyújtanak a párhuzamosság kezelésére akár egy-, akár többprocesszoros rendszereken. Új processzusokat a CreateProcess API-hívással hozunk létre. Ennek a függvénynek 10 paramétere, ezen belül mindegyiknek számos opciója van. Ez a konstrukció nyilvánvalóan sokkal bonyolultabb, mint a UNIX-séma, ahol a fork-nak nincs is paramétere, sőt az exec-nek is csupán 3 van: a végrehajtandó fájl nevére mutató pointer, az (elemzett) parancssor paramétereinek tömbje és a környezeti változók tömbje. A CreateProcess 10 paraméterének rövid ismertetése:

1. A végrehajtandó fájl nevére mutató pointer.
2. Az eredeti (nem elemzett) parancssor.
3. A processzus biztonságleírójára mutató pointer.
4. A kezdeti szál biztonságleírójára mutató pointer.
5. Jelzőbit, amely azt mondja meg, hogy az új processzus örökli-e létrehozójának kezelőit.
6. Különféle jelzők, például a hibamód, prioritás, nyomkövetés, konzolok.
7. A környezeti változókra mutató pointer.
8. Az új processzus munkakönyvtárának nevére mutató pointer.
9. A képernyőn megjelenő kiindulási ablakot leíró struktúrára mutató pointer.
10. A hívónak 18 értéket visszaadó struktúrára mutató pointer.

Az XP nem követel meg semmiféle szülő-gyerek vagy egyéb hierarchiát. Minden processzus születésénél fogva egyenlő. Mivel azonban a létrehozó processzusnak visszaadott 18 paraméter egyike az új processzus kezelője (ez az új processzus fölötti számottevő ellenőrzést tesz lehetővé), mégis van egy implicit hierarchia annak megfelelően, hogy ki kit vezérelhet. Bár ezek a kezelők közvetlenül nem adhatók át más processzusoknak, a processzusok készíthetnek más processzusok számára megfelelő kezelőket, és ezeket aztán tovább is adhatják. Az implicit processzushierarchia tehát nem mindig marad meg tartósan.

Kezdetben minden XP-processzus egyetlen szállal születik, de később újabb szálakat is létrehozhat. A szálak létrehozása egyszerűbb a processzusokénál. A CreateThread függvénynek 10 helyett csak 6 paramétere van: a biztonságleíró, a verem mérete, a kezdőcím, egy felhasználó által definiált paraméter, a szál kezdeti állapota (futáskész vagy blokkolt) és a szál azonosítója. A kernel végzi a szálak létrehozását, tehát nyilvánvalóan tud létezésükről (vagyis nem teljesen felhasználói módban valósították meg, mint más rendszerekben).

A kernel ütemezője nem csak a legközelebb futtatandó processzust, hanem ezen belül a futtatandó szálat is kijelöli. Vagyis a kernel mindig tudja, mely szálak futhatnak és melyek blokkoltak. Mivel a szálak is kernelobjektumok, azoknak is van biztonságleírójuk és kezelőjük. Egy processzus vezérelheti más processzusok szálait is, ugyanis ezek a kezelők más processzusoknak továbbadhatók. Nyomkövetőkben (debugger) jól kihasználható ez a tulajdonság.

Az XP-szálak használata viszonylag költséges, mivel a szálak közti átkapcsoláshoz a kernelbe való belépés, majd később az onnan való kilépés szükséges. A nagyon könnyűsúlyú pszeudopárhuzamossághoz az XP **fonalak**at **(fibers)** biztosít. Ezek a szálakhoz hasonlítanak, de ütemezésüket felhasználói módban végzi az őket létrehozó processzus (vagy annak futtató rendszere). Ugyanúgy, mint ahogy a processzusok többszálúak lehetnek, egy szál is tartalmazhat több fonalat. Eltérés ott van, hogy ha egy fonál logikailag blokkolódik, akkor a blokkolt fonalak sorába teszi be magát, és az őt tartalmazó szálon belüli környezetből választ másik futó fonalat. A kernel nem veszi észre ezt az átmenetet, mert a szál tovább fut, bár lehet, hogy először az egyik fonalat futtatta, most meg egy másikat. A kernel csak a processzusokat és a szálakat kezeli, a fonalakat nem. A fonalak hasznosak lehetnek például, ha olyan programokat akarunk XP-re átvinni, amelyek saját szálaikat kezelték eredetileg.

A processzusok sokféle módon kommunikálhatnak: csövekkel, nevesített csövekkel, levélszekrényekkel, socketekkel, távoli eljáráshívásokkal és megosztott fájlokkal. A csöveknek két módja lehet, bájt vagy üzenet (ezt létrehozásukkor kell eldönteni). A bájt módú csövek ugyanúgy működnek, mint a UNIX-ban. Az üzenetmódú csövek lényegében hasonlók, de megőrzik az üzenetek határait, tehát a négyszer beírt 128 bájt négy 128 bájtos üzenetként olvasható, és nem olvasható egyetlen 512 bájtos üzenetként, mint a bájt módú csöveknél. Léteznek nevesített csövek is, ugyanúgy két módjuk lehet, mint a közönséges csöveknek. A nevesített csövek hálózaton keresztül is használhatók, a közönségesek nem.

A **mailslot (levélbedobó nyílás)** olyan XP-sajátosság, amely hiányzik a UNIX-ból. Bizonyos szempontból hasonlít a csövekhez, de nem mindenben. Először is ez egyirányú, míg a cső kétirányú. Hálózaton át is használható, de az üzenet eljutása nem garantált. Végül a küldő processzus „üzenetszórást" is végezhet segítségével; nem egy, hanem sok fogadóhoz eljuttatva ugyanazt a küldeményt.

A socketek a csövekhez hasonlítanak azzal az eltéréssel, hogy rendszerint különböző gépeken futó processzusokat kapcsolnak össze. De használhatók ugyanazon a gépen futó processzusok összekapcsolására is. Általában kevés előnnyel jár a gépen belüli kommunikációban a socketek csővel vagy elnevezett csővel történő összekapcsolása.

A távoli eljáráshívások arra adnak módot, hogy az A processzus a B processzussal annak címtartományában végrehajthasson A helyett egy eljárást, és a futás eredményét visszajuttassa A-nak. Különböző megszorítások léteznek a paraméterekre vonatkozóan. Például értelmetlen dolog mutatót átadni egy másik processzusnak.

Végül a processzusok úgy is használhatnak megosztott memóriát, hogy egy időben ugyanazt a fájlt képezik le memóriájukba. Az egyik processzus által végrehajtott minden írás megjelenik az összes többi címtartományában. Ezzel a mechanizmussal könnyen megvalósítható a termelő-fogyasztó példában használt puffer.

Ugyanúgy, ahogy számos processzusok közti kommunikációs eszközt kínál az XP, sokféle szinkronizációs mechanizmust is felajánl. Idetartoznak a szemaforok, a mutexek, a kritikus régiók és az események. Ezek mind szálakon és nem processzusokon hatnak, tehát amikor egy szál blokkolódik a szemaforon, a processzus többi szálát (ha vannak ilyenek) mindez nem érinti, nyugodtan futhatnak tovább. Szemafort a CreateSemaphore API-függvénnyel hozhatunk létre, amellyel inicializálhatjuk, és a maximális értékét is megadhatjuk. A szemaforok kernelobjektumok, tehát van biztonságleírójuk és kezelőjük. A szemafor kezelőjéről másolat készíthető a DuplicateHandle hívással. Ennek eredményét más processzusoknak átadva, több processzus is hozzáigazíthatja futását ugyanahhoz a szemaforhoz. Létezik az up és a down hívás, bár az elég furcsa ReleaseSemaphore (up) és WaitForSingleObject (down) nevekre hallgatnak. A WaitForSingleObject hívásakor időkorlát (timeout) is megadható, hogy a hívó szál végül fölszabaduljon akkor is, ha a szemafor értéke 0 marad (bár az időzítések használata újra versenyhelyzetekhez vezethet).

A mutexek szintén szinkronizációra szolgáló kernelobjektumok, de annyival egyszerűbbek, hogy nincsen számlálójuk. Lényegében zárolást végeznek a zárolásra (WaitForSingleObject) és a felszabadításra (ReleaseMutex) vonatkozó két API-függvény segítségével. A szemaforok kezelőihez hasonlóan a mutexek kezelői is

duplikálhatók és processzusok között átadhatók, így különböző processzusok szálai is elérhetik ugyanazt a mutexet.

A harmadik szinkronizációs mechanizmus a **kritikus szakaszokon** (**critical sections**) alapul, melyek hasonlítanak a mutexekhez, de a létrehozó szál címtartományára nézve lokálisak. Mivel a kritikus szakaszok nem kernelobjektumok, nincsen sem biztonságleírójuk, sem kezelőjük és nem adhatók át más processzusoknak. A zárolás és a felszabadítás az EnterCriticalSection, illetve a LeaveCriticalSection hívásokkal végezhető. Mivel ezek a függvények teljesen felhasználói módban hajtódnak végre, sokkal gyorsabbak a mutexeknél.

Az utolsó szikronizációs mechanizmus **eseményeknek** (**events**) nevezett objektumokat használ. A szálak a WaitForSingleObject hívásával várhatnak egy esemény bekövetkezésére. Fölszabadítható egyetlen eseményre váró szál a SetEvent függvénnyel vagy az összes, a megadott eseményre váró szál a PulseEvent-tel. Az események sokfélék lehetnek és sok opciójuk is lehet.

Az események, a mutexek és a szemaforok a nevesített csövekhez hasonlóan mind névvel láthatók el és a fájlrendszerben tárolhatók. Két vagy több processzus úgy is szinkronizálható, hogy ugyanazt az eseményt, mutexet vagy szemafort nyitják meg, és nem egyetlen objektum másolatainak kezelőit adjuk át több processzusnak, bár az utóbbi megközelítés is járható.

6.5. Összefoglalás

Az operációs rendszer felfogható az ISA-szinten hiányzó bizonyos architekturális sajátosságok értelmezőjeként. Ezek közül legfontosabb a virtuális memória, a virtuális B/K utasítások és a párhuzamos programozási lehetőségek.

A virtuális memória olyan architekturális sajátosság, amelynek célja, hogy a programok a gépben lévő memóriánál nagyobb címtartományt használhassanak, vagy hogy konzisztens és rugalmas mechanizmust biztosítson a memória védelmére és megosztására. Megvalósítható tiszta lapozással, tiszta szegmentálással vagy a kettő kombinálásával. A tiszta lapozás esetén a címtartományt azonos méretű lapokra osztjuk fel. Ezek közül némelyeket leképezünk a fizikai lapkeretekre. Mások nincsenek leképezve. A leképezett lapra vonatkozó hivatkozást az MMU fordítja le a helyes fizikai címre. A le nem képezett lapra való hivatkozás laphibát okoz. A Pentium 4-ben és az UltraSPARC III-ban lévő MMU támogatja a virtuális memóriát és a lapozást.

A legfontosabb B/K absztrakció ezen a szinten a fájl fogalma. A fájl bájtok vagy logikai rekordok sorozatából áll, melyek anélkül írhatók és olvashatók, hogy ismernénk a lemezes, a szalagos és más B/K egységek működését. A fájlok elérhetők sorosan és véletlenszerűen a rekordszámok vagy kulcsok használatával. A könyvtárak a fájlok csoportosítására használhatók. A fájlok tárolhatók a lemezek egymás utáni szektorain vagy szétszórtan. Az utóbbi esetben, melyet általánosan alkalmaznak a merevlemezes egységeken, megfelelő adatstruktúrákra van szükség, hogy megtaláljuk a fájlhoz tartozó összes blokkot. A szabad lemezterület listák vagy bittérképek segítségével tartható számon.

A párhuzamos feldolgozást gyakran egyetlen CPU-n időosztással több CPU-t szimulálva valósítják meg. A processzusok közti ellenőrizetlen kölcsönhatások versenyhelyzetekhez vezethetnek. Ennek a problémának a megoldására szinkronizációs primitíveket vezettek be, közülük a legegyszerűbbek a szemaforok. A termelő-fogyasztó problémák szemaforok használatával egyszerűen és elegánsan megoldhatók.

A bonyolult operációs rendszerek két példája a UNIX és az XP. Mindkettő támogatja a lapozást és a memóriába leképezett fájlokat. Szintén mindegyikükben használhatók bájtsorozatokból álló fájlokból felépülő hierarchikus fájlrendszerek. Végül mindkettőben vannak processzusok és szálak, valamint a szinkronizálásukhoz szükséges eszközök.

6.6. Feladatok

1. Miért csak részben értelmezi az operációs rendszer a 3. szintű utasításokat, holott a mikroprogram az összes ISA-szintű utasítást értelmezi?
2. Tegyük fel, hogy a számítógép 32 bites, bájtonként címezhető, 4 KB-os lapméretű virtuális memóriát használ. Hány lapból áll a virtuális címtartomány?
3. Szükségszerű, hogy a lapok mérete 2 hatványa legyen? Elméletileg megvalósítható lenne mondjuk a 4000 bájtos lapméret is? Ha igen, mennyire lenne praktikus?
4. Legyen a virtuális memória lapmérete 1024 szó, álljon 8 virtuális lapból, a fizikai memória pedig tartalmazzon 4 lapkeretet. A laptábla a következő:

Virtuális lap	Lapkeret
0	3
1	1
2	Nincs a memóriában
3	Nincs a memóriában
4	2
5	Nincs a memóriában
6	0
7	Nincs a memóriában

   a) Soroljuk fel az összes laphiányt okozó virtuális címet.
   b) Milyen fizikai címek felelnek meg a 0, 3728, 1023, 1024, 1025, 7800 és 4096 címeknek?
5. Egy számítógép 16 lapos virtuális memóriát és csupán 4 lapkeretet tartalmaz. Kezdetben a memória üres. A program a következő sorrendben hivatkozik a virtuális lapokra:
   0, 7, 2, 7, 5, 8, 9, 2, 4.

a) Az LRU esetén melyik hivatkozás okoz laphiányt?
b) A FIFO alkalmazásakor mikor lép fel laphiány?

6. A 6.1.4. alfejezetben a FIFO lapcserélő stratégiát megvalósító algoritmust ismertettünk. Tervezzünk ennél hatékonyabbat. *Tipp:* az éppen betöltött oldal számlálóját külön is frissíthetjük anélkül, hogy a többiét változtatnánk.
7. A szövegben tárgyalt lapozásos rendszereknél a lapkezelő az ISA-szinthez tartozott, ezért egyetlen OSM-szintű program címtartományában sem volt benne. A valóságban a lapkezelő szintén elfoglal néhány lapot, és bizonyos körülmények között (például a FIFO lapkezelési elvnél) maga is kikerülhet a memóriából. Mi történne, ha olyankor következne be laphiány, amikor a laphiánykezelő nincs a memóriában? Hogyan kezelhető ez az eset?
8. Nincsenek minden számítógép hardverében olyan bitek, amelyek automatikusan beállítódnak, ha írunk a lapokra. Ámbár célszerű nyilvántartani, hogy mely lapok módosultak, hogy ne kelljen a legrosszabb esetet feltételeznünk, vagyis minden lapot visszamásolnunk a diszkre. Hogyan tartja nyilván az operációs rendszer a tiszta és a szennyezett lapokat, ha azt tételezzük fel, hogy minden laphoz olyan hardverbitek tartoznak, amelyekkel külön engedélyezhető az olvasás, az írás és a végrehajtás?
9. Tegyük fel, hogy lapozott szegmensekkel működő szegmentált memóriánk van. A virtuális címek 2 bites szegmensszámból, 2 bites lapszámból és a lapon belüli eltolásból (offsetből) állnak. A 32 KB memóriát 2 KB-os lapokra osztottuk. A szegmensek csak olvashatók, olvashatók/végrehajthatók, olvashatók/írhatók vagy olvashatók/írhatók/végrehajthatók lehetnek. A laptáblák és a jogosultságok legyenek a következők:

0. szegmens		1. szegmens		2. szegmens	3. szegmens	
Csak olvasható		Olvasható/végrehajtható		Olvasható/írható/végrehajtható	Olvasható/írható	
Virtuális oldal	Lapkeret	Virtuális oldal	Lapkeret		Virtuális oldal	Lapkeret
0	9	0	A lemezen	A laptábla nincs a memóriában	0	14
1	3	1	0		1	1
2	A lemezen	2	15		2	6
3	12	3	8		3	A lemezen

Soroljuk fel, hogy a következő virtuálismemória-hozzáférések milyen fizikai címekre vonatkoznak. Ha hozzáférési hiba lép fel, adjuk meg a típusát is.

Elérés módja	Szegmens	Lap	Lapon belüli eltolás
1. adat betöltése	0	1	1
2. adat betöltése	1	1	10
3. adat betöltése	3	3	2047
4. adat tárolása	0	1	4
5. adat tárolása	3	1	2
6. adat tárolása	3	0	14
7. programelágazás ide	1	3	100
8. adat betöltése	0	2	50
9. adat betöltése	2	0	5
10. programelágazás ide	3	0	60

10. Némely számítógép felhasználói szinten is megengedi a direkt B/K műveleteket. Például a program közvetlenül a lemezről kezdhet adatokat áttölteni a felhasználó processzusában található pufferbe. Okozhat-e ez bármilyen problémát, ha a virtuális memória megvalósításánál tömörítést alkalmazunk?
11. A fájlok memóriába való leképezését megengedő operációs rendszerek mindig megkövetelik, hogy a fájl képe laphatáron kezdődjön. 4 KB méretű lapok esetén például a fájl leképezhető a 4096 címtől kezdődően, de 5000-től kezdve nem. Miért?
12. Amikor a Pentium 4-en szegmensregisztert töltünk be, a megfelelő leíró is betöltődik a szegmensregiszter láthatatlan részébe. Mit gondol, miért választották az Intel tervezői ezt a megoldást?
13. A Pentium 4-en futó program a 10. lokális szegmensen belüli 8000 eltolásra (offsetre) hivatkozik. A szegmenshez tartozó LDT BASE mezőjében 10000 áll. A lapkönyvtár melyik bejegyzését használja a Pentium 4? Mennyi a lapszám? Mennyi az eltolás?
14. Vizsgáljunk meg néhány olyan algoritmust, amely szegmensek eltávolítására alkalmas nem lapozott szegmentált memória esetén.
15. Hasonlítsa össze a külső és a belső elaprózódást. Hogyan csökkenthető hatásuk?
16. A szupermarketek is rendszeresen szembesülnek a virtuális memóriát használó rendszerek lapkezeléséhez hasonló problémákkal. A polcaik összterülete állandó, ezen kell egyre több terméket elhelyezniük. Ha fontos új termék, például egy 100%-os hatékonyságú kutyaeledel, jelenik meg, valamelyik régebbi termék eltávolításával kell helyet szorítani neki. Az LRU és a FIFO a két kézenfekvő helyettesítési algoritmus. Melyiket részesítené előnyben?
17. A gyorsítótárazás és a lapozás bizonyos értelemben nagyon hasonló. Mindkét esetben két memóriaszint van (az elsőnél a gyorsítótár és a főmemória, az utóbbinál pedig a főmemória és a lemez). A fejezetben kis és nagy (lemez) lapokat támogató érveket egyaránt megvizsgáltunk. A gyorsítósorokkal kapcsolatban is érvényesek ezek az érvelések?
18. Miért követeli meg sok operációs rendszer, hogy olvasás előtt az open rendszerhívással explicit módon megnyissuk a fájlokat?

19. Hasonlítsuk össze a szabad helyek nyilvántartásának két módszerét (a bittérképet és a lyuklistát) egy olyan diszk esetében, amelynek 800 cilindere van, mindegyiken 5 sávval s ezeken egyenként 32 szektorral. Hány lyuk keletkezése után lenne a lista nagyobb a bittérképnél? Feltételezzük, hogy a helyfoglalás egysége a szektor, a lyukak címét pedig 32 bites formában tároljuk.
20. A diszkkezelés hatékonyságának előzetes becsléséhez célszerű a helyfoglalást modellezni. Tekintsük a diszket olyan $N >> 1$ számú szektorból álló lineáris címtartománynak, amelyben adatblokkok sora után egy lyuk, majd megint valahány adatblokk, újra egy lyuk stb. ismétlődik. Ha a kísérleti mérések szerint az adatblokkokat tartalmazó részek és a lyukak hossza ugyanazt a valószínűségi eloszlást követi: az i szektoros hosszúság valószínűsége 2^{-i}. Mekkora a lyukak számának várható értéke?
21. Egy bizonyos számítógépen a programok tetszés szerinti számú, méretét futás közben dinamikusan változtató fájlt hozhatnak létre. A fájlok végső méretéről sem kell előzetesen tájékoztatni az operációs rendszert. Tárolhatók-e egymás utáni szektorokban a fájlok? Indokolja válaszát.
22. A különböző fájlrendszerek tanulmányozása azt mutatta, hogy a fájlok több mint fele néhány KB-os, vagy még kisebb, sőt a fájlok elsöprő többsége kisebb nagyjából 8 KB-nál. Másrészt viszont általában az összes fájl méret szerint legnagyobb 10%-a foglalja el az összes felhasznált lemezterület nagyjából 90%-át. Ezekből az adatokból milyen következtetést vonhatunk le a lemez blokkok méretére vonatkozóan?
23. Tekintsük a szemaforok operációs rendszerekben történő megvalósításának következő lehetőségét. Valahányszor up vagy down művelet végrehajtásához kezd a CPU egy szemaforon (a memóriában tárolt egész számon), előtte a prioritási vagy maszkolási bitek beállításával letiltja a megszakításokat. Azután betölti a szemafort, módosítja, majd végrehajtja a megfelelő programelágazást. Működik-e ez a módszer, ha
    a) egyetlen CPU van, amely 100 ms-os időközönként kapcsol át a processzusok között;
    b) 2 CPU van, s ezek megosztva használják a szemafort tartalmazó memóriát?
24. A Bombabiztos Operációs Rendszereket Gyártó Cég ügyfeleitől panaszok érkeztek a szemafor műveleteket is végző legújabb termékükkel kapcsolatban. Azt nehezményezték, hogy szerintük „erkölcstelen dolog" a processzusok blokkolása, vagy ahogy ők mondták, „munka közbeni elaltatása". Mivel a cég fő célkitűzése a felhasználók igényeinek maradéktalan kielégítése, az up és a down mellett harmadikként javasolta a peek nevű kiegészítő művelet bevezetését. A peek egyszerűen csak megvizsgálja a szemafort, anélkül hogy változtatna rajta, vagy blokkolná a processzust. Így a blokkolást erkölcstelennek ítélő programok a down művelet előtt megvizsgálhatják a szemafort, hogy biztonságos-e az operáció végrehajtása. Működne-e ez az elképzelés, ha három vagy több processzus használja a szemafort? És két processzus esetén?
25. Készítsen olyan táblázatot, amely az idő függvényében (0-tól 1000 ms-ig) mutatja, hogy az alábbi P1, P2 és P3 processzusok közül éppen melyik fut, és melyik van blokkolt állapotban. Mindhárom processzus up és down műveleteket végez ugyanazon a szemaforon. Két blokkolt processzus mellett végrehajtott up után a kisebb sorszámú indul újra, vagyis előnyben részesítjük P1-et P2-vel és P3-mal szemben stb. Kezdetben a szemafor értéke 1, és fut mindhárom processzus.
    t = 100-nál P1 down-t hajt végre
    t = 200-nál P1 down-t hajt végre
    t = 300-nál P2 up-ot hajt végre
    t = 400-nál P3 down-t hajt végre
    t = 500-nál P1 down-t hajt végre
    t = 600-nál P2 up-ot hajt végre
    t = 700-nál P2 down-t hajt végre
    t = 800-nál P1 up-ot hajt végre
    t = 900-nál P1 up-ot hajt végre

26. A repülési helyfoglalási rendszereknél biztosítani kell, hogy amíg az egyik processzus egy fájlon dolgozik, másik processzus ne használhassa ugyanezt a fájlt. Máskülönben előfordulhatna, hogy két különböző jegyiroda két processzusa véletlenül egyszerre adná el valamelyik járat utolsó jegyét. Javasoljon olyan szemaforokat használó szinkronizációs módszert, amely biztosítja, hogy bármely fájlt egyszerre csak egy processzus érhessen el (feltesszük, hogy a processzusok betartják a megadott szabályokat).
27. Közös memóriát használó, több CPU-t tartalmazó gépeken a rendszermérnökök gyakran segítik a **Tesztelés és Zárolás (Test and Set Lock)** utasítás bevezetésével a szemaforok megvalósítását. TSL X az X memóriarekeszt teszteli. Ha X tartalma 0, akkor egyetlen, oszthatatlan memóriaciklus során 1-re állítódik be, és a program átugorja az utána következő utasítást. Ha X nem 0, TSL üres utasításként viselkedik. TSL-t felhasználva megírhatók a következő tulajdonságokkal bíró *lock* és *unlock* eljárások. A *lock*(x) azt ellenőrzi, hogy x zárolva van-e. Ha nem, zárolja x-et, és visszaadja a vezérlést. Amennyiben x már zárolva van, akkor addig várakozik, míg x fel nem szabadul, akkor zárolja és visszaadja a vezérlést. *unlock* hívása a meglévő zárolást szabadítja fel. Ha használat előtt minden processzus zárolja a szemafor tábláját, akkor egyszerre csak egy processzus babrálhatja a változókat és a pointereket, tehát megelőzhető a versenyhelyzetek kialakulása. Írja meg assembly nyelven a *lock* és az *unlock* eljárásokat (összegyűjtve az ehhez szükséges feltételeket).
28. Sorolja fel a 65 szó hosszúságú körkörös pufferre kapott *in* és *out* értékeket a következő műveletek végrehajtása után (mindkét mutató 0-ról indul):
    a) 22 szót beteszünk
    b) 9 szót kiveszünk
    c) 40 szót beteszünk
    d) 17 szót kiveszünk
    e) 12 szót beteszünk
    f) 45 szót kiveszünk
    g) 8 szót beteszünk
    h) 11 szót kiveszünk

29. Tegyük fel, hogy olyan UNIX-verziónk van, amely 2 KB-os diszk blokkokat használ, és az (egyszeres, kétszeres vagy háromszoros) indirekt blokkokban 512 diszk címet tárol. Mi lenne a maximális fájlméret? Feltesszük, hogy a fájlmutatók 64 bitesek.
30. Tegyük fel, hogy az unlink("/usr/ast/bin/games3") UNIX-rendszerhívást a 6.36. ábrának megfelelő környezetben hajtottuk végre. Részletesen írja le, hogy milyen változások mentek végre a könyvtárrendszerben.
31. Képzelje el, hogy a UNIX-rendszert kevés memóriával rendelkező mikroszámítógépen kellett megvalósítania. Hosszú ideig tartó erőlködés után sem sikerült még cipőkanállal sem begyömöszölni a rendszert a memóriába. Ezért úgy döntött, hogy a siker érdekében feláldoz egy véletlenszerűen kiválasztott rendszerhívást. A pipe lett az áldozat, amely a processzusok közti bájtsorozatok továbbításához szükséges csöveket hozza létre. Megoldható-e ekkor is a B/K átirányítása? Mi lesz a csővezetékekkel?
32. Az Egyenlő Jogokat a Fájlleíróknak Bizottság tüntetést szervez a UNIX-rendszer ellen, mivel az fájlleíróként mindig a legkisebb, éppen nem használt számot adja vissza. Következésképpen a nagyobb fájlleírókat alig-alig használják. Tervük szerint a legkisebb, éppen nem használt szám helyett inkább a program által még nem használt legkisebb számot kellene választani. Azt állítják, hogy ez a változat könnyen megvalósítható, nincs hatással a meglévő programokra, és méltányosabb is. Mi erről a véleménye?
33. Készíthető-e az XP-n olyan hozzáférést vezérlő lista, hogy egy bizonyos fájlhoz Roberta egyáltalán ne férhessen hozzá, ugyanekkor bárki más teljes hozzáférést kapjon? Mit gondol, hogyan valósítható ez meg?
34. Adjon meg két módszert a termelő-fogyasztó probléma megosztott pufferekkel és szemaforokkal való programozására XP-n. Gondolja át a megosztott pufferek megvalósítását mindkét esetben.
35. A lapkezelő algoritmusokat általában szimulációval tesztelik. Ebben a gyakorlatban az a feladata, hogy készítse el egy olyan gép szimulátorát, amely 64 darab 1 KB méretű lapból álló virtuális memóriát kezel. A szimulátor egyetlen 64 elemű táblázatban tartsa nyilván a lapoknak megfelelő fizikai lapkereteket. A szimulátornak a decimálisan megadott virtuális címeket egy fájlból soronként kell beolvasnia. Ha a megfelelő lap a memóriában van, csak egy találatot kell feljegyeznie. Ha nem ez a helyzet, akkor meg kell hívnia a laphelyettesítő eljárást, amely kiválasztja az eltávolítandó lapot (vagyis a felülírandó táblaelemet), és egy laphiányt kell regisztrálnia. Tényleges lapcsere nem történik. Generáljon egy véletlen számokat tartalmazó fájlt, és tesztelje mind az LRU, mind a FIFO hatékonyságát. Ezután generáljon a lokalitás szimulálására olyan fájlt, amelyben a címek x százaléka négy bájttal nagyobb a megelőzőnél. Futtasson teszteket x különböző értékeire, és összesítse az eredményeket.
36. A 6.25. ábrán látható program végzetes versenyhelyzetet tartalmaz, mivel a két szál a közös változókat szabályozatlanul éri el, anélkül hogy szemaforokat vagy más kölcsönös kizárási technikát használnának. Futtassuk le a programot, és nézzük meg, mennyi idő múlva fagy le. Ha nem tudjuk lefagyasztani, módosítsuk úgy, hogy az *m.in* és az *m.out* beállítása közé iktassunk be valami

számolást, ezzel növelve a „sebezhetőségi ablakot". Mennyi számolást kell beiktatnunk ahhoz, hogy mondjuk, legalább óránként egyszer lefagyjon?
37. Írjon olyan UNIX- vagy XP-programot, amelynek bemenő paramétere egy könyvtárnév. A program feladata a könyvtárban található fájlok listázása. Soronként egy fájl nevét és méretét írja ki. A fájlneveket a könyvtárban való előfordulásuk sorrendjében írassa ki. A könyvtár fel nem használt helyeit (kihasználatlan tartalmú) sorokkal jelezze.

7. Az assembly nyelv szintje

A legmodernebb számítógépeken meglevő három különböző szintet a 4., 5. és 6. fejezetben tárgyaltuk. Ez a fejezet egy olyan további szinttel foglalkozik, amely lényegében minden korszerű számítógépnél megtalálható, ez az assembly nyelv szintje. Az assembly nyelv szintje egy lényeges dologban eltér a mikroarchitektúra, az ISA- és az operációs rendszer gép szintjétől – megvalósítása ugyanis értelmezés helyett inkább fordítással történik.

Az olyan programokat, amelyek egy adott nyelven írt alkalmazói programot egy másik nyelvre alakítanak át, **fordítók**nak nevezzük. Azt a nyelvet, amelyben az eredeti program íródik, **forrásnyelv**nek hívják, **célnyelv**nek pedig azt, amelyre az átalakítás történik. A forrásnyelv és a célnyelv is egy-egy szintet definiálnak. Ha a processzor képes forrásnyelvű program végrehajtására, akkor szükségtelen a forrásnyelvű program célnyelvre való átalakítása.

Akkor van szükség fordításra, ha a processzor (vagy a hardver, vagy az értelmező) értelmezni tudja a célnyelvet, de a forrásnyelvet nem. Helyes fordítás esetén az átalakított program futásának eredménye ugyanaz, mintha a forrásnyelvi programot közvetlenül értelmezni tudó processzor végezte volna a futtatást. Következésképpen létrehozható olyan szint, amelyen írt programokat a processzorok számára nem kell először egy célszintre átalakítani, hogy aztán az eredményül kapott célszintű programot már végre tudják hajtani.

Fontos látni a fordítás és az értelmezés közötti különbséget. Fordítás esetén az eredeti forrásnyelvű program végrehajtása nem közvetlen. Először az eredeti programmal ekvivalens **tárgyprogram**ra vagy **végrehajtható bináris program**ra való átalakítás történik, és ennek a folyamatnak csak a teljes befejezése után kerül sor a tárgyprogram végrehajtására. Tehát a fordításnak az alábbi két elkülönülő lépése van:

1. Egy célnyelvű ekvivalens program előállítása.
2. Az újként generált program végrehajtása.

Ezek a lépések párhuzamosan nem fordulnak elő. A második lépés addig nem kezdődhet, amíg az első be nem fejeződött. Az értelmezőnél csak egyetlen lépés van: az eredeti forrásprogram végrehajtása. Előzetesen nincs szükség egy ekvivalens program generálására, bár néha a forrásprogramot a könnyebb értelmezhetőség miatt egy közbülső formára alakítják át (például Java-bájtkód).

A tárgyprogram végrehajtásakor csak három szint érdekes, a mikroarchitektúra szintje, az ISA-szint és az operációs rendszer gép szintje. Vagyis futtatáskor a számítógép memóriájában három program van, a felhasználó tárgyprogramja, az operációs rendszer és a mikroprogram (ha egyáltalán van). Nyoma sincs az eredeti forrásprogramnak. Ebből adódik, hogy a végrehajtáskor jelentkező szintszámok különbözhetnek a fordítást megelőző szintszámoktól. Meg kell jegyeznünk, hogy a szintek definiálásánál mi a programozók által használatos utasításokat és nyelvi szerkezeteket alkalmazzuk (nem pedig implementációs technikát), míg más szerzők a végrehajtási idejű értelmező és a fordító szintjeit hangsúlyozottabban megkülönböztetik.

7.1. Bevezetés az assembly nyelvbe

A fordítókat durván két csoportba osztják aszerint, hogy milyen a kapcsolat a forrásnyelv és a célnyelv között. Ha a forrásnyelv lényegében szimbolikus formája a numerikus gépi nyelvnek, akkor a fordítót **assembler**nek hívják, a forrásnyelvet pedig **assembly nyelv**nek. Abban az esetben, amikor a forrásnyelv a Javához vagy a C-hez hasonló magas szintű nyelv, és a célnyelv vagy numerikus gépi nyelv, vagy egy ilyennek a szimbolikus formája, akkor a transzformátort **fordítóprogram**nak nevezik.

7.1.1. Mi az assembly nyelv?

Egy egyszerű assembly nyelv olyan nyelv, amelyben mindegyik utasításnak egyetlen gépi utasítás felel meg. Más szavakkal, a gépi utasítások és az assembly program utasításai között egy-egy értelmű megfeleltetés van. Ha az assembly program minden sora pontosan egy utasításból áll, és minden gépi szó pontosan egy gépi utasítást tárol, akkor egy n soros assembly programnak egy n szavas gépi kódú program felel meg.

Mivel assemblyben programozni sokkal könnyebb, ezért a gépi kódú (hexadecimális) nyelv helyett inkább az assembly nyelvet használják. Óriási a különbség, hogy szimbolikus nevek és címek használhatók, vagy csak bináris vagy oktális alakú nevek és címek. A legtöbb ember képes megjegyezni, hogy az ADD, SUB, MUL, illetve a DIV az összeadás, a kivonás, a szorzás, illetve az osztás rövidítése, de nem tudja megjegyezni az ezeknek megfelelő, a gép által használt numerikus értékeket. Az assembly nyelven programozóknak csak a szimbolikus neveket kell tudniuk, mivel gépi kóddá alakításukat az assembler átvállalja.

Hasonlók mondhatók el a címzéssel kapcsolatban is. Az assembly programozó szimbolikus neveket adhat a memóriahelyeknek, amelyeket az assembler megfelelő numerikus értékké alakít át. A gépi kódban programozónak mindig a címek numerikus értékével kell dolgoznia. Ezért aztán manapság senki sem programoz gépi kódban, jóllehet évtizedekkel korábban, mielőtt az assembler megjelent, ezt tették.

Az assembly nyelveknek a magas szintű nyelvektől eltérő más tulajdonsága is van azonkívül, hogy az assembly utasítások egyértelműen leképezhetők gépi kódú utasításokba. Az assembly programozó a célgép minden szerkezeti eleméhez és utasításához hozzáférhet. A magas szintű nyelven programozó ezt nem teheti meg. Például, ha a célgép rendelkezik egy túlcsordulásbittel, akkor ez egy assembly programmal ellenőrizhető, míg egy Java-program ilyen vizsgálatra közvetlenül képtelen. Az assembly programmal a célgép utasításkészletének bármely utasítása végrehajtható, míg a magas szintű nyelven írt programra ez nem igaz. Röviden: bármi, ami gépi kódban megtehető, az assembly nyelvben is, de sok regiszter és más hasonló szerkezet a magas szintű nyelven programozó számára nem hozzáférhető. Vannak olyan nyelvek, ilyen a C programozási nyelv, amelyek ötvözik a két csoport tulajdonságait, vagyis magas szintű nyelvi szintaxisuk van, ugyanakkor egy assembly nyelvű gép számos hozzáférési lehetőségével is bírnak.

Még egy utolsó eltérés, amelyre érdemes rávilágítani, hogy az assembly programok a számítógépeknek csak bizonyos családjában futtathatók, míg egy magas szintű nyelven írt program potenciálisan sok gépen használható. Sok alkalmazás esetén nagy a gyakorlati jelentősége a szoftver számítógépek közötti mozgathatóságának.

7.1.2. Miért használnak assembly nyelvet?

Az assembly nyelven való programozás nehézkes. Nem könnyű hiba nélkül dolgozni vele. Továbbá egy program megírása assembly nyelven sokkal időigényesebb tevékenység, mint ugyanannak a programnak valamely magas szintű nyelven történő megírása. Sokkal hosszadalmasabb a hibamentesítés és a karbantartás.

Ilyen körülmények között miért programozna bárki assembly nyelven? Két oka is van ennek: a hatékonyság és a gép elérhetősége. Először, egy gyakorlott assembly programozó gyakran sokkal kisebb és gyorsabb programkódot készíthet, mint a magas szintű nyelven programozó. Ebbe a kategóriába tartozik sok beágyazott alkalmazás, mint egy intelligens plasztikkártya kódja, az eszközmeghajtók, a BIOS-eljárások.

Másodszor, bizonyos eljárások a hardver teljes elérését igénylik, ami rendszerint a magas szintű nyelveken lehetetlen. Például ebbe a csoportba tartoznak az operációs rendszer alacsony szintű megszakításai és csapdakezelései, sok valós idejű beágyazott rendszernél az eszközvezérlők.

Az assembly nyelven való programozás első oka (a magas hatékonyság) általában az egyik legfontosabb szempont, így közelebbről is megnézzük ezt. A legtöbb programra igaz, hogy a végrehajtási idő egy nagy százalékáért a teljes kódnak csupán egy kis része felelős. Rendszerint a program 1%-ára esik a végrehajtási idő 50%-a, és a program 10%-a felelős a végrehajtási idő 90%-áért.

Tegyük fel, hogy egy program magas szintű nyelven való megírásához 10 programozói időegység kell, és az így elkészült program végrehajtási ideje egy tipikus tesztágyban 100 másodperc. (A **tesztágy** egy olyan tesztelő program, amellyel számítógépeket, fordítóprogramokat hasonlítanak össze.) Mivel az assembly programozók termelékenysége alacsonyabb, ezért a teljes program megírásához 50 programozói időegységre van szükségük. Az így elkészült programról feltehető, hogy a tesztágyban a futási ideje kb. 33 másodperc, mivel egy okos assembly programozó egy okos fordítóprogramot 3-szorosan felülmúlhat (jóllehet ezek az arányok mások is lehetnek). A 7.1. ábra ezt a helyzetet mutatja be.

A fentebbi észrevétel szerint a kódnak egy parányi része felelős a végrehajtási idő túlnyomó részéért, ami lehetőséget ad egy további megközelítésre. Legyen a program először magas szintű nyelven írva. Ezután egy méréssorozat végezhető, amelyből meghatározható, hogy a program mely részei dolgoznak a futási idő legnagyobb részében. Az egyes eljárásokban eltöltött idő mérésére rendszerint a számítógép rendszeróráját használják, nyilvántartva, hogy az egyes ciklusok hányszor hajtódtak végre, és ehhez hasonló más jellemzőket. Példaként tegyük fel, hogy a teljes program 10%-a dolgozik a végrehajtási idő 90%-ában. Ez azt jelenti, hogy egy 100 másodperces feladat 90 másodpercet tölt ebben a kritikus 10%-ban és 10 másodpercet a program fennmaradó 90%-ban. Most ennek a 10%-nyi kritikus résznek a hatékonysága jelentősen megnövelhető assembly kódba való átírással. Ezt a folyamatot a program **hangolás**ának nevezik, amit a 7.1. ábra szemléltet. A kritikus eljárások újraírásához további 5 programozói időegységre van szükség, de ezt követően az eljárások végrehajtási ideje 90 másodpercről 30 másodpercre csökken.

	A program elkészítése programozói időegységben	A program végrehajtási ideje másodpercben
Assembly nyelv	50	33
Magas szintű nyelv	10	100
Kevert megközelítés hangolás nélkül		
Kritikus rész 10%	1	90
Egyéb rész 90%	9	10
Összesen	10	100
Kevert megközelítés hangolással		
Kritikus rész 10%	6	30
Egyéb rész 90%	9	10
Összesen	15	40

7.1. ábra. *Az assembly és a magas szintű programozási nyelven való programozás összehasonlítása hangolás nélkül és hangolással*

Tanulságos a magas szintű/assembly nyelv kevert megközelítést összehasonlítani a tisztán assembly nyelvű változattal (lásd 7.1. ábra). Az utóbbi kb. 20%-kal gyorsabb (33 másodperc a 40 másodperccel szemben), de több mint 3-szor olyan drága (50 programozói időegység a 15-tel szemben). Azonban a kevert megközelítés előnye igazából az eddig jelzettnél nagyobb, mivel egy magas szintű nyelven megírt hibátlan eljárás assembly kódba átírása sokkal könnyebb feladat, mint ugyanezt az assembly kódú eljárást a semmiből megírni. Más szavakkal, a kritikus eljárások újraírására fordítandó 5 programozói időegység túlságosan óvatos becslés. Ha az újrakódolás valójában csak 1 programozói időegységig tart, akkor a költség aránya a kevert megközelítés és a tiszta assembly nyelvű megközelítés között a 4:1 aránynál is jobb a kevert megközelítés javára.

A magas szintű nyelven programozók a bitek mozgatásába nem mélyednek el, mivel csak néha adódik számukra olyan helyzet, amelyben a hatékonyságot jelentősen javítani tudnák. Az assembly programozóknál éppen ellenkező a helyzet, mivel ők rendszerint az utasításokkal manipulálnak, hogy ezáltal esetleg néhány ciklust megtakarítsanak.

Mindent egybevetve még legalább négy olyan ok említhető, amely indokolttá teszi az assembly nyelv tanulmányozását. Először, mivel egy nagyméretű projekt sikere vagy sikertelensége függhet attól, hogy a kritikus eljárások hatékonyságában 2-szeres vagy 3-szoros javulást érünk el, ezért fontos, hogy amikor valóban szükséges, képesek legyünk jó assembly kód írására.

Másodszor, a szűkös memóriakapacitás miatt néha az assembly kód az egyetlen lehetőség. Az intelligens plasztikkártyák rendelkeznek egy CPU-val, de ritkán van néhány megabájtos memóriájuk, és még ritkábban tartalmaznak merevlemezt is a lapozáshoz. Ezen korlátozott erőforrások mellett kell bonyolult, titkosítással kapcsolatos számításokat végezniük. Az alkalmazásokba beépített processzorok gyakran a költségek miatt csak minimális memóriával rendelkeznek. A menedzserkalkulátoroknak és más vezeték nélküli tápfeszültséggel működő elektronikus berendezéseknek általában kicsi a memóriájuk a tápfeszültséggel való takarékosság miatt, így aztán kicsi és hatékony kódra van itt is szükség.

Harmadszor, egy fordítóprogramnak vagy egy assembler számára értelmezhető kimenetet kell előállítania, vagy saját magának kell az assembly eljárásokat végrehajtania. Épp ezért a fordítóprogramok működésének megértéséhez lényeges az assembly nyelv megértése. Végül is, valakinek a fordítóprogramot (az assemblyét is) meg kell írnia.

Negyedszer, az assembly nyelv vizsgálata során feltárulhat előttünk a valós számítógép. A számítógép-architektúrát tanulók számára a gép architekturális szintjének megismerésére az egyetlen mód az, hogy bizonyos assembly kódú programokat írnak.

7.1.3. Az assembly utasítások alakja

Bár az assembly utasítás szerkezete a megfelelő gépi kódú utasítás szerkezetét tükrözi, mégis az eltérő számítógépek és szintek assembly nyelve lényegében egymáshoz hasonlók, és így az assembly nyelvek vizsgálata egységesen történhet. A 7.2. ábra a Pentium 4, a Motorola 680 x 0 és az (Ultra)SPARC esetében különkülön mutatja az $N = I + J$ értékadást végrehajtó megfelelő assembly programrészeket. Mindegyiknél az üres sor feletti utasítások végzik a számolást. Az üres sor alatti utasítások az assemblert utasítják, hogy foglaljon memóriahelyet az I, J és N változóknak, vagyis ezek nem gépi kódú utasítások szimbolikus reprezentánsai.

Az Intel családban több eltérő szintaxisú assembler van. Ebben a fejezetben a Microsoft MASM assembly nyelvét használjuk a példákban. Bár vizsgálataink középpontjában a Pentium 4 áll, a rávonatkozó megállapítások érvényesek a 386-osra, a 486-osra, a Pentiumra és a Pentium Prora is. A SPARC-kal kapcsolatos példáknál a Sun assemblert használjuk. Itt is mindegyik megállapítás érvényes

Címke	Műveleti kód	Operandusok	Megjegyzés
FORMULA:	MOV	EAX,I	; regiszter EAX = I
	ADD	EAX,J	; regiszter EAX = I + J
	MOV	N,EAX	N = I + J
I	DW	3	; lefoglal 4 bájtot, beállítja 3-ra
J	DW	4	; lefoglal 4 bájtot, beállítja 4-re
N	DW	0	; lefoglal 4 bájtot, beállítja 0-ra

(a)

Címke	Műveleti kód	Operandusok	Megjegyzés
FORMULA	MOVE.L	I, D0	; regiszter D0 = I
	ADD.L	J, D0	; regiszter D0 = I + J
	MOVE.L	D0, N	N = I + J
I	DC.L	3	; lefoglal 4 bájtot, beállítja 3-ra
J	DC.L	4	; lefoglal 4 bájtot, beállítja 4-re
N	DC.L	0	; lefoglal 4 bájtot, beállítja 0-ra

(b)

Címke	Műveleti kód	Operandusok	Megjegyzés
FORMULA:	SETHI	%HI(I),%R1	! R1 = I címének magasabb helyi értékű bitjei
	LD	[%R1+%LO(I)],%R1	! R1 = I
	SETHI	%HI(J),%R2	! R2 = J címének magasabb helyi értékű bitjei
	LD	[%R2+%LO(J)],%R2	! R2 = J
	NOP		! várakozik a memóriából érkező J-re
	ADD	%R1,%R2,%R2	! R2 = R1 + R2
	SETHI	%HI(N),%R1	! R1 = N címének magasabb helyi értékű bitjei
	ST	%R2,[%R1+%LO(N)]	
I:	.WORD 3		! lefoglal 4 bájtot, beállítja 3-ra
J:	.WORD 4		! lefoglal 4 bájtot, beállítja 4-re
N:	.WORD 0		! lefoglal 4 bájtot, beállítja 0-ra

(c)

7.2. ábra. *N = I + J kiszámítása. (a) Pentium 4. (b) Motorola 680x0. (c) SPARC*

a SPARC korábbi (32 bites) változataira. Az egyöntetűség miatt a műveleti kódoknál és a regiszterek nevénél nagybetűket használunk mindenütt (ez Pentium 4 konvenció), annak ellenére, hogy a Sun assembler kisbetűt vár.

Az assembly utasítások négy részből tevődnek össze: egy címkemezőből, egy műveleti kód mezőből (opcode), egy operandus mezőből és egy megjegyzés mezőből. A memóriacímeknek adott szimbolikus nevek a címkék, amelyekre az elágaztatást végrehajtható utasításokban van szükség. Ugyanitt jelennek meg az adatnevek, amelyek lehetővé tesznek szimbolikus hivatkozást tárolt adatokra. Címkézett utasításban a címke (szokásosan) az első oszlopban helyezkedik el.

A 7.2. ábra három részének mindegyikében négy címke van: *FORMULA*, *I*, *J* és *N*. Vegyük észre, hogy a SPARC assemblynél a címkéket kettőspont követi, míg a Motorola esetében ilyen nincs. Az Intel az utasításcímkéknél megköveteli a kettőspontot, de az adatcímkék után nem. Ezek az eltérések a lényeget nem érintik. Az assemblerek tervezőinek ízlése gyakran meglehetősen eltérő. Az alapul szolgáló

architektúrában semmi sem indokolja jobban az egyik vagy a másik választást. A kettőspontos jelölés előnye, hogy ilyenkor egy sorban önállóan is megjelenhet egy címke, a következő sor első oszlopában, pedig a műveleti kód. A fordítóprogramok számára ez a megoldás néha kényelmes. A kettőspontos jelölés nélkül nincs mód a címke és a műveleti kód megkülönböztetésére, ha önmagukban vannak egy sorban. Ilyen félreérthetőség nem fordul elő a kettőspont alkalmazása esetén.

Néhány assemblynek megvan az a nem éppen szerencsés tulajdonsága, hogy a címkék hossza hat vagy nyolc karakterre korlátozott. Ezzel szemben a legtöbb magas szintű nyelv tetszőleges hosszúságú neveket engedélyez. A hosszú, jól megválasztott nevek a programokat sokkal olvashatóbbá, érthetőbbé teszik.

Minden számítógép rendelkezik néhány regiszterrel, de ezek nevei nagyban eltérhetnek egymástól. A Pentium 4 regisztereinek nevei EAX, EBX, ECX és így tovább. A Motorola regiszterei többek között a D0, D1, D2 neveket kapták. A SPARC-regisztereknek több nevük is van. Itt a %R1 és %R2 neveket használjuk.

A műveleti kód mező a műveleti kód szimbolikus rövidítését tartalmazza – ha az utasítás egy gépi kódú utasítás megfelelője – vagy egy assemblernek szóló parancsot. Egy alkalmas névválasztás csupán ízlés kérdése, és az assembly nyelv tervezőinek ízlése gyakran eltér egymástól.

Az Intel assembler tervezői úgy döntöttek, hogy a MOV kódot használják mind a regiszternek memóriából történő értékadására, mind egy regiszter tartalmának a memóriába való tárolására. A Motorola assembler tervezői mindkét műveletre a MOVE kódot választották. A SPARC assembler tervezői az előbbi művelethez az LD, míg az utóbbinál az ST mellett döntöttek. Természetesen ezeknek a választásoknak semmi közük a gépi architektúrához.

Ezzel ellentétben, a SPARC architektúrájának tulajdonságából következik, hogy a memória eléréséhez két gépi kódú parancsra van szükség, ebből az első a SETHI, mivel 32 bites (SPARC 8. verzió) vagy 44 bites (SPARC 9. verzió) virtuális címeket használ, míg az utasítások legfeljebb 22 bites közvetlen adatot tárolhatnak csak. Így mindig két utasításra van szükség a virtuális címek kezeléséhez.

```
SETHI %HI(I),%R1
```

utasítás az (64 bites) R1 regiszter felső 32 bitjét és alsó 10 bitjét nullára állítja, ezután az *I* 32 bites címének felső 22 bitjét az R1 10-től 31-ig terjedő bitpozícióiba teszi el. A következő utasítás a

```
LD [%R1+%LO(I)],%R1
```

I teljes címének megállapításához R1-t összeadja *I* címének alsó 10 bitjével, kiveszi az itt található szót a memóriából, és eltárolja R1-ben. Egy utasítások közötti szépségversenyben 1–10-ig terjedő pontozási skála esetén ezek az utasítások körülbelül a –20-at érnék el, de hát a SPARC assembly nyelv tervezőinek fő szempontja nem a szépség volt. Az elsődleges cél a gyors utasításvégrehajtás volt, és ezt a feladatot nagyon jól teljesíti.

A Pentium család, a 680x0-as és a SPARC mindegyike megengedi a bájt-, illetve a szóhosszúságú és a hosszú operandusok használatát. Honnan tudja az assembler, hogy melyik hosszat használja? Az assembler tervezői ismét eltérő megoldásokat választottak. A Pentium 4-nél a különböző hosszúságú regiszterek neve különböző, így az EAX 32 bites, az AX 16 bites, az AL és AH 8 bites adatok mozgatására szolgál. Nem így tettek a Motorola tervezői, akik inkább a *.L*, *.W*, *.B* utótagokkal egészítették ki a neveket a hosszú, a szó- és a bájtméretnek megfelelően, ahelyett hogy a D0 stb. nevekben jelezték volna ezt. A SPARC különböző műveleti kódokat használ az eltérő hosszokhoz (például LDSB, LDSH vagy LDSW jelzi a bájtnyi, félszavas, illetve szavas betöltést egy 64 bites regiszterbe). Mindhárom megoldás indokolt, de ez ismét a nyelv tervezésének önkényes voltára mutat rá.

A három assembler abban is eltér, ahogy helyet foglalnak az adatoknak. Az Intel assembly nyelv tervezői a DW-t (Define Word) választották, bár később mintegy alternatívaként hozzávették a .WORD jelölést. A Motorolásoknak a DC (Define Constant) tetszett. A SPARC emberei kezdettől fogva a .WORD jelölést részesítették előnyben. Az eltérő választások ismét önkényesek.

Egy assembly utasítás operandus mezője arra szolgál, hogy meghatározza a címek és regiszterek operandusokként való használatát a gépi utasításban. Egy egész összeadó utasítás operandus mezője megmondja, hogy mit mihez kell hozzáadni. Egy elágazó utasításban megadja, hogy hova kell ugrani. Az operandusok lehetnek regiszterek, konstansok, memóriahelyek és így tovább.

A megjegyzés mező helyet biztosít a programozóknak, hogy hasznos magyarázatokat illesszenek be a program működéséről más programozók segítésére, akik később használják vagy módosítják a programot (vagy a program készítője saját maga számára, ami hasznos lehet, ha például egy év múlva módosítani akarja a programot). Egy dokumentáció nélküli assembly program nagyon nehezen érthető a programozók számára, gyakran még a készítő számára is. A megjegyzés mező kizárólag az érthetőséget segíti, nincs semmi hatása az assembly eljárásokra vagy az előállított kódra.

7.1.4. Pszeudoutasítások

Egy assembly program a végrehajtandó gépi utasításokon kívül tartalmazhat az assemblernek szóló parancsokat is, például kérhet memóriafoglalást vagy a listán egy lapdobást. Az assemblernek szóló utasításokat **pszeudoutasítások**nak vagy néha **assembler direktívák**nak nevezzük. Már láttunk egy tipikus pszeudoutasítást a 7.2. ábrán, a DW-t. A 7.3. ábra is tartalmaz néhány pszeudoutasítást. Ezek az Intel családnak készült Microsoft MASM assembleréből valók.

A SEGMENT pszeudoutasítás egy új szegmens kezdetét jelzi, az ENDS pedig egy ilyennek a végét. Megengedett, hogy elkezdjünk egy szövegszegmenst kódolni, majd indíthatunk egy adatszegmenst, utána visszatérhetünk a szövegszegmenshez és így tovább.

Az ALIGN – rendszerint adat elhelyezésekor – olyan címet kényszerít ki a következő sornak, amely argumentumának többszöröse. Például, ha az aktuális szeg-

Pszeudoutasítás	Hatás
SEGMENT	Egy új szegmenst kezd (szöveg, adat stb.) bizonyos attribútumokkal
ENDS	Az aktuális szegmens bezárása
ALIGN	A következő utasítás vagy adat igazítása
EQU	Új szimbólum definiálása egy adott kifejezéssel
DB	Egy vagy több bájt lefoglalása (inicializálva)
DD	Egy vagy több 16 bites fél szó számára helyfoglalás (inicializálva)
DW	Egy vagy több 32 bites szó számára helyfoglalás (inicializálva)
DQ	Egy vagy több 64 bites dupla szó számára helyfoglalás (inicializálva)
PROC	Egy eljárás kezdete
ENDP	Egy eljárás vége
MACRO	Egy makródefiníció kezdete
ENDM	Egy makródefiníció vége
PUBLIC	Szimbólumok láthatóvá tétele más modulok számára
EXTERN	Más modulban definiált szimbólum felhasználása
INCLUDE	Más fájl beillesztése
IF	Egy adott kifejezés teljesülése esetén fordítandó rész kezdete
ELSE	Az IF feltétel nem teljesülése esetén fordítandó rész kezdete
ENDIF	A feltételesen fordítandó rész vége
COMMENT	Egy új megjegyzés kezdő karakterének definiálása
PAGE	Listázásban lapdobás kikényszerítése
END	Az assembly program vége

7.3. ábra. *A Pentium 4 assembler (MASM) néhány pszeudoutasítása*

mens már tartalmaz 61 bájtnyi adatot, akkor az ALIGN 4 után a következő elérhető cím a 64.

Az EQU-val egy kifejezésnek adhatunk szimbolikus nevet. Például a

```
BASE   EQU   1000
```

pszeudoutasítás után a BASE szimbólumot használhatjuk mindenhol a programunkban az 1000 helyett. Az EQU utáni kifejezés több definiált szimbólumot is tartalmazhat, aritmetikai vagy más műveleti jelekkel összekapcsolva, mint például az alábbi kifejezésben:

```
LIMIT   EQU   4 * BASE + 2000
```

A legtöbb assemblernél, ideértve a MASM-ot is, egy szimbólumnak már definiáltnak kell lennie, mielőtt egy kifejezés felhasználná; lásd az előzőket.

A következő négy pszeudoutasítás, a DB, DD, DW és a DQ rendre 1, 2, 4, illetve 8 bájtnyi helyet foglalnak le egy vagy több változó számára. Például a

```
TABLE   DB 11, 23, 49
```


helyet foglal 3 bájtnak, és ezeknek rendre a 11, 23, illetve a 49 kezdőértéket adja. Ezen kívül definiálja a *TABLE* szimbólumot, és értékét beállítja a 11-et tartalmazó bájt címére.

A PROC és ENDP pszeudoutasítások egy assembly eljárás kezdetét és végét definiálják. Az assembly eljárások ugyanolyan szerepet töltenek be, mint más programozási nyelveknél az eljárások. Hasonlóan, a MACRO és ENDM egy makródefiníció hatáskörét jelöli ki. A makrókkal ebben a fejezetben később még foglalkozunk.

A következő két pszeudoutasítás, a PUBLIC és az EXTERN a szimbólumok láthatóságát szabályozzák. Rendszerint programjaink több fájlból állnak. Gyakran előfordul, hogy az egyik fájlban levő eljárás más fájlban levő eljárást szeretne hívni, vagy más fájlban levő adatra lenne szüksége. Az ilyen hivatkozás megoldására PUBLIC-ként kell definiálni azt a szimbólumot, amelyet elérhetővé akarunk tenni más fájlok számára. Hasonlóan, ha meg akarjuk előzni az assembler definiálatlan szimbólumra vonatkozó hibaüzeneteit, akkor a szimbólumot EXTERN-né kell nyilvánítani, amely az assemblernek jelzi, hogy a szimbólum egy másik fájlban kerül definiálásra. Azoknak a szimbólumoknak a hatásköre, amelyek nem e két pszeudoutasítás valamelyikével definiáltak, csak az adott fájlra terjed ki. Ez az alapértelmezés azt jelenti, hogy például a *FOO* szimbólum több fájlban is használható anélkül, hogy hiba lépne fel, mert mindegyik definíció lokális a saját fájljára nézve.

Az INCLUDE pszeudoutasítás az assemblert egy másik fájl elérésére utasítja, amelyet teljesen bemásol az aktuális helyre. Az ilyen beágyazott fájlok gyakran tartalmaznak több fájlban használatos definíciókat, makrókat és más szimbólumokat.

Több assembler, beleértve a MASM-ot is, támogatja a feltételes végrehajtást. Például a következő utasítássorozat helyet foglal egy 32 bites szónak és címét *WSIZE*-nak nevezi el.

```
WORDSIZE  EQU  16
IF WORDSIZE GT 16
WSIZE:  DW  32
ELSE
WSIZE:  DW  16
ENDIF
```

A *WSIZE* szó kezdőértéke 32 vagy 16 a *WORDSIZE* konstans értékétől függően, amely esetünkben 16. Ezen konstrukció segítségével írható olyan program, amely 16 bites gépeken (mint a 8088) vagy 32 bites gépeken (mint a Pentium 4) is lefordítható. A gépfüggő részek IF ... ENDIF szerkezetbe való beágyazással, majd egyetlen, a *WORDSIZE*-ra vonatkozó definíció megváltoztatásával a program automatikusan lefordítható akármilyen gépre. Ezt a megközelítést választva egy forrásprogram használhatóvá tehető különböző célgépek számára, ami megkönnyíti a szoftver fejlesztését és karbantartását. Sok esetben a gépfüggő definíciókat, mint a *WORDSIZE* definiálása, egyetlen fájlba gyűjtik, ezek különböző változatait aztán a különböző gépekhez rendelik. A megfelelő definíciós fájl beszerkesztésével a program könnyen lefordítható a különböző gépekre.

A COMMENT pszeudoutasítás lehetővé teszi a felhasználónak, hogy a megjegyzés határait a pontosvessző helyett más szimbólummal jelezze. A PAGE-dzsel az assemblertől kért listázást vezérelhetjük. Végül az END a program végét jelöli.

A MASM-ban vannak még más pszeudoutasítások is. Más Pentium 4 assemblerek más pszeudoutasítás-gyűjteménnyel rendelkeznek, nemcsak az architektúra eltérő volta miatt, hanem az assembler készítőjének ízlése miatt is.

7.2. Makrók

Az assembly nyelven programozók gyakran utasítássorozatok többszöri ismétlésére kényszerülnek egy programon belül. A legnyilvánvalóbb megoldása ennek, ha egyszerűen bemásoljuk a megfelelő utasításokat minden olyan helyre, ahol szükséges. Ha az utasítássorozatunk hosszú vagy sokszor kell alkalmazni, akkor fárasztóvá válik az ismételt beírás.

Egy alternatív megközelítése ennek a problémának az, hogy az utasítássorozatból eljárást készítünk, és az eljárást ott hívjuk meg, ahol szükségünk van rá. Ennek a stratégiának az a hátránya, hogy mindig kell egy hívó és egy visszatérő utasítás, amikor az eljárást végre akarjuk hajtani. Ha az utasítássorozat rövid, például két utasításból áll, viszont sokszor alkalmazzuk, akkor az eljáráshívások többletköltsége jelentősen lelassíthatja a programot. A makrók egyszerű és hatékony megoldást jelentenek olyan esetekben, amikor ugyanazokat vagy közel ugyanazokat az utasításokat ismételten akarjuk alkalmazni.

7.2.1. A makrók definíciója, hívása, kifejtése

A **makródefiníció** mód arra, hogy egy kódrészletnek nevet adjunk. Miután a makró definiálása megtörtént, a programozó a programrészlet helyett a makró nevét írhatja. A makró valójában egy kódrészlet rövidítéseként fogható fel. A 7.4. (a) ábra egy Pentium 4-re írt assembly nyelvű programot mutat, amely két változó, a *p* és *q* tartalmát kétszer felcseréli. Ezek az utasítások makróban is definiálhatók, mint ahogy a 7.4. (b) ábra mutatja. A makró definíciója után minden *SWAP* előfordulása a következő négy sorral helyettesítődik:

```
MOV  EAX,P
MOV  EBX,Q
MOV  Q,EAX
MOV  P,EBX
```

A programozó a *SWAP* makrót a fentebbi négy utasítás rövidítéseként definiálta.

Bár a különböző assemblerek kissé eltérően jelölik a makródefiníciókat, azonban minden makródefiníció ugyanazon alapvető részekből épül fel, ezek a következők:

1. A makrófej, amely megadja a makró nevét.
2. A makrótörzs, amely a kódot tartalmazza.
3. A makró végét jelző pszeudoutasítás (például ENDM).

Amikor az assembler elér egy makródefiníciót, azt elmenti egy makródefiníciós táblába a későbbi használat számára. Ettől a ponttól kezdve valahányszor a makró neve megjelenik (*SWAP* a 7.4. ábrán látható példában), az assembler mindannyiszor a makrótörzzsel helyettesíti. **Makróhívás**ként ismeretes a makrónév műveleti kódként történő alkalmazása, és **makrókifejtés**nek nevezik a makrótörzzsel való helyettesítés folyamatát.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV   EAX,P
      MOV   EBX,Q
      MOV   Q,EAX
      MOV   P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

7.4. ábra. *Assembly kód a P és a Q változók értékeinek kétszeri felcserélésére. (a) Makró nélkül. (b) Makróval*

A makrókifejtés nem a program végrehajtása alatt, hanem a fordítás során megy végbe. Ez lényeges szempont. Mind a 7.4. (a), mind a 7.4. (b) pontosan ugyanazt a gépi kódot eredményezi. Csupán a gépi kódot vizsgálva lehetetlen megmondani, hogy a program írásakor használtunk makrókat vagy sem. Ennek az az oka, hogy az összes makrókifejtés után az assembler eldobja a makródefiníciókat. A kész programban ezeknek semmi nyoma nem marad.

A makróhívásokat és az eljáráshívásokat nem szabad összetéveszteni. Az alapvető különbség közöttük, hogy a makróhívás egy assemblernek szóló parancs, amellyel a makrónévnek a makrótörzzsel való helyettesítését kérjük. Az eljáráshívás egy gépi utasítás, amely beépül a tárgykódba és később végrehajtódik, amikor is az eljárás meghívásra kerül. A 7.5. ábra a makróhívásokat és az eljáráshívásokat hasonlítja össze.

Lényegében az assembly program fordítását legjobb egy kétmenetes folyamatnak tekinteni. Az első menetben minden makródefiníció tárolásra kerül, a makróhívások pedig kifejtésre kerülnek. A második menetben a kapott kód feldolgozása úgy történik, mintha ez lett volna az eredeti program. Így nézve a folyamatot, először a forráskódú program kerül beolvasásra, majd ennek egy olyan programmá való átalakítása történik, amelyben az összes makróhívás helyettesítve van a megfelelő makrótörzzsel és a makródefiníciók ezt követően törlődnek. Az így előálló assembly programban már nincsenek makrók, és az assembler fel tudja dolgozni.

Szempontok	Makróhívás	Eljáráshívás
Mikor történik a hívás?	A fordítás közben	A program végrehajtása közben
A törzs minden híváskor bemásolódik a tárgyprogramba?	Igen	Nem
A hívó utasítás elhelyezésre kerül a tárgyprogramban és később hajtódik végre?	Nem	Igen
Szükséges visszatérő utasítás a hívás befejezése után?	Nem	Igen
A törzsnek hány másolata jelenik meg a tárgyprogramban?	Makróhívásonként egy	1

7.5. ábra. *A makróhívások és az eljáráshívások összehasonlítása*

Jusson eszünkbe, hogy egy program valójában betűkből, számjegyekből, szóközökből, írásjelekből és a „kocsi vissza" (új sorra váltás) karakterekből felépülő karakterlánc. A makrókifejtés egy karakterlánc bizonyos részláncainak más karakterlánccal való helyettesítése. A makrók használata egy olyan karakterlánc-manipulációs technika, amely nem törődik a karakterlánc jelentésével.

7.2.2. Paraméteres makrók

Az előzőkben leírt makrózási lehetőség olyan programok rövidítésére használható, amelyben az ismételten előforduló utasítássorozat pontosan ugyanaz. Gyakran egy program több olyan utasítássorozatot tartalmaz, amelyek nem pontosan azonosak, csak majdnem. Ilyen látható a 7.6. (a) ábrán. Ennél a példánál az első sorozat a *P*-t felcseréli a *Q*-val, a második sorozat az *R*-t cseréli fel az *S*-sel.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,R
MOV   EBX,S
MOV   S,EAX
MOV   R,EBX
```

(a)

```
CHANGE   MACRO P1, P2
         MOV EAX,P1
         MOV EBX,P2
         MOV P2,EAX
         MOV P1,EBX
         ENDM

         CHANGE P, Q

         CHANGE R, S
```

(b)

7.6. ábra. *Két közel azonos utasítássorozat. (a) Makró nélkül. (b) Makróval*

A makróassemblerek kezelni tudják a közel azonos utasítássorozatokat úgy, hogy **formális paraméterek** alkalmazását engedik a makródefiníciókban és **aktuális paramétereket** a makróhívásokban. Makrókifejtéskor a makrótörzsben levő összes formális paraméter a megfelelő aktuális paraméterrel helyettesítődik. Az aktuális paramétereket a makróhívás operandus mezőjében kell elhelyezni.

A 7.6. (b) ábra kétparaméteres makrót használva mutatja a 7.6. (a) részben levő program átírt változatát. A *P1* és *P2* szimbólumok a formális paraméterek. A makrókifejtéskor a makrótörzsben minden *P1* előfordulás az első aktuális paraméterrel helyettesítődik. Hasonlóan, minden *P2* előfordulás a második aktuális paraméterrel helyettesítődik.

```
CHANGE P, Q
```

makróhívásban a *P* az első, a *Q* a második aktuális paraméter. Így a 7.6. ábra programjaiból generált futtatható programok azonosak. A programok utasításai és az operandusai teljesen megegyeznek.

7.2.3. Előnyös tulajdonságok

A legtöbb makróassembler seregnyi előnyös tulajdonsággal könnyíti meg az assembly nyelven programozók helyzetét. Ebben a fejezetben szemügyre vesszük a MASM néhány ilyen jellegű sajátosságát. Minden makrózást támogató assemblernél megjelenik a címke többszöröződési probléma. Tegyük fel, hogy egy makróban van egy feltételes vezérlésátadó utasítás és egy címke, amelyre az ugrás történik. Ha a programban erre a makróra két vagy több hívás is van, akkor a makrótörzsben levő címke többszöröződik, ami fordítási hibát okoz. A probléma egyik megoldási módja, ha a programozó minden makróhívást különböző címkével, mint aktuális paraméterrel ad ki. Egy másik megoldás (amelyet a MASM is alkalmaz), hogy lokális címke definiálását engedi meg, és így a makró minden egyes kifejtésekor az assembler automatikusan különböző címeket generál. Vannak olyan assemblerek, amelyeknél az a szabály, hogy a numerikus címkék automatikusan lokálisak.

A legtöbb assembler, így a MASM is megengedi a makródefiniálást a makrókon belül. Ez a sajátosság a feltételes assembly utasítással kombinálva nagyon hasznos konstrukció. Gyakori, hogy ugyanannak a makrónak a definiálása van az IF utasítás mindkét ágában, ezt szemlélteti a következő példa:

```
M1   MACRO
     IF WORDSIZE GT 16
M2     MACRO
       ...
       ENDM
     ELSE
M2     MACRO
       ...
       ENDM
     ENDIF
       ENDM
```

Mindkét esetben az *M2* makró definiálására kerül sor, de a makródefiníció attól függ, hogy a programot 16 bites vagy 32 bites számítógépen hajtjuk végre. Ha az *M1* makróra nincs hívás, akkor az *M2* makródefiníció teljesen elmarad.

Utolsóként említjük, hogy a makrók hívhatnak más makrókat, beleértve önmagukat is. Ha egy makró rekurzív, vagyis önmagát hívja, akkor kell lennie olyan paraméterének, amelynek az értéke változik minden makrókifejtéskor, és ha a paraméter egy bizonyos értéket elér, akkor a rekurzió befejeződik. Máskülönben az assembler végtelen ciklusba kerülhet. Ha ez bekövetkezik, akkor a felhasználói beavatkozással kell az assemblert megállítani.

7.2.4. A makróassembler működése

Egy makrózási lehetőséget biztosító assemblernek a következő két feladatot is el kell látnia: a makródefiníciók tárolását és a makróhívások kifejtését. Ezeknek a funkcióknak a vizsgálatára térünk most rá.

Az assemblernek kezelni kell egy olyan táblázatot, amely az összes makrónevet tartalmazza, és mindegyik névvel együtt egy mutatót a tárolt makródefinícióra, hogy visszakereshesse, amikor szükséges. Vannak assemblerek, amelyek a makróneveket egy külön táblázatban tárolják, és vannak olyanok, amelyek egy közös táblázatot kezelnek az összes műveleti kód számára, amelybe a gépi utasítások, a pszeudoutasítások és a makrónevek is beletartoznak.

Amikor az assembler egy makródefinícióhoz ér, akkor a táblázatba felveszi a makró nevét, a formális paraméterek számát és egy mutatót, amely egy másik táblázatra – a makródefiníciós táblázatra – fog mutatni, ahova a makrótörzset tárolni fogja. Ekkor a formális paraméterek alapján készül egy lista is, amelyre a definíció feldolgozásakor van szükség. Ezután a makrótörzset olvassa, és a makródefiníciós táblázatban tárolja. A makrótörzsben előforduló formális paramétereket speciális szimbólummal jelöli meg. Ezt láthatjuk a *CHANGE* makródefiníció alábbi tárolt formájában, ahol a pontosvessző a „kocsi visszát" jelöli, és az et (&) karakter a formális paramétereket jelzi:

```
MOV EAX, &P1; MOV EBX, &P2; MOV &P2,EAX; MOV &P1,EBX;
```

A makródefiníciós táblázat a makrótörzset egyszerű karakterláncként tárolja.

Az assembler az első menetben a műveleti kódokat figyeli és a makrókat kifejti. A talált makródefiníciók mindegyikét tárolja a táblázatban. Makróhíváskor az assembler ideiglenesen abbahagyja a beviteli eszközről az input olvasását, és ehelyett a tárolt makrótörzset kezdi olvasni. A makrótörzsből a formális paramétereket elhagyja és helyükbe a hívásban szereplő aktuális paramétereket helyettesíti be. Az assembler a formális paramétereket az azokat jelző & karakterből könnyedén felismeri.

7.3. Az assembler menetei

A következő fejezetekben az assembler működését írjuk le röviden. Bár különböző assembly nyelvvel rendelkeznek a számítógépek, mégis az assembler menetei az eltérő gépeken hasonlók, és így ezek együttesen vizsgálhatók.

7.3.1. Kétmenetes assemblerek

Mivel az assembly nyelvű programok egysoros utasításokból állnak, ezért természetesnek tűnik az elgondolás, hogy az assembler, miután egy utasítást beolvasott, azt gépi nyelvre fordítsa, és a generált kódot egy fájlba helyezze el, míg a megfelelő listarészeket, ha vannak ilyenek, egy másik fájlba. Ezt a folyamatot aztán addig ismételné, amíg a teljes program fordítása elkészül. Sajnos, ez a stratégia nem használható.

Tekintsünk ugyanis egy olyan helyzetet, amikor az első utasítás egy ugrás az *L*-re. Ekkor az assembler ezt az utasítást addig nem tudja lefordítani, amíg az *L* utasítás címét nem ismeri. Előfordulhat, hogy az *L* utasítás a program végéhez van közel, és az assembler képtelen meghatározni a címet anélkül, hogy majdnem a teljes programot előzőleg végigolvasná. Ezt a helyzetet **előre hivatkozási problémá**nak nevezzük, mivel egy szimbólum, esetünkben ez az *L*, azt megelőzően kerül alkalmazásra, mielőtt definiálva lenne, vagyis egy olyan szimbólumra történik hivatkozás, amelynek a definíciója később történik meg.

Az előre hivatkozási probléma két módon kezelhető. Az első megközelítés szerint az assembler a forrásprogramot kétszer olvassa. A forrásprogram olvasási folyamatát **menet**nek hívjuk; azokat a fordítókat, amelyek a bemeneti programot kétszer olvassák, **kétmenetes fordítók**nak nevezzük. A kétmenetes fordító az első menetben a szimbólumok definícióit, beleértve a címkeszimbólumokat is, összegyűjti és egy táblázatban tárolja. A második menet kezdetekor már minden szimbólum értéke ismert, így előre hivatkozási probléma nem fordulhat elő, és az assembler mindegyik utasítást képes olvasni, fordítani és tárolni. Annak ellenére, hogy ez a megközelítés egy extra olvasási menetet igényel, az alapgondolat egyszerű.

A második megközelítés az assembly programot csak egyszer olvassa, egy közbülső formába transzformálja, és ezt a formát a memória egy táblázatába helyezi. Majd egy második menet következik, de ez nem a forráskódon, hanem a táblázaton dolgozik. Ha elegendő memória áll rendelkezésre (vagy virtuális memória), akkor ezzel a megközelítéssel B/K időmegtakarítás érhető el. Listázási igény esetén a teljes forráskódú utasítást a megjegyzéssel együtt tárolni kell. Ha nincs igény listázásra, akkor a közbülső forma egy leegyszerűsített vázzá redukálható.

Mindkét módszernél az első menet feladata az összes makródefiníció tárolása, és a makróhívások kifejtése elérésük pillanatában. Így a szimbólumok definiálása és a makrókifejtések általában keveredve jelentkeznek az első menetben.

7.3.2. Első menet

Az első menet fő feladata az ún. szimbólumtábla felépítése, ami tartalmazza minden szimbólum értékét. Az alábbihoz hasonló pszeudoutasítás segítségével értékül adhatunk egy szimbolikus névnek akár egy címkét, akár egy értéket (konstans):

```
BUFSIZE EQU 8192
```

Amikor az utasítás címkemezőjében levő szimbólumnak konstanst adunk értékül, akkor az assemblernek tudnia kell, hogy a program végrehajtásakor mi lesz az utasítás címe. **Utasítás-helyszámlálóként (ILC, Instruction Location Counter)** ismert változót kezel az assembler a fordítás alatt, amelyben nyomon követi annak az utasításnak a futtatáskori címét, amelyet éppen fordít. Az első menet kezdetekor ennek a változónak az értéke 0-ra van beállítva, és minden egyes utasítás feldolgozásakor az utasítás hosszával növekszik az értéke, ez látható a 7.7. ábrán. Ez egy Pentium 4-es példa. Nem fogunk SPARC- (vagy Motorola-) példákat adni, mivel nincs lényeges különbség az assembly nyelvek között, és így egy példa is elegendő. Azon kívül a SPARC lenne az igazi esélyes a legkevésbé olvasható assembly nyelvek világversenyén.

Címke	Műveleti kód	Operandusok	Megjegyzés	Hossz	ILC
MARIA:	MOV	EAX,I	EAX = I	5	100
	MOV	EBX,J	EBX = J	6	105
ROBERTA:	MOV	ECX,K	ECX = K	6	111
	IMUL	EAX,EAX	EAX = I * I	2	117
	IMUL	EBX,EBX	EBX = J * J	3	119
	IMUL	ECX,ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX,EBX	EAX = I * I + J * J	2	125
	ADD	EAX,ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	Ugrás a DONE címkére	5	129

7.7. ábra. *Az utasítás-helyszámláló (ILC) jelzi az utasítások memóriába töltési címeit. Ebben a példában a MARIA-t megelőző utasítások 100 bájtot foglalnak le*

A legtöbb assembler első menete legalább három táblázatot használ: a szimbólumtáblát, a pszeudoutasítások táblázatát és a műveleti kódok tábláját. Ha szükséges, akkor még egy konstans táblázatot is. Mint ahogy a 7.8. ábra is mutatja, a szimbólumtáblában minden szimbólumhoz egy bejegyzés tartozik. Szimbólumokat vagy úgy definiálunk, hogy címkeként használjuk, vagy közvetlenül definiáljuk (például az EQU pszeudoutasítással). A szimbólumtábla mindegyik bejegyzése tartalmazza magát a szimbólumot (vagy egy mutatót rá), a numerikus értékét, és más egyéb információkat. Ez az egyéb információ a következő lehet:

1. A szimbólumhoz tartozó adatmező hossza.
2. Az áthelyezési (relocation) bitek. (Jelzik, hogy ha a program más címre töltődik be, mint amit az assembler feltételez, akkor a szimbólum értéke változik-e?)
3. Vajon a szimbólum külső eljárásból elérhető-e, vagy nem?

Szimbólum	Érték	Egyéb információk
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

7.8. ábra. *A 7.7. ábrában látható program szimbólumtáblája*

A műveleti kód táblája az assembly nyelv minden szimbolikus műveleti kódja (mnemonic) számára legalább egy bejegyzést nyit. A 7.9. ábrán látható a műveletikód-tábla egy részlete. Ezek mindegyike tartalmaz egy-egy mezőt a szimbolikus kód, a két operandus, a műveleti kód numerikus értéke, az utasítás hossza és egy típusszám számára, mely utóbbi csoportba sorolja a műveleti kódokat aszerint, hogy hány darab van és milyen az operandusuk.

Műveleti kód	Első operandus	Második operandus	Műveleti kód hexadecimálisan	Az utasítás hossza	Az utasítás típusa
AAA	–	–	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

7.9. ábra. *Egy rövid kivonat a Pentium 4 assembler műveleti kód táblájából*

Példaként nézzük az ADD utasítást. Ha egy ADD utasításban az első operandus az EAX, és a második operandus egy 32 bites közvetlen konstans (immed32), akkor a műveleti kód 0x05, és az utasítás hossza 5 bájt. (8 vagy 16 bites konstans esetén más a műveleti kód, de ezt nem mutatjuk.) Ha az ADD utasításban két regisztert használunk operandusként, akkor az utasítás 2 bájt hosszúságú, és a műveleti kódja 0x01. A 19-es utasítástípus tartozik az összes olyan műveleti kód-operandus kombinációhoz, amelyek a két regiszteroperandusú ADD utasításnál alkalmazott szabályt követik, és a feldolgozásuk ugyanúgy történik. Az utasítástípus lényegében azt mondja meg, hogy a fordítás során melyik eljárást kell meghívni egy adott típusú, tetszőleges utasítás feldolgozásához.

Bizonyos assemblerek megengedik a programozóknak, hogy közvetlen címzést használjanak az utasításokban, annak ellenére, hogy a célnyelvben megfelelő utasítás nincs. Az ilyen „pszeudoközvetlen" utasítások kezelése a következőképpen történik. Az assembler a közvetlen operandus számára a program végén memóriahelyet foglal, és egy erre hivatkozó utasítást generál. Például az IBM 3090 nagygépnek nincsenek közvetlen utasításai. Azonban a programozó írhatja a 14-es regiszter egyszavas 5 konstanssal való feltöltéséhez a következőt:

```
L 14,=F'5'
```

Ezzel a programozó megspórol egy pszeudoutasítást, amely egy szót lefoglalna az 5 értéknek, és adna ennek egy címkét, amelyet aztán az L utasításban használna fel. Azokat a konstansokat, amelyeknek az assembler automatikusan memóriahelyet foglal, **literálok**nak nevezik. Azon túl, hogy így a programozónak kevesebbet kell írnia, a literálok alkalmazásával a program olvashatóbbá válik, mert a forráskódban a konstans értéke látható. Az assembler első meneténck kell a programban előforduló összes literálból egy táblázatot készíteni. Az általunk példaként vett mindhárom számítógép rendelkezik közvetlen utasításokkal, így ezeknél literál nem jelenik meg. Manapság a közvetlen utasítások teljesen megszokottak, de korábban ezeket nem használták. Valószínűleg, a literálok elterjedt alkalmazása ösztönözte a számítógép tervezőit arra, hogy közvetlen utasításokat valósítsanak meg. Literálok alkalmazása esetén a fordítás alatt egy literáltáblázat is felépül úgy, hogy valahányszor a fordító egy literálhoz ér, felveszi a táblázatba. Az első menet után a táblázatban előforduló többszöröződéseket megszüntetik.

A 7.10. ábra egy assembler első meneténck alapvető feladatait tartalmazó eljárást mutat. Figyelemre méltó ebben a programozási stílus. Az eljárások neveit úgy választottuk meg, hogy azok egyben jelzik az eljárás feladatát. A legfontosabb azonban az, hogy az első menet egy olyan vázát tartalmazza az ábra, amely jó kiindulási pontként szolgálhat. Elég rövid, hogy érthető legyen, és megvilágítsa a következő lépést, nevezetesen a benne előforduló eljárások elkészítését.

Az eljárások között lesznek viszonylag rövidek, ilyen a *check_for_symbol*, amely csupán a szimbólum karakterláncát adja vissza, ha a szimbólum létezik, és 0-t ad vissza egyébként. Más eljárások, mint a *get_length_of-type1* és a *get_length_of_type2*, hosszabbak lehetnek, és más eljárásokat is hívhatnak. Általában a típusok száma természetesen nem kettő, hanem attól függ, hogy a fordítandó nyelvben hányféle típusú utasítás van.

A programok ilyen strukturáltsága a programozói munka könnyítésén túl más előnyökkel is jár. Abban az esetben, ha az assemblert egy csoport készíti el, akkor a különféle eljárásokat a programozók egymás között szétoszthatják. A *read_next_line* eljárással a bejövő adatsor lényegtelen részei elrejthetők. Ha ezek változnának, például az operációs rendszer változása miatt, akkor is csak egy segédeljárást befolyásolna, és nem tenné szükségessé magának a *pass-one* eljárásnak a módosítását.

Az első menet a programot olvassa, és soronként elemzi, megkeresi a műveleti kódot (például ADD), meghatározza a típusát (alapvetően az operandusok fajtája alapján), és kiszámítja az utasítás hosszát. Ezekre az információkra a második menetnek is szüksége lesz, így egy lehetséges megoldás ezek közvetlen kitárolása, hogy ezáltal elkerülhető legyen, hogy a következő alkalommal egy vázszerkezetből kelljen a sort elemezni. Azonban a bemeneti fájl újraírása megnöveli a B/K tevékenységet. A CPU és a lemez viszonylagos gyorsaságától, a fájlrendszer hatékonyságától és más egyéb tényezőktől függ, hogy mi a jobb, több B/K-t végezni, hogy ne kelljen elemezni, vagy kevesebb B/K-t végrehajtani, ami viszont több elemzést von maga után. Ebben a példában egy ideiglenes fájlba helyezzük el a típust, a műveleti kódot, az utasítás hosszát és az aktuális bemeneti sort. Ezt olvassa majd a második menet az eredeti bemeneti fájl helyett.

Elérve az END pszeudoutasítást, az első menet befejeződik. Ha szükséges, akkor a szimbólumtáblát és a literáltáblát rendezi. A rendezett táblázatban előforduló ismétlődő bejegyzéseket pedig kitörli.

```
public static void pass_one( ) {
  // Egy egyszerű assembler első meneténck vázlata.
  boolean more_input = true;                    // jelző az első menet megállítására
  String line, symbol, literal, opcode;         // egy utasítás mezői
  int location_counter, length, value, type;    // egyéb változók
  final int END_STATEMENT = -2;                 // jelzi a bemenő adatok végét

  location_counter = 0;                         // az első utasítás fordítása 0 címnél
  initialize_tables( );                         // a kezdeti beállításokat csinálja

  while (more_input) {                          // a more_input értékét END állítja hamisra
    line = read_next_line( );                   // vesz egy bemeneti adatsort
    length = 0;                                 // az utasítás hossza kezdetben 0
    type = 0;                                   // az utasítás típusa kezdetben 0

    if (line_is_not_comment(line)) {
      symbol = check_for_symbol(line);          // a sorban van címke?
      if (symbol != null)                       //ha igen, bejegyzi a szimbólumot és az értéket
        enter_new_symbol(symbol, location_counter);
      literal = check_for_literal(line);        // tartalmaz a sor literált?
      if (literal != null)                      // ha igen, akkor bejegyzi a táblázatba
        enter_nex_literal(literal);
      // Most a műveleti kód típusát határozza meg. -1 az érvénytelen műveleti kódot jelzi.
      opcode = extract_opcode(line);            // kikeresi a műveleti kód mnemonic-ot
      type = search_opcode_table(opcode);       // az utasítás formája, pl. OP REG1, REG2
      if (type < 0)                             // ha nem műveleti kód, akkor pszeudoutasítás?
        type = search_pseudo_table(opcode);
      switch(type) {                            // meghatározza az utasítás hosszát
        case 1: length = get_length_of_type1(line); break;
        case 2: length = get_length_of_type2(line); break;
        // további esetek
      }
    }

    write_temp_file(type, opcode, length, line);      // hasznos a második menet számára
    location_counter = location_counter + length;     // frissíti a loc_ctr változót
    if (type == END_STATEMENT) {                      // vége van a bemeneti adatoknak?
      more_input = false;                             // ha igen, akkor járulékos feladatokat hajt végre
      rewind_temp_for_pass_two( );                    // többek között a temp állomány visszatekerése
      sort_literal_table( );                          // a literál tábla rendezése
      remove_redundant_literals( );                   // a többszörös előfordulás megszüntetése
    }
  }
}
```

7.10. ábra. *Egy egyszerű assembler első menete*

7.3.3. Második menet

A második menet feladata a tárgyprogram előállítása, és esetleg a fordítási (assembly) lista kinyomtatása. Ezen túlmenően a második menetnek egyéb információt is ki kell adnia, amelyre a szerkesztőnek lesz szüksége, amikor futtatható fájlt szerkeszt a különböző időben fordított eljárásokból. A 7.11. ábra a második menet egy vázlatát tartalmazza.

A második menet többé-kevésbé az első menethez hasonlóan működik: ez is egyesével olvassa, és egyesével dolgozza fel a sorokat. Mivel mindegyik utasítás a típusára, a műveleti kódra és a hosszra vonatkozó információval kezdődik (az ideiglenes fájlban), így ezeket beolvashatja, és ezzel megmenekül bizonyos elemzé-

```
public static void pass_two( ) {
   // Egy egyszerű assembler második menetének vázlata
   boolean more_input = true;                  // jelző, ami a második menetet megállítja
   String line, opcode;                        // egy utasítás mezői
   int location_counter, length, type;         // egyéb változók
   final int END_STATEMENT = -2;               // jelzi a bemenő adatok végét
   final int MAX_CODE = 16;                    // az utasításonkénti kód maximális bájtszáma
   byte code[] = new byte[MAX_CODE];           // utasításonként a generált kódot tárolja

   location_counter = 0;                       // az első utasítás fordítása a 0 címnél

   while (more_input) {                        // a more_input értékét END állítja hamisra
      type = read_type( );                     // veszi a következő sor típus mezőjét
      opcode = read_opcode( );                 // veszi a következő sor műveleti kód mezőjét
      length = read_length( );                 // veszi a következő sor hosszmezőjét
      line = read_line( );                     // veszi a bemenet aktuális sorát

      if (type != 0) {                         // a megjegyzés sor 0 típusú
         switch(type) {                        // a kimeneti kód generálása
            case 1: eval_type1(opcode, length, line, code); break;
            case 2: eval_type2(opcode, length, line, code); break;
            // további esetek itt
         }
      }

      write_output(code);                      // kiírja a bináris kódot
      write_listing(code, line);               // listába helyez egy sort
      location_counter = location_counter + length;  // frissíti a loc_ctr változót
      if (type == END_STATEMENT) {             // elfogytak a bemeneti adatok?
         more_input = false;                   // ha igen, akkor járulékos feladatokat hajt végre
         finish_up( );                         // hibát ellenőriz és befejeződik
      }
   }
}
```

7.11. ábra. *Egy egyszerű assembler második menete*

sektől. A kódgenerálást az *eval_type1*, *eval_type2* (és így tovább) eljárások végzik. Ezek mindegyike valamely mintának megfelelő esetet kezel, például két regiszteroperandussal rendelkező műveleti kód esetét. Az utasításhoz generált bináris kódot a *code* változóban adja vissza. Ez aztán kiírásra kerül. Nagy valószínűséggel a *write_output* csak összegyűjti egy pufferben a bináris kódot, és a lemezre nagyobb egységekben írja ki, hogy ezzel csökkentse a lemezes műveletek számát.

Az eredeti forráskódú utasítás, és a belőle generált (hexadecimális) kód ezután kinyomtatható, vagy egy pufferbe helyezhető későbbi nyomtatás céljából. Az utasítás-helyszámláló (ILC) módosítása után kezdődhet a következő utasítás feldolgozása.

Eddig feltettük, hogy a forráskódú programban nincs hiba. Azonban aki valaha is írt valamilyen nyelven programot, az tudja, hogy mennyi a realitása ennek a feltevésnek. A gyakori hibák közül néhányat az alábbiakban felsorolunk:

1. Egy még nem definiált szimbólum alkalmazása.
2. Egy szimbólum többszörösen definiált.
3. A műveleti kód mezőben levő név nem megengedett műveleti kód.
4. A műveleti kód mellett kevés operandus van megadva.
5. A műveleti kód mellett túl sok operandus van megadva.
6. Egy oktális számban a 8-as vagy a 9-es számjegy előfordul.
7. Jogtalan regiszterhasználat (például ugrás egy regiszterre).
8. Hiányzik az END utasítás.

A programozók a legváltozatosabb és váratlanabb hibák elkövetésére képesek. Gyakran gépelési hiba okozza a definiálatlan szimbólumhiba előfordulását. Ilyenkor egy okos assembler megpróbálja kitalálni, hogy melyik definiált szimbólum hasonlít leginkább a definiálatlanhoz, és ezt használja helyette. A legtöbb hiba korrigálásában azonban az assembler nem túl sokat tehet. Egy értelmezhetetlen utasítás esetén az assembler nem tehet mást, minthogy kiír egy hibaüzenetet, és megpróbálja folytatni a fordítást.

7.3.4. Szimbólumtábla

Az assembler az első menetben a szimbólumokkal és értékükkel kapcsolatos információkat összegyűjti, és a második menet számára a szimbólumtáblában tárolja. A szimbólumtábla felépítésére többféle módszer ismeretes. Ezek közül az alábbiakban néhányat röviden ismertetünk. Mindegyik módszer az **asszociatív memória** működését próbálja utánozni, amely lényegében párok (szimbólum és érték) halmazából áll. Az asszociatív memória a szimbólum ismeretében megmondja a kapcsolódó értéket.

Ennek legegyszerűbb megoldása, amikor a szimbólumtáblapárokból álló tömb adatstruktúra, a párok első komponense a szimbólum (vagy egy mutató rá), a második komponense pedig az érték (vagy egy mutató rá). Egy szimbólum keresése-

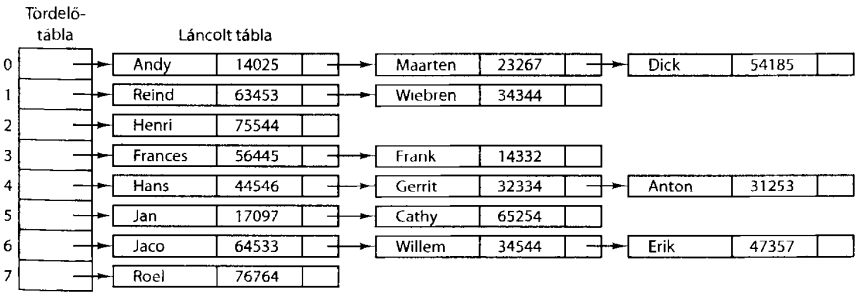
kor a szimbólumtábla rutinja a táblázatban csak lineárisan tudja a szimbólumot keresni. Ez a módszer könnyen programozható, de lassú, mivel átlagban egy-egy szimbólum kikereséséhez a táblázat felét át kell vizsgálni.

A másik módszernél a szimbólumtábla a szimbólumok szerint rendezett és a szimbólum keresését **bináris kereső algoritmus**sal végzi. Ez az algoritmus a szimbólumot a táblázat középső elemével hasonlítja. Ha a szimbólum lexikografikusan kisebb a középső elemben tárolt szimbólumnál, akkor a táblázat első felében kell szerepelnie. Ha a keresett szimbólum a középső elembelinél nagyobb, akkor pedig a táblázat második felében kell előfordulnia. Végül, ha a középső elembeli szimbólum megegyezik a keresettel, akkor a keresés befejeződik.

Abban az esetben, amikor a középső elem nem a keresett szimbólumot tartalmazza, akkor is legalább az kiderül, hogy a táblázat melyik felében kell tovább keresnünk. A bináris keresést most a táblázatnak a megfelelő felében kell folytatni, és ekkor vagy megtaláljuk a szimbólumot, vagy a további keresést már csak a táblázat megfelelő negyedére kell folytatni. Ezt a keresést egy n elemű táblázatban rekurzí-

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

7.12. ábra. *Tördelő kódolás. (a) Szimbólumok, értékek és a szimbólumok tördelési értékei. (b) 8 réses tördelőtáblázat a szimbólumok és értékek láncolt listáival*

van alkalmazva, legfeljebb $\log_2 n$ lépésre van szükség. Nyilvánvaló, hogy ez a módszer a lineáris keresésnél sokkal gyorsabb, de egy rendezett táblázat kell hozzá.

Az asszociatív memória szimulálásának egy teljesen más módja **tördelő kódolási módszer**ként, **hash-elés**ként vagy **hasítás**ként ismert. Ehhez tördelőfüggvény kell, amely a szimbólumokat leképezi a 0-tól k – 1-ig terjedő egész számok halmazába. Az a függvény megfelelő lehet tördelőfüggvénynek, amely a szimbólumban előforduló karakterek ASCII kódját összeszorozza, figyelmen kívül hagyja az esetleges túlcsordulást, majd képezi a k szerinti modulo értékét, vagy elosztja egy prímszámmal. Valójában majdnem minden függvény alkalmas, amely az input tördelési értékeit egyenletesen osztja szét. A szimbólumok 0-tól k – 1-ig terjedő egész számokkal sorszámozott k darab **rés**ből álló táblázatba kerülnek. Az összes i tördelési értékű szimbólum a táblázat i-edik sorszámú réséből induló listába tárolódik. n darab szimbólummal és k darab réssel rendelkező tördelőtáblázat esetén a listák átlagos hossza n/k. Ha k értékét n-hez közelinek választjuk, akkor átlagban a szimbólumok egy lépésben megtalálhatók. A k változtatásával a táblázat mérete csökkenthető, amiért a keresés lassulásával kell fizetni. A tördelő kódolási módszert a 7.12. ábra mutatja.

7.4. Szerkesztés és betöltés

A legtöbb program egynél több fájlból (modulból) áll. A fordítóprogramok és az assemblerek általában egyszerre egy modult fordítanak, és az eredményt lemezen tárolják. A program futtatását megelőzően az összes lefordított modult meg kell keresni, és megfelelően össze kell szerkeszteni. Amennyiben nem áll rendelkezésre virtuális memória, akkor az összeszerkesztett programot be kell tölteni közvetlenül a memóriába. Az ezeket a feladatokat ellátó programok különböző neveken ismertek, például hívják ezeket **szerkesztő**nek, **szerkesztő-betöltő**nek és **szerkesztő-editor**nak is. Egy forrásprogram teljes fordítása két lépésben végezhető el, ez látható a 7.13. ábrán.

1. A forráskódú eljárások fordítása.
2. A tárgykódú modulok összeszerkesztése.

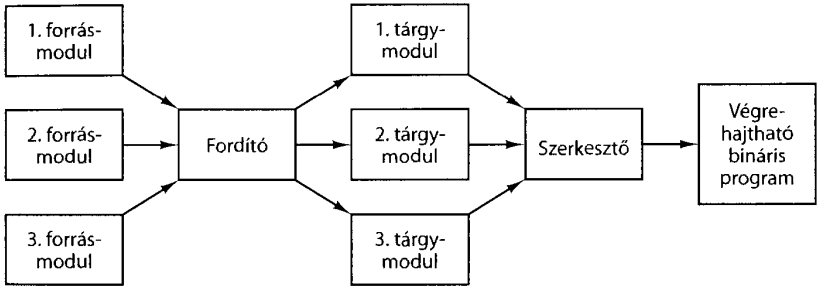
Az első lépést a fordítóprogram vagy az assembler végzi, míg a második lépést a szerkesztőprogram hajtja végre.

A forráskódú modul tárgykódú modullá való átalakítása egy szintváltással valósul meg, mivel a forráskódú nyelv és a célnyelv eltérő utasításokat és jelöléseket alkalmaz. Ezzel szemben a szerkesztés folyamatát nem kíséri szintváltás, mivel a szerkesztő bemeneti és kimeneti programjai ugyanazon virtuális gép programjai. A szerkesztő feladata a külön-külön lefordított eljárások összegyűjtése és ezek ún. **végrehajtható bináris program**má való összeszerkesztése. Az MS-DOS-nál, a Windows 95/98-nál és az NT-nél a tárgymodulok kiterjesztése *.obj*, a végrehajtható

bináris programok kiterjesztése *.exe*. A UNIX-nál a tárgymodulok kiterjesztése *.o*, a futtatható bináris programoknak pedig nincs kiterjesztésük.

Nem véletlen, hogy a fordítóprogramok és az assemblerek a forráskódú modulokat egymástól függetlenül fordítják le. Ha ugyanis a fordítóprogram vagy az assembler úgy működne, hogy beolvasná a forráskódú modulok sorozatát, és közvetlenül elkészítené a végrehajtható gépi kódú programot, akkor egyetlen eljárás egyetlen utasításának megváltozása esetén az összes forráskódú modult újra kellene fordítani.

A 7.13. ábra által szemléltetett független tárgymodul technika alkalmazásakor csak a megváltozott modult kell újrafordítani, a változatlanokat nem, de szükség van az összes tárgymodul ismételt szerkesztésére. Rendszerint a szerkesztés folyamata sokkal gyorsabb, mint a fordításé, így egy program fejlesztésekor jelentős idő takarítható meg a fordítás és szerkesztés kétlépéses módszerével. Ez a tulajdonság rendkívül fontos olyan programoknál, amelyek modulok százaiból vagy esetleg ezreiből állnak.

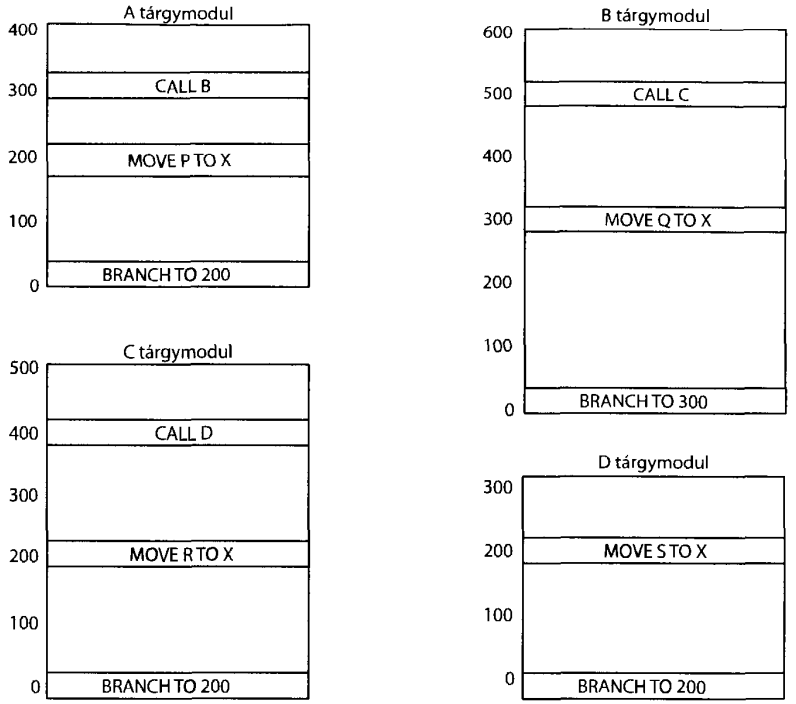


7.13. ábra. *Függetlenül lefordított eljárásokból egy végrehajtható bináris kódú program generálása szerkesztő alkalmazásával*

7.4.1. A szerkesztő feladatai

Az assembly folyamat első menete indításakor az utasítás-helyszámláló értéke 0. Ennek felel meg az a feltételezés, hogy végrehajtáskor a tárgymodul betöltési (virtuális) címe 0. A 7.14. ábrán egy általános gép négy tárgymodulját láthatjuk. Ebben a példában mindegyik modul egy, a modulban levő MOVE utasításra való BRANCH utasítással kezdődik.

A program futtatásához a szerkesztő a tárgymodulokat behozza a memóriába, hogy kialakítson egy végrehajtható bináris programot, ezt mutatja a 7.15. (a) ábra. A szerkesztő a végrehajtandó bináris program virtuális címeinek pontos értékét meghatározza, figyelembe véve a tárgymodulok betöltési címeit. Ha kevés a (virtuális) memóriahely a program számára, akkor egy lemezes fájlt használ. Rendszerint a 0. címtől kezdve a memória egy kis része a megszakításvektorok számára, az operációs rendszerrel való kommunikációra, kezdeti értékkel nem rendelkező mutatók rögzítésére és más egyéb célokra van fenntartva, így a prog-



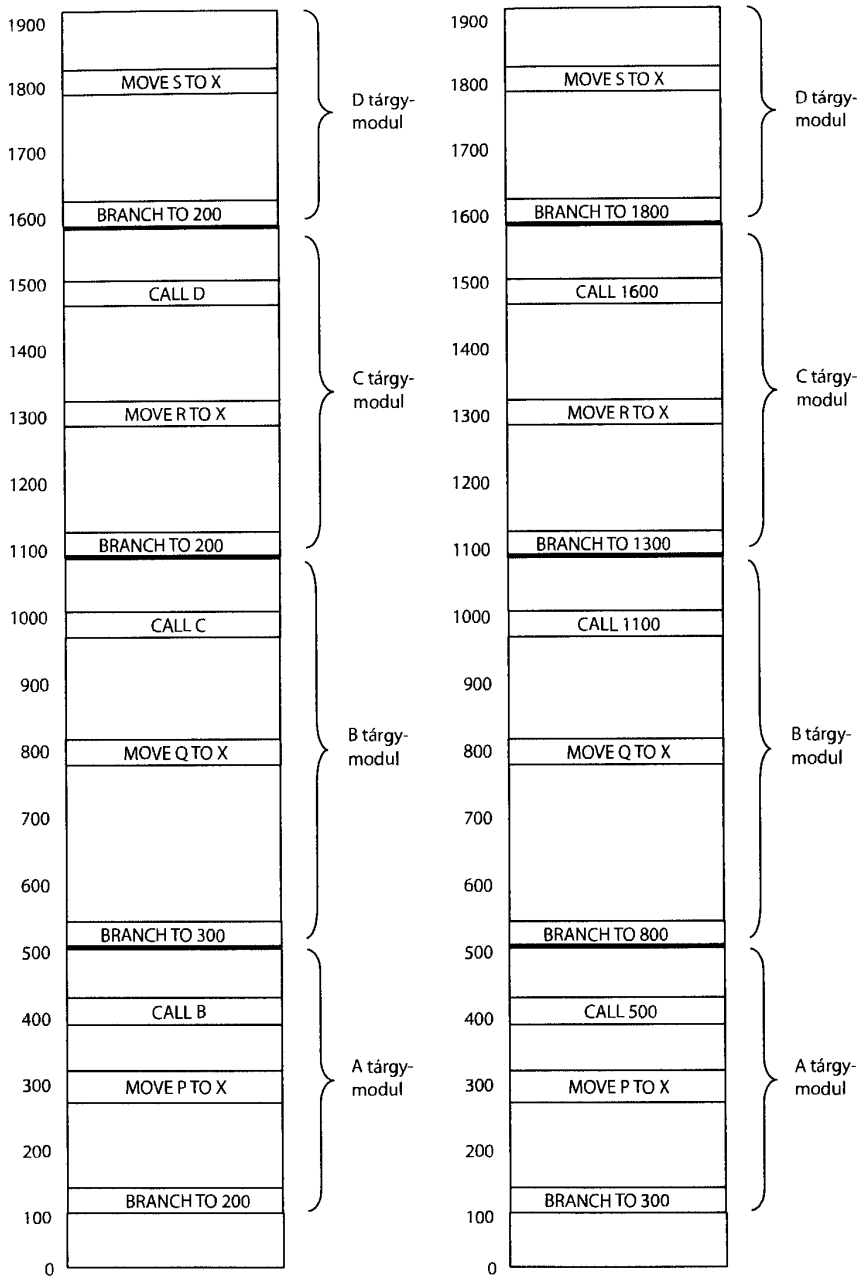
7.14. ábra. *Mindegyik modul saját, 0-val kezdődő címtartománnyal rendelkezik*

ramok gyakran 0-nál nagyobb címen kezdődnek. Ábránkon a program a 100-as címnél kezdődik.

Bár a végrehajtandó bináris program a 7.15. (a) ábrán látható módon be van töltve a memóriába, a program mégsem futtatható. Nézzük, mi történne, ha a végrehajtás az *A* modul elején levő utasítással kezdődne. A program nem a MOVE utasításra ugrana, pedig azt kellene tennie, mivel ez az utasítás most a 300-as címnél van. Valójában, ugyanebből az okból az összes memóriahivatkozást tartalmazó utasítás hibás. Nyilvánvaló, hogy valamit tenni kell.

Ez a probléma, amely **áthelyezési problémaként** ismert, azért jelentkezik, mert a 7.14. ábrán a tárgymodulok saját címtérben vannak. A Pentium 4-hez hasonló szegmenscímzésű gépeken elméletileg mindegyik tárgymodul saját címzést alkalmazhat a szegmensében. A Pentium 4-nél csak az OS/2 operációs rendszer támogatja ezt a koncepciót. Az összes Windows-verzió és a UNIX csak egyetlen lineáris címtartományt támogat, így az összes tárgymodult erre az egyetlen helyre kell összefésülni.

A 7.15. (a) ábrán az eljáráshívás utasítások sem fognak működni. A 400-as címnél a programozó a *B* tárgymodult akarta hívni, azonban mivel mindegyik eljárás fordítása külön történt, így az assembler nem tudhatja, hogy a CALL B utasításba milyen címet írjon be. A *B* tárgymodul címe a szerkesztésig nem ismert. Ez a prob-



7.15. ábra. *(a) A 7.14. ábra tárgymoduljainak elhelyezése utáni, de az áthelyezést és szerkesztést megelőző képe. (b) Ugyanazon tárgymodulok szerkesztés és áthelyezés utáni képe*

léma **külső hivatkozási problémaként** ismert. Mindkét problémát egyszerűen meg tudja oldani a szerkesztő.

A szerkesztő a tárgymodulok saját címtereit egyetlen lineáris címzéssé fésüli össze az alábbi lépésekben:

1. Készít egy táblázatot az összes tárgymodulról és azok hosszáról.
2. Ezt a táblát alapul véve, mindegyik tárgymodulhoz egy kezdőcímet rendel.
3. Megkeresi az összes memóriára hivatkozó utasítást, és mindegyikhez hozzáad egy **áthelyezési konstans**t, amely a modul kezdőcímével egyenlő.
4. Megkeresi az összes más modulra hivatkozó utasítást, és a megfelelő eljárások címét beírja ezekbe.

A 7.15. ábra moduljaihoz tartozó első lépésben készülő tárgymodultábla a következőképpen néz ki:

Modul	Hossz	Kezdőcím
A	400	100
B	600	500
C	500	1100
D	300	1600

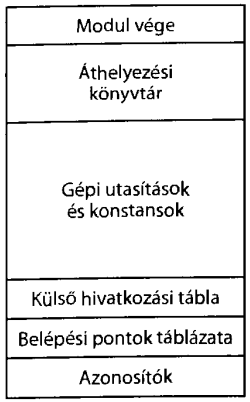
Ebben szerepel mindegyik modul neve, hossza és kezdőcíme. A 7.15. (b) ábra azt mutatja, hogy a 7.15. (a) ábra címei a szerkesztés előbb felsorolt lépései után hogyan alakulnak.

7.4.2. A tárgymodul szerkezete

Egy tárgymodul gyakran hat részből épül fel, ezt mutatja a 7.16. ábra. Az első rész tartalmazza a modul nevét, a szerkesztő számára szükséges bizonyos információkat, mint a modul különféle darabjainak a hosszát és néha a létrehozás dátumát.

A tárgymodul második része a modulban definiált azon szimbólumok listája az értékükkel együtt, amelyekre más modulok hivatkozhatnak. Például, ha a modulban van egy *nagyhiba* nevű eljárás, akkor a belépési pontok táblázata a „nagyhiba" karakterláncot a hozzá tartozó eljárás címével együtt tartalmazza. Az assembly programozó feladata jelezni, hogy mely szimbólumokat akarja **belépési pont**ként deklarálni; ezt a 7.3. ábrán szereplő PUBLIC-hoz hasonló pszeudoutasítással adhatja meg.

A tárgymodul harmadik része olyan szimbólumok listájából áll, amelyek a modulban használhatók, de a definíciójuk más modulokban vannak. A szimbólumok itt azon gépi utasítások listájával együtt szerepelnek, amelyek használják. A szerkesztőnek erre az utóbbi listára azért van szüksége, hogy be tudja írni a külső szimbólumokat használó utasításokba a helyes címeket. Egy eljárás hívhat olyan tőle függetlenül fordított eljárásokat, amelyeket külső névként definiált. Az assembly programozónak jelezni kell, hogy mit szeretne **külső szimbólum**ként használni; ezt a 7.3. ábrában látható EXTERN vagy ehhez hasonló pszeudoutasítás-



7.16. ábra. *Egy fordító által készített tárgymodul belső felépítése*

sal teheti meg. Néhány számítógépnél egyetlen táblázatban szerepelnek a belépési pontok és a külső hivatkozások.

A tárgymodul negyedik részében van a lefordított kód és a konstansok. A tárgymodul részei közül ez az egyetlen, amely végrehajtáskor a memóriába töltődik. A többi öt részt a szerkesztőprogram használja a munkája elvégzéséhez, azonban a futtatás kezdete előtt eldobja ezeket.

A tárgymodul ötödik része az áthelyezési könyvtár. Mint ahogy a 7.15. ábrán is látható, vannak memóriacímeket tartalmazó utasítások, amelyekben ezeket a címeket egy áthelyezési konstanssal meg kell növelni. Mivel a szerkesztőnek nincs módja arra, hogy a negyedik részben levő adatszavakat megvizsgálja abból a szempontból, hogy gépi utasítás vagy konstans, ezért az áthelyezést igénylő címekről ez a táblázat tartalmaz információt. Az információ tárolódhat egy bittáblázatban, áthelyezendő címenként egy bittel vagy az áthelyezendő címek egy közvetlen listájában.

A hatodik rész a modul vége jelzés, tartalmazhatja a hibaellenőrző összeget, ami segít a modul olvasásakor fellépő esetleges hibák észlelésében, és a végrehajtás kezdőcímét.

A legtöbb szerkesztő két menetben dolgozik. A szerkesztő az első menetben beolvassa az összes tárgymodult, és felépíti a modulok nevét és hosszát tartalmazó táblázatot, továbbá egy globális szimbólumtáblát az összes belépési pontból és külső hivatkozásból. A második menetben olvassa a tárgymodulokat, elvégzi az áthelyezéseket, és ekkor szerkeszti össze őket egy modullá.

7.4.3. Hozzárendelési idő és dinamikus áthelyezés

Egy multiprogramozásos rendszer egy programot betölthet a memóriába, egy rövid ideig tartó futtatás után lemezre tárolhatja, hogy aztán onnan ismét a memóriába tölthesse további futtatáshoz. Sok programból álló nagy rendszereknél nehezen biztosítható, hogy ugyanarra a helyre történjen a program visszatöltése minden alkalommal.

A 7.17. ábrán látható, hogy mi történik, ha a 7.15. (b) ábrán szereplő már lefordított és összeszerkesztett programot a 400-as címre tölti a 100-as cím helyett, ahová eredetileg a szerkesztő helyezte. Az összes memóriacímzés hibás lesz, és már az áthelyezési információ sincs meg. Még ha az áthelyezési információk elérhetők is lennének, akkor is az összes cím ismételt áthelyezési költsége, amit a program minden egyes memóriába töltésekor el kellene végezni, nagyon nagy lenne.

A már összeszerkesztett és áthelyezett programok mozgatásának problémája abból adódik, hogy ekkor a szimbolikus nevek már abszolút fizikai címekkel vannak helyettesítve. Programíráskor szimbolikus neveket használunk memóriacímek helyett, például `BR L`. Azt az időpontot, amikor az *L*-hez az aktuális memóriacím hozzárendelődik, **hozzárendelési idő**nek nevezzük. Legalább hatféle hozzárendelési idő létezik:

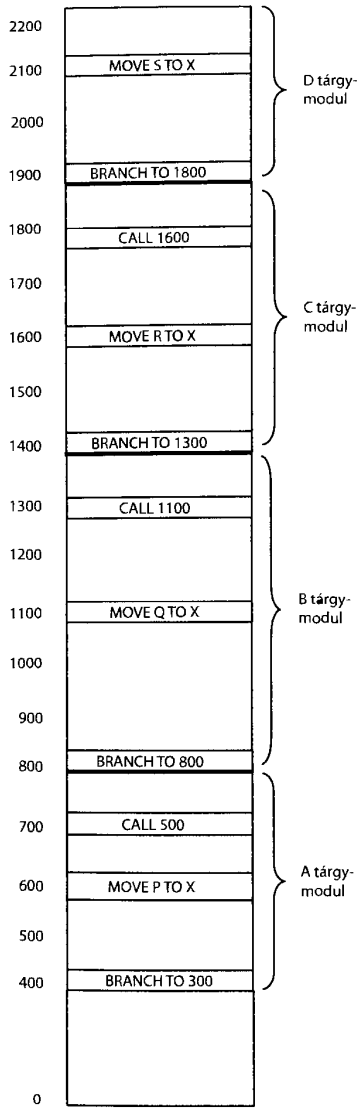
1. A program írásakor.
2. A program fordításakor.
3. A program szerkesztésekor, de a betöltés előtt.
4. A program betöltésekor.
5. A címzésnél használt bázisregiszter beállításakor.
6. A címet tartalmazó utasítás végrehajtásakor.

Ha egy memóriacímet tartalmazó utasítás mozgatása a hozzárendelés után történik, akkor érvénytelen, hibás lesz (feltéve, hogy a hivatkozott objektum is elmozdul). Ha a fordító kimenetként végrehajtható bináris kódot generál, akkor fordítási időben kerül sor a hozzárendelésekre, és a programot a fordító által meghatározott címen kell futtatni. Az előző részben tárgyalt szerkesztéses módszernél az abszolút címek szimbolikus nevekhez való hozzárendelése a szerkesztés alatt történik, így a program szerkesztés utáni mozgatása hibát okoz, ezt szemlélteti a 7.17. ábra.

Két kérdés jelentkezik ezzel kapcsolatban. Az első, hogy a szimbolikus nevekhez mikor történik a virtuális címek hozzárendelése. A második, hogy a virtuális címekhez mikor történik az abszolút címek hozzárendelése. A teljes hozzárendelés csak mindkét művelet végrehajtása után valósul meg. A virtuális címek hozzárendelése akkor történik, amikor a szerkesztő a tárgymodulok egymástól független címzéseit egyetlen lineáris címzéssé alakítja át. Az áthelyezés és szerkesztés során történik a szimbolikus nevek speciális virtuális címekhez rendelése. Ez a megfigyelés attól függetlenül igaz, hogy van-e virtuális memória, vagy nincs.

Egy pillanatra tegyük fel, hogy a 7.15. (b) ábrán a címterület lapozott. Világos, hogy az *A*, *B*, *C* és *D* szimbólumoknak megfelelő virtuális címek már definiáltak, azonban a fizikai memóriacímek a laptábla akkori tartalmától függnek, amikor hivatkozunk rájuk. Egy végrehajtható bináris program valójában a szimbolikus nevek virtuális nevekhez kötése.

Az olyan mechanizmusokban, amelyeknél a virtuális címek fizikai memóriacímekhez kötése könnyen változtatható, a programok mozgathatók a memóriában



7.17. ábra. *A 7.15. (b) ábrán szereplő összeszerkesztett program 300 címmel feljebb mozgatva. Sok utasítás most hibás memóriacímre hivatkozik*

a virtuális címek hozzárendelése után is. Egy ilyen módszer a lapozási technika. Egy program memóriában való elmozdulása esetén csak a laptábláját kell módosítani, nem magát a programot.

Egy másik módszer futás idejű áthelyezési regisztert használ. A CDC 6600-nál és ennek későbbi verzióinál van ilyen regiszter. Az ilyen áthelyezési technikát alkalmazó gépeken a regiszter mindig az aktuális program kezdő memóriacímére mutat. A memóriába töltés előtt az összes memóriacímhez az áthelyezési regiszter hozzáadódik hardveres úton. A teljes áthelyezési folyamat láthatatlan a felhasználói programok számára. Még tudni sem kell a létezéséről. A program elmozdulásakor az operációs rendszer az áthelyezési regisztert megfelelően módosítja. Ez a módszer a lapozási technikánál kevésbé általános, mivel a teljes programot egy egységként mozgatja (kivéve, ha van külön kód- és adatáthelyezési regiszter, mint például az Intel 8088-nál, mert ekkor két egységként mozgatja).

A harmadik módszer olyan gépeken lehetséges, amelyek az utasításszámlálótól függő memóriacímzésre képesek hivatkozni. Sok PC relatív elágazó utasítás van, ezek segítenek. Valahányszor a memóriában egy program elmozdul, csak az utasításszámlálóját kell frissíteni. **Helyfüggetlen**nek nevezik azokat a programokat, amelyeknek minden memóriára hivatkozása vagy relatív az utasításszámlálóhoz viszonyítva vagy abszolút (például a B/K eszköz abszolút címen lévő regiszterei esetében). A helyfüggetlen kódú eljárások a virtuális címtartományon belül bárhová elmozdulhatnak anélkül, hogy áthelyezési tevékenységet igényelnének.

7.4.4. Dinamikus szerkesztés

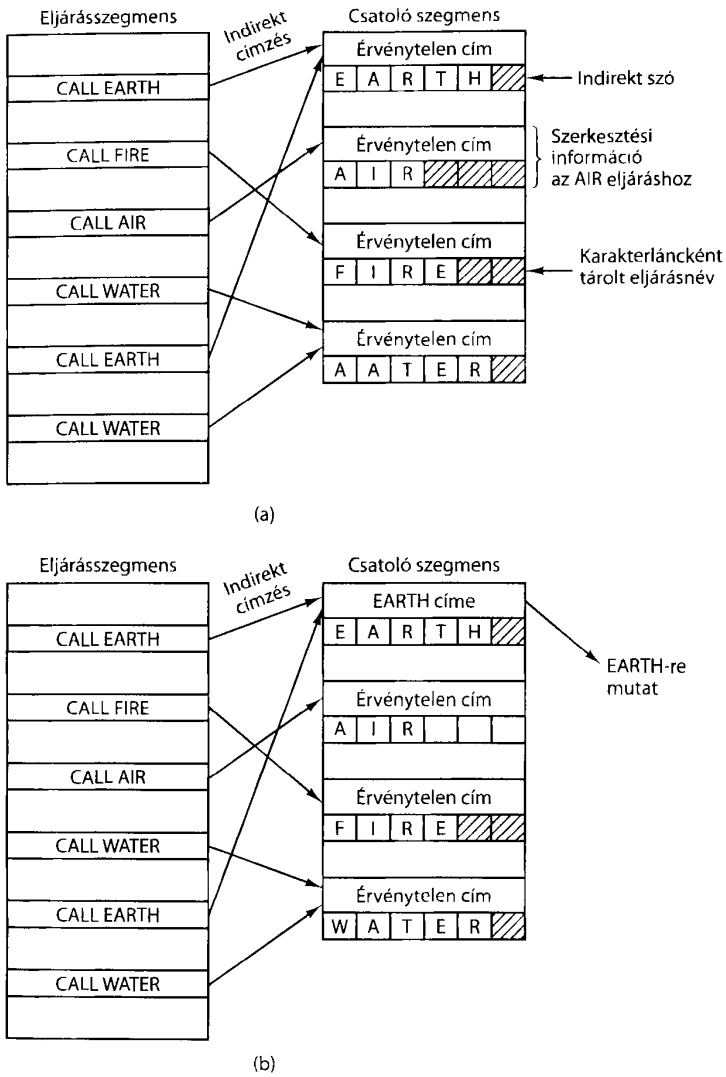
A 7.4.1. részben tárgyalt szerkesztési stratégiának az a tulajdonsága, hogy még a program futtatását megelőzően a programhoz hozzászerkeszti az összes olyan eljárást, amelyet esetleg hív a program. Egy virtuális memóriájú számítógép azáltal, hogy futtatás előtt minden szerkesztést elvégez, nem használja ki a virtuális memóriából adódó lehetőségeket. Sok programnak vannak olyan eljárásai, amelyeket csak váratlan körülmények esetén hív. Például a fordítóprogramoknál a ritkán használt utasítások fordítását végző eljárások, vagy az elvétve jelentkező hibahelyzeteket kezelő eljárások ilyenek.

A külön lefordított eljárások szerkesztésének rugalmasabb módja az, amelynél az egyes eljárásokat az első hívásukkor szerkesztik be. Ez a folyamat **dinamikus szerkesztés**ként ismert. A MULTICS alkalmazta elsőként ezt a módszert, amelyet bizonyos értelemben még ma sem múltak felül. A következőkben több rendszernél megnézzük a dinamikus szerkesztést.

A MULTICS dinamikus szerkesztése

A MULTICS dinamikus szerkesztése minden programhoz egy ún. **csatoló szegmens**t társít, amelyben a hívható eljárások mindegyikének van egy információs blokkja. Az információs blokk elején egy szó van fenntartva az eljárás virtuális címe számára, ezt az eljárás neve követi, amely karakterláncként tárolt.

Dinamikus szerkesztéskor a forrásnyelvű eljáráshívás olyan utasításokká fordítódik, amelyek indirekt módon címzik a megfelelő csatoló blokk első szavát; ezt



7.18. ábra. *Dinamikus szerkesztés. (a) EARTH hívása előtt. (b) EARTH hívása és szerkesztése után*

mutatja a 7.18. (a) ábra. A fordítóprogram ebbe a szóba vagy egy érvénytelen címet ír vagy egy megszakítást generáló speciális bitformát.

Egy másik szegmens eljárásának hívásakor az ekkor érvénytelen szóra tett címzési kísérlet közvetve egy megszakítást okoz a dinamikus szerkesztő számára.

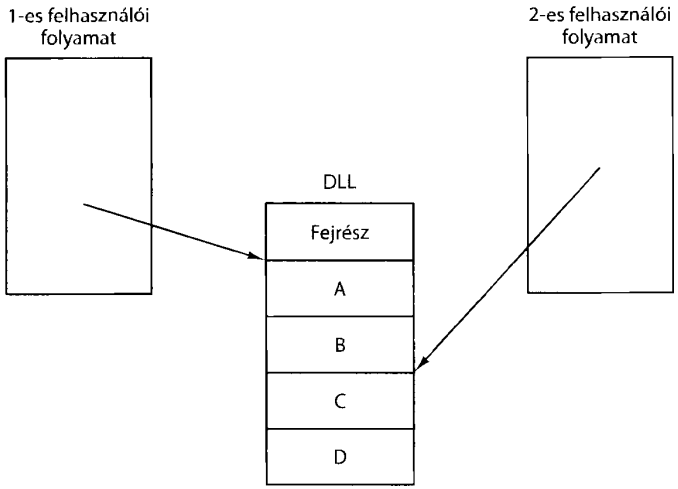
A szerkesztő ekkor kiveszi az érvénytelen címet követő szóban levő karakterláncot, és egy ilyen nevű lefordított eljárást keres a felhasználó könyvtárában. Az eljáráshoz ezután egy virtuális cím rendelődik, ez rendszerint a saját szegmenséhez tartozik, és ez a virtuális cím került a csatoló szegmensben az érvénytelen cím helyére, amint az a 7.18. (b) ábrán is látható. Ezt követően a szerkesztési hibát okozó utasítást ismételten elvégzi, vagyis a program a megszakítást megelőző helyzettől folytatódik.

Erre az eljárásra vonatkozó összes újabb hivatkozás nem okoz majd szerkesztési hibát, mivel az indirekt címet tartalmazó szó most már egy érvényes virtuális címet tartalmaz. Következésképpen, a dinamikus szerkesztő csak az eljárás első meghívásakor aktivizálódik, utána már nem.

A Windows dinamikus szerkesztése

A Windows operációs rendszer mindegyik verziója, beleértve az XP-t is, támogatja a dinamikus szerkesztést, és erősen támaszkodik rá. A dinamikus szerkesztéshez egy speciális fájlformátumot használ, amelyet **DLL**-nek **(Dynamic Link Library, dinamikus szerkesztő könyvtár)** hívnak. A DLL-ek tartalmazhatnak eljárásokat, adatokat vagy mindkettőt. Rendszerint arra használják, hogy lehetőséget biztosítsanak két vagy több folyamat számára a könyvtár eljárásainak és adatainak megosztott alkalmazására. Sok DLL-nek *.dll* a kiterjesztése, de más kiterjesztések is előfordulnak, ilyen a *.drv* (meghajtó könyvtáraknál) és a *.fon* (font könyvtáraknál).

Egy DLL leggyakrabban olyan eljárások gyűjteményéből álló könyvtár, amelyet a memóriába töltve több folyamat egy időben elérhet. A 7.19. ábrán két program osztozik egy DLL fájlon, amely az *A*, *B*, *C* és *D* eljárásokból áll. Az 1-es program



7.19. ábra. *Két folyamat által használt DLL fájl*

az *A* eljárást, a 2-es program a *C* eljárást használja, de használhatják ugyanazt az eljárást is.

DLL fájlt a szerkesztő hoz létre az input fájlok egy gyűjteményéből. Valójában egy DLL fájl kialakítása nagyban hasonlít egy végrehajtható bináris program kialakításához, kivéve, hogy egy speciális jel segítségével a szerkesztővel közölni kell, hogy abból DLL-t csináljon. A DLL-ek szokásosan olyan eljáráskönyvtárak gyűjteményeiből készülnek, amelyekre nagy valószínűséggel több folyamatnak is szüksége lesz. A Windows-interfészeljárások rendszerhívó könyvtára és a nagy grafikus könyvtárak gyakran DLL-ekben valósulnak meg. A DLL-ek alkalmazásával memória és lemezterület spórolható meg. Ha minden programhoz az általa használt könyvtárat statikusan kötnék, akkor egy gyakran alkalmazott könyvtárnak a futtatható bináris kódja a lemezen és a memóriában sokszor előfordulna, ami helypazarlás. A DLL-ek alkalmazásával a lemezen és a memóriában csak egyszer fordul elő minden könyvtár.

A hellyel való takarékoskodáson túl, ez a módszer megkönnyíti a könyvtári eljárások frissítését, még az ezeket használó programok lefordítását és szerkesztését követően is. A DLL-eket használó kereskedelmi szoftvercsomagoknál, amelyekhez a vásárlók ritkán kapnak forráskódot, a szoftverterjesztők a könyvtárakban észlelt hibák javítását könnyen megoldhatják úgy, hogy új DLL fájlokat juttatnak el interneten keresztül a felhasználókhoz, így a főprogram bináris kódja nem igényel változtatást.

A fő különbség egy DLL és egy futtatható fájl között, hogy egy DLL nem képes önmagát futtatni (mivel nincs főprogramja). Ennek is van egy fejrésze különféle információkkal. Továbbá a DLL-nek, mint önálló egységnek, vannak saját eljárásai, amelyek nem kapcsolódnak a könyvtár eljárásaihoz. Például az egyik eljárása automatikusan meghívódik egy új folyamatnak a DLL-hez kötésekor, és egy másik eljárása pedig akkor, amikor egy folyamat a DLL-ről leválasztódik. Ezek az eljárások képesek a memóriát lefoglalni és felszabadítani, illetve más erőforrásokat kezelni, amelyeket a DLL igényel.

Két módon köthető egy program egy DLL-hez. Az **implicit szerkesztés**nek hívott módszernél a felhasználói program az **import könyvtár**nak nevezett speciális fájlhoz statikusan van kapcsolva, amit egy segédprogram valósít meg a DLL-ből nyert bizonyos információk alapján. A felhasználói program az import könyvtáron keresztül éri el a DLL-t. Egy felhasználói program több import könyvtárhoz is köthető. Amikor egy implicit (közvetett) szerkesztést alkalmazó program futtatáskor a memóriába töltődik, a Windows megnézi, milyen DLL-eket használ a program és a memóriában ezek közül melyek vannak már bent. A hiányzó DLL-eket azonnal betölti a memóriába (nem szükséges a teljes egységeket, mivel lapozási technikát használ). Bizonyos módosításokra kerül sor ezután az import könyvtár adatszerkezetében, hogy a hívott eljárások elérhetők legyenek, ezek némileg analóg módon történnek a 7.18. ábrán látottakhoz. Most már a felhasználói program készen áll a futtatásra, és úgy hívhat DLL-beli eljárásokat, mintha azok statikusan lennének hozzákötve.

Az implicit szerkesztés alternatívája az **explicit szerkesztés**. Ennél a módszernél nincs szükség import könyvtárakra, és a DLL-eket sem kell a felhasználói programmal együtt betölteni. A felhasználói program ehelyett futási időben egy közvetlen (explicit) hívással hozza létre a kötését a DLL-hez, majd ezt követően további hívások útján jut hozzá a szükséges eljárások címéhez. Amint ezeket megtalálta, hívhatja az eljárásokat. Amikor mindennel végez, akkor a DLL-ről való leválasztáshoz még egy utolsó hívást csinál. Amikor egy DLL-ről az utolsó folyamat is leválik, akkor a DLL kikerül a memóriából.

Fontos látni, hogy egy DLL-beli eljárásnak nincs semmi önállósága (ellentétben egy szállal vagy folyamattal). A hívó száljában fut, és a hívó vermét használja a lokális változói számára. Lehetnek folyamatspecifikus statikus adatai (például megosztott adatok), de máskülönben ugyanolyan, mint egy statikusan kapcsolt eljárás. Az egyetlen lényeges különbség az eléréséhez szükséges hozzárendelés kialakításában van.

A UNIX dinamikus szerkesztése

A UNIX módszere hasonló a Windows DLL-es koncepciójához. Az általa használt speciális fájlt **megosztott könyvtár**nak hívják. Egy megosztott könyvtár – a DLL fájlhoz hasonlóan – egy archivált fájl, amely megosztott eljárásokat vagy adatmodulokat tartalmaz, amelyek futási időben vannak jelen a memóriában és a folyamatok számára egyidejűleg elérhetők. A standard C könyvtár és a legtöbb hálózati kód megosztott könyvtárakban van.

A UNIX csak az implicit szerkesztést támogatja, ezért egy megosztott könyvtár két részből áll: egy **gazdakönyvtár**ból, amely a végrehajtható fájlhoz statikusan van szerkesztve, és egy **célkönyvtár**ból, amelynek hívása futtatáskor történik. Bár a részletekben eltér, de a koncepciója lényegében ugyanaz, mint a DLL fájloké.

7.5. Összefoglalás

Bár a legtöbb programot magas szintű nyelven lehet, illetve kell írni, mégis vannak olyan helyzetek, amikor az assembly nyelvre van szükség, legalábbis részben. Ilyen programokra van igény a hordozható, szegényes erőforrással rendelkező számítógépek esetén, mint például az intelligens kártyák, az alkalmazásokba beépített processzorok vagy a vezeték nélküli, digitális hordozható segédeszközök. Egy assembly nyelvű program egy bizonyos gépi kódú program szimbolikus formája. Ezt a gép nyelvére egy program fordítja le, amelyet assemblernek hívnak.

Azoknál az alkalmazásoknál, amelyeknek sikere döntően a végrehajtás gyorsaságától függ, jobb megközelítést jelent a teljes program assembly nyelven való megírásánál, ha először a teljes programot egy magas szintű nyelven megírják, utána mérik, hogy mely részek használják el a legtöbb futási időt és végül csak ezeket az erősen használt részeket írják újra assembly nyelven. A gyakorlatban a kód kis része felelős rendszerint a végrehajtási idő nagy hányadáért.

Sok assembler támogatja a makrótechnikát, ami lehetőséget ad a programozóknak, hogy a gyakran használt kódsorozatoknak neveket adjanak, és a későbbiek-

ben ezekkel hivatkozhassanak rájuk. Általában ezek a makrók a szokásos módon paraméterezhetők. A makrók megvalósítása a karakterláncot szavanként feldolgozó algoritmussal történik.

A legtöbb assembler két menetben fordít. Az első menet a címkékből, literálokból és a közvetlenül deklarált azonosítókból felépít egy szimbólumtáblát. A szimbólumok tárolhatók rendezetlenül, és ekkor lineárisan kereshetők, vagy rendezetten, és ekkor a keresésre bináris keresési algoritmus használható, vagy tördelőtáblában. Ha a szimbólumokat az első menetnek nem kell törölni, akkor rendszerint a tördelőtábla alkalmazása adja a legjobb módszert. A második menet generálja a kódot. Bizonyos pszeudoutasításokat az első menet hajt végre, míg másokat a második.

A függetlenül fordított programokat a futtatáshoz egyetlen végrehajtható bináris programmá kell szerkeszteni. Ezt a feladatot látja el a szerkesztő. Elsődleges feladatai az áthelyezés és a nevek hozzárendelése. A dinamikus szerkesztésnél az eljárások a hívásukig nem szerkesztődnek be. A Windows DLL fájljai és a UNIX megosztott könyvtárai dinamikusan szerkesztődnek.

7.6. Feladatok

1. Egy adott programban a kód 2%-a használja a végrehajtási idő 50%-át. Hasonlítsa össze az alábbi három stratégiát a programozási és végrehajtási idő szerint. Tegyük fel, hogy C-ben a programíráshoz 100 programozói időegység kell, míg az assembly kód megírása ennél 10-szer lassabb, de négyszer gyorsabban fut.
   a) C-ben készül az egész program.
   b) Assembly nyelven készül az egész program.
   c) Először minden C-ben, utána a 2%-nyi kritikus részt újraírják assembly nyelven.
2. A kétmenetes assemblerre érvényes megfontolások a fordítóprogramokra is igazak?
   a) Tételezze fel, hogy a fordítóprogramok tárgymodulokat állítanak elő, nem pedig assembly kódot.
   b) Tételezze fel, hogy a fordítóprogramok szimbolikus assembly nyelvet állítanak elő.
3. Az Intel CPU legtöbb assemblerénél az első operandus egy célcím, a második operandus egy forráscím. Milyen problémákat kellene megoldani, ha más módon dolgoznának?
4. Lehet-e az alábbi programot két menetben fordítani? Az EQU olyan pszeudoutasítás, amely a címkét egyenlővé teszi az operandus mezőben levő kifejezéssel.

```
P    EQU  Q
Q    EQU  R
R    EQU  S
S    EQU  4
```


5. A Dirtcheap Software Company egy assembler elkészítését tervezi egy olyan számítógéphez, amelyen a szavak 48 bitesek. A költségek csökkentése miatt dr. Scrooge projektvezető azt a döntést hozta, hogy a szimbólumok hosszát úgy korlátozzák, hogy mindegyik szimbólum tárolható legyen egyetlen szóban. Scrooge közölte, hogy a szimbólumok csak betűkből állhatnak, kivéve a Q-t, amelyet tilos használni (hogy a vásárlóknak demonstrálja a hatékonyságra való törekvésüket). Mi lehet egy szimbólum maximális hossza? Írja le a kódolási sémáját.
6. Mi a különbség az utasítás és a pszeudoutasítás között?
7. Mi a különbség az utasítás-helyszámláló és az utasításszámláló között, ha egyáltalán van különbség? Elvégre mindkettő a programban a következő utasítás helyét tárolja.
8. Adja meg a szimbólumtáblát, miután a következő Pentium 4-es utasításokat már elérte. Az első utasítás kezdőcíme 1000.

```
EVEREST:    POP BX          (1 bájt)
K2:         PUSH BP         (1 bájt)
WHITNEY:    MOV BP,SP       (2 bájt)
MCKINLEY:   PUSH X          (3 bájt)
FUJI:       PUSH SI         (1 bájt)
KIBO:       SUB SI,300      (3 bájt)
```

9. Tudna olyan körülményt említeni, amelyben egy assembly nyelv megengedi, hogy egy címke ugyanaz legyen, mint egy műveleti kód (például a *MOV* címkeként)? Fejtse ki az álláspontját.
10. Sorolja fel a Berkeley listaelem megtalálásához szükséges lépéseket, ha az alábbi listában a bináris keresést alkalmazza: Ann, Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood és Yellow Springs. Páros elemszámú lista középső elemeként az elemszám felénél eggyel nagyobb indexű elemet vegye.
11. Lehet-e binárisan keresni egy prímszám méretű táblában?
12. Számítsa ki a következő szimbólumok tördelési kódját úgy, hogy a betűket (A = 1, B = 2 és így tovább) összeadja, és ennek az összegnek veszi a tördelőtáblázat mérete szerinti modulóját. A tördelőtáblának legyen 0-tól 18-ig sorszámozva 19 rése. A szimbólumok a következők:
    els, jan, jelle, maaike
    A szimbólumok tördelési kódjai egyediek? Ha nem, akkor hogyan lehetne az ütközéseket kezelni?
13. A tördelőtábla módszer alapján az azonos tördelési értékű dolgok együttesen egy láncolt listába kerülnek. Egy alternatív módszernél csak egyetlen egyszerű n réses tábla kell, amelyben mindegyik rés csak egyetlen kulcs és értéke (vagy egy arra mutató mutató) számára biztosít helyet. Ha a tördelőfüggvény olyan értéket generál, amely már foglalt, akkor veszi a második tördelőfüggvényt, és ismét próbálkozik az elhelyezéssel. Ha az elhelyezés ismét nem sikerül, akkor veszi a következő tördelőfüggvényt és így tovább, egészen addig, míg üres helyet nem talál. Ha a megtelt rések hányada R, akkor átlagosan hány próbálkozásra van szükség egy új szimbólum elhelyezéséhez?

14. Ahogy a technológia fejlődik, talán lehetséges lesz egy napon, hogy egy áramköri lapkán több ezer azonos CPU helyezkedjen el, mindegyik néhány szavas lokális memóriával. Ha mindegyik CPU olvashat és írhat három megosztott regisztert, akkor hogyan lehet egy asszociatív memóriát megvalósítani?
15. A Pentium 4 szegmentált felépítésű, több független szegmenssel. Egy ilyen gép assemblere esetleg rendelkezik a SEG N pszeudoutasítással, amely közli az assemblerrel, hogy a következő kódot vagy adatot az N szegmensbe tegye. Ennek a sémának lenne bármi hatása az utasítás-helyszámlálóra?
16. Gyakran a programok több DLL-hez is kapcsolódnak. Egy olyan módszer nem lenne-e hatékonyabb, amelynél az összes eljárás egyetlen nagy DLL-ben lenne elhelyezve, és ehhez lehetne kapcsolódni?
17. Le lehet-e képezni egy DLL-t két különböző virtuális címtartományba tartozó virtuális címre? Ha igen, akkor milyen problémák jelentkezhetnek? Megoldhatók? Ha nem, akkor mit lehet tenni, hogy ne forduljanak elő?
18. A (statikus) szerkesztés egy módszere az alábbi. A könyvtárvizsgálatot megelőzően a szerkesztő elkészíti a szükséges eljárások listáját, amely az összeszerkesztendő modulokban az EXTERN utasítással definiált nevekből áll. Ezután a szerkesztő lineárisan vizsgálja a könyvtárat, és kivesz minden olyan eljárást, amely a névlistában előfordul. Működik ez a módszer? Ha nem, miért nem, és hogyan lehetne orvosolni?
19. Makróhívás aktuális paramétereként használható-e egy regiszter? És egy konstans? Miért igen, vagy miért nem?
20. A feladata éppen egy makróassembler megvalósítása. A főnöke esztétikai okokból úgy dönt, hogy a makródefinícióknak nem szükséges megelőzniük a hívásaikat. Milyen hatással van ez a döntés a megvalósításra?
21. Gondolkozzon el azon, hogy milyen módon kerülhet egy makróassembler végtelen ciklusba.
22. Egy szerkesztő 5 modult olvas, amelyeknek hossza egyenként 200, 800, 600, 500 és 700 szó. Ha ebben a sorrendben történik a betöltés, akkor mik lesznek az áthelyezési konstansok?
23. Írjon egy két rutinból álló szimbólumtábla programcsomagot. Az *enter(symbol, value)* rutin egy új szimbólumot vegyen fel a táblába, a *lookup(symbol, value)* rutin egy szimbólumot keressen meg a táblában. Használjon tördelőtáblát.
24. Írjon egy egyszerű assemblert a 4. fejezetben szereplő Mic–1 számítógépre. A gépi utasítások kezelésén túl, tegye lehetővé fordítási időben konstansok szimbólumhoz rendelését és gépi szóba való elhelyezésüket.
25. Az előző problémát oldja meg úgy, hogy az assemblert egy egyszerű makrózási lehetőséggel bővítse.

8. Párhuzamos számítógép-architektúra

Bár a számítógépek egyre gyorsabbak lesznek, a velük szemben támasztott elvárások legalább olyan gyorsan nőnek. A csillagászok a világegyetem fejlődését szeretnék szimulálni az ősrobbanástól az idők végezetéig. A gyógyszerkutatók nagyon szeretnék, ha bizonyos betegségekhez a számítógépeiken tudnának gyógyszereket tervezni, mintsem hogy patkányok tömegét kelljen feláldozni ebből a célból. A repülőgépek tervezői üzemanyag-takarékosabb termékekkel tudnának előrukkolni, ha a számítógépek mindent elvégeznének helyettük, és nem kellene szélcsatorna vizsgálatokhoz a prototípusokat megépíteniük. Röviden szólva, akármekkora is a rendelkezésre álló számítási kapacitás, főleg a kutatók, a mérnökök és az ipari felhasználók között sokan vannak, akiknek sohasem elég.

Bár a gépek órajel-frekvenciája folyamatosan nő, az áramkörök sebességét azonban nem lehet a végtelenségig növelni. A legnagyobb teljesítményű számítógépek tervezői számára a fénysebesség már most is nagy probléma, és nincs túl sok remény arra, hogy az elektronok és a fotonok gyorsabb mozgásra bírhatók. A hőelvezetési megoldások a szuperszámítógépeket a legmodernebb légkondicionáló berendezésekké is teszik egyben. Végül, mivel a tranzisztorok mérete folyamatosan csökken, elérhetnek egy olyan kicsi méretet, amikor már olyan kevés atomból állnak, hogy a kvantummechanikai hatások (például a Heisenberg-féle bizonytalansági elv) válhatnak problémává.

Éppen e megoldásra váró egyre nagyobb és nagyobb problémák miatt fordulnak a számítógép-tervezők a párhuzamos számítógépek felé. Míg egyáltalán nem biztos, hogy megépíthető egyetlen CPU-val és 0,001 ns ciklusidővel rendelkező számítógép, annál elképzelhetőbb egy olyan, amiben 1000 darab CPU van egyenként 1 ns ciklusidővel. Noha az utóbbiban a CPU-k lassabbak, mint az előzőben, mégis a teljes számítási teljesítményük elméletileg azonos. Ez ad alapot a reményre.

Párhuzamosítás különböző szinteken vezethető be. A legalsó szinten a CPU-n belül csővezeték vagy szuperskaláris architektúra és több funkcionális egység alkalmazható. Elképzelhető az is, hogy nagyon hosszú utasításszavakat használunk implicit párhuzamossággal kiegészítve. A CPU-t felvértezhetjük speciális funkciókkal annak érdekében, hogy több végrehajtási szálat tudjon kezelni egyszerre. Végül, több CPU-t is elhelyezhetünk egy lapkán. Ezen módszerek együttes alkalmazásával a tisztán szekvenciális tervezésű gépek teljesítményének talán 10-szerese is elérhető.

A következő szinten kiegészítő CPU-k illeszthetők a rendszerbe, növelve a számítási kapacitást. Az ilyen CPU-k általában speciális feladatot látnak el, mint például hálózati csomagfeldolgozást, multimédia-feldolgozást vagy titkosítási funkciókat. Speciális alkalmazások esetén ezek további 5-10-szeres gyorsulást eredményezhetnek.

Ahhoz azonban, hogy a teljesítmény 100, 1000 vagy 1 000 000-szorosára nőjön, sok CPU-t kell felhasználni, és ezeket alkalmassá kell tenni a hatékony együttműködésre. Ez az ötlet vezet a nagy multiprocesszoros gépekhez és a multiszámítógépekhez (klaszter számítógépek). Mondanunk sem kell, hogy több ezer processzor összekapcsolása egy rendszerbe maga is felvet problémákat, amelyeket meg kell oldani.

Végül, ma már lehetséges, hogy teljes szervezeteket kössünk össze az internet segítségével, amelyek így nagyon lazán kapcsolódó számítási hálózatokat alkotnak. Ezek a rendszerek még csak most kezdenek feltűnni, de érdekes lehetőséget rejtenek magukban a jövőre nézve.

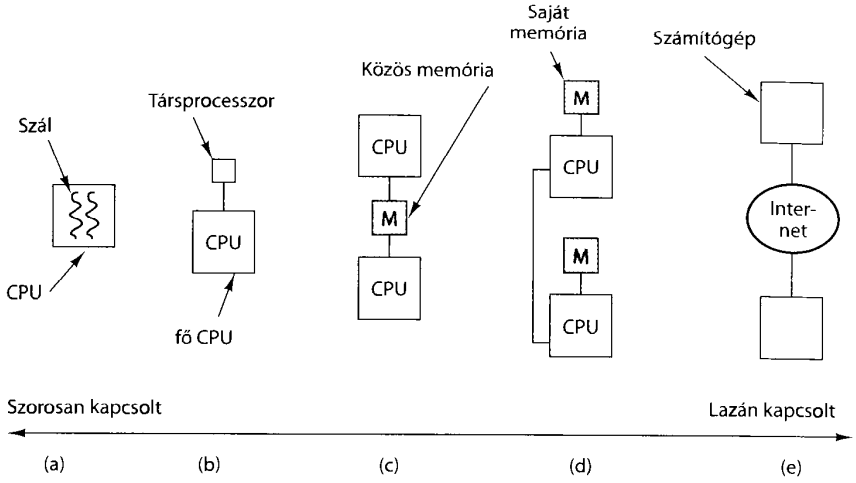
Ha két CPU vagy feldolgozó elem közel van egymáshoz, nagy átvivő képességű és kis késleltetésű kapcsolat van közöttük, és ugyanazon a feladaton dolgoznak, akkor **szorosan kapcsolt**nak nevezzük. Ezzel ellentétben, ha távol vannak egymástól, kis áteresztő képességű és nagy késleltetésű kapcsolatuk van, és számítási feladataiknak kevés közük van egymáshoz, akkor **lazán kapcsolt**nak nevezzük.

Ebben a fejezetben a párhuzamosság különböző fajtáihoz tartozó tervezési alapelveket fogjuk megvizsgálni és tanulmányozzuk a legkülönfélébb példákat. A legszorosabban kapcsolt rendszerekkel kezdjük, azokkal, amelyek lapkaszintű párhuzamosságot alkalmaznak, és fokozatosan haladunk egyre lazábban kapcsolt rendszerek felé, hogy aztán a grid rendszerekről ejtett néhány szóval fejezzük be. Ez a spektrum nagyjából a 8.1. ábrán látható.

A párhuzamosság kérdése a spektrum egyik végétől a másikig a kutatások középpontjában van. Ennek megfelelően ebben a fejezetben sok hivatkozást adunk meg, főleg a témáról szóló friss cikkekre. A bevezetőbb jellegű hivatkozások a 9.1.8-as alfejezetben találhatók.

8.1. Lapkaszintű párhuzamosság

Egy lapka áteresztőképességét például úgy növelhetjük, hogy egyszerre több dolgot végeztetünk el vele. Ebben a fejezetben annak járunk utána, hogy miként lehet növelni a sebességet lapkaszintű párhuzamosság alkalmazásával, többek között utasításszintű párhuzamossággal, többszálú végrehajtással, illetve több CPU egy lapkára helyezésével. Ezek a módszerek egészen különbözők, de mindegyiknek van előnye. Minden esetben az alapötlet az, hogy több tevékenységet végezzünk egyszerre.



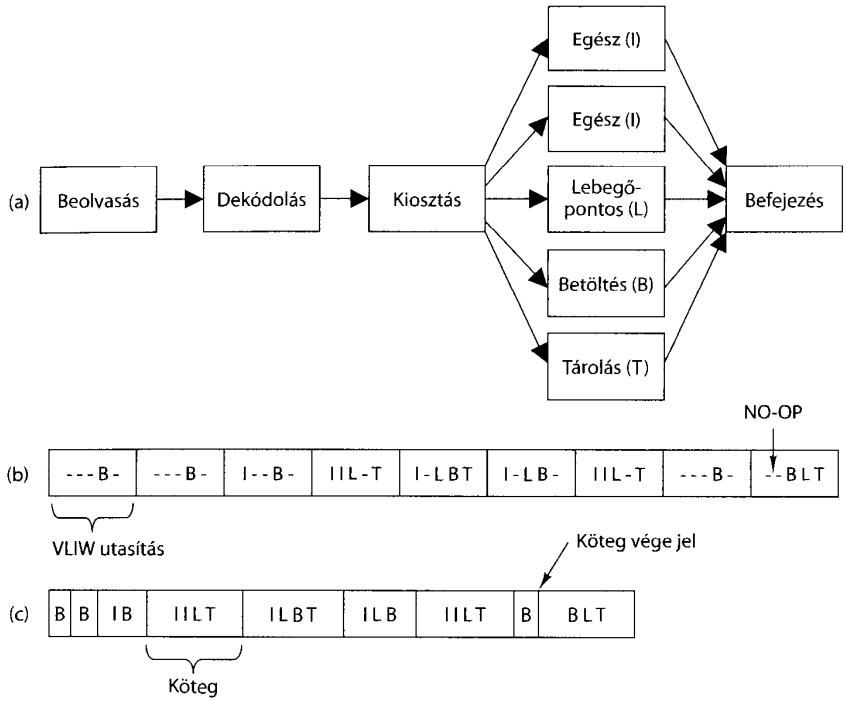
8.1. ábra. *(a) Lapkaszintű párhuzamosság. (b) Társprocesszor. (c) Multiprocesszor. (d) Multiszámítógép. (e) Grid*

8.1.1. Utasításszintű párhuzamosság

A legalsó szinten egy párhuzamosítási lehetőség, hogy óraciklusonként több utasítást kezdünk el végrehajtani. Az ilyen CPU-knak két változata létezik: szuperskaláris processzorok és VLIW processzorok. Mindkettőt érintettük korábban a könyvben, de hasznos lehet, ha röviden újra áttekintjük ezeket.

Korábban láttunk már szuperskaláris processzorokat (lásd 2.5. ábra). A leggyakoribb konfigurációban a csővezeték egy bizonyos pontján egy utasítás készen áll a végrehajtásra. A szuperskaláris CPU-k képesek több utasítást kiadni a végrehajtó egységeknek minden óraciklusban. Egy adott helyzetben kiadott utasítások száma függ a processzor felépítésétől és a konkrét körülményektől is. A hardver a kiadott utasítások maximális számát határozza meg, ez általában kettő és hat között van. Előfordulhat azonban, hogy egy utasításnak olyan funkcionális egységre van szüksége, amely éppen nem áll rendelkezésre, vagy olyan adatra, amely még nincs kiszámítva. Ezekben az esetekben az utasítás végrehajtása nem kezdődik el.

A másik fajta utasításszintű párhuzamosságot a **VLIW (Very Long Instruction Word, nagyon hosszú utasításszavú)** processzorokban találjuk. Eredeti formájukban a VLIW gépek valóban hosszú szavakat használtak, ezek mindegyike több funkcionális egységet igénybe vevő utasításokat tartalmazott. Tekintsük például a 8.2. (a) ábrán látható csővezetéket, ahol a gépnek 5 funkcionális egysége van, két egész, egy lebegőpontos, egy betöltő és egy tároló utasítást tud elvégezni egyszerre. Ennek a gépnek egy VLIW utasítása öt műveleti kódot és öt operanduspárt tartalmaz, egy műveleti kódot és egy operanduspárt funkcionális egységenként. Műveleti kódon-



8.2. ábra. *(a) Egy CPU csővezeték. (b) Egy VLIW utasítássorozat. (c) Egy utasítássorozat kötegjelzésekkel*

ként 6 bitet, regiszter operandusonként 5 bitet, valamint memóriahivatkozásonként 32 bitet feltételezve egy utasítás 134 bit hosszú lehet – ez tényleg elég hosszú.

Ez a felépítés azonban túl merevnek bizonyult, mert nem minden utasítás tudta kihasználni az összes funkcionális egységet, ez pedig sok helykitöltő NO-OP (no operation) beillesztését tette szükségessé, ahogy a 8.2. (b) ábrán is látható. Emiatt a modern VLIW gépek képesek arra, hogy egymás után következő utasításokat összetartozó kötegként jelöljenek meg, például egy „köteg vége" bittel, ahogy a 8.2. (c) ábrán látható. A processzor aztán beolvashatja az egész köteget, és egyszerre indíthatja az utasításokat. A kompatibilis utasítások kötegeinek előkészítése a fordítóprogram feladata.

Az egyszerre végrehajtható utasítások meghatározásának feladatát a VLIW módszer tulajdonképpen végrehajtási idő helyett fordítási időre tolja el. Ez nem csak a hardvert teszi egyszerűbbé és gyorsabbá, de mivel egy optimalizáló fordítóprogram szükség esetén akár nagyon hosszú ideig is futhat, jobb csoportok alakíthatók ki ezzel a módszerrel, mint amiket a hardver végrehajtási időben össze tudna állítani. Természetesen a CPU architektúra ilyen radikális megváltoztatása nehezen lesz keresztülvihető, ahogy azt az Itanium térhódításának lassúsága is jelzi.

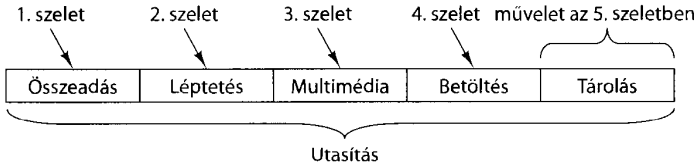
Érdemes röviden megjegyeznünk, hogy az utasításszintű párhuzamosság nem az alacsony szintű párhuzamosság egyetlen formája. Egy másik a memóriaszintű párhuzamosság, amikor több memóriaművelet áll végrehajtás alatt egyszerre (Chou és társai, 2004).

A TriMedia VLIW CPU

Az 5. fejezetben szerepelt egy VLIW CPU, az Itanium-2. Vizsgáljunk meg most egy attól teljesen eltérő VLIW processzort. Ez a **TriMedia**, amelyet a Philips tervezett, ugyanaz a holland elektronikai cég, amelynek nevéhez a zenei CD és a CD-ROM feltalálása is fűződik. A TriMedia arra a célra készült, hogy beágyazott processzorként használják olyan alkalmazásokban, amelyekben intenzíven használnak grafikát, hangot és mozgóképet. Ilyenek találhatók a CD-, DVD- és MP3-lejátszókban, CD- és DVD-írókban, interaktív tv-készülékekben, digitális fényképezőgépekben, kamkorderekben és egyéb eszközökben. Ezeket a felhasználási területeket tekintve nem meglepő, hogy felépítését tekintve lényegesen más, mint az Itanium-2, amely egy nagy teljesítményű szerverekbe szánt általános célú CPU.

A TriMedia egy igazi VLIW processzor, minden utasítása nem kevesebb, mint öt **művelet**et tartalmazhat. Teljesen optimális feltételek esetén, minden óraciklusban egy utasítás végrehajtása elkezdődik, vagyis mind az öt művelete kiadásra kerül. Az óra frekvenciája 266 MHz vagy 300 MHz, de az effektív sebessége ennél akár ötször nagyobb lehet, mivel ciklusonként öt műveletet képes kiadni. A következőkben a TriMedia TM3260 verziót tárgyaljuk, más verziók ettől kismértékben eltérnek.

A 8.3. ábrán egy tipikus utasítás látható. A skála 8, 16, és 32 bites egész műveletektől kezdve, IEEE 754 szabvány szerinti lebegőpontos műveleteken át párhuzamos multimédia-utasításokig terjed. Az óraciklusonkénti öt műveletnek és a párhuzamos multimédia-műveleteknek köszönhetően a TriMedia elég gyors ahhoz, hogy kamkorderből érkező teljes méretű teljes sebességű digitális videót szoftveresen dekódoljon.



8.3. ábra. *Egy tipikus TriMedia-utasítás, öt lehetséges műveletet bemutatva*

A TriMedia memóriája bájtszervezésű, a B/K regiszterek a memória címtartományába vannak ágyazva. A (16 bites) félszavak és a (32 bites) egész szavak csak a nekik megfelelő címekre igazítva helyezkedhetnek el. Egy az operációs rendszer által beállítható PSW bitnek megfelelően kis endián és nagy endián módban is működhet. Ez a bit csak a memóriából betöltő és az oda tároló műveletekre van

hatással. A CPU tartalmaz egy szétválasztott nyolcutas halmazkezelésű gyorsítótárat, 64 bájtos sorméretttel mind az utasítások, mind az adatok számára. Az utasítások gyorsítótára 64 KB, az adatoké 16 KB.

128 általános célú 32 bites regisztere van. Az R0 regiszter tartalma mindig 0, az R1 regiszter tartalma mindig 1. Bármelyik megváltoztatására irányuló kísérlet CPU szívrohamot okoz. A fennmaradó 126 regiszter teljesen egyenrangú és bármilyen célra felhasználható. Van még négy speciális célú 32 bites regiszter. Ezek a programszámláló (utasításszámláló), programállapotszó, valamint két megszakításokkal kapcsolatos regiszter. Végül, egy 64 bites regiszter számlálja az utolsó indítás óta eltelt óraciklusokat. 300 MHz-es frekvencia esetén közel 2000 évig tart, míg a számláló körbeér.

A TriMedia TM3260-nak 11 funkcionális egysége van az aritmetikai, logikai és vezérlési műveletek elvégzésére (van egy a gyorsítótár kezeléséhez is, ezt nem fogjuk tárgyalni); ezek a 8.4. ábrán láthatók. Az első két oszlop az egység nevét és rövid leírását tartalmazza. A harmadik oszlop adja meg, hogy hány hardverpéldány van belőle. A negyedik oszlop a késleltetést tartalmazza, vagyis azt, hogy hány óraciklust igényel, amíg elvégzi a feladatát. Ebben az összefüggésben érdemes megjegyezni, hogy a lebegőpontos gyökvonást és osztást végző egység kivételével mindegyik csővezetékes. A táblázatban megadott késleltetés azt adja meg, hogy hány ciklusra van szükség a művelet eredményének meghatározásához, de minden ciklusban ki lehet adni végrehajtásra egy új műveletet. Így például három egymás után következő utasítás mindegyike tartalmazhat két betöltő műveletet, ami hat egyidejű, a végrehajtás különböző fázisában lévő betöltést eredményez.

Végül az utolsó öt oszlopból kiolvasható, hogy az utasításszó mely pozícióján (melyik utasításszeletben) állhatnak az adott funkcionális egységet használó műveletek. Például lebegőpontos összehasonlító művelet csak a harmadik szeletben fordulhat elő.

Egység	Leírás	Db	Késleltetés	1	2	3	4	5
Konstans	Közvetlen műveletek	5	1	x	x	x	x	x
Egész ALU	32 bites aritmetika, Boole-műveletek	5	1	x	x	x	x	x
Léptető	Léptetés egy vagy több bittel	2	1	x	x	x	x	x
Betöltő/tároló	Memóriaműveletek	2	3				x	x
Egész/FP szorzó	32 bites egész és lebegőpontos szorzások	2	3		x	x		
FP ALU	Lebegőpontos aritmetika	2	3	x			x	
FP összehasonlító	Lebegőpontos összehasonlítás	1	1			x		
FP gyökvonó/osztó	Lebegőpontos gyökvonás és osztás	1	17		x			
Elágazó	Vezérlésátadás	3	3		x	x	x	
DSP ALU	2 × 16 bites, 4 × 8 bites multimédia-aritmetika	2	3	x		x		x
DSP MUL	2 × 16 bites, 4 × 8 bites multimédia-szorzások	2	3		x	x		

8.4. ábra. *A TM3260 funkcionális egységei, ezek darabszáma, késleltetése, valamint az általuk használható utasításszeletek*

A konstans egység a közvetlen címzésű műveletekhez használatos, mint amilyen az utasítás részeként tárolt szám betöltése egy regiszterbe. Az egész ALU összeadást, kivonást, a szokásos Boole-féle műveleteket és becsomagoló/kicsomagoló műveleteket végez. A léptető egység a megadott számú bittel való léptetést végez valamely regiszteren.

A betöltő/tároló egység memóriaszavakat olvas regiszterbe és ír vissza onnan. A TriMedia lényegében egy kibővített RISC CPU, így a normál műveletek regisztereken hajtódnak végre, a memória kezelését a betöltő/tároló egység végzi. Az átvitt adatok 8, 16 vagy 32 bitesek lehetnek. Az aritmetikai és logikai műveletek nem érik el a memóriát.

A szorzó egység (egész/FP szorzó) végzi mind az egész, mind a lebegőpontos szorzást. A következő három egység kezeli sorrendben a lebegőpontos összeadást/kivonást, összehasonlítást, gyökvonást és osztást.

A vezérlésátadásokat az elágazó egység kezeli. Minden ugrás után van egy 3 ciklust igénylő fix hosszúságú késleltetési rés, így az ugrás utáni 3 utasítás (legfeljebb 15 művelet) mindig végrehajtásra kerül még feltétel nélküli ugrás esetén is.

Végül elérkeztünk a két multimédia-egységhez, amelyek a speciális multimédia-műveleteket végzik. A funkcionális egységek nevében a DSP a Digital Signal Processor (digitális jelfeldolgozó) kezdőbetűiből áll össze, amelyet ezek az egységek felváltani hivatottak. A következőkben röviden leírjuk a multimédia-műveleteket. Ezek egyik kiemelendő tulajdonsága, hogy **telített módú aritmetikát** használnak, ellentétben az egész műveletek kettes komplemens aritmetikájával. Amikor egy művelet túlcsordulás miatt az ábrázolási tartományon kívüli számot eredményezne, akkor ahelyett, hogy kivételt generálna vagy valamilyen „szemetet" hagyna a regiszterben, a legközelebbi érvényes számot adja. Például 8 bites előjel nélküli számok esetén 130 + 130 eredménye 255.

Mivel nem minden művelet szerepelhet minden utasításszeletben, gyakran előfordul, hogy egy utasítás a lehetséges ötnél kevesebb műveletet tartalmaz. Amikor egy szelet kihasználatlan, akkor a veszendőbe menő hely minimalizálása érdekében tömörítés történik. A kihasznált szeletekben lévő műveletek 26, 34 vagy 42 bitet foglalnak el. A műveletek számától függően a TriMedia-utasítások hossza – egy fix többletet is beleszámítva – 2 és 28 bájt között lehet.

A TriMedia nem ellenőrzi futás közben, hogy az utasításban lévő műveletek kompatibilisek-e. Akkor is végrehajtja, ha nem azok, és hibás eredményt ad. Az ellenőrzéseket szándékosan hagyták ki idő- és tranzisztortakarékossági okokból. A Pentium ellenőrzi, hogy a szuperskaláris utasítások kompatibilisek-e, de ennek hatalmas a költsége összetettségben, időben és tranzisztorban. A TriMedia ezeket a költségeket megtakarítja azáltal, hogy az ütemezést áthárítja a fordítóprogramra, amelynek annyi ideje van a műveleteket optimálisan utasításszavakba rendezni, amennyit csak akar. Másrészről viszont, ha egy műveletnek egy éppen foglalt funkcionális egységre lenne szüksége, akkor az utasítás várakozni fog az egység szabaddá válásáig.

Ugyanúgy, mint az Itanium-2, a TriMedia is predikátumos műveleteket alkalmaz. (Két apró kivételtől eltekintve) minden művelet megnevez egy regisztert,

amelyet ellenőrizni kell a művelet végrehajtása előtt. Ha a regiszter legkisebb helyértékű bitje be van állítva, akkor a művelet végrehajtásra kerül, különben kimarad. Az utasítás minden művelete (legfeljebb öt) egymástól függetlenül predikátumozható. Predikátumos műveletre példa az

```
IF R2 IADD R4, R5 -> R8
```

amely ellenőrzi az R2 regisztert, majd, ha a legkisebb helyértékű bit 1, akkor összeadja R4 és R5 tartalmát, és az eredményt R8-ban tárolja. Egy művelet feltétel nélkülivé tehető azáltal, hogy predikátumregiszterként R1-re hivatkozik (amelynek tartalma mindig 1). R0 (értéke mindig 0) üres műveletté tesz mindent.

A TriMedia multimédia-műveletek a 8.5. ábrán látható 15 csoportba sorolhatók. Sok művelet tartalmaz vágást, ami azt jelenti, hogy fogad egy operandust és egy tartományt, majd a tartományba kényszeríti az operandust azáltal, hogy a megengedett legkisebb vagy legnagyobb értékkel helyettesíti. Vágást lehet végezni 8, 16, vagy 32 bites operandusokon. Például ha a 0–255 tartománnyal vágást végzünk 40-en és 340-en, akkor az eredmény 40 és 255 lesz.

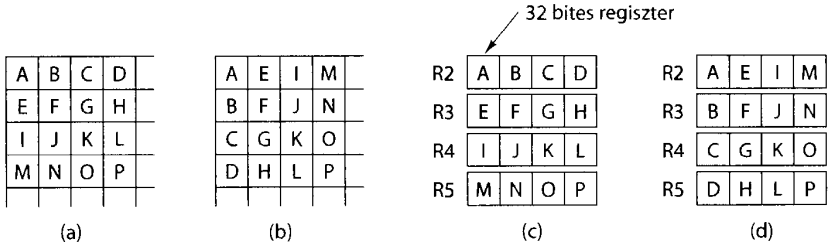
Csoport	Leírás
Vágás	4 bájtos vagy 2 félszavas vágás
DSP abszolút érték	Előjeles érték abszolút értéke, vágás
DSP összeadás	Előjeles összeadás, vágás
DSP kivonás	Előjeles kivonás, vágás
DSP szorzás	Előjeles szorzás, vágás
Min, max	Négy bájtpár minimuma vagy maximuma
Összehasonlítás	Két regiszter bájtonkénti összehasonlítása
Léptetés	16 bites operanduspárok léptetése
Szorzatok összege	8 vagy 16 bites szorzatok előjeles összege
Összefésülés, csomagolás, csere	Bájtok és félszavak kezelése
Bájtnégyes átlagok	Előjel nélküli bájtnégyes átlaga
Bájtátlagok	4 előjel nélküli bájt átlaga
Bájtszorzás	Előjel nélküli 8 bites szorzás
Mozgásbecslés	8 bites előjeles különbségek abszolút értékének összege
Egyéb	Egyéb aritmetikai műveletek

8.5. ábra. *A TriMedia speciális utasításainak nagyobb csoportjai*

A vágás csoport vágási műveleteket végez. A 8.5. ábra következő négy csoportja a megadott műveleteket végzi különféle méretű operandusokon, az eredményt egy bizonyos tartományra vágja. A min, max csoport megvizsgál két regisztert, és minden bájtra megkeresi a legkisebb, illetve a legnagyobb értéket. Hasonlóan, az összehasonlítás csoport két regisztert 4 bájtpárnak értelmezve minden párt összehasonlít.

A multimédia-műveleteket ritkán kell 32 bites egészeken elvégezni, mert a legtöbb kép a vörös, zöld és kék színösszetevőknek megfelelő 8 bites RGB értékekből áll. Amikor egy kép feldolgozásra (például tömörítésre) kerül, akkor legtöbbször három komponenssel reprezentálják, vagy színenként (RGB kódolásnál), vagy valamilyen logikailag ekvivalens formában (YUV kódolás, később szó lesz róla ebben a fejezetben). Bárhogy is történik, rengeteg számítást kell végezni 8 bites előjel nélküli számokat tartalmazó mátrixokon.

A TriMedia sok olyan műveletet tartalmaz, amelyeket speciálisan 8 bites előjel nélküli számokat tartalmazó mátrixok hatékony feldolgozására terveztek. Egy egyszerű példaként a 8.6. (a) ábrán tekintsük a (nagy endián) memóriában tárolt 8 bites értékek egy tömbjének bal felső sarkát. A sarokban lévő 4 × 4-es blokk 16 darab 8 bites értéket tartalmaz, ezeket *A*-tól *P*-ig betűkkel jelöltük meg. Tegyük fel például, hogy a képet transzponálni kell, hogy a 8.6. (b) ábrán látható elrendezést kapjuk. Hogyan tudjuk ezt a feladatot megoldani?



8.6. ábra. *(a) 8 bites értékek egy mátrixa. (b) A transzponált mátrix. (c) Az eredeti mátrix négy regiszterbe töltve. (d) A transzponált mátrix a négy regiszterben*

A transzponálás végrehajtásának egyik módja, ha 12 olyan műveletet használunk, ahol mindegyik egy bájtot tölt egymástól különböző regiszterekbe, majd 12 másik műveletet, amelyek a bájtokat a memória megfelelő helyére teszik. (Figyeljük meg, hogy a diagonális mentén elhelyezkedő négy bájt nem mozdul.) Ezzel a megközelítéssel az a probléma, hogy 24 (hosszú és lassú) memóriaműveletet igényel.

A 8.6. (c) ábrán szemléltetett alternatív megközelítés lehet, hogy kiindulásként négy művelettel egy-egy szót töltünk az R2-től R5-ig terjedő négy regiszterbe. Ezután a négy eredmény szót megkaphatjuk maszkoló és léptető műveletekkel, ahogy a 8.6. (d) ábrán látható. Végül ezeket a szavakat visszaírjuk a memóriába. Habár ez a módszer a szükséges memóriaműveleteket 24-ről 8-ra csökkentette, a maszkolás és léptetés költséges, mert az egyes bájtokat elég sok művelettel lehet csak kinyerni, és a helyükre illeszteni.

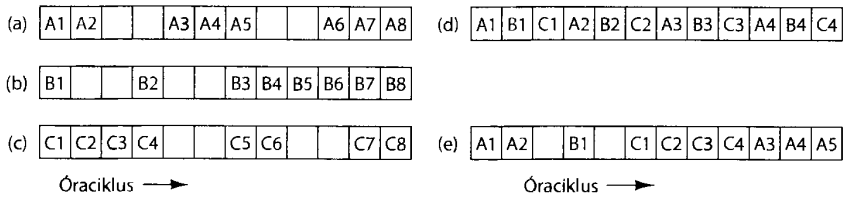
A TriMedia az előző kettőnél is jobb megoldást tesz lehetővé. Kezdetben betölti a négy szót négy regiszterbe. Az eredmény kiszámításához azonban maszkolás és léptetés helyett speciális műveleteket használ a bájtok kinyerésére és beillesztésére. Mindezek miatt összesen 8 memóriahivatkozás és 8 speciális multimédia-művelettel elvégezhető a transzponálás. A programkód egy olyan utasítással kezdődik, amelyik a 4. és az 5. szeletében az R2 és R3 regisztert betöltő műveleteket tar-

talmaz. Ezt követi egy hasonló az R4 és R5 betöltésére. Ez a két utasítás az 1., 2. és 3. szeletet más célra felhasználhatja. A szavak betöltése után a nyolc multimédia-művelet elhelyezhető két utasításban, majd a 4 tárolás újabb kettőben. Mindent összevetve csak hat utasításra van szükség, és a 30 szeletből 14 más műveletek számára felhasználható. Ez lényegében azt jelenti, hogy a feladat nagyjából 3 utasításnak megfelelő programkóddal megoldható. A többi multimédia-műveletek hasonlóan hatékonyak. Az erős műveleteknek és az ötszeletes utasításoknak köszönhetően a TriMedia rendkívül hatékonyan oldja meg a multimédia-alkalmazásokban felmerülő számítási feladatokat.

8.1.2. Lapkaszintű többszálúság

Minden modern csővezetékes CPU-ra jellemző probléma, hogy amikor olyan memóriacímre történik hivatkozás, amelyik sem az első, sem a második szintű gyorsítótárban nincs benne, akkor sok idő telik el a kért szó (és a hozzá tartozó gyorsítósor) betöltődéséig, így a csővezeték elakad. Ennek a helyzetnek egyik lehetséges kezelési módja az ún. **lapkaszintű többszálúság**, amely a CPU számára lehetővé teszi, hogy az elakadásokat megpróbálja elfedni több végrehajtási szál egyidejű kezelésével. Röviden arról van szó, hogy ha egy szál blokkolódik, akkor a CPU a hardver teljesen kihasználtsága érdekében még mindig futtathat egy másikat.

Jóllehet az alapötlet meglehetősen egyszerű, több változata is létezik, amelyeket most megvizsgálunk. Az első megközelítés neve **finom szemcsézettségű többszálúság**, amelyet a 8.7. ábrán szemléltetünk egy olyan CPU-ra, amely ciklusonként egy utasítást képes kiadni. A 8.7. (a)–(c) ábrákon az *A*, *B* és *C* szálakat láthatjuk 12 gépi ciklus idejére. Az első ciklus alatt az *A* szál az *A1* utasítást hajtja végre. Ez egy ciklus alatt be is fejeződik, ezért a második ciklusban az *A2* kezdődik. Sajnos ez az utasítás nem találja az adatát az elsőszintű gyorsítótárban, ezért két ciklus veszendőbe megy, amíg az igényelt szó megérkezik a második szintű gyorsítótárból. A szál az ötödik ciklusban folytatódik. Az ábrán látható, hogy hasonlóképpen a *B* és *C* szál is megakad időnként. Ebben a modellben egy megakadt utasítás után következők nem adhatók ki. Természetesen összetettebb megoldásokkal néha mégis kiadhatók új utasítások, de ezt a lehetőséget most figyelmen kívül hagyjuk.



8.7. ábra. *(a)–(c) Három szál. Az üres négyzetek azt jelzik, hogy a szál a memóriára várva megakadt. (d) Finom szemcsézettségű többszálúság. (e) Durva szemcsézettségű többszálúság*

A finom szemcsézettségű többszálúság úgy fedi el az elakadásokat, hogy a szálakat körben egymás után futtatja, minden ciklusban másik szál utasítását veszi elő, ahogy az a 8.7. (c) ábrán látható. Az *A1* által kezdeményezett memóriaművelet befejeződik mire a negyedik ciklushoz érünk, tehát az *A2* utasítás még akkor is végrehajtható, ha szüksége van az *A1* eredményére. Ebben az esetben a maximális elakadás két ciklushossznyi, ezért három szállal az elakadt művelet mindig idejében befejeződik. Ha egy memóriára várakozás négy ciklust venne igénybe, akkor a folyamatos működéshez öt szálra lenne szükség és így tovább.

Mivel a szálaknak semmi közük nincs egymáshoz, mindegyiknek saját regiszterkészlet kell. Amikor egy utasítás kiadásra kerül, a regiszterkészletét azonosító mutatót is mellékelni kell, hogy a hardver tudja melyik regiszterkészlethez kell nyúlnia, ha valamelyik regiszterre hivatkozna. Ezért az egyszerre futtatható szálak maximális száma a tervezéskor eldől.

Nemcsak a memóriaműveletek miatt vannak elakadások. Néha egy utasításnak szüksége van egy korábbi utasítás eredményére, amely még nem áll rendelkezésre. Néha egy utasítás azért nem kezdődhet el, mert olyan feltételes elágazást követ, amelynek a kimenetele még kétséges. Általánosságban, ha egy csővezetéknek *k* fázisa van, de van legalább *k* darab, körben egymás után futtatható szál, akkor sosem lesz a csővezetékben szálanként egynél több utasítás, tehát konfliktus sem léphet fel. Ebben a helyzetben a CPU megakadás nélkül, teljes sebességgel működhet.

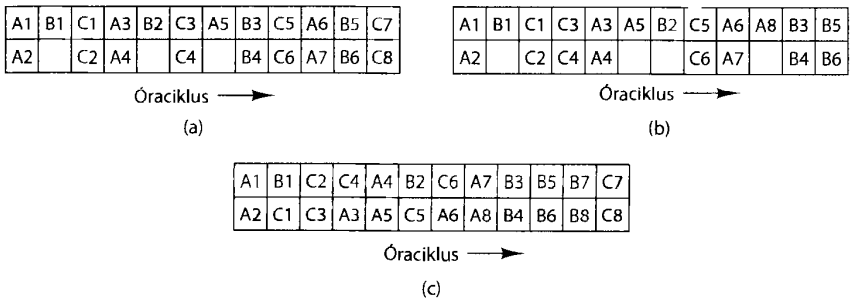
Természetesen nem biztos, hogy mindig rendelkezésre áll annyi szál, ahány fázisa van a csővezetéknek. Ezért a tervezők egy része egy másik, **durva szemcsézettségű többszálúság**ként ismert és a 8.7. (e) ábrán bemutatott megközelítést részesíti előnyben. Itt az *A* szál elindul, és mindaddig fut, amíg el nem akad, ezzel elveszítve egy ciklust. Ennél a pontnál váltás történik, és *B1* kerül végrehajtásra. Mivel *B* első utasítása elakad, újabb szálváltás történik és *C1* hajtódik végre a hatodik ciklusban. Mivel utasítás elakadásakor egy ciklust elveszítünk, megvan annak a lehetősége, hogy a durva szemcsézettségű többszálúság kevésbé hatékony legyen, mint a finom szemcsézettségű többszálúság, de megvan az a nagy előnye is, hogy jóval kevesebb szál kell a CPU kihasználásához. A durva szemcsézettségű többszálúság jobb olyan helyzetekben, amikor nincs elegendő futtatható szál.

Habár a durva szemcsézettségű többszálúságot úgy írtuk le, mint amelyik csak elakadás esetén vált a szálak között, ez nem az egyetlen lehetőség. Egy másik módszer, hogy azonnal váltunk olyan utasítások esetén, amelyek elakadást idézhetnek elő, mint például a betöltés, a tárolás vagy az elágazás, még mielőtt kiderülne, hogy valóban elakadást idéznének-e elő. Ez utóbbi módszer korábbi (az utasítás dekódolása utáni) váltásokat tesz lehetővé, és elkerülheti a kihasználatlan ciklusokat. Tulajdonképpen azt mondja, hogy „fusson addig a szál, amíg tudjuk, hogy nem lesz probléma, egyébként váltsunk, biztos, ami biztos." Utóbbi módosítással a durva szemcsézettségű többszálúság hasonlóbbá válik a gyakoribb váltásokkal operáló finom szemcsézettségűhöz.

Függetlenül attól, hogy milyen többszálúságot alkalmazunk, valamilyen módon nyilván kell tartanunk, hogy melyik utasítás melyik szálhoz tartozik. Finom szemcsézettség esetén az egyetlen komoly lehetőség az, hogy egy szálazonosítót rendelünk minden utasításhoz, mert így a csővezetéken való áthaladása közben mindig

egyértelmű, hogy melyik szálhoz tartozik. Durva szemcsézettség esetén van egy másik lehetőség is: szálváltáskor először hagyjuk kiürülni a csővezetéket, és csak ezután kezdjük el az új szálat. Ilyen módon a csővezetékben mindig csak egy szál van és egyértelmű, hogy melyik. Természetesen szálváltáskor csak akkor van értelme hagyni, hogy a csővezeték kiürüljön, ha a szálváltások csak a csővezeték kiürülési idejénél jóval nagyobb időközönként következnek be.

Eddig feltételeztük, hogy a CPU ciklusonként csak egy utasítást tud kiadni Korábban azonban láttuk, hogy a modern CPU-k több utasítást is képesek kiadni. A 8.8. ábrán feltételezzük, hogy a CPU két utasítást tud kiadni óraciklusonként, de fenntartjuk, hogy ha egy utasítás megakad, akkor továbbiak nem indulhatnak. A 8.8. (a) ábrán azt látjuk, hogy hogyan működik a finom szemcsézettségű többszálúság olyan szuperskaláris CPU esetén, amely két utasítást indíthat egyszerre. Az *A* szál első két utasítása az első ciklusban kiadható, de a *B* szál a következő ciklusban azonnal akadályba ütközik, ezért csak egy utasítás indulhat, és így tovább.



8.8. ábra. *Többszálúság egyszerre két utasítást indítani képes szuperskaláris CPU esetén. (a) Finom szemcsézettségű többszálúság. (b) Durva szemcsézettségű többszálúság. (c) Egyidejű többszálúság*

A 8.8. (b) ábrán azt látjuk, hogy a durva szemcsézettségű többszálúság hogyan működik olyan CPU esetén, amely két utasítást indíthat egyszerre, de most egy olyan statikus ütemezővel, amely nem enged üres ciklust elakadt utasítás után. Alapvetően az történik, hogy a szálak egymás után futnak, a CPU az aktív szál két utasítását indítja minden ciklusban addig, amíg valamelyik el nem akad. Ekkor a következő ciklus elején átvált a következő szálra.

Szuperskaláris CPU-k esetén a többszálúság kezelésére van egy harmadik lehetőség is, amelyet **egyidejű többszálúság**nak nevezünk, és a 8.8. (c) ábrán szemléltetünk. Ez a megközelítés a durva szemcsézettség finomításának tekinthető, amennyiben egy szál addig indíthat ciklusonként két utasítást, amíg erre képes, de ha megakad, akkor a CPU teljes kihasználtsága érdekében azonnal a következő szál utasításai következnek. Az egyidejű többszálúság a funkcionális egységek teljes kihasználása szempontjából is előnyös. Ha egy utasítás azért nem indítható, mert a szükséges funkcionális egység éppen foglalt, akkor választhatjuk helyette egy másik szál soron következő utasítását. Ezen az ábrán feltételezzük, hogy a *B* szál *B8* után megakad a 11. ciklusban, ezért a 12-ben *C7* indul.

A többszálúságról további információ találhatók a (Dean, 2004; Kalla és társai, 2004; Kapil és társai, 2004). A többszálúság és a spekulatív végrehajtás együttes alkalmazásáról olvashatunk (Sohi és Roth, 2001).

A Pentium 4 többszálúsága (hyperthreading)

Miután a többszálúságot elméletben áttekintettük, nézzünk most egy konkrét példát, a Pentium 4 CPU-t. A Pentium 4 gyártása már megkezdődött, amikor az Intel tervezőmérnökei lehetőségeket kerestek a sebesség növelésére úgy, hogy a programozói interfészt érintetlenül lehessen hagyni, hiszen annak megváltoztatását úgysem lehetett volna elfogadtatni. Öt lehetőség elég gyorsan adódott:

1. Az órajel frekvenciájának növelése;
2. Két CPU elhelyezése egy lapkán;
3. Funkcionális egységek hozzáadása;
4. A csővezeték hosszának növelése;
5. Többszálúság használata.

A teljesítmény növelésének nyilvánvaló módja a frekvencia növelése anélkül, hogy bármi máshoz hozzányúlnánk. Ez aránylag egyszerű és könnyű, így minden új lapka általában egy kicsit gyorsabb, mint az elődje. Sajnos azonban a nagyobb órajelnek két komoly hátránya is van, ezek pedig korlátozzák a gyorsulás mértékét. Egyrészt a gyorsabb óra több energiát használ, ami elég nagy probléma a hordozható számítógépek és más, akkumulátorról működő eszközök számára. Másrészt a nagyobb energiafelvétel miatt a lapka jobban melegszik, és több hőt kell elvezetni.

Két CPU elhelyezése egy lapkán aránylag könnyű, de ez közel kétszer akkora lapkaterületet kíván, ha mindkettőnek saját gyorsítótára van. Ekkor pedig felére csökken az egy szilikonostyából előállítható lapkák száma, vagyis lényegében megduplázódik az egy egységre vetített előállítási ár. Ha a két CPU-nak az eredetivel megegyező méretű gyorsítótára van, akkor a lapka mérete nem duplázódik, de az egy CPU-ra jutó gyorsítótárméret felére csökken a teljesítmény rovására. Ráadásul, míg a nagy teljesítményű szerverek teljesen ki tudnak használni két komplett CPU-t, az asztali számítógépek alkalmazásai közül nem mindegyikben van annyi párhuzamosság, ami két teljes CPU-t igényelne.

További funkcionális egységek hozzáadása is elég könnyű, de fontos az egyensúly. Nem sok haszna van 10 ALU-nak, ha nem tudjuk kihasználni, mert nem vagyunk képesek elég gyorsan utasításokat juttatni a csővezetékbe.

Szintén növeli a teljesítményt egy hosszabb csővezeték, amelyben minden fázis a rá eső kevesebb munkát rövidebb idő alatt tudja elvégezni. Felerősödnek azonban a negatív hatások is hibás elágazásjövendölés, gyorsítótárhiány, megszakítás és minden olyan egyéb esemény következtében, amelyek megzavarják a csőveze-

ték normális működését. Ráadásul a hosszabb csővezeték teljes kihasználásához növelni kell az órajel frekvenciáját is, ez pedig nagyobb energiafelvételt és ezzel együtt nagyobb hőtermelést jelent.

Végül, be lehet vezetni a többszálúságot. Előnye, hogy egy második szál kihasználhatja a hardvert olyankor, amikor az egyébként parlagon heverne. Némi kísérletezés után világossá vált, hogy a lapkaterület 5 százalékos növelése fejében a többszálúság sok alkalmazásnál 25 százalékos teljesítménynövekedést eredményezett, így ezt a megoldást választották. Az Intel első többszálú CPU-ja a Xeon volt 2002-ben, de aztán a többszálúságot a 3,06 GHz-es változattól kezdve a Pentium 4-nél is bevezették. A többszálúság Pentium 4-ben megvalósított formájára az Intel a **hyperthreading** elnevezést vezette be.

Az alapötlet az, hogy két szál (vagy folyamat, a CPU nem tudja megkülönböztetni ezeket) egyszerre fut. Az operációs rendszer felé egy hyperthreading Pentium 4 két olyan CPU-nak látszik, amelyek közös gyorsítótárat és memóriát használnak. Az operációs rendszer a szálakat egymástól függetlenül ütemezi. Ha két alkalmazás fut egyszerre, akkor az operációs rendszer ezeket egyszerre futtathatja. Például, ha egy levélkezelő háttérfolyamat éppen levelet küld vagy fogad, amíg a felhasználó valamilyen programot használ, akkor ezek párhuzamosan futhatnak, mintha két CPU lenne a gépben.

A többszálúra tervezett felhasználói programok is használhatják mindkét virtuális CPU-t. Például a videoszerkesztő programoknak rendszerint van olyan funkciója, amellyel a felhasználó megadhat bizonyos szűrőket, melyeket aztán a program egy tartomány minden képére alkalmaz. A szűrők módosíthatják a fényerőt, a kontrasztot, a színtelítettséget, vagy a képkockák más tulajdonságait. A program megtehcti, hogy az egyik CPU-t a páros, a másikat a páratlan sorszámú képkockákhoz rendeli, a kettő pedig egymástól teljesen függetlenül működhet.

Mivel a két szál osztozik az összes hardver erőforráson, a megosztáshoz szükség van valamilyen stratégiára. A hyperthreading kapcsán az Intel négy használható erőforrás-megosztási stratégiát definiált: erőforrás többszörözés, erőforrás-felosztás, küszöbölt megosztás és teljes megosztás. Most ezeket tekintjük át egyenként.

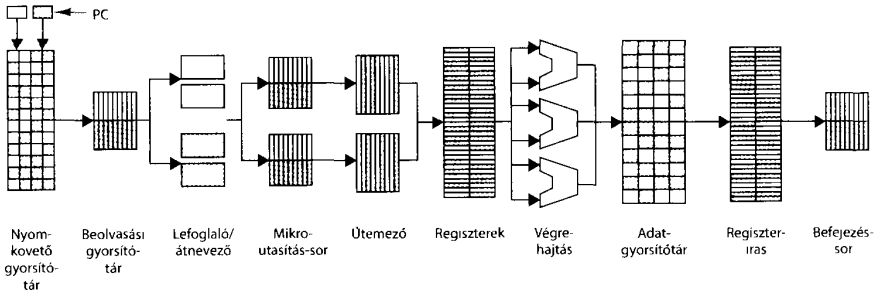
Azzal kezdjük, hogy néhány erőforrás éppen a többszálúság miatt létezik több példányban. Például egy második programszámlálót (utasításszámlálót) is be kell építeni, hiszen a két szálnak külön végrehajtandó utasítássorozata van. Ezen túlmenően az architekturális regisztereket (*EAX*, *EBX* stb.) a fizikai regiszterekhez rendelő megfeleltetési táblázatot is meg kell kettőzni, ahogy a megszakításvezélőt is, hiszen a két szál egymástól függetlenül megszakítható.

A következő módszer az **erőforrás-felosztás**, ami azt jelenti, hogy a hardvererőforrások megváltoztathatatlanul fel vannak osztva a szálak között. Például, ha két csővezetékfázis között a CPU-nak van egy várakozási sora, akkor a rendelkezésre álló helyek egyik fele kizárólagosan az egyik, a másik fele kizárólagosan a másik szálhoz rendelhető. A felosztást egyszerű megvalósítani, nem jár többletterheléssel, és a taszkok sem kerülnek egymás útjába. Ha minden erőforrást felosztunk, akkor tulajdonképpen két CPU-nk lesz. Hátrány lehet, hogy bizonyos helyzetben az egyik szál nem használja ki teljesen a rendelkezésére álló erőforrásokat, a máskor pedig szüksége lenne rá, de nem engedjük hozzáférni. Tehát olyan erőforrásokat hagyunk kihasználatlanul, amelyeket hasznosítani lehetett volna.

A felosztás ellentéte a **teljes erőforrás-megosztás**. Amikor ezt használjuk, akkor a szálak az igénylés sorrendjében hozzájuthatnak bármelyik számukra szükséges erőforráshoz. Képzeljünk el azonban egy főleg összeadásokból és kivonásokból álló gyors szálat, és egy főleg szorzásokból és osztásokból álló lassú szálat. Ha az utasítások beolvasása a memóriából gyorsabb, mint a szorzások és osztások elvégzése, akkor a lassú szál beolvasott, de még végre nem hajtott utasításai a várakozási sorban rekednek, mert nem kerülhetnek be a csővezetékbe. Végül a várakozási sor betelik ezekkel az utasításokkal, és hely hiányában leállásra kényszerül a gyors szál. Teljes erőforrás-megosztásnál nem fordulhat elő, hogy egy erőforrás kihasználatlan marad míg egy szálnak szüksége lenne rá. Viszont felmerül egy új probléma, nevezetesen az, hogy az egyik szál annyi erőforrást harácsol össze, hogy a másikat lelassítja, vagy teljesen meg is akadályozza a futásban.

Egy közbenső séma a **küszöbölt erőforrás-megosztás**, amelynél egy szál dinamikusan szerezhet meg erőforrásokat (nincsenek előre felosztva), de csak egy bizonyos határig. A több példányban létező erőforrások esetében ez a módszer anélkül biztosít rugalmasságot, hogy fennállna annak a veszélye, hogy valamelyik szál nem jut erőforráshoz. Ha például az utasítások várakozási sorában egyik szál sem szerezhet meg többet a helyek ¾-énél, akkor mindegy mit csinál a lassú szál, a gyors futása nem állhat le.

A Pentium 4 hyperthreading különböző erőforrások esetén különböző stratégiákat alkalmaz a fent vázolt problémák elkerülése érdekében. Megkettőzték azokat az erőforrásokat, amelyeket a szálak állandóan használnak, mint például a programszámlálót, a regiszter hozzárendelési táblázatot és a megszakításvezérlőt. Ez a kettőzés csupán a lapkaterület 5 százalékos növekedésével járt, ami nem nagy ár a többszálúságért. Teljes megosztással dinamikusan osztják ki azokat az erőforrásokat, amelyek olyan nagy mennyiségben állnak rendelkezésre, hogy reálisan egyik szál sem szerezheti meg az összeset (például ilyenek a gyorsítósorok). Másrészről a csővezeték működését vezérlő minden erőforrás (mint például a csővezetékhez tartozó különféle várakozási sorok) fele-fele arányban megosztottak a két szál között. A Pentium 4-ben használt Netburst architektúra fő csővezetékét



8.9. ábra. *A szálak közötti erőforrás-megosztás a Pentium 4 NetBurst mikroarchitektúrában*

a 8.9. ábrán mutatjuk be, a fehér és szürke színezés jelzi, hogy az erőforrások miként vannak felosztva a két szál között.

Az ábrán láthatjuk, hogy a várakozási sorok mind fele-fele arányban fel vannak osztva a szálak között, itt egyik sem tud elvenni a másik elől. A regiszter lefoglaló és átnevező szintén felosztott. Az ütemező dinamikusan megosztott, de egy küszöbértékkel elejét veszik annak, hogy az egyik szál minden helyet kitöltsön. A csővezeték fennmaradó fázisai teljesen megosztottak.

De nem minden szép és jó a többszálúságban, van rossz oldala is. Míg a felosztás egyszerű, addig bármely erőforrás dinamikus megosztása (különösen, ha korlátozni akarjuk, hogy egy szál mennyit foglalhat le), a felhasználás nyilvántartása miatt futásidejű adminisztrációval jár. Ehhez hozzájön még az is, hogy olyan helyzetek is előfordulhatnak, amikor többszálúság nélkül egyes programok jobban futnak. Például, képzeljünk el két olyan szálat, amelyeknek a gyorsítótár ¾-ére szükségük van a megfelelő működéshez. Külön futtatva őket mindkettő rendben halad, és csak ritkán kényszerülnek költséges memóriaműveletre gyorsítótár hiány miatt. Együtt futtatva őket mindkettő sokszor kényszerül a gyorsítótár helyett a memóriához fordulni, és az összesített eredmény sokkal rosszabb lesz, mint többszálúság nélkül.

A Pentium 4 többszálúságáról további információ található (Gerber és Binstock, 2004; Koufaty és Marr, 2003; Tuck és Tullsen, 2003).

8.1.3. Egylapkás multiprocesszorok

Jóllehet a többszálúság mérsékelt többletköltség mellett jelentős teljesítménynövekedést biztosít, vannak olyan alkalmazások, amelyeknek sokkal nagyobb növekedés kell, mint amit a többszálúság biztosítani képes. A nagyobb teljesítménynövekedés eléréséhez multiprocesszoros lapkákat terveznek. Ezek a kettő vagy több CPU-t tartalmazó lapkák érdeklődésre tarthatnak számot például a nagy teljesítményű szerverek vagy a szórakoztatóelektronikai cikkek piacán. Röviden kitérünk most ezekre.

Homogén multiprocesszorok egy lapkán

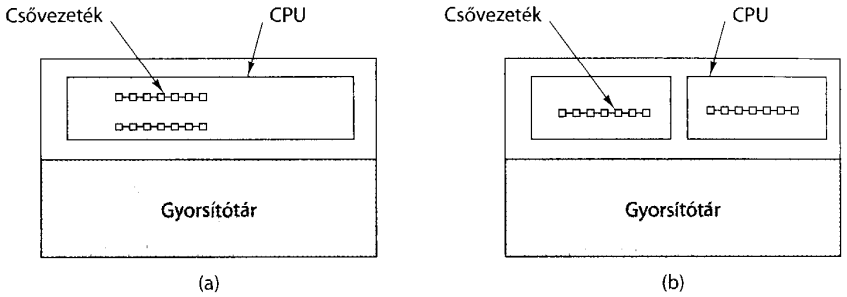
A VLSI technológia fejlődésével ma már lehetséges egy lapkán kettő vagy több nagy teljesítményű CPU-t elhelyezni. Mivel ezek a CPU-k mindig osztoznak az első és második szintű gyorsítótáron és a központi memórián, ezért multiprocesszornak nevezhetjük ezeket a 2. fejezetben tárgyaltak alapján. Tipikus alkalmazási területük a sok gépből álló nagy webszerver együttesek. Ha két CPU-t úgy pakolunk be egy dobozba, hogy nemcsak a memórián, hanem a lemezeken és a hálózati csatlakozókon is osztoznak, akkor gyakran a szerver teljesítménye megkétszerezhető anélkül, hogy a költségek megduplázódnának (mert hiába kell kétszer annyit kiadni CPU-ra, ez az összköltségnek csak töredéke).

A kisebb, egylapkás multiprocesszorok esetén kétféle felépítéssel találkozunk leginkább. Az egyik a 8.10. (a) ábrán látható, valójában egy lapka két csővezetékkel,

ami magában rejti a kétszeres végrehajtási sebesség lehetőségét. A 8.10. (b) ábrán látható a másik megoldás, amely két magot tartalmaz két teljes CPU-val. A **mag** egy nagy áramkör, mint amilyen egy CPU, egy B/K vezérlő vagy egy gyorsítótár, amelyet külön modulként el lehet helyezni a lapkán, általában más magok mellé.

Az első elrendezés lehetővé teszi, hogy olyan erőforrások, mint például a funkcionális egységek, a CPU-k között megosztottak legyenek, így az egyik CPU használhatja azokat az erőforrásokat, amelyekre a másiknak nincs szüksége. Másrészről ez a megközelítés szükségessé teszi a lapka újratervezését, és a módszer nem is igazán működőképes kettőnél több CPU-val. Ezzel ellentétben két vagy több mag elhelyezése egy lapkán viszonylag könnyű.

A fejezet egy későbbi részében tárgyaljuk majd a multiprocesszorokat. Jóllehet az a rész jobbára egyetlen CPU-t tartalmazó lapkákból összeállított multiprocesszorokra koncentrál, mégis nagy része érvényes a több CPU-t tartalmazó lapkákra is.



8.10. ábra. *Egylapkás multiprocesszorok. (a) Két csővezetékes lapka. (b) Kétmagos lapka*

Heterogén multiprocesszorok egy lapkán

Egy teljesen más, egylapkás multiprocesszorokat igénylő alkalmazási terület az olyan beágyazott rendszereké, amelyeket főleg az audiovizuális elektronikai cikkekben, például tv-készülékekben, DVD-lejátszókban, kamkorderekben, játékgépekben, mobiltelefonokban és hasonló termékekben használnak. E rendszerekkel szemben magasak a teljesítménybeli elvárások, és szűk specifikációs korlátok között kell működniük. Habár ezek az eszközök külsőre nem hasonlítanak egymásra, valójában egyre több közülük nem más, mint kisebb fajta számítógép egy vagy több CPU-val, memóriával, B/K vezérlővel és néhány speciális B/K eszközzel. Egy mobiltelefon például nem más, mint egy apró PC, benne CPU, memória, kicsinyke billentyűzet, mikrofon, hangszóró és vezeték nélküli hálózati csatlakozó.

Vagy tekintsünk példaként egy hordozható DVD-lejátszót. A benne lévő számítógépnek az alábbi feladatokat kell elvégeznie:

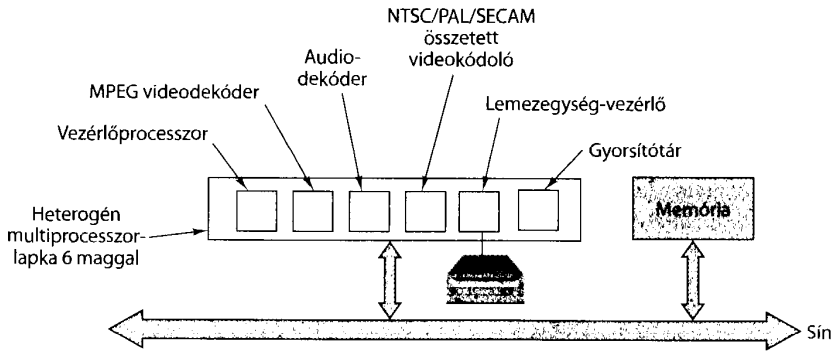
1. Egy olcsó, kis megbízhatóságú fejmozgató szervomechanika vezérlése;
2. Analóg jelek digitálissá átalakítása;

3. Hibajavítás;
4. Dekódolás és szerzői jogok kezelése;
5. MPEG-2 video kibontása;
6. Tömörített hang kibontása;
7. Kimenet kódolása NTSC, PAL és SECAM rendszerű tévékészülékekhez.

Ezeket a feladatokat szigorú követelményeknek megfelelve kell megoldani: valós időben, megfelelő minőségben, megszabott energiafelhasználással és hőleadással, mindemellett korlátozott mérettel, súllyal és árban.

A DVD-lemezek egy hosszú spirál mentén tartalmazzák az adatokat, ahogy a 2.24. ábrán látható (ott CD-re). Az olvasófejnek pontosan kell követnie a spirált a lemez forgása közben. Az árat azzal tartják alacsonyan, hogy viszonylag egyszerű mechanikát használnak, de a fejet szoftveres úton állandó ellenőrzés alatt tartják. A fejből analóg jel érkezik, amelyet további feldolgozás előtt digitálisra kell alakítani. Digitalizálás után alapos hibajavításra van szükség, mert a DVD-k a préseléses előállítás miatt sok hibát tartalmaznak, amelyeket szoftverrel kell kijavítani. A kép az MPEG-2 szabvány szerint tömörített, ezért bonyolult (Fourier-transzformációhoz hasonló) átalakítást igényel a kibontáshoz. A hang egy pszichoakusztikus modell alkalmazásával kerül tömörítésre, ami szintén összetett számításokat igényel a kibontáshoz. Végül, a képet és a hangot olyan formátumra kell alakítani, amely lehetővé teszi, hogy NTSC, PAL vagy SECAM tévékészüléken megjeleníthető legyen, attól függően, hogy a DVD-lejátszót a világ mely országában használják. Nem meglepő, hogy mindezt a sok valós idejű feldolgozást igénylő munkát nem lehet elvégezni olcsó, általános célú CPU-n futó programmal. Szükség van egy heterogén, többmagos multiprocesszorra, melyben minden mag egy meghatározott speciális feladatot lát el. A 8.11. ábrán példaként egy DVD-lejátszó látható.

A 8.11. ábrán a magok mind különböző feladatokat látnak el, mindegyiket gondosan arra tervezték, hogy a lehető legolcsóbb legyen, és emellett rendkívül hatékonyan oldja meg a rábízott feladatot. Például a DVD-kép tömörítése az **MPEG-2**



8.11. ábra. *Egy egyszerű DVD-lejátszó logikai felépítése. A heterogén multiprocesszor a különböző funkciókat külön magokkal oldja meg*

(a létrehozó **Motion Picture Expert Group** kezdőbetűiből) séma szerint történik. Ez abból áll, hogy minden képkockát pixelekből álló blokkokra osztunk, és a blokkok mindegyikén egy összetett transzformációt végzünk el. Egy képkocka állhat teljes egészében transzformált blokkokból, de megadható az is, hogy egy bizonyos blokk megegyezik az előző képkocka egy blokkjával, esetleg az eredeti helyétől valamilyen (Δx, Δy) vektorral el van tolva, és néhány pixel megváltozott benne. Ezek a számítások szoftveresen nagyon lassúk, de lehet jó gyors MPEG-2 hardverdekódoló egységet építeni. Hasonlóképpen, a hang dekódolása és az egyesített kép-hang jel a világ valamelyik tv-szabványa szerinti újrakódolása hardveresen gyorsabban elvégezhető. Ezek a megfigyelések hamar elvezettek az audiovizuális alkalmazásokhoz tervezett többmagos, heterogén multiprocesszoros lapkákhoz. Mivel azonban a vezérlőprocesszor egy általános célú, programozható CPU, ezért a multiprocesszor-lapka más, hasonló alkalmazási területeken is felhasználható, például DVD-felvevőkben.

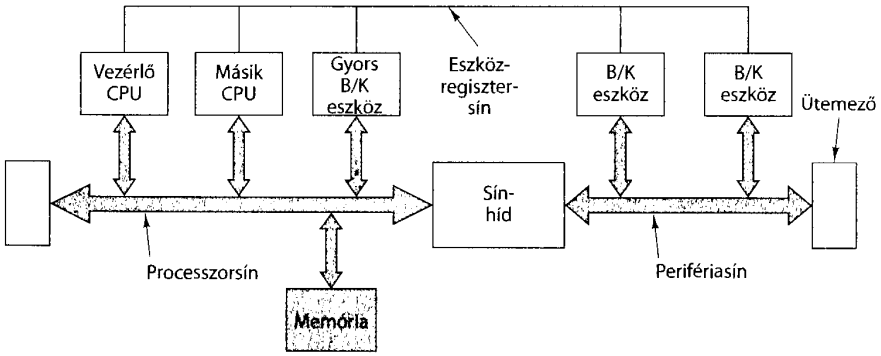
A fejlett mobiltelefonok is olyan eszközök, amelyekbe heterogén multiprocesszor kell. A jelenlegi típusokban van fényképezőgép, filmfelvevő, játékok, webböngésző, e-mail olvasó és digitális műholdvevő rádió. Ezeket a funkciókat vagy mobiltelefon- (CDMA vagy GSM) technológiát, vagy vezeték nélküli internet- (IEEE 802.11, más néven WiFi) technológiát felhasználva építik be; lehet, hogy a jövő készülékei mindezeket együtt fogják tartalmazni. Ahogy az eszközök egyre több és több funkciót látnak el, és az órák GPS-alapú térképekké, a szemüvegek rádióvá válnak, a heterogén multiprocesszorok iránti igény egyre növekedni fog.

Hamarosan a lapkák 500 millió tranzisztort fognak tartalmazni. Az ilyen lapkák túl nagyok ahhoz, hogy kapunként és vezetékenként tervezzük meg őket. Ehhez akkora emberi erőfeszítés kellene, hogy elavulnának, mire elkészülnének. Az egyetlen lehetséges út a magok használata (ezek lényegében könyvtárak, a programkönyvtárak mintájára); a magok elég nagy részelemek, amelyeket csak egymás mellé kell helyezni, és megfelelően össze kell kötni a lapkán. A tervezőknek csak azt kell eldönteniük, hogy melyik CPU magot használják vezérlőprocesszornak, és melyik speciális célú processzort tegyék mellé segítségül. Minél több vezérlőfunkciót lát el a vezérlőprocesszor szoftvere, annál lassabb lesz a rendszer, de egyben annál kisebb (és olcsóbb) a lapka. A speciális célú kép- és hangfeldolgozó processzorok elfoglalnak valamennyi területet a lapkán, ami növeli a költségeket, de nagyobb teljesítményt biztosít alacsonyabb órajel mellett, ami viszont kisebb fogyasztást és kisebb hőtermelést jelent. Így a lapkatervezőknek egyre inkább ezeken a magasabb szinteken kell kompromisszumokat kötniük ahelyett, hogy azzal foglalkoznának, hogy az egyes tranzisztorok hova kerüljenek.

Az audiovizuális alkalmazások nagyon adatorientáltak. Óriási mennyiségű adatot kell rövid idő alatt feldolgozni, ezért tipikusan a lapkaterület 50-70 százaléka valamilyen formában memória számára van fenntartva, és ez az arány növekvőben van. Sok itt a tervezési probléma. Hány gyorsítótárszint legyen? A gyorsítótár(ak) közösek vagy osztottak legyenek? Mekkorák legyenek a gyorsítótárak? Milyen gyorsak legyenek? Legyen-e a központi memóriából valamennyi a lapkán? SRAM vagy DRAM legyen? Az ezekre a kérdésekre adott válaszok nagyban befolyásolják a lapka teljesítményét, energiafogyasztását és hőleadását.

A processzorok és a memóriák mellett jelentős következménnyel járnak a kommunikációs alrendszer tervezése során hozott döntések is – hogyan fognak a magok kommunikálni egymással? Kis rendszerekben egy sín elég, de nagyobbakban ez hamar szűk keresztmetszetté válik. A probléma gyakran megoldható több sínnel, vagy egy magokat összekötő gyűrűvel. Az utóbbi esetben a sínkiosztást egy kis adatcsomag, az úgynevezett **token** körbeutaztatásával oldják meg. A sín használatához a magnak először meg kell szereznie a tokent. Miután végzett, továbbadja a tokent, amely így folytathatja körútját. Ezeket a szabályokat követve kiküszöbölhetők az ütközések.

Lapkán belüli összeköttetésre példa a 8.12. ábrán bemutatott IBM **CoreConnect**. Ez egy olyan architektúra, amely egylapkás heterogén multiprocesszorok magjainak összekötésére szolgál, különösen komplett egylapkás rendszerekhez alkalmazható. Bizonyos értelemben a CoreConnect ugyanaz az egylapkás multiprocesszoroknak, mint a PCI sín a Pentiumnak — kötőanyag, amely összetartja a részeket. A PCI sínnel ellentétben azonban a CoreConnect tervezésekor nem kellett visszafelé kompatibilisnek maradni régebbi eszközökkel vagy protokollokkal, illetve nem korlátozták a fejlesztést olyan megszorítások, mint az alaplapi sínek esetén, ahol például kötött, hogy egy csatlakozónak hány érintkezője kell legyen.



8.12. ábra. *Példa az IBM CoreConnect architektúrára*

A CoreConnect három sínből áll. A **processzorsín** egy nagy sebességű, szinkron, csővezetékes sín 32, 64 vagy 128 adatvezetékkel, és 66, 133 vagy 183 MHz-es órajellel. A legnagyobb áteresztőképessége így 23,4 Gbps (a PCI síné 4,2 Gbps). A csővezetékes felépítés lehetővé teszi, hogy a magok egy folyamatban lévő átvitel ideje alatt kérjenek hozzáférést a sínhez, vagy hogy a PCI sínhez hasonlóan különböző magok különböző vonalakat egyszerre használjanak. A processzorsín rövid blokkok átvitelére van optimalizálva. Gyors magok összekapcsolására szánták, ilyenek a CPU-k, MPEG-2 dekóderek, nagy sebességű hálózatok és hasonló rendszerek.

A processzorsín szétterítése az egész lapkán csökkentené a teljesítményét, ezért van egy második sín a lassú B/K eszközöknek, ilyenek például az UART-ok, időzítők, USB-vezérlők, soros B/K eszközök stb. Ezt a **perifériális sín**t azzal a céllal

tervezték, hogy könnyű legyen hozzákapcsolni legfeljebb néhány száz kapu felhasználásával 8, 16, és 32 bites perifériákat. Ez is szinkron sín, legnagyobb áteresztőképessége 300 Mbps. A két sínt egy híd köti össze, ahhoz hasonlóan, ahogy a PCI és az ISA síneket hidak kötötték össze a PC-kben, még mielőtt néhány éve az ISA sín teljesen kiszorult volna.

A harmadik az **eszközregisztersín**, egy nagyon lassú, aszinkron, kapcsolatorientált sín, amelynek segítségével a processzorok hozzáférhetnek bármely periféria eszközregiszteréhez, és így vezérelhetik azokat. Ritka, egyszerre néhány bájtos adatátvitelekre szánták.

A lapkákhoz használható szabványos sín, interfész és keretrendszer kidolgozásától az IBM azt reméli, hogy a PCI-rendszer miniatűr változatát hozhatja létre, melyben sok gyártó készít együttműködésre képes processzorokat és vezérlőket. Különbség azonban, hogy a PCI-eszközöket a gyártók maguk állítják elő és adják el a kereskedőknek, illetve a felhasználóknak. A CoreConnect esetén a tervezők megtervezik a magokat, de nem gyártják azokat. Ehelyett a használati jogot adják el a szórakoztatóelektronikai és más vállalatoknak, akik aztán megtervezik a saját heterogén multiprocesszoros lapkáikat a saját és a licencelt magok felhasználásával. Mivel az ilyen nagy és összetett lapkák gyártása hatalmas befektetést igényel, a legtöbb esetben a szórakoztatóelektronikai cég csak a terveket készíti el, majd egy félvezetőket előállító alvállalkozóval legyártatja azokat. Számos CPU (ARM, MIPS, PowerPC stb.) mag érhető el, ezen kívül MPEG dekódereké, digitális jelfeldolgozóké és a szabványos B/K vezérlőké is.

Az IBM CoreConnect nem az egyetlen népszerű lapkasín a piacon. Az **AMBA (Advanced Microcontroller Bus Architecture, fejlett mikrokontroller sínarchitektúra)** is széles körben használt (Flynn, 1977). Más, valamennyivel kevésbé népszerű lapkasínek a **VCI (Virtual Component Interconnect, virtuális komponens összeköttetés)** és az **OCP-IP (Open Core Protocol-International Partnership, nyílt magprotokoll nemzetközi társaság)**, amelyek szintén versenyben vannak a piaci részesedésért (Kogel és Meyr, 2004; Ouadjaout és Houzer, 2004). A lapkasínek csak a kezdet, vannak akik akár teljes hálózatokat terveznek a lapkákra (Benini és De Micheli, 2002).

Mivel hőelvezetési problémák miatt a lapkagyártók egyre nagyobb nehézségekbe ütköznek az órajel frekvenciájának növelése terén, ezért az egylapkás multiprocesszorok területe nagyon népszerű. További információ található (Classen, 2003; Jerrays és Wolf, 2005; Kumer és társai, 2004; Lavagno, 2002; Lines, 2004; Ravikumar, 2004).

8.2. Társprocesszorok

Miután megvizsgáltuk a lapkaszintű párhuzamosság néhány fajtáját, lépjünk most egy szinttel feljebb, és vizsgáljuk meg, hogyan növelhető egy számítógép sebessége azáltal, hogy egy második, speciális célú processzort is teszünk bele. Ilyen **társprocesszor** sokféle van, kicsitől a nagyig. Az IBM 360 nagygépekben és ezek

összes leszármazottjában a bemeneti/kimeneti műveleteket független B/K csatornák végzik. Hasonlóképpen, a CDC 6600-nak 10 független processzora van a B/K műveletek elvégzésére. A grafika és a lebegőpontos aritmetika két olyan terület még, ahol eddig alkalmaztak társprocesszorokat. Még egy DMA lapkát is társprocesszornak tekinthetünk. Néha a CPU adja át a végrehajtandó utasítást vagy utasításokat a társprocesszornak, más esetekben a társprocesszor függetlenebb és többé-kevésbé önállóan dolgozik.

Fizikailag a társprocesszorok lehetnek külön szekrényben (a 360-as B/K csatornái), lehetnek külön kártyán (hálózati processzorok), vagy elfoglalhatják a főprocesszor lapkájának egy részét (lebegőpontos társprocesszor). Minden esetben arról lehet felismerni, hogy van egy másik processzor, a főprocesszor, amelyet segítenek. Most megvizsgálunk három olyan területet, ahol lehetséges a sebesség növelése: hálózati feldolgozás, multimédia és titkosítás.

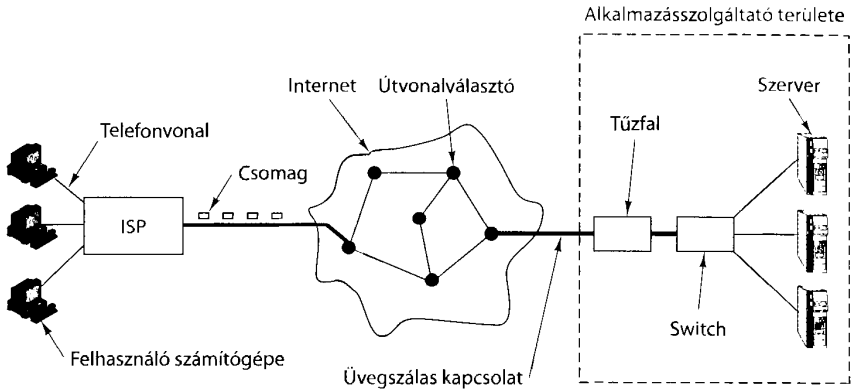
8.2.1. Hálózati processzorok

Ma a legtöbb számítógép csatlakoztatva van valamilyen hálózathoz vagy az internethez. A hálózati hardver terén bekövetkezett technológiai haladás eredményeként a hálózatok ma olyan gyorsak, hogy a bejövő és a kimenő adatokat egyre nehezebbé vált szoftverrel feldolgozni. Ennek következtében speciális hálózati processzorokat fejlesztettek ki az adatforgalom kezelésére, és sok nagy teljesítményű számítógépben van is ilyen processzor. Ebben az alfejezetben először áttekintjük a hálózatok működését, majd rátérünk a hálózati processzorokra.

Bevezetés a hálózatok működésébe

A számítógép-hálózatok két típusba sorolhatók: **helyi hálózat (LAN, Local Area Network)**, amely egy épületen vagy épületegyüttesen belül köt össze számítógépeket, és a **nagy területű hálózat (WAN, Wide Area Network)**, amely földrajzilag nagy területen elhelyezkedő számítógépeket kapcsol össze. A legnépszerűbb LAN az ún. **Ethernet**. Az eredeti Ethernet egy vastag kábel volt, amelybe a számítógépekből jövő vezetéket egy eufemisztikusan **vámpírcsatlakozó**nak nevezett eszközzel csatlakoztatták. A mai Ethernetben a számítógépek egy központi útvonalkapcsolóhoz vannak kötve, amint az a 8.13. ábra jobboldali részén is látható. Az eredeti Ethernet 3 Mbps sebességgel csoszogott, de az első kereskedelmi verzió sebessége már 10 Mbps volt. Ezt végül leváltotta a gyors Ethernet 100 Mbps, majd ezt a gigabit Ethernet 1 Gbps sebességgel. Már elérhető a 10 gigabites Ethernet, és előkészületben van a 40 gigabites is.

A WAN hálózatok felépítése ettől eltérő. Ezek speciális útvonalválasztó számítógépekből állnak, amelyeket hagyományos vezeték vagy optikai szál köt össze, ahogy a 8.13. ábra középső részén látható. Az adatok tipikusan 64 és 1500 bájt közötti méretű **csomag**okban utaznak a kiindulási ponttól egy vagy több útvonalválasztón keresztül a célállomásig. Minden lépésben a csomagot először eltárolják



8.13. ábra. *A felhasználók kapcsolódása a szerverekhez az interneten*

az útvonalválasztó memóriájában, és csak akkor továbbítják a következőhöz, ha az ehhez szükséges kommunikációs vonal rendelkezésre áll. Ennek a módszernek a neve **tárold-és-továbbítsd csomagkapcsolás**.

Jóllehet sokan úgy tekintenek az internetre, mintha egy WAN lenne, technikailag azonban sok WAN összekapcsolva. A mi szempontunkból azonban ez a megkülönböztetés nem fontos. A 8.13. ábra egy otthoni felhasználó szempontjából mutatja az internet szerkezetét. A felhasználó számítógépe rendszerint egy webszerverhez kapcsolódik telefonvonalon keresztül, 56 kbps sebességű betárcsázós modemmel vagy a 2. fejezetben tárgyalt ADSL-lel. (Elképzelhető még kábeltévés kapcsolat is, ez esetben a 8.13. ábra bal oldala egy kicsit másképpen néz ki, továbbá a kábeltévé-társaság az ISP). A felhasználó gépe az elküldeni kívánt adatokat csomagokra bontja és elküldi az **internetszolgáltató (ISP, Internet Service Provider)** gépére. Az internetszolgáltató az a cég, amely internet-hozzáférést nyújt az ügyfelei részére. A szolgáltatónak nagy sebességű (rendszerint üvegszálas) kapcsolata van az internet részét képező egyik területi- vagy gerinchálózattal. A felhasználó adatai az útvonalválasztók között haladva jutnak el az interneten keresztül a webszerverhez.

A legtöbb webszolgáltatást nyújtó cégnek van egy speciális számítógépe, amelyet **tűzfal**nak neveznek, ez szűri a bejövő forgalmat és eltávolítja a nem kívánt csomagokat (például a betörni szándékozó számítógépes kalózoktól érkezőket). A tűzfal a helyi hálózathoz csatlakozik, rendszerint egy Ethernet-switch-hez (kapcsolóhoz), amely a kívánt szerverhez továbbítja az adatokat. Természetesen a valóság sokkal bonyolultabb ennél, de a 8.13. ábra nagy vonalakban helyes képet mutat.

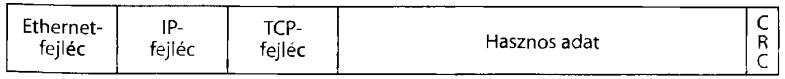
A hálózati szoftver **protokoll**okból áll, ezek mindegyike formátumok és szabályok halmaza, amelyek meghatározzák az adatok jelentését, és szabályozzák az adatcserét. Például, amikor a felhasználó egy weboldalt akar lehívni egy szerverről, akkor a felhasználó böngészője egy *GET PAGE* kérést tartalmazó csomagot küld **HTTP (HyperText Transfer Protocol)** formátumban a szervernek, amely tudja, hogyan kell feldolgozni az ilyen kéréseket. Sok protokoll van használatban, és sokszor

kombinálva is használják ezeket. A legtöbb helyzetben a protokollokat rétegekbe szervezik, a felsőbb rétegek az alacsonyabban fekvők felé továbbítják a csomagokat feldolgozásra, és végül a legalsó szinten lévő végzi el a tényleges adatátvitelt. A fogadó oldalon a csomagok fordított sorrendben haladnak a felsőbb rétegek felé.

Mivel a hálózati processzorok foglalkozásukat tekintve protokoll feldolgozók, ezért el kell mondanunk néhány dolgot a protokollokról, mielőtt magukra a hálózati processzorokra sor kerülhetne. Térjünk vissza egy pillanatra a *GET PAGE* kérésre. Hogyan kerül ez a webszerverhez? Az történik, hogy a böngésző először kapcsolatot teremt a webszerverrel a **TCP (Transmission Control Protocol)** segítségével. Az ezt a protokollt megvalósító szoftver ellenőrzi, hogy az összes csomag hibátlanul és a megfelelő sorrendben megérkezett-e. Ha egy csomag elvész, a TCP szoftver gondoskodik róla, hogy annyiszor el legyen küldve újra és újra, amíg rendben meg nem érkezik. A gyakorlatban az történik, hogy a webböngésző a *GET PAGE* üzenetet a HTTP szabályoknak megfelelő formátumban a TCP szoftvernek átadja, hogy az a felépített kapcsolatot használva továbbítsa. A TCP szoftver az üzenet elé egy fejlécet tesz, amelyben sorozatszám és egyéb információ van. Ezt a fejlécet természetesen **TCP fejléc**nek nevezik.

Ezután a TCP szoftver fogja a TCP fejlécet és az üzenetet (amely a *GET PAGE* kérést tartalmazza), ezt átadja egy másik szoftverkomponensnek, amely az **internetprotokollt (IP)** valósítja meg. Ez egy IP-fejlécet illeszt az adatok elé, amely tartalmazza a kiindulási címet (melyik gépről jön a csomag), a célállomás címét (melyik gépnek szánjuk a csomagot), hány útvonalválasztón haladhat még át megsemmisítés előtt (ez megakadályozza, hogy az elveszett csomagok örökké keringjenek a hálózatban), egy ellenőrző összeget (átviteli és memóriahibák felderítésére), valamint egyéb mezőket.

Ezt követően az így kialakított csomag (amely most már az IP-fejlécet, a TCP-fejlécet és a *GET PAGE* kérést tartalmazza) átkerül az adatkapcsolati réteghez, ahol egy adatkapcsolati fejléc kerül elé a tényleges adatátvitel előtt. Az adatkapcsolati réteg egy **CRC (Cyclic Redundancy Code, Ciklikus redundanciakód)** nevű ellenőrző összeget is elhelyez a végén, hogy az átviteli hibákat fel lehessen ismerni. Úgy tűnhet, hogy felesleges az adatkapcsolati rétegben és az IP-rétegben is ellenőrző összegeket elhelyezni, de ez növeli a megbízhatóságot. A CRC minden átvitelnél ellenőrzésre kerül, a fejléccel együtt leválasztják, majd újragenerálják olyan formátumban, amely megfelel a kimenő kapcsolatnak. A 8.14. ábra azt mutatja, hogy a csomag miként néz ki az Etherneten; telefonvonalon hasonló, csak az Ethernet-fejléc helyett „telefonvonal-fejléc" van az elején. A fejlécek kezelése fontos terület, ez a hálózati processzorok egyik funkciója. Mondanunk sem kell, hogy a számítógép-hálózatok működésének témakörét csak felszínesen érintettük. Alapos tárgyalása található (Tanenbaum, 2003).



8.14. ábra. *Egy csomag formátuma az Etherneten*

Bevezetés a hálózati processzorok működésébe

Sokfajta eszköz van a hálózatokhoz csatlakoztatva. A felhasználók személyi (asztali és hordozható) számítógépei természetesen, de egyre inkább játékgépek, PDA-k (tenyérszámítógépek) és mobiltelefonok is. A cégek PC-i és szerverei a hálózat végpontjai, de számtalan olyan eszköz is van, amelyek közvetítő szerepet játszanak a hálózatokban, ezek közé tartoznak az útvonalválasztók, switchek, tűzfalak, helyettesítő kiszolgálók és terheléselosztók. Érdekes módon ezek a közvetítő rendszerek vannak a legnagyobb igénybevételnek kitéve, mert ezeknek kell másodpercenként a legtöbb csomagot megmozgatniuk. A szerverek szintén le vannak terhelve, de a felhasználók gépei nincsenek.

A hálózattól és magától a csomagtól függően, egy bejövő csomag sokféle feldolgozást igényelhet, mielőtt továbbítják a kimenő vonalon, vagy átadják a felhasználói programnak. A feldolgozás lehet a csomag további sorsának eldöntése, feldarabolása vagy összeállítása darabjaiból, a szolgáltatás minőségének kezelése (főleg hang és mozgókép esetén), biztonsági kérdések kezelése (például titkosítás vagy titkosított üzenet visszaállítása), tömörítés/kibontás stb.

A lokális hálózatok sebessége kezdi megközelíteni a 40 gigabit/sec sebességet, 1 KB méretű csomagokkal számolva egy hálózatba kapcsolt számítógépnek másodpercenként közel 5 millió csomagot kell feldolgoznia. Ha 64 bájtos csomagokat tételezünk fel, akkor a másodpercenként feldolgozandó csomagok száma közel 80 millió. Az előbb említett feldolgozási műveleteket 12–200 ns alatt szoftverrel nem lehet elvégezni (ráadásul a csomagot menthetetlenül le kell másolni több példányban a feldolgozás alatt). A hardveres segítség elengedhetetlen.

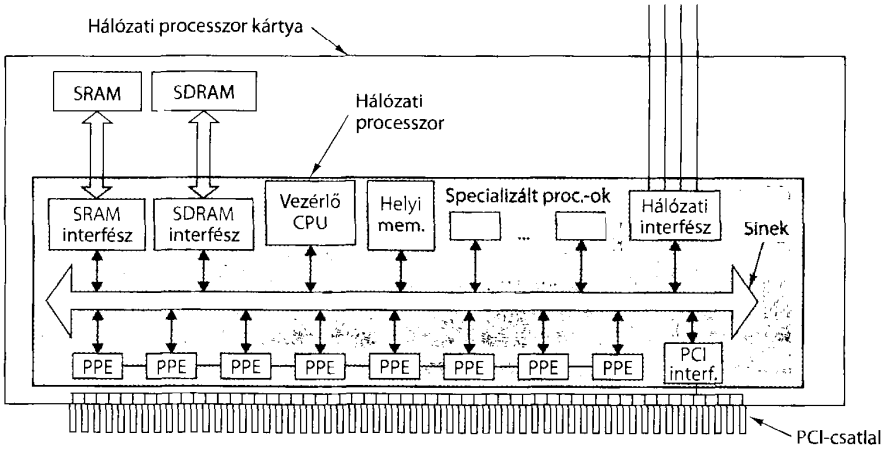
A gyors csomagfeldolgozás egyik hardveres megoldása egy **ASIC (Application-Specific Integrated Circuit, alkalmazáshoz kifejlesztett egyedi integrált áramkör)**. Egy ilyen áramkör olyan, mint egy bedrótozott program, amely a megtervezett feldolgozási funkciókat hajtja végre. Sok mai útvonalválasztó használ ASIC-ot, ezekkel azonban több probléma is van. Először is hosszú ideig tart megtervezni és legyártani. Rugalmatlanok is, ha egy új funkcióra van szükség, akkor új lapkát kell tervezni és gyártani. Továbbá a hibakeresés igazi rémálom, hiszen a javítás egyetlen módja, hogy megterveznek, legyártanak, leszállítanak és beszerelnek egy új lapkát. Drágák is, hacsak nem gyárthatók elég nagy mennyiségben ahhoz, hogy a költségek megtérüljenek.

Egy másik megoldás az **FPGA (Field Programmable Gate Array, helyszínen programozható kapumátrix)** alkalmazása. Ez olyan logikai kapukat tartalmaz, amelyeket a helyszínen lehet tetszőleges módon összekötni, és ezáltal a kívánt áramkört létrehozni. Ezek az ASIC-oknál sokkal rövidebb idő alatt kerülnek piacra, és helyben módosíthatók úgy, hogy kiemelik a rendszerből, és egy speciális újraprogramozó eszközbe helyezik. Másrészről viszont eléggé bonyolultak, lassúk és drágák, ami néhány speciális alkalmazási területtől eltekintve nem teszi őket vonzóvá.

Végül a **hálózati processzor**okhoz érkeztünk, ezek olyan programozható eszközök, amelyek képesek a bejövő és kimenő csomagokat valós időben kezelni, vagyis olyan gyorsan, ahogy az adatok a vezetékben haladnak. A leggyakoribb megoldás

az, hogy a hálózati processzor lapkáját egy kiegészítő kártyán helyezik el memória és egyéb szükséges logikai áramkör társaságában. Az alaplap egy vagy több hálózati csatlakozását a hálózati processzorhoz irányítják. A beérkezett és feldolgozott csomagokat vagy továbbküldi egy másik hálózati vonalon (például útvonalválasztó esetén), vagy a rendszersínre (például PCI sín) továbbítja, ha végfelhasználói eszközről van szó, mint amilyen egy felhasználó PC-je is. Egy tipikus hálózati processzor lapka és kártya látható a 8.15. ábrán.

SRAM és SDRAM is található a kártyán, ezeket tipikusan más-más célra használják. Az SRAM gyorsabb, de drágább az SDRAM-nál, így csak kevés van belőle. Az SRAM az útvonalválasztáshoz szükséges táblázatokat és más kulcsfontosságú adatokat, míg az SDRAM a feldolgozás alatt álló csomagokat tárolja. Azáltal, hogy az SRAM és SDRAM a hálózati lapkán kívül helyezkedik el, a kártya tervezői eldönthetik, hogy melyikből mennyit használnak. Így az egyetlen hálózati vonalat kezelő kis teljesítményű kártyákba (például egy PC-hez vagy szerverhez) kevesebb memóriát lehet tenni, míg egy komolyabb útvonalválasztó nagy teljesítményű kártyájába jóval többet.



8.15. ábra. *Egy tipikus hálózati processzor és kártya*

A hálózati processzorok lapkáit arra optimalizálják, hogy nagyszámú bejövő és kimenő csomagot gyorsan fel tudjanak dolgozni. Ez hálózati vonalanként és másodpercenként több millió csomagot jelent, és egy útvonalválasztónak könnyen lehet féltucat kiszolgálandó vonala. Ilyen feldolgozási sebesség csak úgy érhető el, ha a hálózati processzorok belül a lehető legnagyobb mértékben párhuzamosan működnek. És valóban, minden hálózati processzort több PPE alkot. Ennek a rövidítésnek az eredete nem teljesen tisztázott, lehet **protokoll-/programozott/csomagfeldolgozó egység (Protocol/Programmed/Packet Processing Engine)**. Mindegyik egy (esetleg módosított) RISC-mag és egy kisméretű belső memória a program és a változók tárolására.

A PPE-ket kétféleképpen lehet optimalizálni. A legegyszerűbb felépítés az, ha minden PPE egyforma. Amikor csomag érkezik a hálózati processzorhoz (akár bejövő valamelyik vonalon, akár kimenő a sínről), azt az egyik szabad PPE-hez irányítják feldolgozásra. Ha nincs szabad PPE, akkor a csomag a kártya SDRAM-jában tárolt várakozási sorba kerül, amíg egy PPE fel nem szabadul. Ha ezt a módszert alkalmazzák, akkor a PPE-k között a 8.15. ábrán látható vízszintes kapcsolatok nem léteznek, mert a PPE-knek nem kell kommunikálniuk egymással.

A másik PPE-elrendezés a csővezeték. Ebben az esetben mindegyik PPE egy feldolgozási lépést hajt végre, majd az eredményül kapott csomag mutatóját átadja a csővezetékben rákövetkező PPE-nek. Ily módon a PPE-csővezeték a 2. fejezetben tanulmányozott CPU-csővezetékekhez hasonlóan viselkedik. Mindkét elrendezésben a PPE-k teljesen programozhatók.

Fejlettebb rendszerekben a PPE-k többszálúságot is használnak, vagyis több regiszterkészletük van, amelyek közül egy másik hardver regiszter jelöli ki az éppen használt aktív készletet. Ez a felépítés lehetőséget ad arra, hogy több program fusson egyszerre, mert program (vagyis szál) váltáshoz csak az aktív regiszterkészletet kijelölő regisztert kell átállítani. Amikor egy PPE megakad, mert például adatot kell olvasnia az SDRAM-ból (ami sok óraciklust igényel), akkor leggyakrabban azonnal váltani tud másik szálra. Ily módon egy PPE akkor is közel maximális kihasználtsággal üzemelhet, ha egyébként gyakran kellene az SDRAM elérése vagy más lassú külső művelet végrehajtása miatt várakoznia.

A PPE-k mellett minden hálózati processzor tartalmaz egy vezérlőprocesszort, amely rendszerint egy normál általános célú RISC CPU, a csomagkezeléssel szorosan össze nem függő feladatok elvégzésére, mint például az útvonalválasztó táblázatok frissítése. Ennek a programja és adatai a lapka memóriájában vannak. Ezen kívül sok hálózati processzor lapkán vannak még speciális processzorok mintaillesztés vagy más kritikus művelet elvégzésére. Ezek a processzorok valójában kis ASIC-ok, amelyek nagyon jók egyetlen egyszerű művelet elvégzésében, például az útvonalválasztó táblából egy cím megtalálására. A hálózati processzor komponensei a lapkán elhelyezett egy vagy több párhuzamos sínen keresztül kommunikálnak egymással, ezek sebessége több gigabit/másodperc.

Csomagfeldolgozás

Egy csomag megérkezésekor attól függetlenül keresztülmegy néhány feldolgozási fázison, hogy a hálózati processzor csővezetékes felépítésű-e, vagy sem. Néhány hálózati processzor a munkát **bejövő csomagokon végzett feldolgozás**ra (akár egy hálózati vonalról jön, akár a rendszersínről), illetve **kimenő csomagokon végzett feldolgozás**ra osztja. Ha ilyen módon teszünk megkülönböztetést, akkor minden csomag először bemenő feldolgozáson, majd kimenő feldolgozáson megy keresztül. A kétfajta feldolgozás között nem éles a határ, mert bizonyos műveleteket bármelyiknél el lehet végezni (például adatforgalmi statisztikák számítása).

Alább felállítjuk a különböző lépések egy lehetséges sorrendjét, de jegyezzük meg, hogy nem minden csomagnál szükséges elvégezni az összes műveletet, és sok más sorrend is elképzelhető lenne.

1. **Ellenőrző összeg vizsgálata.** Ha a bejövő csomag az Ethernetről érkezik, akkor a CRC-je újra kiszámításra kerül, hogy össze lehessen hasonlítani a csomagban lévővel. Ha a kettő egyezik, akkor nem történt átviteli hiba. Ha az Ethernet CRC helyes vagy hiányzik, akkor az IP ellenőrző összeg kerül kiszámításra és ellenőrzésre, hogy biztosak lehessünk abban, hogy az IP-csomag nem sérült meg a küldő memóriájának hibás bitje miatt, miután az az IP ellenőrző összeget kiszámította. Ha minden ellenőrző összeg helyes, akkor a csomag további feldolgozása folytatódhat, különben egyszerűen megsemmisítésre kerül.
2. **Mező kinyerése.** A megfelelő fejléc elemzése, és a kívánt mezők értékének kiolvasása történik itt. Egy Ethernet-kapcsoló csak az Ethernet-fejlécet vizsgálja, míg egy IP-útvonalválasztó csak az IP-fejlécet. A kívánt mezők értékei regiszterekben (párhuzamos PPE-felépítés) vagy SRAM-ban (csővezetékes felépítés) tárolódnak.
3. **Csomagosztályozás.** A csomagok egy sor programozható szabálynak megfelelően osztályozásra kerülnek. A legegyszerűbb osztályozás, amikor adatcsomagokat és vezérlőcsomagokat különböztetünk meg, de rendszerint ennél sokkal finomabb a felosztás.
4. **Útválasztás.** A legtöbb hálózati processzor fel van készítve arra, hogy a mezei adatcsomagokat gyorsan lekezelje, míg az összes többit ezektől eltérően gyakran a vezérlőprocesszor veszi kezelésbe. El kell dönteni, mi legyen a csomaggal ilyen szempontból.
5. **Célhálózat megállapítása.** Az IP-csomagok tartalmazzák a célállomás 32 bites címét. Nem lehet (és nem is lenne célszerű) egy 2^{32} számú bejegyzést tartalmazó táblázatot használni az IP-csomagok célállomásának megállapítására, ezért az IP cím bal oldali része a hálózat száma, a maradék része pedig egy gépet határoz meg a hálózaton belül. A hálózat száma akármekkora hosszúságú lehet, ezért a célállomás meghatározása nem triviális feladat, és még az is bonyolítja, hogy több lehetőség is adódhat, de ezek közül a leghosszabb számít. Gyakran erre a célra ASIC-ot használnak.
6. **Útvonal megállapítása.** Ha a célállomás hálózata már ismert, akkor a hozzá tartozó kimenő vonal egy SRAM-ban található táblázatból kiolvasható. Ebben a lépésben is használható speciális ASIC.
7. **Feldarabolás és összerakás.** A rendszerhívások számának csökkentése végett a programok előszeretettel adnak át egyszerre nagy mennyiségű adatot a TCP rétegnek, de a TCP, az IP és az Ethernet-protokoll csak egy megadott maximális méretű csomagot tud kezelni. A korlátok miatt elképzelhető, hogy a küldésre váró adatokat, illetve csomagokat a küldő oldalon fel kell darabolni, majd a darabokat a fogadó oldalon összerakni. Ezeket a feladatokat is el tudja végezni a hálózati processzor.
8. **Feldolgozás.** Néha nagy számítási igényű műveleteket kell elvégezni, például tömörítés/kibontás vagy titkosítás/visszafejtés. Ezeket a feladatokat is el tudja végezni a hálózati processzor.
9. **Fejléckezelés.** Alkalmanként a csomagokhoz fejléceket kell előállítani, onnan el kell távolítani vagy valamelyik mezőjüket módosítani kell. Például az IP-fejlécben van egy mező, amely megadja, hogy hány lépést tehet még meg a csomag mielőtt megsemmisítésre kerülne. Ezt a mezőt minden elküldés előtt csökkenteni kell, amit a hálózati processzor el tud végezni.
10. **Várakozási sor kezelése.** A bejövő és kimenő csomagokat várakozási sorba kell helyezni, amíg rájuk nem kerül a sor. A multimédia-alkalmazásoknak szükségük lehet a csomagok között egy bizonyos egyenletes időközre a változó késleltetés okozta hibák (jitter) megelőzésére. Egy tűzfal vagy útvonalválasztó a bejövő csomagokat bizonyos szabályok szerint szétoszthatja a kimenő vonalak között. Mindezeket a feladatokat elvégezheti egy hálózati processzor.
11. **Ellenőrző összeg generálása.** A kimenő csomagokat többnyire el kell látni ellenőrző összeggel. Az IP ellenőrző összeget előállíthatja a hálózati processzor, de az Ethernet CRC-t általában hardver számítja ki.
12. **Elszámolási információ gyűjtése.** Bizonyos esetekben szükség van a csomagforgalom adatainak gyűjtésére, különösen, amikor egy hálózat üzleti alapon továbbít csomagokat más hálózatoknak. A hálózati processzor elvégezheti az adatgyűjtést.
13. **Statisztikák készítése.** Végül, sok szervezet szeret statisztikákat készíteni az adatforgalomról, a hálózati processzor megfelelő hely az adatok összegyűjtésére.

A teljesítmény növelése

A teljesítmény a legfontosabb a hálózati processzorok számára. Mit lehet tenni ennek növelése érdekében? De mielőtt növeljük, definiálnunk kell, hogy mi is az. Egy mérték a másodpercenként továbbított csomagok száma. Egy másik a másodpercenként továbbított bájtok száma. Ezek különböznek, és egy kisméretű csomagokhoz megfelelő séma nem biztos, hogy ugyanolyan jól működik nagyméretűre is. Például kis csomagok esetén sokat segíthet, ha gyorsítjuk a célállomások címeinek előkeresését, de nagy csomagok esetén nem biztos.

A teljesítmény növelésének legegyszerűbb módja a hálózati processzor órajelének növelése. Természetesen a teljesítmény nem nő lineárisan az órajellel, mert a memória ciklusideje és még egyéb tényezők is befolyásolják. A nagyobb órajel még több elvezetendő hőt is jelent.

Sokszor célravezetőbb, ha több PPE-t és párhuzamosságot alkalmazunk, főleg egy olyan elrendezésben, ahol a PPE-k működnek párhuzamosan. Egy hosszabb csővezeték is segíthet, de csak akkor, ha egy csomag feldolgozása kisebb részekre bontható.

Egy másik módszer a speciális processzorok vagy ASIC-ok beépítése az olyan ismétlődő és időigényes feladatok elvégzésére, amelyeket hardverrel gyorsabban el lehet végezni, mint szoftverrel. Sok egyéb mellett keresések táblázatokban, ellenőrző összegek képzése és titkosítási algoritmusok lehetnek a jelöltek között.

Több belső sín beépítése, illetve a meglévők szélesítése is segíthet a sebesség növelésben, hiszen a csomagok gyorsabban mozognak a rendszerben. Végül teljesítménynövelésnek számít az is, ha az SDRAM-ot lecseréljük SRAM-ra, de természetesen ennek is ára van.

Természetesen sokkal többet lehetne elmondani a hálózati processzorokról; bővebben lásd (Comer, 2005; Crowley és társai, 2002; Lekkas, 2003; Papaefstathiou és társai, 2004).

8.2.2. Médiaprocesszorok

A társprocesszorok alkalmazásának második területe a nagy felbontású fényképek, hang és mozgókép kezelése. A hagyományos CPU-k nem különösebben jók ezekben a feladatokban, mert az ilyen alkalmazások nagy mennyiségű adatának feldolgozása nagyon nagy műveletigényű. Emiatt néhány mai és a legtöbb jövőbeni PC fel lesz szerelve média-társprocesszorral, amelynek átadhatja a munka jelentős részét.

A Nexperia médiaprocesszor

Ezt az egyre növekvő jelentőségű területet egy példán keresztül fogjuk tanulmányozni: a Philips Nexperia lapkacsalád lesz vizsgálódásunk tárgya, amelynek tagjai többféle órajel-frekvenciával is rendelkezésre állnak. A **Nexperia** egy komplett egylapkás heterogén multiprocesszor a 8.11. ábra szerinti értelemben. Több magot tartalmaz, beleértve egy TriMedia VLIW CPU-t is vezérlőnek, de sok más magot is kép, hang, video és hálózati feldolgozás céljára. Használható önállóan fő processzorként CD-, DVD-, MP3-lejátszóban vagy -felvevőben, tv-készülékben vagy vevőegységben, fényképezőgépben, videokamerában stb. Használható ezen kívül társprocesszorként PC-ben képek vagy audiovizuális adatok feldolgozására. Mindkét konfigurációban a saját kis, valós idejű operációs rendszerét futtatja.

A Nexperiának három funkciója van: bejövő adatáram begyűjtése és tárolása belső adatszerkezetekben, az adatszerkezetek feldolgozása és végül az adatok kiadása a különféle kimeneti eszközöknek megfelelő formátumban. Például, amikor egy PC-t DVD-lejátszónak használunk, akkor a Nexperia programozható úgy, hogy a DVD-lemezről a titkosított, tömörített video-adatáramot olvassa be, ezután dekódolja és bontsa ki, majd a megjelenítő ablaknak megfelelő méretben adja tovább. Mindez folyhat a háttérben, a számítógép fő processzorának nem kell részt vennie benne miután a DVD-lejátszó programot letöltötte a Nexperiába.

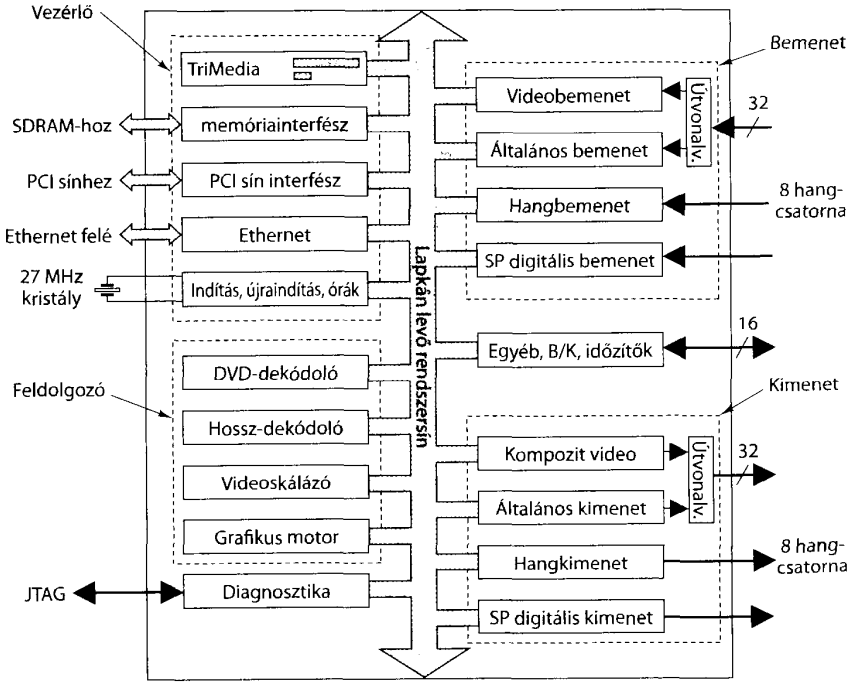
Minden bejövő adat először feldolgozásra a memóriában tárolódik; nincs közvetlen kapcsolat a bementi és a kimeneti eszközök között. Adatok begyűjtése különböző forrásokból is történhet, többféle videoformátum (MPEG-1, MPEG-2 és MPEG-4) és hangformátum (többek közt AAC, Dolby és MP3) mindegyikéből a megfelelő adatszerkezetbe konvertál tárolás és feldolgozás céljából. A bejövő adatok jöhetnek a PCI sínről, Ethernetről vagy kitüntetett inputcsatornákról (például közvetlenül a lapkához csatlakoztatott mikrofonból vagy sztereó rendszerből). A Nexperia lapkának 456 lába van, ezek között vannak olyanok, amelyek közvetlenül bejövő vagy kimenő adatáramok számára állnak rendelkezésre.

Az adatfeldolgozást a TriMedia CPU szoftvere irányítja, amely bármilyen szükséges program lehet. A tipikus feladatok közé tartozik a mozgóképsorozatok élességének növelése „deinterlace"-eléssel, képek fényerejének, kontrasztjának és színeinek javítása, képméret módosítása, különböző videoformátumok közötti konverzió és a zajcsökkentés. Rendszerint a CPU veszi át a munkát, majd a legtöbb részfeladatot kiosztja a lapkán található különböző speciális magoknak.

A kimenettel kapcsolatos funkciók magukba foglalják a belső adatszerkezetek átkódolását a kimeneti eszközöknek megfelelő formátumra, több adatforrás (video, hang, kép, 2D grafika) összefésülését és a kimeneti eszközök vezérlését. Mint ahogy a bemenet esetében is, a kimenő adatok kerülhetnek a PCI sínre, Ethernetre vagy egy kitüntetett kimenő vonalra (például hangszóróra vagy erősítőbe).

A 8.16. ábrán a Nexperia PNX 1500 lapka blokkdiagramja látható. Más verziók ettől kis mértékben eltérnek, így az egységesség kedvéért a fejezet hátralévő részében a „Nexperia" ezt a PNX 1500 implementációt fogja jelenteni. Ennek négy nagyobb része van: vezérlő, bemenet, feldolgozó és kimenet. A CPU a 8.1.1 alfejezetben tárgyalt 32 bites TriMedia VLIW processzor 300 MHz-es változata. Ennek programja (amit általában C-ben vagy C++-ban írnak) határozza meg a Nexperia működését.

A Nexperia a lapkán nem tartalmaz memóriát, kivéve a TriMedia CPU-ban lévő két gyorsítótárat. Ehelyett van külső memóriainterfésze, amelynek segítségével 8–256 MB közötti DDR SDRAM-ot érhet el. Ez a mennyiség bőven elegendő ti-



8.16. ábra. *A Nexperia heterogén multiprocesszor-lapkája*

pikus multimédia-alkalmazásokhoz. A 200 MHz-es órajellel a memória adatátviteli képessége 1,6 GB/sec.

Egy teljes PCI interfész is helyet kapott a lapkán 8, 16 és 32 bites átvitellel 33 MHz-en. Ha fő CPU-ként használják egy szórakoztatóelektronikai eszközben (például DVD-lejátszóban), akkor a PCI interfész sínkiosztóként is üzemelhet. Ez az interfész például egy DVD-meghajtóval való kapcsolattartásra is használható.

Közvetlen Ethernet-kapcsolatot biztosít egy erre a célra elhelyezett mag, amely képes 10 és 100 Mbps sebességre is. Ebből következően egy Nexperia-alapú kamkorder képes digitális videót szolgáltatni Etherneten keresztül egy távoli adatgyűjtő vagy megjelenítő eszköznek.

A következő mag kezeli a rendszerindítást, az újraindítást, az órákat és még néhány apróbb működési jellemzőt. Ha a Nexperia egy bizonyos kivezetésére jel érkezik, akkor elkezdődik az újraindítás. Ezt a magot arra is be lehet programozni, hogy ha a CPU nem küld neki jelet szabályos időközönként, akkor feltételezi, hogy a rendszer lefagyott, és magától kezdeményezi az újraindítást. Önállóan működő berendezésekben az újraindítás flash memóriából történhet.

A magot egy 27 MHz-es kristályoszcillátor hajtja, ez a frekvencia belül 64-gyel szorzódik, így áll elő a lapkán mindenhol használt 1,728 GHz-es jel. Az energiagazdálkodás kezelése is itt történik. Alapesetben a CPU teljes sebességgel, a többi komponens pedig feladatának ellátásához szükséges sebességgel üzemel. A CPU azonban energiamegtakarítás céljából le tudja lassítani az órát. Alvó üzemmód is van, amikor a legtöbb funkció lekapcsolódik, ha nincs semmi tennivaló. Ez a hordozható eszközök akkumulátorát hivatott kímélni.

Ennek az egységnek része még 16 „szemafor", amelyek több eszköz közötti szinkronizáció megvalósítására használhatók. Amikor egy mag nemnulla értéket ír egy éppen nulla értéket tartalmazó szemaforba, akkor az írás sikeres lesz; különben sikertelen, és a korábbi érték változatlan marad. Nulla beírása mindig sikeres. Mivel egyszerre csak egyetlen mag férhet hozzá a rendszersínhez, ezért ezek a műveletek atomiak, így alapját képezhetik a kölcsönös kizárás megvalósításának. Ha egy mag meg akar szerezni egy erőforrást, akkor nemnulla értéket próbál írni egy meghatározott szemaforba. Ha a beírás sikeres, akkor onnantól kezdve kizárólagos hozzáférési joga van az erőforráshoz egészen addig, amíg le nem mond róla úgy, hogy nullát ír a szemaforba. Ha egy írás sikertelen, akkor a magnak rendszeres időközönként újra próbálkoznia kell, amíg sikeres nem lesz. Emiatt természetesen ezek nem a 6. fejezet értelmében vett klasszikus szemaforok.

Most nézzük a bemeneti részt. A videobemeneti mag 10 bit széles digitális video-adatáramot kap, simító algoritmussal átalakítja 8 bitesre, majd a külső DRAM-ban eltárolja. A legtöbb esetben a digitális bemenet egy külső analóg-digitális átalakító kimenetéről érkezik, amely pedig analóg tv-jelet kap hagyományos antennáról vagy kábelről.

Az általános bemeneti mag képes 100 MHz-en strukturálatlan 32 bites adatot gyűjteni tetszőleges bemenetről és az SDRAM-ban eltárolni. Ugyanezt olyan strukturálatlan adatokkal is meg tudja tenni, amelyben a rekordok jelekkel szét vannak választva. A két digitális videobemenet előtt elhelyezett útvonalválasztó demultiplexálást végez, illetve egyúttal képes némi videoátalakítást is elvégezni. A

demultiplexálásra azért van szükség, mert ugyanazokat a kivezetéseket használják általános és videobemenet céljára is.

A hangbemeneti mag képes begyűjteni legfeljebb 8 sztereó hangcsatornán érkező zenét vagy beszédet, és egészen 96 kHz-ig 8, 16 vagy 32 bites pontossággal az SDRAM-ban eltárolni. Tud tömörített formátumokat visszafejteni, csatornákat keverni, a mintavétel sűrűségét megváltoztatni, szűrőket alkalmazni, és mindezt menet közben, mielőtt a hangadatokat eltárolná. Az SP digitális bemeneti mag olyan digitális hangjeleket fogad, amelyek megfelelnek a Sony-Philips digitális hangszabványának (IEC 1937). Így a digitális hanganyagok a minőség romlása nélkül átvihetők egyik eszközről a másikra.

A hang-, video- és más adatok a beolvasás után feldolgozást igényelnek, ezt tárgyalja a következő rész. A kölcsönzött vagy megvásárolt DVD-filmek kódolva vannak a másolást megakadályozandó. A DVD-dekódoló eltávolítja a kódolást, így hozzájuthatunk az eredeti filmhez MPEG-2 formátumban tömörítve. A dekódolás egy memóriából memóriába történő másolási művelet, a bemenet jön az egyik pufferből, a kimenet megy egy másikba.

A hossz-dekódoló még tovább megy, és eltávolítja az MPEG-2 (vagy MPEG-1) által alkalmazott változó hosszúságú kódolást az érintett részekről, ezzel előáll egy részben tömörített adatsorozat a tényleges MPEG feldolgozáshoz, amelyet már a TriMedia végez szoftverrel. Ennek a felosztásnak az az oka, hogy a változó hosszúságú dekódolás (Huffmann- és futóhossz-dekódolás) nem használja ki hatékonyan a TriMedia képességeit, ezért jobbnak ítélték hardverrel megoldani néhány négyzetmilliméter szilikont felhasználva erre a célra. Ezek a memóriából memóriába dolgozó műveletek egy egyszerű pixeltérképet hoznak létre.

Egy pixeltérkép három általános formátum bármelyikében lehet, ezek mindegyikének további három vagy négy alfaja van a különböző méretek és paraméterek szerint. Az első formátum az **indexelt színelőállítás**, amelynél minden érték egy **CLUT (Color Look Up Table, színkereső tábla)** index. A táblázat minden bejegyzése egy 24 bites színértékből és egy 8 bites **áttetszőségi csatorna maszk**ból áll, utóbbit akkor használják, amikor több réteg kerül egymásra. A második formátum az RGB, így működnek a számítógép-monitorok is, minden pixelt szétválasztva a vörös, zöld és kék komponensekre. A harmadik formátum az YUV, amelyet a televíziózásban használt jelek kódolására kifejlesztettek ki. A vörös, zöld és kék komponensek külön-külön kódolása helyett a kamerában egy olyan transzformációt hajtanak végre, amely egy fényerősség- és két színcsatornát hoz létre. Ezzel a megoldással a színnél nagyobb sávszélesség rendelhető a fényerősséghez, ez viszont nagyobb zajtűrő képességet jelent adatátvitel során. Az YUV formátum jó választás olyan alkalmazás esetén, amely televíziós jeleket fogad és állít elő. A tárolható formátumok számának korlátozása lehetővé teszi, hogy minden mag be tudja olvasni bármelyik másik kimenetét.

A videoskálázó egy skálázási feladatlistát fogad és maximálisan 120 millió pixel/sec sebességgel végrehajtja. Ezek a feladatok az alábbiakat tartalmazhatják:

1. Deinterlacing;
2. Vízszintes és függőleges skálázás;

3. Lineáris és nemlineáris oldalarány-konverzió;
4. Különböző pixelformátumok közötti konverzió;
5. Fényerősség hisztogram adatgyűjtés;
6. Villogáscsökkentés.

A televízió-adások **váltottsoros** jelekből állnak, ami azt jelenti, hogy minden 525 (PAL és SECAM esetén 625) letapogatási sorból álló képkocka esetén először a páros sorok, majd a páratlan sorok kerülnek továbbításra. A deinterlacing sorduplázó technika egy jobb minőségű **progresszív pásztázást** eredményez, ilyenkor minden sor a maga helyén kerül továbbításra, és a váltottsoroshoz képest kétszer akkora frekvenciával frissül (29,97 fps NTSC-nél, 25 fps PAL-nál és SECAM-nál). A vízszintes és függőleges skálázással a képek méretét lehet növelni vagy csökkenteni, esetleg egy rész kivágása után. A szabványos televíziókép szélességének és magasságának aránya 4:3, de a széles képernyős tévénél az arány 16:9, ami jobban illik a 35 milliméteres mozifilmek 3:2 arányához. A skálázó konvertálni tud a különböző oldalarányok között lineáris vagy nemlineáris algoritmussal. Szintén tud konvertálni az indexelt, az RGB és az YUV formátumok között, valamint képes hisztogramot készíteni a fényességértékből, ami hasznos a kimenet minőségének javítása szempontjából. Végül vannak bizonyos transzformációk, amelyek a képvillogást csökkentik.

A grafikus motor kétdimenziós képelőállítást végez az objektumleírások alapján. Ki tud tölteni zárt alakzatokat, és **bitblt** műveleteket is elvégez, ami lényegében azt jelenti, hogy két pixel téglalapot összekombinál logikai ÉS, VAGY, kizáró VAGY vagy más Boolean-függvény segítségével.

Nincs hangfeldolgozást végző mag. Ami a beolvasás után még hátravan, azt a TriMedia CPU végzi el szoftverrel. A hang annyira kevés adatot igényel, hogy a szoftveres feldolgozás nem jelent problémát. Sok alkalmazás egyáltalán nem is igényel hangfeldolgozást, esetleg csak formátumkonverziót.

A diagnosztikai mag segíti a tervezőket és a programozókat a hardver- és a szoftverhibák felderítésében. Interfészt biztosít az IEEE 1149.1 számú, más néven **JTAG (Joint Test Action Group)** szabványnak megfelelő műszerek és eszközök számára.

A kimeneti rész a feldolgozott adatokat a memóriából a kimenetre küldi. A kompozit video mag vesz néhány pixeleket reprezentáló adatszerkezetet, bizonyos módon normalizálja, majd meghatározott módon keveri, mielőtt az eredmény a kimenetre kerül. Az indexelt formátumú adatok alapján menet közben megállapítja a konkrét pixeleket, valamint a nem kompatibilis formátumokat megfelelően konvertálja. Ez a mag kontrasztot, fényességet és színkorrekciót is tud állítani, ha szükséges. Képes még a chroma keying effektusra, amikor egy teljesen kék háttér előtt álló színészt külön választanak a háttértől, és vizuálisan egy másik forrásból származó háttér elé helyeznek. Hasonló módon, rögzített vagy valamilyen irányba haladó háttér előtt mozgó figurákat tartalmazó rajzfilmek is előállíthatók. Természetesen a végeredményt a kívánt video- vagy televíziós szabvány szerinti formátumba konvertálja (például NTSC, PAL vagy SECAM), beleértve a vízszintes és függőleges szinkronjeleket is.

Mivel nem jelent plusz költséget, a legtöbb Nexperia alapú rendszer minden bizonnyal automatikusan kezelni fogja mind a három tv-szabványt, így a világ bármely pontján eladhatók lesznek. Hasonlóképpen, a **HDTV (High Definition TeleVision, nagy felbontású televízió)** hozzáadása a készlethez csak egy olyan kis szoftver hozzáadásából áll, amely a formátum és a memóriában lévő adatszerkezetek közötti konverziót végzi.

Az általános kimeneti mag csak biteket mozgat, 8, 16 vagy 32 bitet ciklusonként 100 MHz frekvencián, ez 3,2 Gbps maximális sávszélességet jelent. Ha egy Nexperia lapka általános kimentét összekötjük egy másik Nexperia lapka általános bemenetével, akkor a fájlok mozgatása a gigabit Ethernet sebességét (1 Gbps) meghaladó ütemben végezhető. Ez az interfész lehetővé teszi a CPU számára azt is, hogy szoftverrel bármilyen típusú kimenetet előállítson.

A kimeneti útvonalválasztó multiplexálja a két kimenő forrást, és némi funkcionalitást is ad hozzá, ideértve a maximum 1280 × 768 pixel felbontású folyadékkristályos kijelzők frissítését 60 Hz-en, vagy váltottsoros, illetve progresszív tévékészülékek frissítését is. A multiplexálásra azért van szükség, mert a kompozit video és az általános kimenetek ugyanazokon a kivezetéseken osztoznak.

A hangkimeneti mag elő tud állítani akár 8 sztereó csatornát 32 bites pontossággal és akár 96 kHz-es mintavételi sebességgel. Gyakran ez a kimenet egy külső digitális-analóg átalakítót hajt meg. Az SP digitális kimenet a Sony-Philips digitális hangszabványának megfelelő eszközök SP digitális bemenetéhez csatlakoztatható.

Az utolsó mag az általános célú B/K kezelő. Tizenhat kivezetés áll rendelkezésre bármilyen szükséges célra. Nyomógombokhoz, kapcsolókhoz, LED-ekhez csatlakoztathatók, és így ezek szoftverből érzékelhetők, illetve vezérelhetők. Még közepes sebességű (20 Mbps) szoftvervezérelt hálózati protokollokhoz is használhatók. Különböző időzítők, számlálók és eseménykezelők is vannak még itt.

Mindent egybevetve a Nexperia az audiovizuális alkalmazások számára hatalmas számítási kapacitással rendelkezik, és a hálózati processzorokhoz hasonlóan lehetővé teszi, hogy tekintélyes mennyiségű munkától szabaduljon meg a CPU. A számítási kapacitás még nagyobb, mint elsőre tűnhet, mert a magok egymással és a CPU-val is párhuzamosan működhetnek. És talán meglepő, de nagy tételben 20 dollár alatti áron kapható. Mostanra a társprocesszorok hasznosságának, különösen a heterogén multiprocesszoros lapkákon alapulók hasznosságának egyre nyilvánvalóbbá kellene válnia. Egy hasonló, de a multimédia helyett a telefónia felé orientált lapka leírása található (Nickolls és társai, 2003).

8.2.3. Kriptoprocesszorok

A biztonság, elsősorban a hálózati biztonság a harmadik terület, ahol a társprocesszorok népszerűek. Amikor egy kliens és egy szerver között kapcsolat jön létre, először sok esetben hitelesíteniük kell egymást. Ezután egy biztonságos, titkosított kapcsolatot kell felépíteni kettejük között, hogy az egymásnak küldött adatok biztonságban legyenek akkor is, ha a kommunikációt lehallgatják.

A biztonsággal az a probléma, hogy titkosítást kell használni, ami viszont nagyon számításigényes. A titkosításnak két fajtája van, a **szimmetrikus kulcsú titkosítás** és a **nyilvános kulcsú titkosítás**. Az első azon alapszik, hogy a biteket nagyon alaposan összekeverik, nagyjából olyan, mintha az üzenetet a turmixgép elektronikus megfelelőjébe dobnánk. Az utóbbi nagy (például 1024 bites) számok szorzásán és hatványozásán alapszik, és rendkívül időigényes.

Adatátviteli vagy tárolási célból történő titkosításhoz a szükséges számítások elvégzésére különböző cégek gyártottak kriptográfiai társprocesszorokat, néha PCI-kártyák formájában. Ezeknek a társprocesszoroknak speciális hardvere van, hogy a szükséges titkosítási műveleteket a hagyományos CPU-knál sokkal gyorsabban tudják elvégezni. Sajnos e társprocesszorok működésének tárgyalásához először magáról a kriptográfiáról kellene hosszasan értekeznünk, ez pedig meghaladja ennek a könyvnek a kereteit. A kriptográfiai társprocesszorokról bővebb információ található (Daneshbeh és Hasan, 2004; Lutz és Hasan, 2004).

8.3. Közös memóriás multiprocesszorok

Láttuk, hogy a párhuzamosság a lapkákon is megvalósítható, illetve társprocesszor hozzáadásával önálló rendszerekben is megjelenhet. A következő szint az, hogy több komplett CPU-t egy rendszerbe összeépítünk. Az ilyen több CPU-val rendelkező rendszerek lehetnek multiprocesszorok vagy multiszámítógépek. Miután megvizsgáltuk, hogy ezek a kifejezések tulajdonképpen mit is jelentenek, először a multiprocesszorokat, majd a multiszámítógépeket tanulmányozzuk.

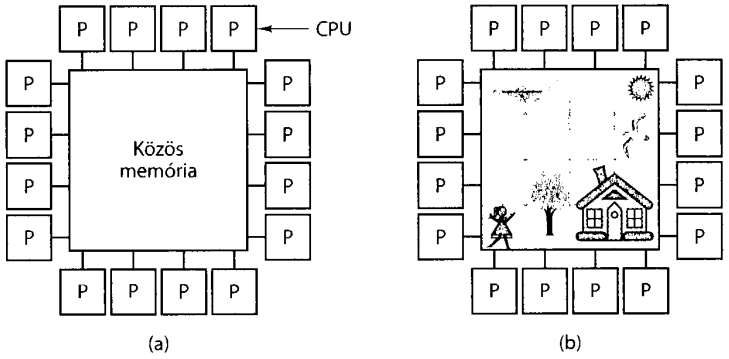
8.3.1. Multiprocesszorok és multiszámítógépek

Bármely párhuzamos számítógéprendszerben a feladat különböző részein dolgozó CPU-knak kommunikálniuk kell ahhoz, hogy információt adhassanak át egymásnak. Hogy ez pontosan milyen módon történjen, arról komoly viták folynak az architektúrákkal foglalkozók körében. Két különböző megközelítést javasoltak és valósítottak meg, a multiprocesszort és a multiszámítógépet. A kettő közötti alapvető különbség az, hogy van-e közös memória, vagy nincs. Ez a különbség kihat ezeknek a gépeknek a tervezésére, megvalósítására, programozására, valamint a méretükre és az árukra is.

Multiprocesszorok

A **multiprocesszor** olyan számítógép, amelyben az összes CPU egy közös memórián osztozik (lásd 8.17. ábra). A multiprocesszoron egyszerre futó folyamatok mindegyike egy közös logikai címtartományt lát, amely a közös fizikai memóriára van leképezve. Bármelyik folyamat írhat vagy olvashat egy tetszőleges memóriaszót, ehhez csak egy STORE vagy LOAD utasítást kell végrehajtania. Semmi más nem kell, a hardver elvégzi a többit. Két folyamat úgy kommunikálhat egymással, hogy egyikük adatokat ír a memóriába, a másik pedig kiolvassa onnan.

A multiprocesszorok népszerűségének oka az, hogy két (vagy több) folyamat egyszerűen a memóriába való írás és olvasás útján kommunikálhat egymással. Ezt a modellt a programozók könnyen megértik, és a problémák széles körében alkalmazható. Tekintsük például azt a programot, amely megvizsgál egy bittérképes képet, és kilistázza a rajta található objektumokat. A képet a memóriába töltjük, ahogy a 8.17. (b) ábrán látható. Mind a 16 processzoron egy-egy folyamat fut, ezek mindegyike megvizsgál a 16 képrészletből egyet. Mindemellett a processzorok elérhetik a teljes képet, amire szükség is van, hiszen egy objektum több képrészletbe is belelóghat. Ha valamelyik folyamat azt veszi észre, hogy egyik objektuma átlóg a szomszédos részbe, akkor egyszerűen követi azt, szavakat olvasva a másik képrészletből is. Ebben a példában lesznek olyan objektumok, amelyeket több folyamat is felfedez, ezért a végén valamilyen egyeztetésre van szükség, hogy ténylegesen hány ház, fa és repülőgép van a képen.



8.17. ábra. *(a) Egy multiprocesszor, amelyben 16 CPU osztozik a közös memórián. (b) Egy 16 részre osztott kép, mindegyiket külön CPU vizsgálja*

Mivel a multiprocesszorban minden processzor ugyanazt a memóriatartalmat látja, ezért operációs rendszerből is csak egy példány van. Következésképpen csak egyetlen laptáblázat és egyetlen folyamattáblázat van. Amikor egy folyamat blokkolódik, a CPU-ja elmenti az állapotát az operációs rendszer táblázataiba, majd ugyanott keres egy másik futtatható folyamatot. A multiprocesszort ez az egypéldányos operációs rendszer különbözteti meg a multiszámítógéptől, amelyben minden számítógépnek saját példánya van az operációs rendszerből.

Mint minden számítógépnek, egy multiprocesszornak is kellenek B/K egységek, mint például lemezek, hálózati csatolók és más berendezések. Némelyik multiprocesszoros rendszerben csak néhány CPU fér hozzá a B/K egységekhez, és így ezeknek speciális B/K funkciójuk van. Más rendszerekben minden CPU egyformán hozzáfér minden B/K eszközhöz. Ha minden CPU egyformán hozzáfér minden

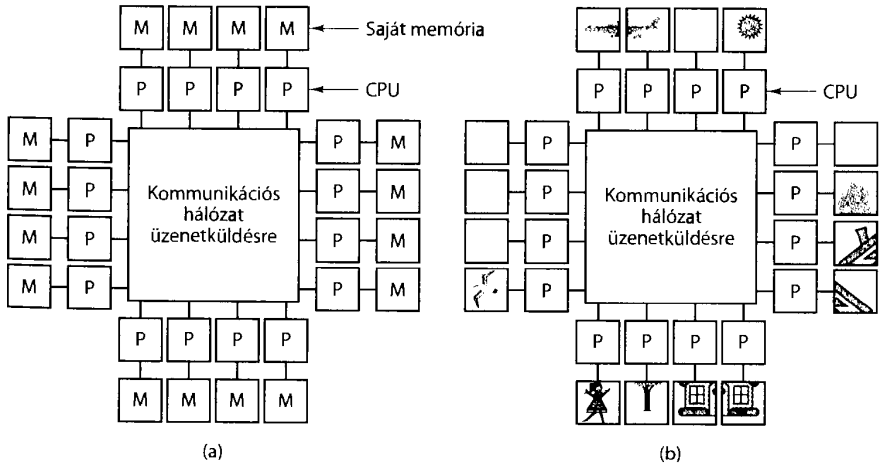
memóriamodulhoz, minden B/K eszközhöz, és az operációs rendszer egyenrangúnak tekinti őket, akkor a rendszert **SMP**-nek **(Symmetric Multiprocesszor, szimmetrikus multiprocesszor)** nevezik.

Multiszámítógépek

A másik lehetséges párhuzamos architektúrában minden egyes CPU-nak saját memóriája van, amelyet csak saját maga érhet el, a többi CPU nem. Az ilyen architektúrát **multiszámítógép**nek vagy néha **osztott memóriájú rendszer**nek nevezik és a 8.18. (a) ábrán látható. A multiprocesszorok és a multiszámítógépek között az a fő különbség, hogy a multiszámítógép minden CPU-jának saját lokális memóriája van, amelyet LOAD és STORE utasításokkal elérhet, de amelyet más CPU-k nem érhetnek el LOAD és STORE utasításokkal. Tehát míg a multiprocesszoros rendszerben egy fizikai címterület van, és ezen osztozik az összes CPU, addig a multiszámítógépes rendszereknél minden egyes CPU-hoz külön fizikai címtartomány van rendelve.

Mivel a CPU-k egy multiszámítógépes rendszerben nem tudnak egymással a közös memórián keresztül érintkezni, ezért más lehetőséget kellett kidolgozni a kommunikációjukra. A folyamatok üzeneteket küldhetnek és fogadhatnak az összekötő hálózaton keresztül. Példák a multiszámítógépes rendszerekre az IBM-BlueGene/L-, a Red Storm- és a Google-klaszter.

A multiszámítógépeknél a közösen használható hardveres memória hiányának fontos következménye van a szoftver szerkezetére nézve. Multiszámítógépen nem lehet egyetlen virtuális címtartományt létrehozni, amelyet minden processzus egyszerűen LOAD és STORE utasítással elérhetne. Például, ha a 0-s CPU (bal felső



8.18. ábra. *(a) Egy multiszámítógép 16 CPU-val, mindegyiknek saját memóriája van. (b) A 8.17. ábra bittérképe a 16 memória között szétosztva*

sarokban) a 8.17. (b) ábrán észreveszi, hogy az általa vizsgált tárgy belóg az 1-es CPU-hoz rendelt területre, akkor folytathatja a memória olvasását, hogy a repülőgép farokrészéhez hozzáférjen. Másrészről, ha a 8.18. (b) ábrán a 0-s CPU jut hasonló felismerésre, akkor ez nem tud az 1-es CPU memóriájából olvasni. Valami egészen mást kell tennie, hogy a szükséges adatokhoz hozzájusson.

Az adott esetben először (valahogyan) ki kell deríteni, hogy melyik CPU birtokolja a szükséges adatot, és küldeni kell neki egy üzenetet, amelyben kérni kell az adatok másolatát. Általában a folyamat ezt követően blokkolódik (azaz várakozik) a válasz megérkezéséig. Amikor az üzenet megérkezik az 1-es CPU-hoz, akkor azt szoftverrel fel kell dolgozni, majd a kért adatokat elküldeni. Amikor a válasz megérkezik a 0-s CPU-hoz, akkor a szoftver blokkoltsága megszűnik, és tovább folytathatja a működését.

Multiszámítógépekben a folyamatok közötti kommunikációban gyakran olyan szoftver primitíveket használnak, mint a send és a receive. Ezáltal az itt alkalmazott szoftver más és bonyolultabb szerkezetű, mint a multiprocesszoros rendszereknél. Ez azt is jelenti, hogy a multiszámítógépeknél nagy probléma a feldolgozandó adatok helyes szétosztása, és azok optimális elhelyezése. Ennek kisebb jelentősége van a multiprocesszoros rendszereknél, mivel ott az elhelyezés nem befolyásolja a helyes működést vagy a programozhatóságot, bár a teljesítményt befolyásolhatja. Röviden, a multiprocesszoros rendszereken sokkal egyszerűbb a programozás, mint a multiszámítógépes rendszereken.

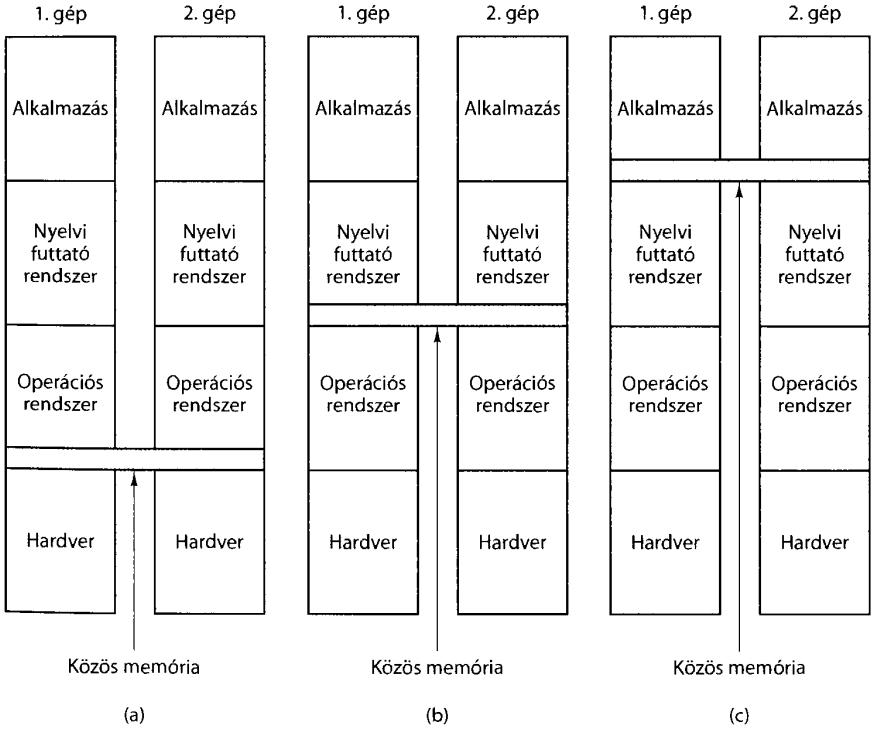
Ilyen körülmények között miért építenek mégis multiszámítógépeket, amikor a multiprocesszoros rendszereken könnyebb programozni? A válasz egyszerű: egy nagy multiszámítógépet sokkal könnyebb és olcsóbb megépíteni, mint egy ugyanannyi CPU-val rendelkező multiprocesszoros rendszert. Igen nagy kihívás olyan memóriát megvalósítani, amelyet néhány száz CPU közösen használ, míg egy 10000 vagy ennél több CPU-s multiszámítógép megépítése könnyen megoldható feladat. A fejezet későbbi részében tanulmányozunk egy olyan multiszámítógépet, amely 50000-nél is több CPU-t tartalmaz.

Így most egy dilemma előtt állunk: multiprocesszoros rendszereket nehéz építeni, de könnyű programozni, míg multiszámítógépeket könnyű építeni, de nehéz programozni. Ez a megfigyelés ahhoz vezetett, hogy nagy erőfeszítéssel olyan kevert rendszereket próbáltak építeni, melyek viszonylag könnyen megépíthetők, és viszonylag könnyen programozhatók. A munka során rájöttek arra, hogy a közös memória megvalósítása többféleképpen történhet, persze ezek mindegyikének van előnye és hátránya is. Valójában a párhuzamos architektúrával kapcsolatos kutatások napjainkban azt célozzák, hogy a multiprocesszoros és a multiszámítógépes szemléletet közelítsék egymáshoz, létrehozva egy köztes, kevert formát, amely ötvözi mindkét alaptípus előnyeit. A célkitűzés az, hogy olyan rendszerfelépítés valósuljon meg, amely **skálázható**, azaz a CPU-k hozzáadásával a teljesítmény a megfelelő arányban növekedjen.

A hibrid rendszerek építésének egyik megközelítése azon a tényen alapszik, hogy a modern számítógépes rendszerek nem monolitikusak, hanem rétegekből állnak – könyvünk éppen ezt tárgyalja. Ez megnyitja a lehetőséget arra, hogy a közös memóriát több réteg valamelyikében megvalósíthassuk, ahogyan a 8.19. ábrán

látható. A 8.19. (a) ábrán egy hardveres közös memória megvalósítása látható; ez egy igazi multiprocesszoros rendszer. Ebben a felépítésben egyetlen operációs rendszer van egyetlen táblázathalmazzal, közöttük a memóriafoglaltsági táblával. Amikor egy folyamatnak memóriára van szüksége, szól az operációs rendszernek, amely a foglaltsági tábla alapján keres egy szabad memórialapot, majd ezt a lapot hozzárendeli a hívó címterületéhez. Ami az operációs rendszert illeti, egyetlen memóriája van, és szoftverrel folyamatosan nyomon követi, hogy ennek melyik lapját melyik folyamat birtokolja. Többféle mód van a hardveres közös memória megvalósítására, ahogy később látni fogjuk.

A második megközelítés multiszámítógép hardvert alkalmaz, és olyan operációs rendszerrel rendelkezik, amelyik szimulálja a közös memóriát úgy, hogy egy lapozott közös virtuális címtartományt kezel. Ennél a megközelítésnél, amelyet **DSM-nek (Distributed Shared Memory, (el)osztott közös memória)** (Li és Hudak, 1989) neveznek, mindegyik lap a 8.18. (a) ábra memóriáinak valamelyikén van. Minden egyes gépnek saját virtuális memóriája és saját laptáblája van. Amikor egy CPU egy olyan LOAD vagy STORE utasítást hajt végre, amely egy másik CPU lapjára vonatkozik, akkor az operációs rendszer aktivizálódik. Az operációs rendszer megkeresi a lapot, megkéri az azt pillanatnyilag birtokló CPU-t, hogy válassza le a lapot a virtuális memóriájáról és küldje el az összekötő hálózaton. Megérkezés után a CPU a lapot magához kapcsolja, és a laphiányt kiváltó utasítást újraindítja. Valójában annyi történik, hogy az operációs rendszer a lapozási hiányt távoli memóriából elégíti ki, nem pedig lemezről. A felhasználó számára a gép úgy néz ki, mintha közös memóriája lenne. A DSM vizsgálatára később visszatérünk.



8.19. ábra. *A közös memória megvalósításainak különböző szintjei. (a) Hardverszint. (b) Operációs rendszer szintje. (c) Nyelvi futtató rendszer szintje*

A harmadik lehetőség a közös memória felhasználói szintű (esetleg nyelvfüggő) megvalósítása. Ebben a megközelítésben a programozási nyelv bizonyos közös memóriás absztrakciókat kínál fel, melyet aztán a fordító és a futtató rendszer valósít meg. Például a Linda modell a közös adategységtér (adatmezőket tartalmazó rekordok) absztrakcióján alapszik. Bármely gépen futó folyamat bekérhet adategységeket a közös adategységtérből, vagy írhat oda adategységeket. Mivel a közös adategységtér elérésének irányítása szoftveres (a Linda futtató rendszere végzi), semmilyen speciális hardver- vagy operációsrendszer-támogatás nem szükséges hozzá.

A közös memória nyelvfüggő futtató rendszer általi megvalósításának egy másik példája a közös adatobjektumokat használó Orca modell. Az Orcában a folyamatok rekordok helyett általános objektumokat osztanak meg egymással, és ezek metódusait is futtathatják. Ha egy metódus megváltoztatja egy objektum belső állapotát, akkor a futtató rendszer feladata, hogy az összes gépen az objektum összes másolatát egyidejűleg frissítse. Mivel az objektumok kizárólag szoftveres absztrakciók, ezért megvalósításukat a futtató rendszer végzi a hardver és az operációs rendszer támogatása nélkül. Később foglalkozunk még a Linda- és az Orca-rendszerrel ebben a fejezetben.

Párhuzamos számítógépek osztályozása

Térjünk most vissza a fejezet fő témájához, a párhuzamos számítógépek architektúrájához. Az évek során számos különböző párhuzamos számítógépet terveztek és építettek, így természetes igényként merül fel, hogy valamiképpen rendszerbe soroljuk őket. Sok kutató próbálkozott már ezzel a feladattal, meglehetősen vegyes eredménnyel (Flynn, 1972; Treleaven, 1985). Sajnos a párhuzamos számítógépek Carolus Linnaeusa* még nem bukkant fel. Az egyetlen gyakran használt séma a Flynn-féle rendszer, és bár ez a legjobb, mégis csak nagyon kezdetleges megközelítést ad. A 8.20. ábrán ez látható.

Flynn osztályozása két fogalomra épül, ezek az utasításáram és az adatáram. Az utasításáram a programszámlálónak (utasításszámlálónak) felel meg. Egy n darab CPU-val rendelkező rendszernek n darab programszámlálója, és így n darab utasításárama van.

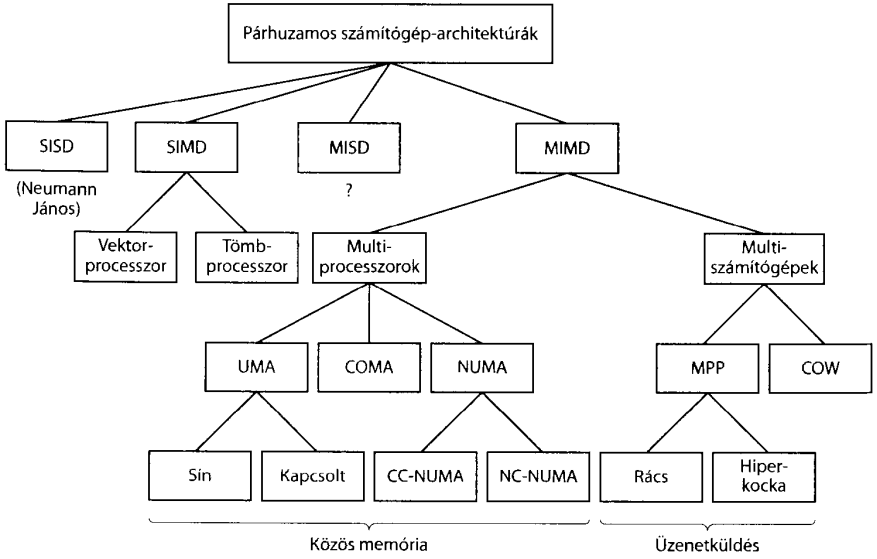
* (1707–1778) svéd biológus, aki kitalálta azt a ma is használatos rendszert, mely az állatokat és növényeket törzsekbe, osztályokba, rendekbe, nemzetségekbe és fajokba sorolja.

Utasításáram	Adatáram	Elnevezés	Példák
1	1	SISD	Klasszikus Neumann-féle gép
1	Többszörös	SIMD	Vektor szuperszámítógép, tömbprocesszor
Többszörös	1	MISD	Létezése vitatható
Többszörös	Többszörös	MIMD	Multiprocesszor, multiszámítógép

8.20. ábra. *Párhuzamos számítógépek Flynn-féle osztályozása*

Egy adatáram egy operandushalmazból áll. A már korábban látott hőmérséklet-kiszámítási példában több adatáram is van, minden érzékelőhöz egy.

Az utasítás- és adatáramok bizonyos mértékig függetlenek egymástól, ezért ahogyan a 8.20. ábrán látható, négy kombináció létezik. A SISD a klasszikus, szekvenciális Neumann-féle számítógép. Egy utasításárama és egy adatárama van, egyszerre egy műveletet végez el. A SIMD gépek egyetlen vezérlőegységgel rendelkeznek, amely egyszerre egy utasítást ad ki, de több ALU-juk van, amelyek a kiadott utasítást több adathalmazon párhuzamosan végzik el. Az ILLIAC IV (lásd 2.7. ábra) a SIMD gépek mintapéldánya. Egyre ritkábbak a SIMD nagygépek, de a hagyományos számítógépeknek néha vannak SIMD-utasításaik audiovizuális anyagok feldolgozására. A Pentium SSE utasítások SIMD-k. Mindazonáltal van egy új terület, ahol a SIMD-terület ötletei fontosak: a stream- (adatáram-) processzorok. Ezeket a gépeket multimédia-anyagok lejátszására tervezik, és még fontos szerephez juthatnak a jövőben (Kapasi és társai, 2003).



8.21. ábra. *A párhuzamos számítógépek osztályozása*

A MISD gépek kategóriája bizonyos tekintetben furcsa, minthogy erre az a jellemző, hogy több utasítás dolgozik ugyanazon adatrészen. Nem teljesen világos, hogy egyáltalán létezik-e ilyen gép, bár egyesek a csővezetékes gépeket MISD gépeknek tekintik.

Végül az utolsó csoport a MIMD gépek csoportja. A MIMD több független CPU egysége, ezek a CPU-k egy nagy rendszer részeiként működnek. A legtöbb párhuzamos számítógép ebbe a kategóriába tartozik. Mind a multiprocesszorok, mind a multiszámítógépek MIMD gépek.

Flynn rendszerezése itt megáll, de mi kiterjesztettük azt a 8.21. ábrán látható módon. A SIMD csoportot két részcsoportra osztottuk. Az egyik részcsoportba a numerikus szuperszámítógépek és más olyan gépek tartoznak, melyek vektorokkal dolgoznak úgy, hogy azonos műveletet végeznek minden vektorelemen. A második részcsoportba olyan párhuzamos jellegű gépek tartoznak, mint az ILLIAC IV, amelyben egy fő vezérlőegység adja ki az utasításokat több független ALU-nak.

A mi rendszerünkben a MIMD két részre van osztva, ezek a multiprocesszorok (közös memóriájú gépek) és a multiszámítógépek (üzenetátadásos gépek). Három multiprocesszor típus van aszerint, hogy a közös memóriát hogyan valósítják meg: **UMA (Uniform Memory Access, egységes memóriaelérés)**, **NUMA (NonUniform Memory Access, nem egységes memóriaelérés)** és **COMA (Cache Only Memory Access, gyorsítótáras memóriaelérés)**. Ennek a csoportbontásnak az az oka, hogy a nagy multiprocesszorokban a memória rendszerint több modulra osztott. Az UMA tulajdonsága, hogy mindegyik CPU mindegyik memóriamodult azonos idő alatt éri el. Más szóval, minden memóriaszót azonos gyorsasággal tud olvasni, függetlenül attól, hogy melyik modulban van. Ha ennek a megvalósítása technikailag lehetetlen, akkor a leggyorsabb hivatkozásokat annyira lelassítják, hogy azonosak legyenek a leglassabbakkal, és így a programozók nem érzékelnek közöttük semmi különbséget. Ezt jelenti itt az „egységes" jelző. Ez az egységesség kiszámíthatóvá teszi a folyamatokat, ami fontos tényező a hatékony kód írásánál.

Ezzel szemben a NUMA típusú multiprocesszorok nem rendelkeznek ezzel a tulajdonsággal. Rendszerint mindegyik CPU-hoz közel van egy-egy memóriamodul, így ennek elérése sokkal gyorsabb, mint a távoli moduloké. Emiatt a hatékonyság szempontjából lényeges, hogy a kód és az adat hol helyezkedik el. A COMA gépek sem egységesek, de másképpen. A későbbiekben részletesen tárgyalunk minden típust és azok alcsoportjait is.

A MIMD gépek másik fő csoportja a multiszámítógépek csoportja, amelyek a multiprocesszorokkal ellentétben nem rendelkeznek architekturális szinten elsődleges közös memóriával. Más szóval, egy multiszámítógép egyik CPU-ján működő operációs rendszer nem érheti el egy másik CPU memóriáját egy LOAD utasítás végrehajtásával. El kell küldenie egy üzenetet, és meg kell várnia a választ. A multiprocesszorokat éppen az különbözteti meg a multiszámítógépektől, hogy az operációs rendszerük képes egy egyszerű LOAD utasítás végrehajtásával beolvasni egy tetszőleges szót. Mint korábban már megjegyeztük, jóllehet egy multiszámítógépen a felhasználói programok képesek lehetnek LOAD és STORE utasítások alkalmazásával távoli memória elérésére, de ez csak látszólagos tulajdonságuk, amelyet az operációs rendszer támogat, nem pedig a hardver. Ez a különbség ugyan nem

tűnik nagynak, de nagyon lényeges. Mivel a multiszámítógépeknek nincs közvetlen hozzáférésük a távoli memóriákhoz, ezért ezeket néha **NORMA (NO Remote Memory Access, távoli memória elérése nélküli)** gépeknek hívják.

A multiszámítógépek durván két csoportba sorolhatók. Az első csoport tartalmazza az **MPP**-ket (**Massively Parallel Processors, erősen párhuzamos processzorok**), az igen drága szuperszámítógépeket, amelyekben a nagyszámú CPU közötti szoros kapcsolatot a gyártó által kifejlesztett nagy sebességű összekötő hálózat biztosítja. A kereskedelmi forgalomban kapható IBM SP/3 egy jól ismert példa.

A másik csoport tartalmazza a szokásos PC-kből vagy munkaállomásokból álló, esetleg egy közös szekrényben elhelyezett és a kereskedelemben kapható összekötési technológiát alkalmazó hálózatokat. Elméletileg a két csoport között nincs nagy különbség, de a sokmillió dollárba kerülő szuperszámítógépeket más feladatokra használják, mint a felhasználó által az MPP-k árának töredékéből összeállított PC-hálózatokat. Ezeknek a házilag gyártott gépeknek több különböző nevük van, ilyen a **NOW (Network of Workstations, munkaállomások hálózata)** és a **COW (Cluster of Workstations, munkaállomások klasztere)**, vagy néha csak **klaszter**.

8.3.2. Memóriaszemantika

Bár az összes multiprocesszor egyetlen közös címtartományt tár a CPU-k elé, gyakran sok memóriamodul van jelen, amelyek mindegyike a fizikai memória bizonyos részét tartalmazza. A CPU-k és a memóriák gyakran bonyolult összekötő hálózattal vannak összekapcsolva, mint ahogy a 8.1.2 részben ezt már láttuk. Több processzor kísérelhet meg egy időben kiolvasni egy szót a memóriából, míg más processzorok megpróbálhatják írni ugyanazt a szót, sőt a kérelmek megelőzhetik egymást menet közben, és a beérkezési sorrend eltérhet a kérések kiadásának sorrendjétől. Ráadásul a memória bizonyos blokkjainak többszörös jelenléte (például gyorsítótárakban) könnyen káoszhoz vezethet, hacsak nem léptetnek érvénybe szigorú rendszabályokat ezek elkerülésére. Ebben a részben megnézzük, hogy mit jelent valójában a közös memória és a memóriák hogyan tudnak ennek a feladatnak megfelelni.

Az egyik értelmezése szerint a memóriaszemantika egy szerződés a szoftver és a fizikai memória között (Adve és Hill, 1990). Ha a szoftver elfogadja, hogy bizonyos szabályokat betart, akkor a memória vállalja, hogy átad bizonyos eredményeket. Most azt kell megnézni, hogy mik ezek a szabályok. Az ilyen szabályokat **konzisztenciamodell**eknek hívják, amelyekből sokfélét javasoltak és valósítottak már meg.

A következő példa megvilágítja a problémát. Tegyük fel, hogy a 0-s CPU 1-et ír egy bizonyos memóriaszóba, majd egy kicsivel később az 1-es CPU 2-est ír ugyanabba a szóba. Most a 2-es CPU olvassa a szót, és 1-et kap eredményül. Vajon a tulajdonosnak ekkor a számítógépét meg kellene javíttatni egy szervizben? Ez attól függ, hogy a memória mit vállalt fel (mi áll a szerződésben).

Szigorú konzisztencia

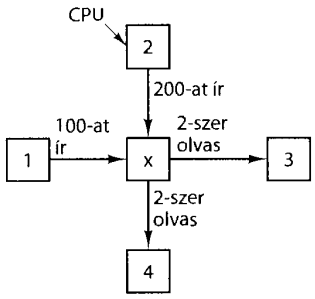
A **szigorú konzisztencia** a legegyszerűbb modell. Ez a modell tetszőleges x hely olvasása esetén, mindig a legutolsóként x-be írt értéket adja vissza. A programozók szeretik ezt a modellt, de ennek megvalósítása szinte lehetetlen másként, mint egyetlen olyan modullal, amely egyszerűen érkezési sorrendben szolgálja ki a kéréseket, bármiféle gyorsítótárazás vagy adattöbbszörözés nélkül. Mivel ez a megvalósítás olyan memóriát eredményez, amely egy hatalmas útszűkülethez hasonló, ezért sajnos ez a modell nem lehet komoly jelölt.

Soros konzisztencia

A **soros konzisztencia** modell (Lamport, 1979) már egy fokkal jobb. Az alapgondolata, hogy ha több olvasási és írási kérés érkezik, akkor a hardver (nemdeterminisztikusan) választ egy sorrendet, de minden CPU ugyanazt a sorrendet látja.

Lássuk egy példán keresztül, hogy mit jelent ez. Tegyük fel, hogy az 1-es CPU a 100-as értéket írja az x szóba, 1 ns múlva a 2-es CPU a 200-as értéket írja az x szóba. Most tegyük fel, hogy a második írás kiadása után (de lehet, hogy még a teljesítése előtt) 1 ns-mal két másik CPU, a 3-as és a 4-es az x szót kétszer gyorsan egymás után olvassa, ahogy a 8.22. (a) ábrán látható. A 8.22. (b)–(d) ábrákon a hat esemény (két írás és négy olvasás) három lehetséges sorrendje látható. A 8.22. (b) szerint a 3-as CPU (200, 200)-at kap, és a 4-es CPU is (200, 200). A 8.22. (c) szerint a 3-as CPU (100, 200)-at, a 4-es pedig (200, 200)-at kap. A 8.20. (d)-ben (100, 100)-at és (200, 100)-at kapnak. Ezek mind bekövetkezhetnek, mint ahogy más lehetőségek is vannak, amelyeket az ábra nem mutat.

Semmilyen körülmények között nem fordulhat elő azonban – és ez a soros konzisztencia lényege –, hogy a 3-as CPU (100, 200)-at olvasson ki, míg a 4-es CPU (200, 100)-at olvas. Ennek előfordulása azt jelentené, hogy a 3-as CPU szerint a 100



(b)	(c)	(d)
W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

8.22. ábra. *(a) Egy közös memóriaszót két CPU ír és két CPU olvas. (b)–(d) Három lehetséges mód a két írás és négy olvasás időbeli összefésülődésére*

írása az 1-es CPU által befejeződött, mielőtt a 2-es CPU beírta volna a 200-at. Ez rendben is van. De ez azt is jelentené a 4-es CPU szerint, hogy a 200 beírása a 2-es CPU által befejeződött, mielőtt a 100-at beírta volna az 1-es CPU. Magában ez az eredmény is lehetséges. A probléma ott van, hogy a soros konzisztencia garantálja, hogy van egy minden CPU számára látható globális sorrendje a beírásoknak. Ha a 3-as CPU a 100 beírását látja elsőnek, akkor a 4-es CPU-nak ugyanezt kell látnia.

Annak ellenére, hogy a soros konzisztencia nem olyan mereven szabályozott, mint a szigorú konzisztencia, mégis nagyon használható modell. Ez gyakorlatilag azt mondja, hogy amikor egymás mellett sok esemény történik, akkor egyetlen előfordulási sorrendjük van, amelyet valószínűleg az időzítés és a véletlen alakít ki, de az összes processzor ugyanazt a sorrendet érzékeli. Bár ez a kijelentés természetesnek tűnik, a továbbiakban olyan konzisztenciamodellekről lesz szó, melyek még ennyit sem garantálnak.

Processzorkonzisztencia

A **processzorkonzisztencia** (Goodman, 1989) egy gyengébb konzisztenciamodell, de könnyebb megvalósítani nagy multiprocesszorokon. Két tulajdonsága van:

1. Bármelyik CPU írásai az összes többi CPU számára a kiadás sorrendjében láthatók.
2. Mindegyik memóriaszóra teljesül, hogy a beírásokat minden processzor ugyanabban a sorrendben érzékeli.

Mindkét tulajdonság fontos. Az első szerint, ha az 1-es CPU írásokat kezdeményez az 1A, 1B és 1C értékekkel ebben a sorrendben valamelyik memóriarekeszbe, akkor az összes többi processzor ugyanezt a sorrendet látja. Más szóval, ha egy másik processzor gyors egymásutánban többször olvas ugyanabból a rekeszből, akkor az 1A, 1B és 1C értékeket kapja ilyen sorrendben, és nem láthatja az 1B után az 1A-t és így tovább. A második tulajdonságra azért van szükség, hogy minden memóriaszónak legyen egy határozott értéke, miután több CPU írja azt. Vagyis egyet kell érteniük abban, hogy ki volt az utolsó.

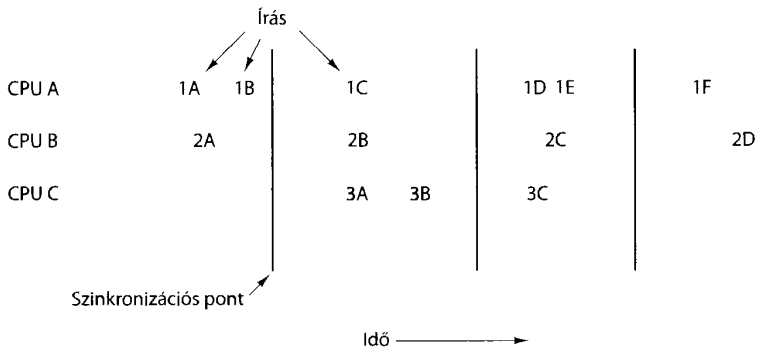
Még ezekkel az előírásokkal is nagy szabadsága van a tervezőnek. Nézzük meg, hogy mi történik, ha a 2-es CPU kiadja a 2A, 2B és 2C írásokat párhuzamosan az 1-es CPU három írásával. A memóriát folyamatosan olvasó más CPU-k egy-egy adatsort észlelnek, mint például 1A, 1B, 2A, 2B, 1C, 2C, vagy 2A, 1A, 2B, 2C, 1B, 1C vagy valami mást. A processzorkonzisztencia *nem* garantálja, hogy mindegyik CPU ugyanazt a sorrendet látja (nem úgy, mint a soros konzisztencia, amely biztosítja ezt). Ily módon teljesen elfogadható az, hogy néhány CPU a fent említett első sorrendet látja, néhány a másodikat, és megint mások ezektől eltérőt. Amit ez a modell garantál az az, hogy egyetlen CPU sem láthat olyan sorrendet, amelyben az 1B az 1A elé kerül stb. Az egyes CPU-k által kezdeményezett írások sorrendje mindenhol ugyanannak látszik.

Érdemes megjegyezni, hogy bizonyos szerzők máshogy értelmezik a processzorkonzisztenciát, és nem követelik meg a második feltételt.

Gyenge konzisztencia

A következő modell a **gyenge konzisztencia**, ez még azt sem garantálja, hogy egyetlen CPU-ból kezdeményezett írások sorrendben legyenek láthatók (Dubois és társai, 1986). Egy gyenge konzisztenciájú memóriában az egyik CPU az 1A-t az 1B előtt láthatja, míg egy másik CPU az 1A-t az 1B után láthatja. Azonban hogy a káoszba egy kis rendet vigyünk, a gyenge konzisztens memóriák szinkronizációs változókkal vagy egy szinkronizációs művelettel rendelkeznek. Egy szinkronizáció végrehajtásakor minden függőben levő írás befejeződik, és újak nem kezdődnek el addig, amíg az összes korábbi kész nincs, és maga a szinkronizáció is megtörténik. Valójában egy szinkronizáció „kitisztítja a csővezetéket", és a memóriát függőben levő műveletek nélküli stabil állapotba hozza. A szinkronizációs műveletek önmagukban sorosan konzisztensek, azaz amikor több CPU kezdeményez ilyet, akkor egy bizonyos sorrend kiválasztásra kerül, és az összes CPU ugyanazt a sorrendet látja.

A gyenge konzisztenciában a (soros konzisztenciájú) szinkronizációk az időt jól meghatározott időszakokra tagolják; ezt illusztrálja a 8.23. ábra. Semmilyen sorrend nem garantált az 1A és 1B között, és a különböző CPU-k a két írást más-más sorrendben láthatják, vagyis az egyik CPU először látja az 1A-t és utána az 1B-t, míg egy másik CPU először az 1B-t és utána az 1A-t. Ez a szituáció megengedett. Azonban, minden CPU az 1B-t csak az 1C előtt láthatja, mert az első szinkronizációs művelet kikényszeríti az 1A, 1B és 2A írások befejezését, mielőtt megengedné az 1C, 2B, 3A vagy 3B elkezdését. Ily módon a szinkronizációs műveletek alkalmazásával a szoftver az események bizonyos sorrendjét kikényszerítheti, bár ennek van valamennyi költsége, mert a memória-csővezeték kiürítéséhez idő kell.



8.23. ábra. *Gyengén konzisztens memóriánál az idő szekvenciális szakaszokra tagolása szinkronizációs műveletekkel*

Elengedési konzisztencia

A gyenge konzisztencia azzal a problémával küzd, hogy nem elég hatékony, mert az összes függőben levő memóriaműveletet be kell fejezni, és az újakat az éppen működők befejeztéig késleltetni kell. Az **elengedési konzisztencia** ezt a problémát igyekszik elkerülni úgy, hogy a kritikus szekciókhoz hasonló modellt alkalmaz (Gharachorloo és társai, 1990). E modell mögötti elgondolás az, hogy amikor egy folyamat elhagy egy kritikus szekciót, akkor nem szükséges az összes írás befejezésének azonnali kikényszerítése. Azt kell csak biztosítani, hogy az összes írás befejeződjön mielőtt bármely folyamat ugyanebbe a kritikus szekcióba belépne.

Ebben a modellben a gyenge konzisztencia szinkronizációs művelete ketté van bontva. Egy megosztott változó olvasásának vagy írásának megkezdése előtt a CPU-nak (vagyis a szoftverének) először egy acquire műveletet kell végrehajtani a szinkronizációs változón, hogy kizárólagos hozzáférése legyen a megosztott adatokhoz. Ezt követően a CPU tetszése szerint olvashatja és írhatja azt. Mikor elkészült, a CPU végrehajt egy release műveletet a szinkronizációs változón, hogy jelezze, készen van. A release nem kényszeríti ki a függőben levő írások befejezését, de ő maga nem fejeződik be addig, amíg az összes korábban kezdeményezett írás be nem fejeződik. Ráadásul új memóriaműveletek azonnal indulhatnak.

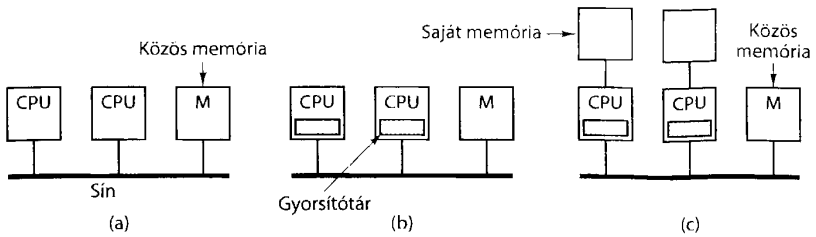
A következő acquire kiadásakor egy ellenőrzés hajtódik végre, amely megnézi, hogy az összes korábbi release befejeződött-e. Ha nem, akkor az acquire ezek mindegyikének befejeztéig (és így az összes írás elvégzéséig, mivel ezek megelőzik a release-ek befejeztét) felfüggesztett helyzetbe kerül. Ily módon, ha a következő acquire megfelelő késéssel követi a legutolsó release-t, akkor indulása előtt nem kell várakoznia, és késlekedés nélkül belépheti a kritikus szekcióba. Ha azonban egy release-t követően túl hamar fordul elő, akkor az acquire-nek (és az összes őt követő utasításnak) várakoznia kell, amíg minden függőben levő release be nem fejeződik, biztosítva ezzel a kritikus szekció összes változójának frissítését. Ez a modell valamivel bonyolultabb, mint a gyenge konzisztencia, azonban nagy előnye, hogy a konzisztencia fenntartása érdekében nem olyan gyakori az utasítások késleltetése.

A memóriakonzisztencia nem lezárt kérdéskör, a kutatók még mindig javasolnak újabb modelleket; lásd (Cain és Lipasti, 2004; Hammond és társai, 2004).

8.3.3. UMA sínrendszerű SMP-architektúrák

A legegyszerűbb multiprocesszorok egyetlen sínt használnak, ahogy a 8.24. (a) ábrán látható. Két vagy több CPU és egy vagy több memóriamodul mindegyike ugyanazt a sínt használja a kommunikációra. Mikor egy CPU ki akar olvasni egy memóriaszót, akkor először ellenőrzi, hogy a sín szabad-e. Ha a sín szabad, akkor a CPU a szó címét kiteszi a sínre, és beállít néhány vezérlőjelet, majd addig vár, amíg a memória a sínre rakja a kért szót.

Ha a sín foglalt, amikor a CPU írni vagy olvasni akarja a memóriát, akkor a sín szabaddá válásáig várakoznia kell. Itt van a probléma ezzel a modellel. Két vagy három CPU esetén a sínért való versengés még kezelhető, de 32 vagy 64 CPU-val



8.24. ábra. *Három sínalapú multiprocesszor. (a) Gyorsítótár nélkül. (b) Gyorsítótárral. (c) Gyorsítótárral és saját memóriákkal*

már elviselhetetlen. A rendszert teljesen korlátozza a sín sávszélessége, és a legtöbb CPU az idő legnagyobb részében áll.

Ennek a problémának a megoldásaként mindegyik CPU-hoz egy gyorsítótárat kapcsolnak, ahogy a 8.24. (b) ábra mutatja. Lehet a gyorsítótár a CPU lapka belsejében, vagy a CPU lapka mellett, vagy az alaplapon, vagy ezek közül több helyen is. Mivel sok olvasás a helyi gyorsítótárból kielégíthető, ezért a sín forgalma jelentősen lecsökken, és így a rendszer több CPU támogatására képes. A gyorsítótár sokat jelent ebben a helyzetben.

A 8.24. (c) ábrán egy másik megoldási lehetőség modellje látható, ebben a CPU-k nemcsak egy-egy gyorsítótárral, hanem külön saját memóriával is rendelkeznek, amelyeket külön erre a célra szolgáló sínen keresztül érnek el. Ahhoz, hogy egy ilyen konfiguráció optimálisan ki legyen használva, a fordítóprogramnak minden programkódot, karakterláncot, konstanst és más, csak olvasható adatot, vermet és lokális változót a CPU saját memóriájába kell elhelyeznie. Így már csak az írható megosztott változók kerülnek a közös memóriába. A legtöbb esetben ezzel a gondos elhelyezéssel nagyban lecsökken a sín forgalma, azonban ilyenkor szükség van a fordító aktív együttműködésére.

Szimatoló gyorsítótár

Míg a fenti, teljesítménnyel kapcsolatos érvek természetesen igazak, közben egy alapvető probléma felett átsiklottunk. Tételezzük fel, hogy a memória sorosan konzisztens. Mi történik, amikor az 1-es CPU a gyorsítótárában tárol egy sort, majd ugyanebből a sorból a 2-es CPU egy szót próbál olvasni? Speciális szabályok hiányában utóbbi is elhelyezhet egy másolatot a saját gyorsítótárában. Elvileg ugyanannak a sornak két különböző gyorsítótárban való megjelenése elfogadható. Most tegyük fel, hogy az 1-es CPU módosítja a sort, és közvetlenül ezt követően a 2-es CPU a saját gyorsítótárából ugyanannak a sornak a másolatát olvassa. Ekkor egy **elavult adat**ot kap, ami sérti a szoftver és a memória közötti megállapodást. A 2-es CPU-n futó program nem fog örülni.

Ez a probléma az irodalomban **gyorsítótár-koherencia**ként vagy **gyorsítótár-konzisztencia**ként ismert, és rendkívül súlyos. Megoldása nélkül nem használhat-

nánk gyorsítótárakat, ami azt jelentené, hogy a sínrendszerű multiprocesszorok két vagy három CPU-s rendszerekre korlátozódnának. A sínrendszerek fontossága következtében az évek során sokféle megoldással próbálkoztak; lásd például (Goodman, 1983; Papamaros és Patel, 1984). Bár ezek a gyorsítótár-algoritmusok, amelyeket **gyorsítótár-koherencia protokoll**oknak hívnak, részleteikben eltérnek egymástól, azonban mindegyikük meggátolja, hogy ugyanannak a sornak eltérő változatai legyenek két vagy több gyorsítótárban.

Az összes megoldásban speciálisan tervezett gyorsítótár-vezérlő van, ennek megengedett a sínen való hallgatódzás, felügyeli az összes CPU-ból és gyorsítótárból jövő kérést, és bizonyos esetekben akcióba lép. Ezeket az eszközöket hívják **szimatoló gyorsítótár**aknak, mivel a sínt „szimatolják". A gyorsítótárak, a CPU-k és a memória működését meghatározó szabályok, amelyek megakadályozzák, hogy több gyorsítótárban az adatok különböző változatai fordulhassanak elő, együttesen alkotják a gyorsítótár-koherencia protokollt. A gyorsítótáraknál az átvitel és a tárolás egységét **gyorsítósor**nak hívják, és általában 32 vagy 64 bájtos.

A legegyszerűbb gyorsítótár-koherencia protokoll az **írásáteresztő**. Ez a legjobban a 8.25. ábrán felsorolt különböző négy eset áttekintésével érthető meg. Amikor egy CPU olyan szót próbál olvasni, amely nincs a gyorsítótárában (olvasáshiány), akkor a gyorsítótár vezérlője betölti a kívánt szót tartalmazó sort a gyorsítótárába. A sor a memóriából kerül kiolvasásra, amely e protokoll szerint mindig érvényes adatot tárol. Az ezt követő olvasások (olvasástalálat) már a gyorsítótárból kielégíthetők.

Egy íráshiány jelentkezésekor a módosított szó a memóriába íródik. A szót tartalmazó sor *nem* töltődik be a gyorsítótárba. Egy írástalálatkor a gyorsítótár frissítésre kerül, és a memóriában a szó is átíródik. Ennek a protokollnak az a lényege, hogy a szóra vonatkozó minden írási művelet a memóriában is átírásra kerül, és így a memória mindig frissített adatokkal rendelkezik.

Nézzük meg újra ezeket az akciókat, de most a szimatoló szemszögéből, amit a 8.25. ábra jobb oldali oszlopa tartalmaz. Az akciót végrehajtó gyorsítótár legyen az 1-es, a szimatoló gyorsítótár pedig a 2-es. Amikor az 1-es CPU-nál olvasáshiány lép fel, akkor egy sínkérést generál, amivel a memóriából a sor behozását kéri. A 2-es gyorsítótár látja ezt, de nem tesz semmit. Amikor az 1-es gyorsítótárnál olvasástalálat van, akkor az olvasás helyben kielégíthető, és nincs szükség sínkérésre, így a 2-es gyorsítótár nem szerez tudomást az 1-es gyorsítótár sikeres olvasásairól.

Esemény	Helyi kérés	Távoli kérés
Olvasáshiány	Behozza a memóriából az adatot.	
Olvasástalálat	Helyi gyorsítótárban levő adatot használja.	
Íráshiány	Frissíti a memóriában az adatot.	
Írástalálat	Frissíti a gyorsítótárat és a memóriát.	Érvényteleníti a gyorsítótár-bejegyzést.

8.25. ábra. *Az írásáteresztő gyorsítótár-koherencia protokoll. Az üres helyek azt jelentik, hogy nincs tevékenység*

Az írás érdekesebb. Amikor az 1-es CPU egy írást hajt végre, akkor az 1-es gyorsítótár a sínen írási kérést generál, mind hiány, mind találat esetében. Minden íráskor a 2-es gyorsítótár ellenőrzi, hogy az írandó szó nála van-e. Ha nincs, akkor az ő szempontjából ez egy távoli kérés/íráshiány, és nem csinál semmit. (Egy fontos részlet megvilágításához megjegyezzük, hogy a 8.25. ábrán szereplő távoli hiány azt jelenti, hogy a szó nincs a szimatoló gyorsítótárban; teljesen mindegy, hogy az akciót indító gyorsítótárban benne van-e, vagy nincs. Így egyetlen kérés lehet helyben kielégíthető találat, és a szimatoló számára hiány, vagy fordítva.)

Most tételezzük fel, hogy az 1-es gyorsítótár egy olyan szót ír, amely a 2-es gyorsítótárban van (távoli kérés/írástalálat). Ha a 2-es gyorsítótár nem tenne semmit, akkor egy elavult adatot tárolna a továbbiakban, ezért megjelöli a módosított szót tartalmazó bejegyzést mint érvénytelent. Valójában eltávolítja a gyorsítótárból az érvénytelen sort. Mivel mindegyik gyorsítótár minden sínkérést kiszimatol, így valahányszor egy szó írása történik, ez előidézi az akciót indító gyorsítótárban és a memóriában az adat frissítését, és az összes többi gyorsítótárból az adat eltávolítását. Ilyen módon meg lehet előzni az inkonzisztens változatok megjelenését.

Természetesen a 2-es gyorsítótár CPU-ja ugyanezt a szót olvashatja akár a következő ciklusban. Ez esetben a 2-es gyorsítótár a szót a memóriából fogja olvasni, amely frissített. Ekkor az 1-es gyorsítótár, a 2-es gyorsítótár és a memória az adat azonos másolataival rendelkeznek. Ha most az egyik CPU végrehajt egy írást, akkor a másiknak a gyorsítótárából ez az adat törlődik, és a memória frissítődik.

Ennek a protokollnak többféle változata lehetséges. Például egy írás találatkor a szimatoló gyorsítótár általában érvényteleníti az írt szót tartalmazó bejegyzését. Egy másik lehetőség az, hogy elfogadja az új értéket, és frissíti magát, ahelyett hogy érvénytelenítene. Lényegében a gyorsítótár frissítése ugyanaz, mintha az érvénytelenítést egy memóriából való olvasás követné. A gyorsítótár-protokollok mindegyike vagy **frissítő stratégiát**, vagy **érvénytelenítő stratégiát** alkalmaz. Ezek a protokollok másként működnek eltérő terheléseknél. A frissítési üzenetek adatokat is tartalmaznak, ezért nagyobbak az érvénytelenítéseknél, de a későbbi gyorsítótárhiányok is megelőzhetők velük.

Egy másik változat szerint a szimatoló gyorsítótár betöltése írás hiány előfordulásakor történik. Ez a betöltés az algoritmus helyességét nem befolyásolja, csak a teljesítményt. A kérdés a következő: „Mi a valószínűsége annak, hogy az éppen írt szó egy rövid időn belül ismét írva lesz?" Ha ennek a valószínűsége nagy, akkor érdemes a gyorsítótárat írási hiánykor feltölteni, amit **íráskori feltöltés (írásallokáció) módszer**ként ismernek. De ha a valószínűség kicsi, akkor nem érdemes frissíteni írási hiánykor. Ha a szó *olvasására* rövid időn belül sor kerül, akkor az olvasási hiány előfordulásakor mindenképpen betöltődik a gyorsítótárba; vagyis kicsi a nyeresége az íráshiány során való betöltésnek.

Mint sok egyszerű megoldás, ez sem kielégítő. Ugyanis minden memóriába írás a sínen keresztül megy végbe, így jelentősebb számú CPU esetén a sín szűk keresztmetszetté válik. A sín forgalmának keretek közt tartására más gyorsítótár protokollokat gondoltak ki. Ezek közös tulajdonsága, hogy nem minden írás megy át egyből a memóriába. Ehelyett amikor egy gyorsítótár sor módosul, akkor a gyorsítótár egy bit beállításával jelzi, hogy a sor frissített a gyorsítótárban, de a

memóriában nem. Valamikor majd az ilyen megjelölt sort vissza kell írni a memóriába, de lehet, hogy csak több írási művelet végrehajtását követően. Az ilyen típusú protokollokat **késleltetett írású protokoll**oknak hívják.

A MESI gyorsítótár-koherencia protokoll

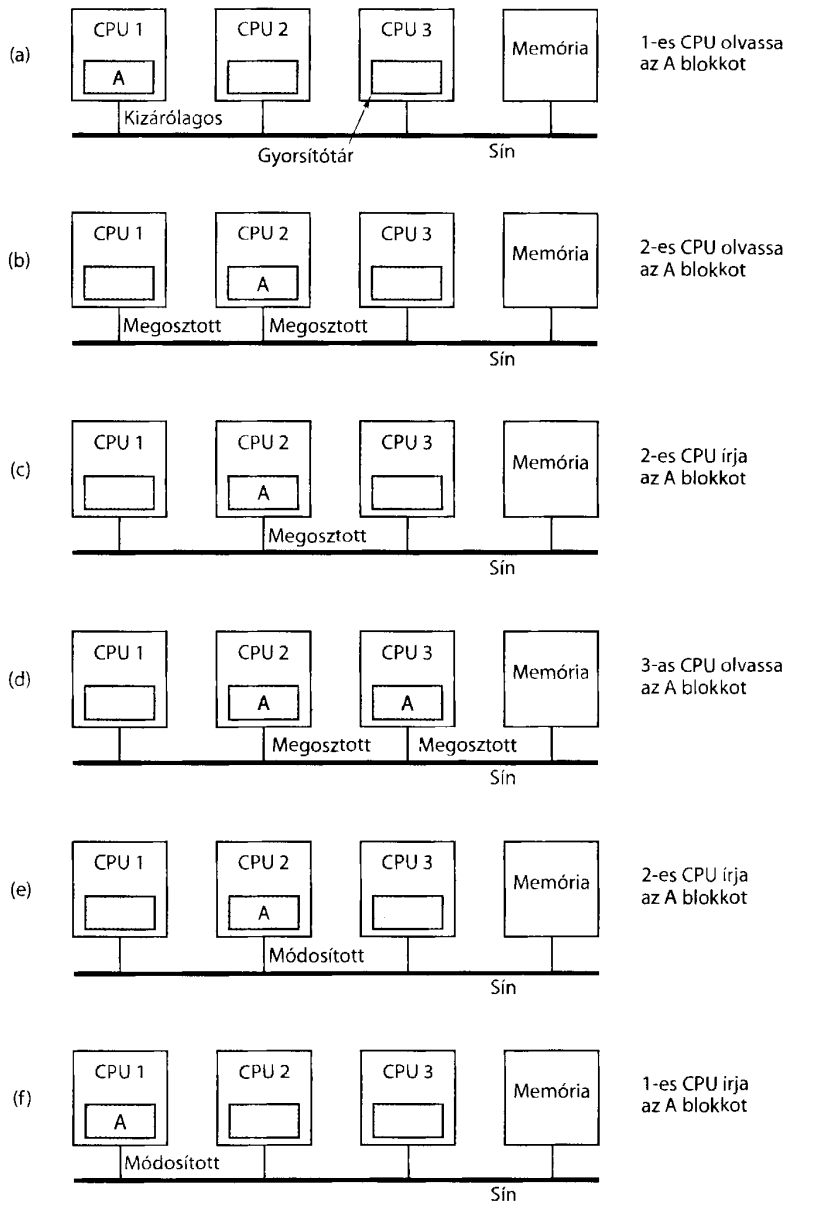
Egy népszerű késleltetett írású gyorsítótár-koherencia protokoll a **MESI**, a nevét az általa használt négy állapot (M, E, S és I) kezdőbetűi után kapta (Papamarcos és Patel, 1984). Alapjául a korábbi **egyszer író protokoll** (Goodman, 1983) szolgált. A Pentium 4 és sok más CPU is használja a MESI protokollt a sín vizsgálatára. Minden gyorsítótár-bejegyzés a következő négy állapot valamelyikében van:

1. Érvénytelen (Invalid) – A gyorsítótár-bejegyzés nem tartalmaz érvényes adatot.
2. Megosztott (Shared) – Több gyorsítótár tartalmazhatja a sort, a memória frissítve van.
3. Kizárólagos (Exclusive) – Más gyorsítótár nem tartalmazza a sort, a memória frissítve van.
4. Módosított (Modified) – A bejegyzés érvényes; a memória nincs frissítve; másolatok nincsenek.

A CPU indulásakor a gyorsítótár összes bejegyzésének állapota érvénytelenre állítódik. A memória első olvasásakor a hivatkozott sor a memóriából kiolvasódik, és a CPU gyorsítótárába kerül E (kizárólagos) állapotjelzéssel, mivel a bejegyzésnek csak egy másolata van a gyorsítótárban, ezt illusztrálja a 8.26. (a) ábra, mikor az 1-es CPU olvassa az *A* sort. A következő olvasásoknál a CPU a gyorsítótárának a bejegyzését veszi, és nem használja a sínt. Egy másik CPU szintén behozhatja ugyanazt a sort, és a gyorsítótárába helyezheti, de a sín vizsgálatából az eredeti (1-es CPU) tároló látja, hogy már nem egyedüli tároló, és a sínen keresztül bejelenti, hogy szintén rendelkezik egy másolattal. Ekkor mindkét bejegyzett másolat állapota S (megosztott) lesz, ezt mutatja a 8.26. (b) ábra. Vagyis az S állapot azt jelenti, hogy a sort egy vagy több gyorsítótár is tárolja olvasáshoz, és a memória aktualizálva van. Egy CPU olyan sorra vonatkozó következő olvasásai, amelyek a gyorsítótárában S állapottal vannak bejegyezve, nem használják a sínt, és nem idéznek elő állapotváltozást.

Gondoljuk át, mi történik, amikor a 2-es CPU egy S állapottal jelzett gyorsítósort ír. Ekkor kitesz egy érvénytelenítő jelzést a sínre, hogy tudassa a többi CPU-val, hogy dobják ki saját példányukat. A gyorsítótárban levő másolat ekkor M-re (módosított) változik, ahogy ezt a 8.26. (c) ábra mutatja. A sor nem íródik ki a memóriába. Érdemes megjegyezni, hogy semmi különös nem történik, ha az írott sor állapota E volt, ekkor ugyanis nincs szükség érvénytelenítő jelzés kiadására, mivel ismert, hogy nincs több példánya a sornak.

A következőben nézzük meg, mi történik, mikor a 3-as CPU olvassa a sort. A 2-es CPU, amelyik most a sort birtokolja, tudja, hogy a memóriában levő példány nem érvényes, ezért kitesz egy jelzést a sínre, így tudatja a 3-as CPU-val, hogy vár-



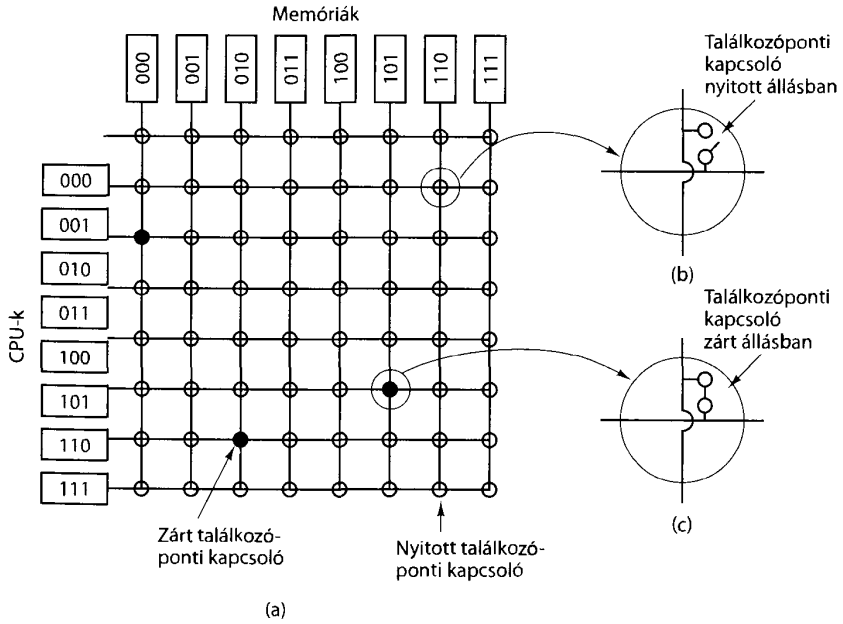
8.26. ábra. *A MESI gyorsítótár-koherencia protokoll*

jon, amíg a memóriába a sort visszaírja. Amikor ez készen van, a 3-as CPU vesz egy másolatot, és mindkét gyorsítótárban a sor megosztott jelzést kap, amint a 8.26. (d) ábrán látható. Ezután a 2-es CPU ismét írja a sort, ami érvényteleníti a 3-as CPU gyorsítótárában levő példányt, ahogy ezt a 8.26. (e) ábra mutatja.

Végül az 1-es CPU ír egy szót a sorba. A 2-es CPU látja ezt, ezért kiad egy jelzést a sínen, hogy tudassa az 1-es CPU-val, hogy várjon, amíg a sort visszaírja a memóriába. Mikor ez készen van, a saját példányát érvénytelenként jelöli meg, mivel tudja, hogy egy másik CPU annak módosítására készül. Ez egy olyan helyzet, amelyben egy CPU egy gyorsítótárban nem tárolt sort ír. Az írásallokáló módszer esetén a sor betöltődik a gyorsítótárba, és az állapota M lesz, amint a 8.26. (f) ábra mutatja. Ha nem alkalmazzuk az írásallokáló módszert, akkor az írás a memóriába történik, és a sor egyik gyorsítótárban sem lesz tárolva.

Crossbar (keresztrudas) kapcsolót alkalmazó UMA-multiprocesszorok

Az összes lehetséges optimalizációval együtt is az egyszerű sínrendszer alkalmazása az UMA-multiprocesszorok méretét körülbelül 16 vagy 32 CPU-ra korlátozza. A továbblépéshez másfajta összekötő hálózatra van szükség. *n* darab CPU-t és *k* darab memóriát összekötő legegyszerűbb áramkör a **crossbar (keresztrudas) kapcsoló**, ezt a 8.27. ábra illusztrálja. A telefonközpontok évtizedek óta használnak crossbar kapcsolót bejövő és kimenő vonalak csoportjainak tetszőleges módon való összekapcsolására.



8.27. ábra. *(a) Egy 8 × 8-as crossbar kapcsoló hálózat. (b) Nyitott találkozópont. (c) Zárt találkozópont*

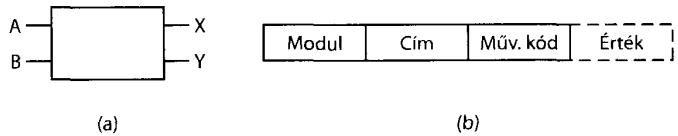
A vízszintes (bejövő) és a függőleges (kimenő) vonalak metszéspontjai a **találkozópont**ok. A találkozópont egy kis kapcsoló, amely lehet elektromosan nyitott vagy zárt, attól függően, hogy a vízszintes és a függőleges vonalakat össze szeretnénk-e kapcsolni, vagy nem. A 8.27. (a) ábrán egyszerre három zárt találkozópont látható, ez egyszerre a (001, 000), (101, 101) és az (110, 010) párok közötti (CPU, memória) kapcsolatot engedélyezi. Sok más kombináció lehetséges még. Valójában a kombinációk száma annyi, ahány különböző módon egy sakktáblán nyolc bástyát biztonságosan el lehet helyezni.

A crossbar kapcsoló egyik legkellemesebb tulajdonsága, hogy **nem blokkoló hálózat**, ami azt jelenti, hogy nincs olyan CPU, amelytől bizonyos találkozópontok és vonalak foglaltsága miatt egy neki szükséges összeköttetés valaha is meg lenne tagadva (feltéve, hogy maga a memóriamodul elérhető). Ráadásul előzetes tervezésre sincs szükség. Még ha hét tetszőleges összeköttetés már fel is van építve, akkor is mindig összekapcsolható a maradék CPU és a maradék memória. Később látni fogunk olyan összekötő sémákat, amelyek nem rendelkeznek ezzel a tulajdonsággal.

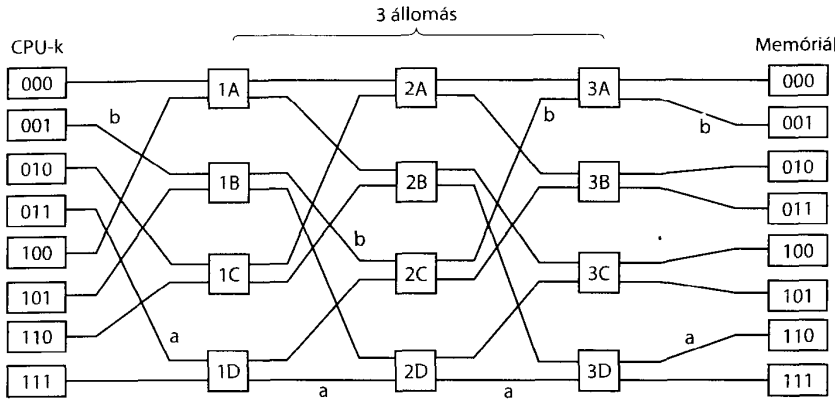
A crossbar kapcsoló egyik legrosszabb tulajdonsága, hogy a találkozópontok száma n^2. Közepes méretű rendszernél még alkalmazható a crossbar kapcsoló. A fejezet későbbi részében bemutatásra kerül egy ilyen rendszer, a Sun Fire E25K. 1000 CPU és 1000 memóriamodulhoz azonban már egymillió találkozópont kell, egy ilyen nagy crossbar kapcsoló nem valósítható meg. Valami egészen másra van szükség.

Többszintű kapcsolóhálózatot használó UMA-multiprocesszorok

Az a „valami egészen más" alapulhat azon a szerény 2 × 2-es kapcsolón, amely a 8.28. (a) ábrán látható. Ennek a kapcsolónak két bemenete és két kimenete van. Bármelyik bemeneti vonalon érkező üzenet rákapcsolható bármelyik kimeneti vonalra. Jelenlegi céljainkhoz illeszkedve az üzenetek legfeljebb négy részre tagolódnak, miként a 8.28. (b) ábra mutatja. A *modul* mező azonosít egy memóriamodult. A *cím* megadja a modulon belüli címet. A *műveleti kód* egy művelet lehet, READ vagy WRITE. Végül az opcionális *érték* operandust adhat meg, amely például a WRITE műveletnél az írandó 32 bites szó lehet. A kapcsoló a *modul* mező vizsgálata alapján dönti el, hogy az üzenetet az *X* vagy az *Y* kimeneti vonalra küldje.



8.28. ábra. *(a) Egy 2 × 2-es kapcsoló. (b) Egy üzenet szerkezete*



8.29. ábra. *Egy omega kapcsoló hálózat*

A 2 × 2-es kapcsolók sokféleképpen rendezhetők, és így lehetőséget adnak nagyobb, **többszintű kapcsoló hálózat**ok kiépítésére. Az egyik lehetséges modell az egyszerű és takarékos **omega hálózat**, ez látható a 8.29. ábrán. Itt nyolc CPU és 8 memória összekapcsolása 12 kapcsolóval megoldható. Ezt általánosítva azt mondhatjuk, hogy n darab CPU és n darab memória összekapcsolása $\log_2 n$ fázison keresztül történik, fázisonként $n/2$ kapcsolóval, így a teljes kapcsolószám $(n/2)\log_2 n$, ami sokkal jobb, mint a crossbar kapcsoló n^2 kapcsolószáma, különösen nagy n értékeknél.

Az omega hálózat mintáját gyakran **tökéletes keverés**nek hívják, mivel a jelek keveredése minden fázisban egy olyan kártyacsomagra hasonlít, amelyet megfeleznek, majd egyenletesen összefésülnek. Hogy lássuk, hogyan működik az omega hálózat, tegyük fel, hogy a 011-es CPU egy szót akar olvasni az 110-s memóriamodulból. A CPU az 1D kapcsolóhoz egy READ üzenetet küld, a *modul* mezőben az 110 értékkel. A kapcsoló útvonalválasztáshoz kiveszi az 110 legelső bitjét (bal szélső). 0 esetén a felső kimeneti vonalra, 1 esetén az alsó kimeneti vonalra irányít. Példánkban ez a bit 1, ezért az üzenet útvonala az alsó kimeneti vonalon keresztül a 2D-be vezet.

A második fázis minden kapcsolója, beleértve a 2D-t is, az útvonalválasztáshoz a második bitet használja. Ez ismét 1, így az üzenet az alsó kimeneten áthaladva a 3D felé megy. Itt a harmadik bitet vizsgálja a kapcsoló, amely most 0. Következésképpen az üzenet a felső kimeneten megy ki, és az 110-s memóriához érkezik úgy, ahogy terveztük. Ennek az üzenetnek az útvonala a 8.29. ábrán az *a* betűvel jelzett.

Ahogy az üzenet mozog a kapcsoló hálózatban, a modulszám bal végén a bitek egymás után feleslegessé válnak. Így ezek a bitek felhasználhatók a kapcsolóba bejövő vonalak számának bejegyzésére, ezáltal a válasz visszaútja készen áll. Az *a*-val jelzett útnál a bejövő vonalak egymás után a 0 (felső bemenet 1D-hez), az 1 (alsó bemenet 2D-hez) és az 1 (alsó bemenet 3D-hez). A válasz a visszaúthoz a 011-et használja, csak most jobbról balra olvasva.

Tegyük fel, hogy ugyanebben az időben a 001-es CPU egy szót akar a 001-es memóriamodulba írni. Ez a folyamat az előzővel analóg módon megy végbe, az üzenet egymás után az alsó, alsó és felső kimeneti vonalakon halad át, ez az útvonal *b*-vel jelzett. A megérkezéskor a *module* mező tartalma az igénybe vett utat reprezentálja, értéke 001. Mivel az eddigi két kérés nem használ azonos kapcsolókat, összeköttetéseket vagy memóriamodulokat, ezért párhuzamosan haladhatnak.

Most gondoljuk végig, mi történik, ha ugyanekkor a 000-s CPU el akarja érni a 000-s memóriamodult. Ennek a kérése a 001-es CPU kérésével a 3A kapcsolónál ütközik. Az egyiknek közülük várnia kell. A crossbar kapcsolóval ellentétben az omega hálózat egy **blokkoló hálózat**. Nem minden kérés halmaz tud a hálózaton párhuzamosan haladni. Az ütközések a vezetékek vagy a kapcsolók használatánál jelentkezhetnek, valamint a memóriához érkező és a memóriából induló válaszok között.

Nyilvánvaló igényként jelenik meg memóriahivatkozások egyenletes szétosztása a modulok között. Az egyik szokásos módszer az alacsony helyértékű biteket használja modulszámként. Példaként vegyünk egy olyan bájtcímzésű számítógépet, amely főleg 32 bites szavakat kezel. A 2 legalacsonyabb helyértékű bit általában 00, de a következő 3 bit egyenletes eloszlású. Modulszámként alkalmazva ezt a 3 bitet, az egymás utáni című szavak egymás utáni modulokba kerülnek. Az olyan memóriarendszert, amelyben az egymás után levő szavak különböző modulokban vannak, **tagolt**nak szokták hívni. A tagolt memóriák maximális párhuzamosítást eredményezhetnek, mert a legtöbb memóriahivatkozás egymást követő címekre vonatkozik. Olyan kapcsoló hálózatot is lehet tervezni, amely nem blokkol, és mindegyik CPU-ból mindegyik memóriamodulhoz több utat is felkínál, hogy ezáltal jobban szétterítse a forgalmat.

8.3.4. NUMA-multiprocesszorok

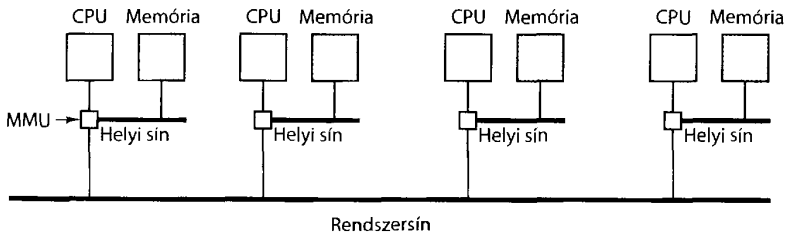
Mostanra világossá kellett válnia, hogy az egyetlen sínnel rendelkező UMA-multiprocesszorok általában néhány tucat CPU-ra korlátozódnak, a crossbar kapcsolós vagy kapcsoló hálózatos multiprocesszorokhoz pedig sok (és drága) hardver kell, és még így sem sokkal nagyobbak. Valamiből engedni kell ahhoz, hogy 100-nál több CPU-val rendelkező rendszer alakuljon ki. Rendszerint abból engedünk, hogy minden memóriamodulnak ugyanakkora az elérési ideje. Ez az engedmény vezetett a **NUMA- (NonUniform Memory Access, nem egységes memóriaelérésű)** multiprocesszorokhoz. Az unokatestvér UMA-hoz hasonlóan ez is egy közös címtartományt kínál fel a CPU-k számára, de az UMA-gépektől eltérő módon a helyi memóriamodulokat gyorsabban érik el, mint a távolikat. Így minden UMA-program változtatás nélkül fut a NUMA-gépeken, de a teljesítményük rosszabb lesz, mint egy azonos ciklusidejű UMA-gépen.

A NUMA-gépeknek három olyan sajátosságuk van, amelyek együttesen megkülönböztetik őket a többi multiprocesszortól:

1. Van egy közös címtartomány, amely az összes CPU-ból látható.
2. Távoli memória elérését a LOAD és a STORE utasításokkal végzi.
3. A távoli memória elérése lassúbb, mint a helyi memóriáé.

Amikor a távoli memória elérése nem rejtett (mert nincs gyorsítótár), akkor **NC-NUMA**-ról beszélünk. **CC-NUMA** a neve a rendszernek, ha vannak benne koherens gyorsítótárak (legalábbis a hardveresek így hívják). A szoftveresek gyakran **hardver DSM**-nek nevezik, mert alapvetően ugyanaz, mint a szoftverfelosztású közös memória, csak kis lapméretet alkalmazó hardverrel van megvalósítva.

Az első NC-NUMA-gépek egyike (jóllehet még az elnevezés sem létezett) a Carnegie-Mellon Cm* volt, amelynek egyszerűsített vázát a 8.30. ábra mutatja (Swan és társai, 1977). LSI–11 CPU-kból áll, amelyek mindegyike lokális sínen megcímzett memóriával is rendelkezik. (Az LSI–11 az 1970-es évek népszerű miniszámítógépének a DEC PDP–11-nek az egylapkás változata volt.) Ezenkívül az LSI–11 rendszereket egy rendszersín kapcsolta össze. Amikor egy memóriakérés érkezett a (speciálisan módosított) MMU-hoz, akkor az megvizsgálta, hogy a kért szó a helyi memóriában van-e. Ha igen, akkor a lokális sínen kérést küldött a szó kiolvasására. Ha nem, akkor kérést a rendszersínen továbbította ahhoz a rendszerhez, amelyiknél a szó volt, és a kérést képes volt teljesíteni. Természetesen ez utóbbi sokkal hosszabb időt vett igénybe, mint az előbbi. Egy program gond nélkül futhatott távoli memóriából, de a végrehajtása 10-szer tovább tartott, mint ugyanannak a programnak a futtatása a helyi memóriából.



8.30. ábra. *Kétszintű sínrendszerrel rendelkező NUMA-gép. A Cm* volt az első ilyen felépítésű multiprocesszor*

A memóriakoherencia egy NC-NUMA-gépen garantált, mivel nem alkalmaz gyorsítótárat. Minden memóriaszó pontosan egy helyen létezik, így nincs az a veszély, hogy egy másolatban elavult adat van: ugyanis nincsenek másolatok. Természetesen itt nagyon fontos, hogy melyik lap melyik memóriában van, mert a rossz elhelyezés nagy teljesítménycsökkenéssel jár. Következésképpen, az NC-NUMA-gépek bonyolult lapmozgató szoftvert használnak, hogy javítsák a teljesítményt.

Általában van egy **lapfelügyelő**nek nevezett démon folyamat, amely néhány másodpercenként fut. Feladata egyrészt megvizsgálni a laphasználati statisztikát, másrészt ennek ismeretében megkísérli a lapokat úgy mozgatni, hogy ezáltal a teljesítmény javuljon. Ha egy lap rossz helyen van, akkor a lapfelügyelő megszünteti a lap leképezését, és így a következő erre való hivatkozás laphiányt eredményez. A hiány megjelenésekor a lap elhelyezéséről lehetőleg olyan döntés születik, hogy a korábbitól eltérő memóriamodulba kerüljön. Hogy a túl gyakori oda-vissza mozgatást elkerüljék, rendszerint azt a szabályt alkalmazzák, hogy valahányszor egy

lap elhelyezésre kerül a memóriában, akkor ΔT időre tiltják a mozgathatóságát. Sokféle algoritmussal próbálkoztak már, de azt a következtetést lehet levonni, hogy nincs minden körülmények között legjobban teljesítő algoritmus (LaRowe és Ellis, 1991).

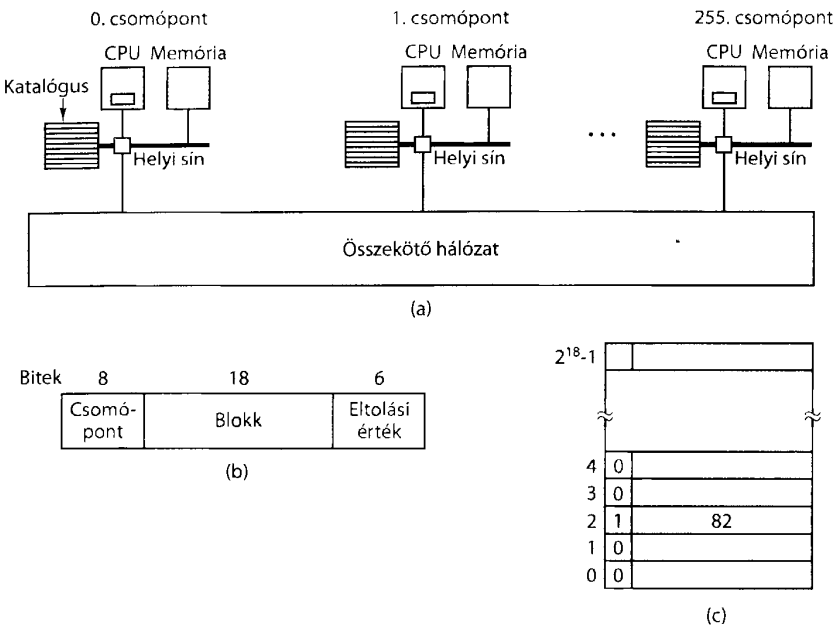
Gyorsítótár-koherens NUMA-multiprocesszorok

A 8.30. ábrán látható architektúrához hasonló multiprocesszorok nem skálázhatók jól, mert nem rendelkeznek gyorsítótárral. A teljesítményt nagyban csökkenti az, hogy távoli memóriát kell elérni valahányszor egy nem lokális memóriaszóra történik hivatkozás. Persze a gyorsítótárazás alkalmazása viszont a gyorsítótárak koherenciájának fenntartását igényli. Ennek egyik lehetséges megoldási módja, ha a rendszersín forgalmát figyeljük. Technikailag ezt nem nehéz megvalósítani, de egy bizonyos processzorszámon felül már nem használható. Valóban nagy multiprocesszorok építése alapvetően más megközelítést tesz szükségessé.

Jelenleg a nagy **CC-NUMA-** (**Cache Coherent NUMA, gyorsítótár-koherens NUMA**) multiprocesszorok legnépszerűbb megközelítése a **katalógusalapú multiprocesszor**. Ennél az ötlet az, hogy egy adatbázisban tartják nyilván mindegyik gyorsítósor helyét és állapotát. Egy gyorsítósorra történő hivatkozáskor az adatbázisból lekérdezhető a sor helye, és az, hogy frissített vagy elavult adatot tárol-e. Mivel az adatbázist minden memóriára hivatkozó utasításnak le kell kérdeznie, ezért ezt egy kivételesen gyors, speciális célú hardverben kell tárolni, amely egy sín ciklusidejének töredéke alatt képes válaszolni.

Hogy jobban megértsük a katalógusalapú multiprocesszor alapgondolatát, tekintsük azt az egyszerű (képzeletbeli) rendszert, amelyben 256 csomópont van, és mindegyik csomópont egy CPU-ból és az ehhez lokális sínnel kapcsolt 16 MB-os RAM-ból áll. A teljes memória 2^{32} bájt, amely 2^{26} darab 64 bájtos gyorsítósorra van felosztva. A csomópontokhoz a memória hozzárendelése statikusan történik, a 0. csomóponthoz a 0–16 MB tartomány, az 1. csomóponthoz a 16–32 MB tartomány tartozik, és így tovább. A csomópontok a 8.31. (a) ábrán látható összekötő hálózattal kapcsolódnak egymáshoz. Az összekötő hálózat lehet rács, hiperkocka vagy más egyéb topológia. A csomópontok tárolják azokat a katalógusbejegyzéseket is, amelyek a 2^{24} bájtnyi lokális memóriát alkotó 2^{18} darab 64 bájtos gyorsítósorhoz tartoznak. Egyelőre tegyük fel, hogy egy sor legfeljebb egy gyorsítótárban jelenhet meg.

Ahhoz, hogy lássuk a katalógus működését, kövessünk nyomon egy LOAD utasítást, amely a 20. CPU-ból indul, és egy gyorsítósorra hivatkozik. Az utasítást a kiadó CPU először a saját MMU-jának adja át, amely előállítja a fizikai címet, legyen ez mondjuk a 0x24000108 fizikai cím. Az MMU ezt a címet három részre bontja fel, mint ahogy a 8.31. (b) ábra mutatja. Decimálisan a három rész jelentése: 36. csomópont, 4. sor, eltolási érték 8. Az MMU látja, hogy a hivatkozás a 36. csomópontra és nem a 20.-ra vonatkozik, így küld egy kérő üzenetet az összekötő hálózaton keresztül a sort birtokló 36. csomóponthoz, megkérdezve azt, hogy vajon a 4-es sor tárolva van-e gyorsítótárban, és ha igen, akkor hol.



8.31. ábra. *(a) Egy 256 csomóponttal rendelkező katalógusalapú multiprocesszor. (b) Egy 32 bites memóriacím mezőkre osztva. (c) A 36. csomópont katalógusa*

Amikor a kérés a 36. csomóponthoz megérkezik az összekötő hálózaton keresztül, akkor ezt a hardverkatalógushoz irányítják. Minden gyorsítósorhoz tartozik a hardver 2^{18} bejegyzésű táblázatában egy bejegyzés. Ezek közül kiemeli a 4. sorhoz tartozót. A 8.31. (c) ábrán láthatjuk, hogy a kérdéses sor még nincs tárolva, így a hardver kiveszi a 4. sort a lokális RAM-ból, elküldi a 20. csomóponthoz, és a katalógusa 4. bejegyzésében megjegyzi, hogy a sor a 20. csomópontnál van tárolva.

Most nézzünk egy másik kérést, amely a 36. csomópont 2. sorára vonatkozik. A 8.31. (c) ábrából láthatjuk, hogy ez a sor a 82. csomópontnál van tárolva. Ekkor a hardver a 2. katalógusbejegyzést módosítja, jelezve, hogy a sor most már a 20. csomópontnál van tárolva és ezután a 82. csomóponthoz azt az üzenetet küldi, hogy adja át a kért sort a 20. csomópontnak és érvénytelenítse a gyorsítótárát. Vegyük észre, hogy a színfalak mögött még az ún. „közös memóriájú multiprocesszor" is nagyszámú üzenetátadást végez.

Rövid kitérőként nézzük meg, mekkora memóriát foglalnak le a katalógusok. Minden csomópontnál van egy 16 MB-os RAM és ennek nyomon követéséhez 2^{18} darab 9 bites bejegyzés. Így a katalógus adminisztráció kb. 9×2^{18} bitje jelentkezik többletként a 16 MB-hoz, ez kb. 1,76 százalék, ami általában elfogadható (bár ehhez nagy sebességű memória kell, ami emeli az árát). Még 32 bájtos gyorsítósorok esetén is az adminisztráció csak 4 százalék lenne. 128 bájtos gyorsítósorok alkalmazásánál viszont 1 százalék alá menne le.

Ennek a szerkezetnek egyik nyilvánvaló korlátja, hogy egy sor tárolását csak egyetlen csomópontnál engedi meg. Ha megengedett a sorok több csomópontnál való tárolása, akkor kell valamilyen módszer ezek mindegyikének megtalálására, hogy például egy íráskor a másolatokat érvényteleníteni vagy frissíteni lehessen. Több megoldás is van, amely megengedi a több csomópontnál való tárolást egy időben.

Egyik lehetőség, hogy a katalógus mindegyik bejegyzéséhez tartozzon k darab mező, a többi másolatot tároló csomópont számára, így téve lehetővé, hogy mindegyik sor (legfeljebb) k csomópontban tárolható legyen. Másik lehetőség, ha az egyszerű tervünkben a csomópontszámokat egy csomópontonként 1 bitet tartalmazó bittérképpel helyettesítjük. Ez a változat a másolatok számát ugyan nem korlátozza, de lényegesen megnöveli az adminisztráció helyigényét. Eszerint egy katalógus mindegyik 64 bájtos (512 bites) gyorsítósorhoz 256 bites bejegyzést tartalmazna, ami az adminisztráció többletköltségét több mint 50%-ra növelné. A harmadik lehetőség, hogy mindegyik katalógusbejegyzésben egy nyolcbites mezőt tárolunk, amelyet csak a gyorsítósor másolatai láncolt listájának fejeként használunk. Ez a stratégia mindegyik csomópontnál extra tároló alkalmazását igényli a mutatók láncolt listája számára, és egy sor összes másolatának elérése a láncolt listán való mozgással jár együtt. Mindegyik lehetőségnek van előnye és hátránya, és mindhármat alkalmazzák valós rendszerekben.

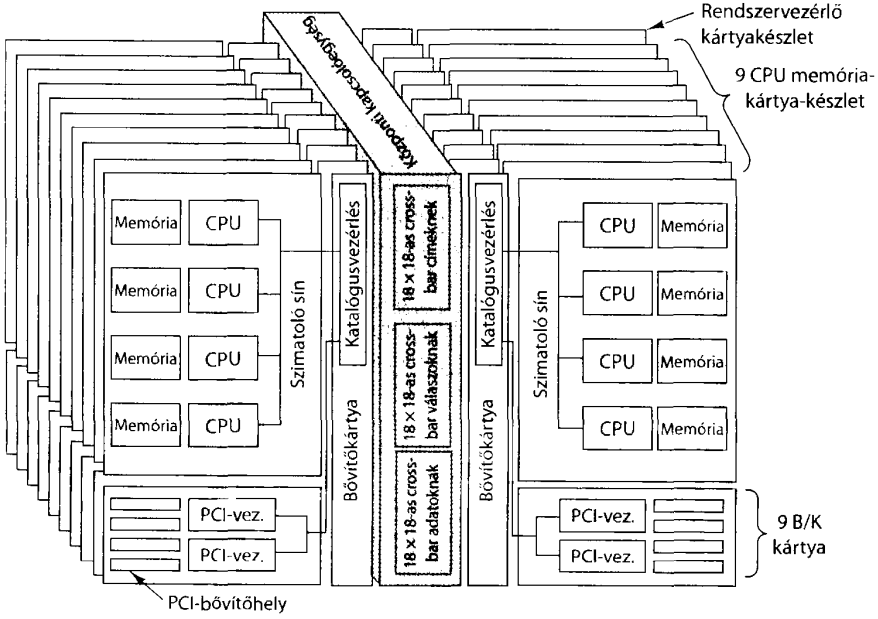
A katalógusos módszer egy másik, fejlettebb modellje nyomon követi, hogy a gyorsítósor tiszta (a memóriában tárolt másolat frissített) vagy piszkos (a memóriában tárolt másolat elavult). Ha olvasási kérés érkezik egy tiszta gyorsítósorra, akkor azt a csomópont ki tudja elégíteni a memóriájából anélkül, hogy a kérést egy gyorsítótárhoz kellene továbbítania. Azonban ha piszkos gyorsítósorra érkezik olvasási kérés, azt továbbítani kell ahhoz a csomóponthoz, amelyiknél a sor van, mert csak nála van érvényes másolat. Ha csak egy gyorsítótár-másolat engedélyezett, mint a 8.31. ábránál, akkor nincs semmi előnye a tisztasági állapot nyilvántartásának, mivel minden új kérés együtt jár a másolat érvénytelenítését kérő üzenet kiküldésével.

Természetesen a gyorsítósorok aktuális vagy elavult voltának nyilvántartása maga után vonja, hogy egy gyorsítósor módosulásakor a birtokló csomópontot tájékoztatni kell, még akkor is, ha csak egy gyorsítósor-másolat létezik. Ha több másolat van, akkor azok egyikének módosítása megköveteli a többi érvénytelenítését, ezért szükség van bizonyos szabályok (protokoll) betartására a versenyhelyzetek elkerülésére. Például egy megosztott gyorsítósor módosításához a módosítás *előtt* kizárólagos hozzáférést kellene kérni. Egy ilyen kérés lehetővé tenné a többi másolat érvénytelenítését még az engedély megadása előtt. A CC-NUMA-gépekkel kapcsolatos további teljesítményoptimalizációs kérdések tárgyalását lásd (Stenstrom és társai, 1997).

A Sun Fire E25K NUMA-multiprocesszor

Most nézzünk egy példát közös memóriás NUMA-processzorra, és tanulmányozzuk a Sun Microsystems Sun Fire családját. Jóllehet ez a család sok modellből áll, mi az E25K-ra koncentrálunk, amely 72 darab UltraSPARC IV CPU lapkát tartalmaz. Egy UltraSPARC IV tulajdonképpen két UltraSPARC III Cu processzor, amelyek osztoznak a gyorsítótáron és a memórián. Az E15K valójában ugyanez a rendszer, csak abban egyprocesszoros CPU lapkák vannak a duplaprocesszorosak helyett. Kisebb tagjai is vannak a családnak, de a mi nézőpontunkból az a fontos, hogy a legtöbb processzort tartalmazó hogyan működik.

Az E25K rendszer akár 18 kártyakészletből is állhat, egy kártyakészlet egy CPU-memóriakártyát, egy 4 PCI bővítőhelyes B/K kártyát és egy bővítőkártyát tartalmaz. Utóbbi kapcsolja össze a CPU-memóriakártyát a B/K kártyával, illetve mindkettőt a központi kapcsolóegységgel, amely befogadja a kártyákat, és a kapcsolási logikát is tartalmazza. Minden CPU-memóriakártya 4 darab CPU lapkát és 4 darab 8 GB-os RAM memóriamodult tartalmaz. Tehát az E25K minden CPU-memóriakártyáján 8 darab CPU és 32 GB RAM van (az E15K kártyáin 4 darab CPU és 32 GB RAM). A teljes E25K így 144 darab CPU-t, 576 GB RAM-ot és 72 darab PCI-bővítőhelyet tartalmaz, ahogy a 8.32. ábrán látható. Érdekes módon a 18-as számot csomagolási megfontolások miatt választották: a 18 kártyakészletes rendszer volt a legnagyobb, ami még egy darabban kifért az ajtón. Míg a



8.32. ábra. *A Sun Microsystems E25K multiprocesszora*

programozók csak a nullákkal és egyesekkel törődnek, a mérnököknek arra is kell gondolniuk, hogy a vásárló hogyan fogja a terméket bevinni az ajtón, és hogyan helyezi el az épületben.

A központi kapcsolóegység 3 darab 18 × 18-as crossbar kapcsolóból áll, ezek segítségével kapcsolható össze egymással a 18 kártyakészlet. Egy crossbar kapcsoló a címvonalakhoz van rendelve, egy másik a válaszokhoz, míg a harmadik az adatátvitelhez. A 18 bővítőkártya mellett a központi kapcsolóegységben van még egy rendszervezérlő kártyakészlet is. Ebben egyetlen CPU van, de hozzá van kapcsolva a CD-ROM, a mágnesszalagegység, a soros vonalak és más olyan perifériák is, amelyek a rendszer indításához, karbantartásához és vezérléséhez szükségesek.

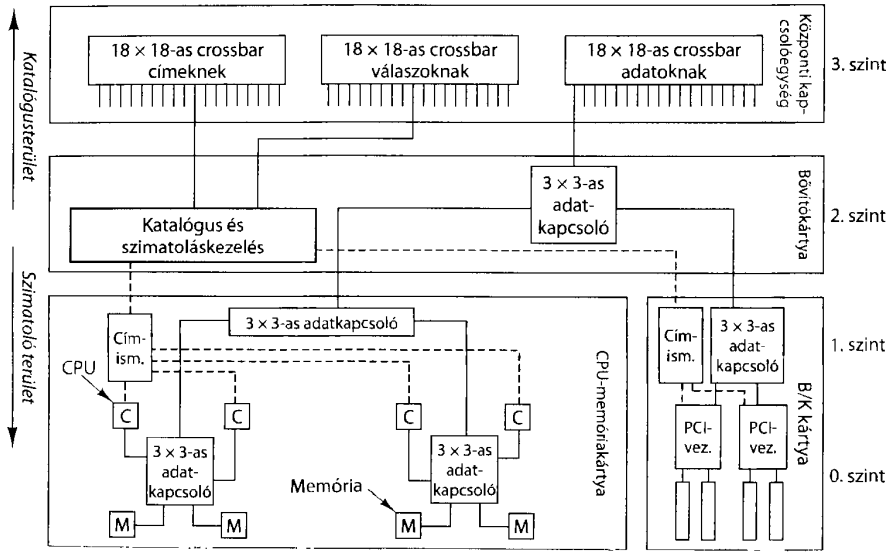
A multiprocesszorok szíve a memória-alrendszer. Hogyan kapcsoljuk össze a 144 CPU-t az osztott memóriával? A legnyilvánvalóbbak – egy nagy közös szimatoló sín vagy egy 144 × 72-es crossbar kapcsoló – nem működnek elég jól. Az első azért, mert a sín szűk keresztmetszet lesz, a második azért, mert a kapcsolót nagyon nehéz és drága megépíteni. Így az E25K és a hozzá hasonló multiprocesszorok kénytelenek jóval bonyolultabb memória-alrendszert használni.

A kártyakészletek szintjén szimatoló logika van beépítve, ezért minden lokális CPU összehasonlíthatja a kártyakészletről jövő memóriakéréseket a saját gyorsítótárában lévő blokkokkal. Amikor egy CPU-nak egy memóriaszóra van szüksége, akkor a virtuális címet először fizikai címmé konvertálja, és ellenőrzi a saját gyorsítótárát. (A fizikai címek 43 bitesek, de a csomagolási korlátok miatt a memória maximálisan 576 GB lehet csak). Ha a szükséges gyorsítósor a saját gyorsítótárában van, akkor a szót megkapja. Különben a szimatoló logika ellenőrzi, hogy a kívánt szó egy másolata megtalálható-e valahol máshol a kártyakészleten. Ha igen, akkor a kártyakészlet a kérést kielégíti. Ha nem, akkor a kérés tovább jut a 18 × 18-as cím crossbar kapcsolón keresztül az alább leírt módon. A szimatoló logika egy ellenőrzést tud végezni ciklusonként. A rendszeróra 150 MHz-en jár, tehát 150 millió ellenőrzés is elvégezhető másodpercenként minden kártyakészleten, ami 2,7 milliárd ellenőrzés másodpercenként rendszerszinten.

Habár a szimatoló logika elvileg egy sín, ahogy a 8.32. ábra is mutatja, fizikailag ez egy olyan eszközfa, amelyen a parancsok le és fel továbbítódnak a szintek között. Ha egy CPU- vagy PCI-eszköz címet generál, akkor az közvetlen kapcsolaton keresztül a 8.33. ábrán is látható címismétlők egyikéhez kerül. A két címismétlő a bővítőkártyán találkozik, amely a címeket visszaküldi a fa alsóbb szintjein elhelyezkedő eszközökhöz találat-ellenőrzésre. Azért használják ezt a felépítést, hogy ne legyen szükség olyan sínre, amely három kártyával van kapcsolatban.

Az adatátvitel négyszintű kapcsoló hálózaton keresztül valósul meg, ahogy a 8.33. ábrán is látható. Ezt a megoldást a nagy teljesítmény miatt választották. A 0. szinten CPU-memóriapárok vannak összekötve egy kis crossbar kapcsolóval, amelynek az 1. szinttel is van kapcsolata. A kétszer két CPU-memóriapár egy másik crossbar kapcsolóval van összekötve az 1. szinten. Ezek a crossbar kapcsolók szokásos ASIC-ok. Mindegyiknél az összes bemenet rendelkezésre áll mind a soroknál, mind az oszlopoknál, habár nem minden kombináció van kihasználva (és nem is mindnek lenne értelme). A kártyákon minden kapcsoló logika 3 × 3-as crossbar kapcsolókkal van megoldva.

Minden kártyakészlet három kártyából áll: egy CPU-memóriakártya, egy B/K kártya és a kettőt összekötő bővítőkártya. A 2. szintű összeköttetés is egy 3 × 3-as crossbar kapcsoló (a bővítőkártyán), amely kapcsolatot teremt a memória és a B/K kapuk között (amelyek mindig memóriába ágyazottak az összes UltraSPARC-on). A kártyakészletről jövő és az oda irányuló adatforgalom a 2. szintű crossbar kapcsolón halad keresztül, akár memóriát, akár B/K kaput érint. Végül a távoli kártyák felé menő és az onnan érkező adatok a 18 × 18-as crossbar kapcsolót használják a 3. szinten. Az átvitel 32 bájt széles, tehát a leggyakrabban előforduló 64 bájtos adategység átvitele két óraciklust igényel.



8.33. ábra. *A Sun Fire E25K négyszintű kapcsoló hálózatot használ. A szaggatott vonalak a címutaknak felelnek meg, a folytonos vonalak pedig az adatutaknak*

Miután áttekintettük a komponensek elhelyezkedését, fordítsuk figyelmünket a közös memória működésére. A legalsó szinten az 576 GB memória 2^{29} darab 64 bájtos blokkra van osztva. Ezek a blokkok a memóriarendszer oszthatatlan építőkövei. Minden blokknak van egy gazdakártyája, ahol elhelyezkedik, amikor nincs másutt használatban. A legtöbb blokk az idő nagy részében a gazdakártyáján van. Amikor azonban egy CPU-nak a saját kártyájáról vagy a 17 másik kártya egyikéről szüksége van egy memóriablokkra, akkor először kér egy másolatot a gyorsítótárába, majd ezt a másolatot használja. Habár az E25K minden CPU lapkája két CPU-t tartalmaz, ezek osztoznak a gyorsítótáron és így a bennük tárolt blokkokon is.

Minden CPU lapkán minden memóriablokk és gyorsítósor három állapot egyikében lehet:

1. Kizárólagos hozzáférés (írásra).
2. Megosztott hozzáférés (olvasásra).
3. Érvénytelen (például üres).

Amikor valamelyik CPU egy memóriaszót akar írni vagy olvasni, akkor először a saját gyorsítótárát ellenőrzi. Ha ott nem találja a keresett szót, akkor a fizikai címre először egy helyi kérést bocsát ki, amely csak a saját kártyakészletére jut el. Ha a kártyakészlet valamelyik gyorsítótárában megvan a keresett sor, akkor a szimatoló logika észreveszi az egyezést, és kiszolgálja a kérést. Ha a sor kizárólagos hozzáférési módban van, akkor átkerül a kérőhöz, az eredeti pedig érvénytelen állapotba kerül. Ha megosztott módban van, akkor a gyorsítótár nem reagál, mert ilyenkor a gazdakártyához kell fordulni.

Ha a szimatoló logika nem találja a gyorsítósort, vagy megtalálja, de megosztott, akkor a központi kapcsoló egységen keresztül megkérdezi a gazdakártyát, hogy hol van a blokk. Minden blokk állapota fel van jegyezve a blokk ECC bitjeiben, ezért a gazdakártya azonnal meg tudja állapítani az állapotot. A blokk akár megosztott egy vagy több távoli kártyával, akár nem, a gazdakártya memóriája friss, és a kérés azonnal kielégíthető a gazdakártya memóriájából. Ebben az esetben a másolat az adatátviteli crossbar kapcsolón át két óraciklus alatt a kérő CPU-hoz kerül.

Ha a kérés olvasásra vonatkozott, akkor új bejegyzés kerül a gazdakártya katalógusába, amely jelzi, hogy új tag került a gyorsítósort megosztva használók közé, és a tranzakció lezárásra kerül. Ha azonban a kérés írásra vonatkozott, akkor érvénytelenítő üzenetet kell küldeni az összes olyan kártyának, amelynek van másolata (ha vannak ilyenek). Ily módon az írást kérő kártyához kerül az egyetlen másolat.

Most tekintsük azt az esetet, amikor a kért blokk egy távoli kártyán kizárólagos módban van. A gazdakártya a kérés megérkezése után a katalógus alapján megállapítja a távoli kártya helyét, és a kérővel közli, hogy hol található a gyorsítósor. A kérő most már a megfelelő kártyához küldheti a kérést. Amikor a kérés megérkezik, akkor a távoli kártya visszaküldi a sort. Ha olvasási kérés volt, akkor a sor megosztott állapotba kerül, és a gazdakártya is kap egy másolatot. Ha írási kérés volt, akkor a távoli kártya érvényteleníti a saját példányát, így a kérőnek lesz csak másolata.

Mivel minden kártyán 2^{29} darab memóriablokk van, mindet nyomon követni a legrosszabb esetben csak egy 2^{29} bejegyzést tartalmazó katalógussal lehetne. Mivel egy (asszociatív keresést megvalósító) katalógus ennél sokkal kisebb, előfordulhat, hogy hely hiányában nem lehet felvenni egy bejegyzést. Ebben az esetben a gazdakártya egy mind a 17 többi kártyának elküldött kéréssel tudja csak megállapítani a blokk helyét. A válaszokat kezelő crossbar kapcsoló szerepe a katalógus-koherencia- és frissítésprotokollban, hogy a kérőhöz irányítja a visszaáramló adatokat. A protokollforgalmat két (cím és válasz) sín között megosztva, az adatokat pedig az adatsínen továbbítva a rendszer áteresztőképessége növekszik.

A Sun Fire E25K a terhelést több kártya eszközei között szétosztva nagyon nagy teljesítményre képes. A korábban említett 2,7 milliárd másodpercenkénti szimatoláson túl a központi kapcsolóegység akár kilenc egyidejű átvitelt képes lebonyolítani, kilenc adó és kilenc fogadó kártya között. Mivel az adatok crossbar kapcsolója 32 bájt széles, minden óraciklusban 288 bájt haladhat keresztül a központi kap-

csolóegységen. 150 MHz-es órajelen ez azt jelenti, hogy az összesített legnagyobb sávszélesség 40 GB/másodperc, ha minden adathozzáférés távoli. Ha a szoftver el tudja helyezni a lapokat úgy, hogy a legtöbb hozzáférés lokális legyen, akkor a rendszer sávszélessége 40 GB/másodpercnél jelentősen nagyobb lehet.

A Sun Fire-ről további technikai jellegű információkat lásd (Charlesworth, 2002; 2001).

8.3.5. COMA-multiprocesszorok

A NUMA- és a CC-NUMA-gépek hátránya, hogy a távoli memória elérése sokkal lassabb, mint a helyi memóriáé. A CC-NUMA ezt a teljesítménybeli különbséget a gyorsítótárazással bizonyos mértékig elrejti. Azonban, ha az igényelt távoli adatok mennyisége nagyban meghaladja a gyorsítótár kapacitását, akkor a gyorsítótárhiány állandósul, és ez a teljesítmény rovására megy.

Így az a helyzet, hogy az UMA-gépek teljesítménye kiváló, de a méretük korlátozott és meglehetősen drágák. Az NC-NUMA-gépek valamivel nagyobb méretűre skálázhatók, de megkövetelik a lapok kézi vagy félig automatikus elhelyezését, gyakran kétes eredménnyel. Az a probléma, hogy nehéz megjósolni, hogy melyik lapra hol lesz szükség, de ezt leszámítva is sok esetben a lapok mérete túl nagy a mozgatáshoz. A Sun Fire E25K-hoz hasonló CC-NUMA-gépek teljesítménye visszaesik, ha sok CPU-nak van szüksége nagy mennyiségű távoli adatra. Mindezek alapján azt mondhatjuk, hogy ezeknek az architektúráknak komoly korlátai vannak.

A multiprocesszorok egy másik típusa megpróbálja ezeket a problémákat kikerülni azáltal, hogy mindegyik CPU főmemóriáját úgy használja, mint egy gyorsítótárat. Ebben a **COMA (Cache Only Memory Access, gyorsítótáras memóriaelérés)** elnevezésű architektúrában a lapokhoz nincs rögzített tulajdonos gép, mint a NUMA és a CC-NUMA esetében. Valójában a lapok egyáltalán nem játszanak jelentős szerepet.

Helyette a fizikai címtartomány van gyorsítósorokba szétosztva, amelyek az igények szerint vándorolnak a rendszerben. A blokkoknak sincs tulajdonosgépük. Néhány harmadik világhoz tartozó ország nomádjaihoz hasonlóan, az otthon ott van, ahol éppen vagy. **Vonzásmemóriá**nak nevezik az olyan memóriát, amely éppen a szükséges sorokat tárolja el (vonzza magához). A fő RAM-ot óriási gyorsítótárként használva, a találati arány nagyban megnő, ennél fogva a teljesítmény is.

Sajnos, mint általában, ez sem olyan dolog, mint egy ingyen vacsora. A COMA-rendszernek két új problémával kell szembenézni:

1. Hogyan találjuk meg a gyorsítósorokat?
2. Mi történik, ha a memóriából egy olyan sort törlünk, ami az utolsó másolat?

Az első probléma azzal függ össze, hogy miután az MMU átalakít egy virtuális címet fizikai címmé, és a sor nincs az igazi hardvergyorsítótárban, akkor nem könnyű megmondani, hogy egyáltalán bent van-e a főmemóriában. A lapozásos hardver itt nem segít, mivel mindegyik lap sok egyedi, egymástól függetlenül vándorló

gyorsítósorból épül fel. Ráadásul, még ha tudná is, hogy a sor nincs a főmemóriában, továbbra is ott a kérdés, hogy hol van? Nem lehet egyszerűen megkérdezni a tulajdonosgépet, mivel olyan nincs.

Van néhány javaslat az elhelyezési problémára megoldására. Új hardvert lehet alkalmazni a főmemóriában lévő gyorsítósorok azonosító címkéinek tárolásához. Az MMU-nak ekkor lehetősége lenne az igényelt sor címkéjét összehasonlítani a memóriabeli összes gyorsítósor címkéjével, hogy kiemelhesse a megfelelőt. Ehhez a megoldáshoz több hardverre van szükség.

Egy némileg más megoldás leképezi a teljes lapokat, de nem követeli meg, hogy az összes gyorsítósor jelen legyen. Ennél a megoldásnál a hardvernek egy laponkénti bittérképre van szüksége, ebben gyorsítósoronként egy bit jelzi, hogy a sor jelen van, vagy hiányzik. Ebben az **egyszerű COMA**-ként ismert architektúrában, ha egy gyorsítósor jelen van, akkor a saját lapján a megfelelő pozícióban kell lennie, de ha nincs jelen, akkor bármilyen alkalmazására vonatkozó kísérlet megszakítást idéz elő, hogy lehetővé tegye a szoftvernek megkeresni, és behozni a sort.

Ez elvezet bennünket a valóban távoli sorok megtalálásának a kérdéséhez. Az egyik megoldás szerint mindegyik laphoz hozzárendelnek egy tulajdonosgépet aszerint, hogy hol van a katalógusbejegyzése, de az adatnak nem kell ott lennie. Ezt követően a tulajdonosgéphez küldhető egy üzenet, hogy legalább keresse meg a gyorsítósort. Egy másik módszer a memóriát fastruktúrába szervezni, és felfelé keresni, a sor megtalálásáig.

A listában a második problémát úgy kell megoldani, hogy a legutolsó másolat nem kerülhet törlésre. Mint a CC-NUMA-gépeknél is, egy gyorsítósor egyszerre több csomópontnál is megjelenhet. Mikor gyorsítótárhiány lép fel, akkor egy sort be kell hozni, ami rendszerint azt jelenti, hogy egy másik sort el kell dobni. Mi történik, ha az éppen kiválasztott sor a legutolsó másolat? Ebben az esetben ezt nem szabad eldobni.

Az egyik megoldás visszamegy a katalógushoz, és azt vizsgálva megnézi, van-e több másolat. Ha igen, akkor a sor biztonsággal eldobható. Egyébként máshova kell elmozgatni. Egy másik megoldásnál mindegyik gyorsítósornak egy másolatát fő másolatként címkéznek meg, és ezt sosem dobják el. Ekkor nem kell a katalógust lekérdezni. Mindent egybevetve, a COMA jobb teljesítménnyel kecsegtet, mint a CC-NUMA, de kevés COMA-gépet építettek eddig, így nincs még elegendő tapasztalat ezekkel kapcsolatban. Az eddig megépült két COMA-gép a KSR–1 (Burkhardt és társai, 1992) és a Data Diffusion Machine (Hagersten és társai, 1992). Egy ezeknél újabb gép az SDAARC (Eschmann és társai, 2002).

8.4. Üzenetátadásos multiszámítógépek

Miként a 8.21. ábrán már láttuk, a MIMD párhuzamos processzoroknak két típusa van, a multiprocesszorok és a multiszámítógépek. Eddig a multiprocesszorokat vizsgáltuk meg. Láttuk, hogy az operációs rendszer számára a multiprocesszorok olyan közös memóriával rendelkeznek, amely a szokványos LOAD és STORE utasí-

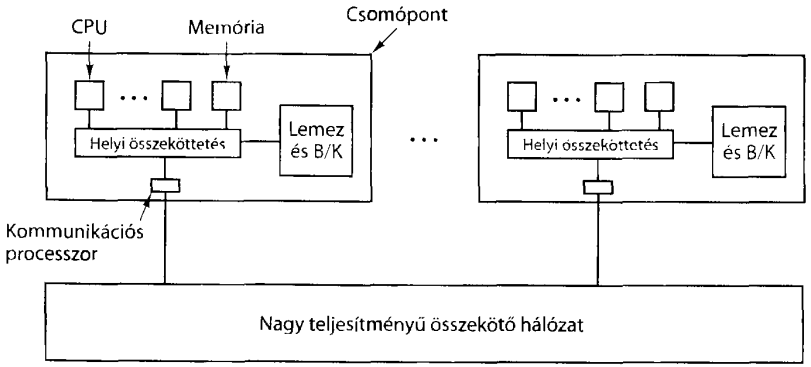
tásokkal érhető el. Azt is láttuk, hogy a közös memória többféleképpen megvalósítható, beleértve a sínek szimatolása, crossbar kapcsoló, többszintű kapcsoló hálózatok és a különböző katalógusalapú rendszerek alkalmazását. Akárhogyan is történik, egy multiprocesszorra írt program anélkül érheti el a memória bármelyik rekeszét, hogy tudomása lenne a belső topológiáról vagy a megvalósítási sémáról. A multiprocesszorokat ez az illúziókeltés teszi vonzóvá számunkra és kedveltté a programozók körében.

Másrészt, a multiprocesszoroknak is vannak olyan hátrányos tulajdonságaik, amelyek miatt a multiszámítógépek is fontos szerepet töltenek be: Mindenekelőtt a multiprocesszorok nagy méretűre nem skálázhatók. Láttuk, hogy a Sunnak milyen hatalmas mennyiségű hardvert kellett alkalmaznia ahhoz, hogy 72 CPU-sra skálázza az E25K rendszert. Ezzel szemben az alábbiakban vizsgálat alá kerülő multiszámítógép 65 536 CPU-t tartalmaz. Évek fognak még eltelni, amíg egy ugyanekkora kereskedelmi célú multiprocesszort épít valaki, addigra viszont a milliófeldolgozós multiszámítógépek már használatban lesznek.

Ráadásul a multiprocesszoroknál a memóriáért folyó versengés hátrányosan befolyásolhatja a teljesítményt. Ha 100 CPU állandóan ugyanazon változókat akarja olvasni vagy írni, akkor a különböző memóriákért, sínekért és katalógusokért folyó versengés óriási teljesítményromlást eredményez.

Ezek és más tényezők következményeként nagy az érdeklődés olyan párhuzamos számítógépek építése és használata iránt, amelyekben mindegyik CPU-nak saját memóriája van, és más CPU-k ezt a memóriát közvetlenül nem érik el. Ezek a multiszámítógépek. A multiszámítógép CPU-inak programjai a send és a receive primitíveket használva explicit üzenetküldéssel érintkeznek egymással, mert egymás memóriáját a LOAD és a STORE utasításokkal nem tudják elérni. Ez a különbség a programozási modellt teljesen megváltoztatja.

Egy multiszámítógépben mindegyik csomópont egy, esetleg néhány CPU-t, bizonyos mennyiségű RAM-ot (amit feltehetőleg csak a csomóponthoz tartozó CPU-k használnak megosztva), egy lemezegységet és/vagy más B/K eszközöket,



8.34. ábra. *Egy általános multiszámítógép felépítése*

valamint egy kommunikációs processzort tartalmaz. A kommunikációs processzorokat egy nagy sebességű összekötő hálózat kapcsolja össze, ilyenekkel a 8.4.1. részben foglalkozunk. Sokféle topológiát, kapcsolási sémát és útvonalválasztó algoritmust alkalmaznak. Minden multiszámítógép közös jellemzője, hogy mikor egy alkalmazói program a send primitívet hajtja végre, akkor a kommunikációs processzor értesül erről, és ez mozgatja át a célgéphez a felhasználó adatblokkját (esetleg azután, hogy kért és kapott erre engedélyt). Egy általános multiszámítógépet mutat a 8.34. ábra.

8.4.1. Összekötő hálózatok

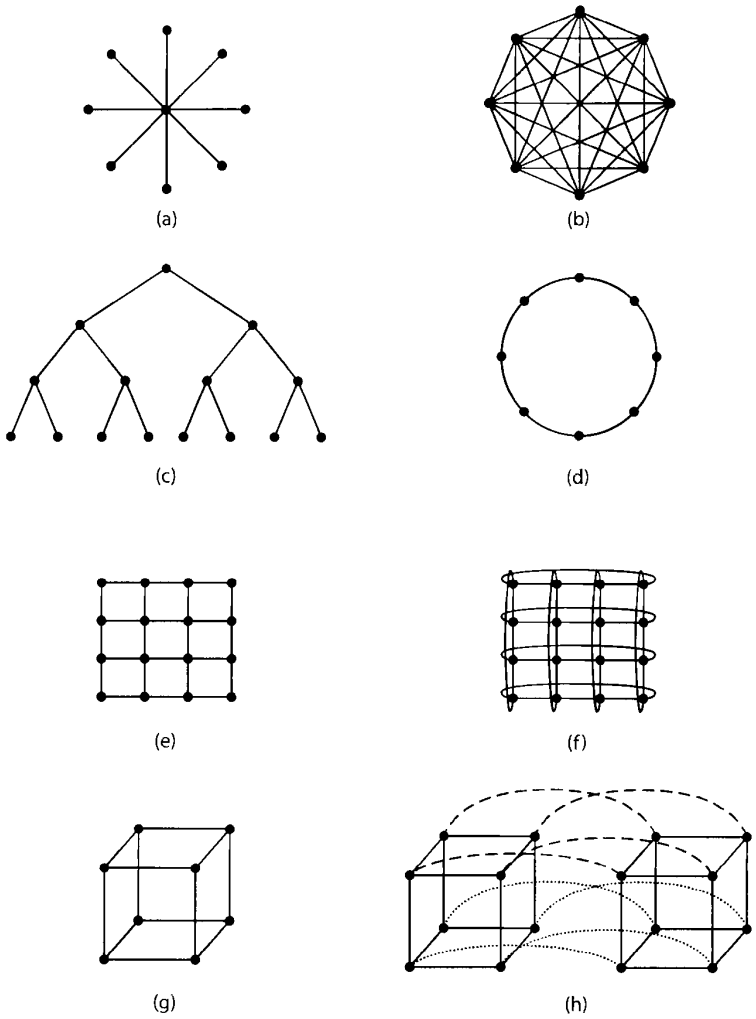
A 8.34. ábrán láttuk, hogy a multiszámítógépeket összekötő hálózatok fogják egységbe. Itt az idő, hogy szemügyre vegyük az összekötő hálózatokat. Elég érdekes, hogy a multiprocesszorok és a multiszámítógépek meglepően hasonlók ebből a szempontból, aminek az az oka, hogy gyakran a multiprocesszoroknak több memóriamoduljuk van, és ezeknek egymással és a CPU-kkal is kapcsolatban kell lenniük. Így aztán az itt tett megállapítások gyakran mindkétfajta rendszerre érvényesek.

A multiprocesszorok és a multiszámítógépek összekötő hálózatai alapvetően azért hasonlítanak egymásra, mert mindkétfajta rendszer a legalsó szinten üzenetküldést alkalmaz. Még egy egyetlen CPU-t tartalmazó rendszerben is, ha a CPU írni vagy olvasni akar a memóriából, akkor a CPU a sín bizonyos vonalain közli a kérést, majd vár a válaszra. Ez a tevékenység alapjában véve olyan, mint egy üzenetküldés: a kezdeményező küld egy kérést, majd vár a válaszra. Nagy multiprocesszoros rendszerekben a CPU-k és a távoli memóriák közötti kommunikáció majdnem mindig abból áll, hogy a CPU küld egy **csomag**nak nevezett üzenetet a memóriához, ebben kéri az adatokat, a memória pedig egy válaszcsomagot küld vissza.

Topológia

Egy összekötő hálózatban a kapcsolók és a kapcsolatok elrendezettségét a topológiával írják le, például, lehet ez egy gyűrű vagy egy rács. A topológiai felépítések gráfokkal modellezhetők, amelyekben az élek a kapcsolatoknak, a csomópontok pedig a kapcsolóknak felelnek meg, ahogyan a 8.35. ábrán látható. Egy összekötő hálózat (vagy gráf) minden egyes csomópontja bizonyos számú kapcsolattal rendelkezik. A matematikusok egy csomópont kapcsolatainak számát a csomópont **fokszám**ának nevezik; a mérnökök ezt **terhelhetőségi szám**nak hívják. Általában minél nagyobb a terhelhetőségi szám, annál több útvonalválasztás lehetséges, és annál nagyobb a hibatűrés, vagyis az az adottság, hogy a hálózat tovább tudjon működni egy hibás kapcsolat esetén is úgy, hogy más útvonalak lépnek a kieső helyébe. Ha minden csomóponthoz k él csatlakozik, és jók az összeköttetések, akkor meg lehet a hálózatot úgy tervezni, hogy még $k - 1$ hibás kapcsolat esetén is összefüggő maradjon.

Egy összekötő hálózat (vagy a gráf) egy másik jellemző tulajdonsága az **átmérő**. Ha két csomópont közötti távolságot azon élek számával mérjük, amelyeken keresztül kell haladni, hogy egyikből a másikat elérjük, akkor egy gráf átmérője a két legtávolabbi csomópont közötti távolság. Egy összekötő hálózat átmérője kifejezi a legrosszabb áthaladási időt, mivel a CPU-ból CPU-ba vagy CPU-ból memóriához küldött csomagok az éleken bizonyos véges idő alatt haladnak keresztül. Minél kisebb az átmérő, annál jobb a teljesítmény a legrosszabb esetben is. A csomópontok közötti átlagos távolság szintén fontos mennyiség, mert kifejezi egy csomag átlagos haladási idejét.



8.35. ábra. *Topológiák. A telt pontok a kapcsolókat jelzik. A CPU-k és a memóriák nincsenek feltüntetve. (a) Csillag. (b) Teljes összekötés. (c) Fa. (d) Gyűrű. (e) Rács. (f) Kettős tórusz. (g) Kocka. (h) 4D-s hiperkocka*

Egy összekötő hálózatnak van még egy további fontos tulajdonsága, az áteresztőképesség, amely a másodpercenként mozgatható adatok mennyisége. Ez a tulajdonság a hálózat **kettévágott sávszélesség**ével fejezhető ki. Ezt a mennyiséget a következőképpen határozzák meg: először a hálózatot képzeletben két (a csomópontok száma szerint) egyenlő részre bontják fel úgy, hogy a gráfból éleket hagynak el. Ezután kiszámítják az eltávolított élek teljes sávszélességét. Mivel egy hálózat két egyenlő részre bontása sokféleképpen történhet, ezért az ezekhez tartozó teljes sávszélességek is különbözők lehetnek. Egy összekötő hálózat kettévágott sávszélessége ezen számok minimuma. Ez a szám azért kifejező, mert ha például a kettévágott sávszélesség 800 bit/s, akkor ha a két fél között élénk a kommunikáció, a legrosszabb esetben sem rosszabb az áteresztőképesség 800 bit/s-nál. Sok tervező úgy véli, hogy egy összekötő hálózat legfontosabb jellemzője a kettévágott sávszélesség. Sok összekötő hálózat megtervezésénél a fő szempont, hogy a kettévágott sávszélesség maximális legyen.

Az összekötő hálózatok jellemezhetők még a **dimenziószám**ukkal. A mi céljainkra a dimenzió a forrásból a célba való eljutási lehetőségek száma. Ha nincs választási lehetőség (azaz csak egyetlen út van az egyes forrásokból a célokhoz), akkor a hálózat dimenziója nulla. Ha van választási lehetőség például aközött, hogy keletre vagy nyugatra megy, akkor a hálózat egydimenziós. Ha van két tengely, vagyis ha egy csomag kelet és nyugat közül is választhat, vagy észak és dél közül is, akkor a hálózat kétdimenziós, és így tovább.

A 8.35. ábra néhány topológia sémát mutat. Csak a kapcsolatok (vonalak) és a kapcsolók (pontok) vannak feltüntetve. A memóriák és a CPU-k (nem láthatók) általában interfésszel illeszkednek a kapcsolókhoz. A 8.35. (a) ábrán egy nulladimenziós **csillag** elrendezés látható, ahol a CPU-k és a memóriák a külső csomópontokhoz köthetők, van benne egy központi csomópont, amelynek a feladata a kapcsolások lebonyolítása. Ez egy egyszerű topológia, de nagy rendszerek esetén a központi kapcsoló valószínűleg szűk keresztmetszetnek bizonyulna. A hibatűrés szempontjából pedig ez egy szegényes terv, mivel a központi kapcsoló kiesése esetén az egész rendszer működésképtelenné válik.

A 8.35. (b) ábrán egy másik elrendezés látható, amely a spektrum másik végéből való, ez a **teljes összekötésű** topológia. Itt minden csomópont minden más csomóponttal közvetlen kapcsolatban van. Ennél maximális a kettévágott sávszélesség, minimális az átmérő, és kiemelkedően nagy a hibatűrés (bármely hat kapcsolat kieshet, mégis összeköttetésben marad a rendszer). Sajnos a szükséges kapcsolatok száma k csomópont esetén $k(k-1)/2$, ami k növelésével kezelhetetlenné válik.

A 8.35. (c) ábrán a **fa** topológia látható. Ennek az a hátránya, hogy a kettévágott sávszélessége megegyezik a kapcsolatok áteresztőképességével. Mivel a fa gyökere közelében az adatforgalom általában nagy, ezért a fa felső néhány csomópontja szűk keresztmetszetűnek bizonyulhat. Ennek a problémának az egyik megoldása a kettévágott sávszélesség növelése úgy, hogy a fentebb levő kapcsolódásokat nagyobb sávszélességgel valósítják meg. Például a legalsó szint kapcsolatai lehetné-

nek b kapacitásúak, a következő szintnél $2b$ kapacitásúak, a felső szintnél pedig $4b$ kapacitásúak. Az ilyen topológiát **kövér fá**nak nevezik, és a kereskedelmi forgalmazású multiszámítógépeknél alkalmazták, ilyen például (a ma már elavult) Thinking Machines' CM–5.

A 8.35. (d) ábrán látható **gyűrű** az általunk bevezetett definíció szerint egydimenziós topológia, mivel minden elküldött csomag két lehetőség közül választhat, vagy jobbra megy, vagy balra. A 8.35. (e) ábrán levő **rács** vagy **háló** sok kereskedelmi rendszernél alkalmazott kétdimenziós szerkezet. Nagyon szabályos, könnyű nagy méretűre növelni, és az átmérője a csomópontok számának négyzetgyökével növekszik. A 8.35. (f) ábrán a rács egy módosított változata, a **kettős tórusz** látható, amely egy rács, de ennek a végpontjai között kapcsolat van. Nemcsak a hibatűrése nagyobb a rácsénál, hanem az átmérője is kisebb, mert az átellenes csomópontok közötti kommunikáció mindössze két kapcsolaton keresztül végbemehet.

Egy másik népszerű topológia a háromdimenziós tórusz. Ez egy háromdimenziós (i, j, k) pontokat tartalmazó rácson alapszik, ahol minden koordináta egész szám, a sarkok koordinátái (1, 1, 1) és (l, m, n). Minden csúcsnak hat szomszédja van, minden tengely mentén kettő. A kocka lapjain lévő csúcsoknak kapcsolata van az átellenes lap csúcsaival, hasonlóan a kétdimenziós tóruszhoz.

A 8.35. (g) ábrán a **kocka**, egy szabályos háromdimenziós topológia látható. Itt egy $2 \times 2 \times 2$-es kocka szerepel, de általános esetben a kocka mérete $k \times k \times k$. A 8.35. (h) ábra egy négydimenziós kockát szemléltet,* ami két háromdimenziós kockából, azok megfelelő csomópontjainak összekötésével alakult ki. Készíthető ötdimenziós kocka a 8.35. (h) ábrában szereplő struktúra két példányából a megfelelő csomópontok összekötésével, és így egy négy kockából álló blokk alakul ki. Alkalmazható ez az eljárás a hatdimenziós esethez, ahol már két négy kockából álló blokkot kell venni, és ezek megfelelő csomópontjait kell összekötni és így tovább. Az így előállítható n dimenziós kockát **hiperkocká**nak hívják. Ezt a topológiát sok párhuzamos számítógép használja, mivel ennek az átmérője lineárisan nő a dimenziójával. Más szóval az átmérő a csomópontok számának 2-es alapú logaritmusa, így például egy 10 dimenziós hiperkockának 1024 csomópontja van, de az átmérője csak 10, ami kiváló a késleltetési időt tekintve. Megjegyezzük, hogy ha az 1024 csomópont 32×32-es rácsba van rendezve, akkor ennek átmérője 62, ami hatszor rosszabb a hiperkockáénál. A kisebb átmérőnek az az ára, hogy a csomópontok fokszáma és ezzel együtt a kapcsolatok száma (és a költség) sokkal nagyobb a hiperkockánál. Ennek ellenére a nagy teljesítményű rendszerekhez gyakran választják a hiperkocka topológiát.

Nagyon sok fajta multiszámítógép van, ezért nehéz egyértelműen csoportosítani őket. Mindenesetre két általános „stílus" látszik kibontakozni: az MPP és a klaszter. A következőkben ezeket fogjuk tanulmányozni.

* A dimenzió nem a hagyományos geometriai dimenziót jelenti. *(A lektor)*

8.4.2. MPP – erősen párhuzamos processzor

Az első kategória az **MPP (Massively Parallel Processor, erősen párhuzamos processzor)**, ezek hatalmas, több millió dolláros szuperszámítógépek. Tudományos kutatásoknál, tervezésnél és az iparban használják nagyon erőforrás-igényes számításokhoz vagy másodpercenkénti nagyon sok tranzakció kezeléséhez, illetve adatbankokhoz (óriási adatbázisok tárolására és kezelésére). Kezdetben az MPP-ket elsősorban tudományos szuperszámítógépekként használták, de mára ezek legtöbbje már kereskedelmi környezetben működik. Bizonyos értelemben ezek a gépek az 1960-as évek hatalmas nagygépeinek az utódai (azonban ez a kapcsolat olyan gyenge, mintha egy paleontológus azt állítaná, hogy egy verébsereg a Tyrannosaurus Rex leszármazottja). Az MPP-k nagyrészt felváltották a SIMD gépeket, a vektoros szuperszámítógépeket és a tömbprocesszorokat a digitális táplálékláncon csúcsán.

Ezeknek a gépeknek a legtöbbje processzorként szabványos CPU-kat használ. Népszerű az Intel Pentium, a Sun UltraSPARC, az IBM PowerPC. Az MPP-k megkülönböztető jegye a nagy teljesítményű összekötő hálózat, amelyen keresztül az üzeneteket rövid késleltetési idővel és nagy sávszélességgel tudják mozgatni. Mindkét tulajdonság fontos, mert az üzenetek túlnyomó többsége kisméretű (jóval 256 bájt alatti), de a teljes forgalom döntő részét a nagyméretű (8 KB-nál nagyobb) üzenetek adják. Az MPP-k rendelkeznek széles körű saját szoftverrel és eljáráskönyvtárakkal is.

Az MPP-k egy másik jellegzetessége a hatalmas B/K kapacitás. Az MPP-ket igénylő méretű problémák megoldása során mindig hatalmas mennyiségű adatot kell feldolgozni, gyakran terabájt nagyságrendben. Ezeket az adatokat muszáj több lemez közt szétosztani, és a gépben való mozgatásukat nagy sebességgel kell végezni.

Végül még MPP-sajátosság az is, hogy nagy figyelmet fordítanak a hibatűrésre. Több ezer CPU-nál hetente néhány meghibásodás elkerülhetetlenül bekövetkezik. Elfogadhatatlan, hogy egy 18 órás futás kárba vesszen egyetlen CPU meghibásodása miatt, különösen, ha minden hétre várható egy ilyen meghibásodás. Éppen ezért a nagy MPP-k mindig rendelkeznek olyan speciális rendszerfigyelő hardverrel és szoftverrel, amellyel észlelik a hibákat, és zökkenőmentesen kijavítják azokat.

Most az lenne a természetes, ha rátérnénk az MPP-felépítés általános elveinek tanulmányozására, de a helyzet az, hogy nem sok alapelv van. Közelebbről vizsgálva egy MPP többé-kevésbé szabványos feldolgozóelemekből áll, amelyek valamilyen, az általunk már tanulmányozottakhoz hasonló nagy sebességű hálózattal vannak összekötve. Ezért inkább most két konkrét MPP-példát nézünk meg; ezek a BlueGene/L és a Red Storm.

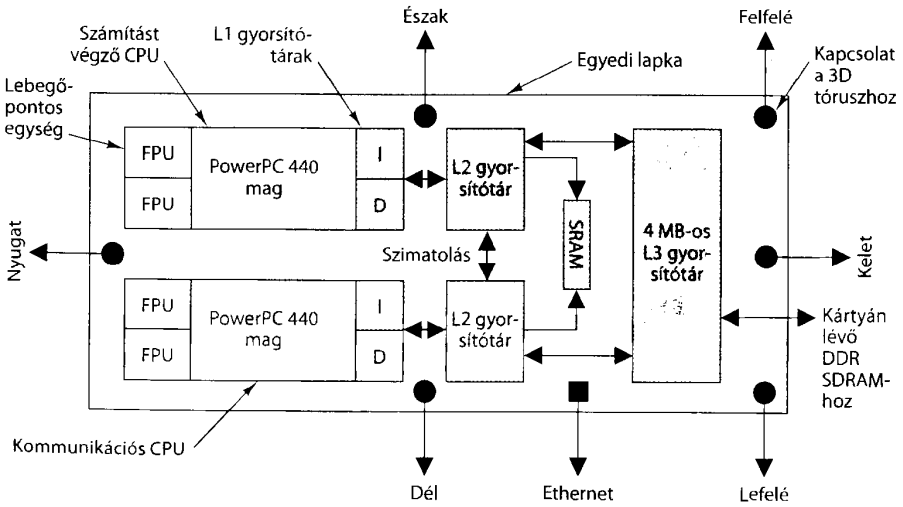
BlueGene

Erősen párhuzamos processzorra első példaként megvizsgáljuk az IBM BlueGene rendszerét. Az IBM 1999-ben indította ezt a projektet, amelynek az volt a célja, hogy kifejlesszenek egy erősen párhuzamos szuperszámítógépet, amelyet több-

bek között az élettudományok számításigényes feladatainak megoldására akartak felhasználni. A biológusok meg vannak győződve arról, hogy a fehérjék háromdimenziós szerkezete meghatározza a funkciójukat, mégis egyetlen kis fehérje háromdimenziós szerkezetének a fizika törvényei alapján történő meghatározása az akkori szuperszámítógépeken éveket vett igénybe. Az emberekben több mint félmillió fehérje található, némelyik rendkívül nagy, és ezek hibás térbeli elrendeződése tehető felelőssé bizonyos betegségekért (például a cisztás fibrózisért). Világos, hogy az összes emberi fehérje szerkezetének meghatározásához a világ számítási kapacitását sok nagyságrenddel meg kellene növelni, és a fehérjék térbeli elrendeződésének modellezése csak egy a BlueGene-nek szánt feladatok közül. Ugyanilyen összetettségű kihívások vannak még a molekuláris dinamika, az éghajlatmodellezés és a csillagászat területén, de még a pénzügyi modellezés is nagyságrendi növekedést igényel a szuperszámítógépektől.

Az IBM-nél úgy érezték, hogy van megfelelő kereslet az erős szuperszámítógépekre, és 100 millió dollárt fektettek be a BlueGene tervezésébe és megépítésébe. 2001 novemberében az Egyesült Államok Energiaügyi Minisztériuma által működtetett Livermore National Laboratory betársult, és egyben első vásárlója is lett a BlueGene család első példányának, a **BlueGene/L**-nek.

A BlueGene projekt célja nem csak a világon leggyorsabb MPP elkészítése volt, hanem azt is el akarták érni, hogy a leghatékonyabb legyen teraflop/dollár, teraflop/watt és teraflop/m^3 szempontból. Emiatt az IBM elvetette a korábbi MPP-filozófiát, miszerint a pénzért kapható legdrágább alkatrészeket kell használni. Ehelyett azt a döntést hozták, hogy alapkomponensként egy egyedi, komplett részrendszert magában foglaló mérsékelt sebességű és fogyasztású lapkát gyártanak, amelyekből aztán sokat zsúfolnak össze kis helyre és egy nagyon nagy gépet építenek. Az első



8.36. ábra. *A BlueGene/L egyedi processzorlapkája*

lapka 2003 júniusában készült el. A BlueGene/L-rendszer első negyede 16 384 feldolgozóegységgel 2004 novemberére már működött, és 71 teraflop/másodperces sebességével a Föld leggyorsabb szuperszámítógépének címét is elnyerte. Csekély 0,4 megawattot fogyasztott, így kategóriájában a fogyasztási hatékonyság győztese is lett 177,5 megaflop/watt-tal. A teljes, 65 536 feldolgozóegységet tartalmazó rendszer maradék részének üzembeállítását 2005 nyarára ütemezték.

A BlueGene/L-rendszer szíve a 8.36. ábrán szemléltetett egyedi feldolgozóegység lapkája, amely két 700 MHz-es PowerPC 440 magból áll. A PowerPC 440 egy csővezetékes, 2 utasítást egyszerre kiadni képes szuperskalár processzor, amely nagyon közkedvelt beágyazott rendszerekben. Minden magban van két darab, egyszerre 2 utasítást kiadni képes lebegőpontos egység, tehát együtt összesen 4 lebegőpontos utasítás kiadására képesek órajelenként. A lebegőpontos egységeket kiegészítették néhány olyan SIMD-jellegű utasítással, amelyek hasznosak bizonyos vektorokon végzett tudományos számításoknál. Noha a lapka nem éppen lomha, azért önmagában nem is a legerősebb multiprocesszor.

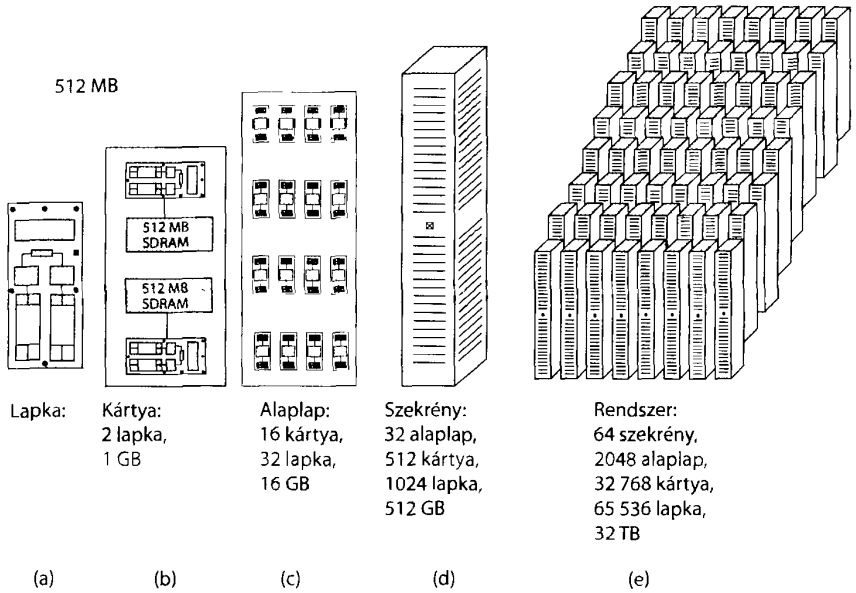
A lapkán található két CPU-mag szerkezetileg teljesen megegyezik, de a programozás során különböző szerepet szántak nekik. Egyikük számításokat végez, míg a másik a 65 536 feldolgozóegység közötti kommunikáció kezeléséért felelős. A lapkán három szinten helyeztek el gyorsítótárat. Az első szinten egy osztott L1 gyorsítótár van, 32 KB áll rendelkezésre az utasítások, és ugyanennyi az adatok számára. A két CPU L1 gyorsítótára között nincs koherencia, mert a szabványos PowerPC 440 magok ezt nem támogatják, és eldöntötték, hogy módosítás nélküli magokat használnak. A második szinten egy egységes 2 KB-os gyorsítótár van. Az L2 gyorsítótárak valójában nem is azok, hanem előolvasó pufferek. Ezek figyelik egymást és konzisztensek. A harmadik szinten egységes, konzisztens 4 MB méretű megosztott gyorsítótár van, amely ellátja mindkét L2 gyorsítótárat. 11 óraciklust igényel egy olyan memóriahivatkozás, amely nem talál az L1 gyorsítótárban, de az L2-ben igen. 28 ciklusba kerül, ha L2-ben nincs, de L3-ban van találat. Végül, ha az L3-ban nincs az adat, és az SDRAM-hoz kell fordulni, az körülbelül 75 ciklust igényel.

Az L2 gyorsítótárakhoz egy kis SRAM van kapcsolva. A JTAG-kivezetésekhez kapcsolódik, és a rendszerindításban, nyomkövetésben, a gazdagéppel való kommunikációban van szerepe, ezen kívül a rendszervermet tárolja, valamint szemafor, sorompó és más szinkronizációs műveletek megvalósítására használják.

A következő szinten IBM által tervezett kártyát találunk, amely a 8.36. ábrán látható lapkák közül kettőt fogad be 1 GB memória társaságában. A kártya jövőbeli változatai akár 4 GB-ot is tartalmazhatnak. A lapka és a kártya a 8.37. (a)–(b) ábrán láthatók.

A kártyákat alaplapokra helyezik, alaplaponként 16 kártyával összesen 32 lapka (és így 32 számításokat végző CPU) van. Mivel minden kártya 1 GB SDRAM-ot tartalmaz, az alaplapon 16 GB van összesen. Az alaplap a 8.37. (c) ábrán látható.

A következő szinten 16-16 ilyen alaplapot találunk egy szekrény alsó és felső részében is, ezzel rendkívül sűrűn 1024 CPU kerül egy 60 × 90 cm-es térrészbe. A két 16-os csoportot egy olyan kapcsoló választja el egymástól, amelyik le tudja választani az egyik csoportot a rendszerről karbantartás céljára, miközben bekapcsol egy tartalék csoportot. A szekrény a 8.37. (d) ábrán látható.



8.37. ábra. *A BlueGene/L (a) lapkája. (b) kártyája. (c) alaplapja. (d) szekrénye. (e) rendszere*

Végül a 64 kabinetből álló teljes rendszer a 8.37. (e) ábrán látható, 65 536 számítást végző CPU-t és 65 536 kommunikációs CPU-t tartalmaz. 131 072 darab, két egész utasítást kiadni képes CPU-val és 262 144 darab, két utasítást kiadni képes lebegőpontos CPU-val ennek a rendszernek elvileg akár 786 432 utasítást is tudnia kellene kiadni órajelenként. Az egyik egész CPU feladata azonban a lebegőpontos CPU ellátása, ezzel az utasítások kiadásának üteme órajelenként 655 360-ra csökken, ami $4{,}6 \times 10^{14}$ utasítás/másodpercet jelent, ezzel ez a gép messze a valaha épített legnagyobb számítógéprendszer.

A rendszer egy multiszámítógép abban az értelemben, hogy egyetlen CPU sem fér hozzá közvetlenül más memóriához, csak a saját kártyáján lévő 512 MB-hoz. Nincs két olyan CPU, amelyek közösen használnának memóriát. Ráadásul lapozásos memóriakezelés sincs, mert nincsenek ehhez szükséges lokális lemezegységek. Ehelyett a rendszerben van 1024 B/K csomópont, ezekhez vannak csatlakoztatva a lemezegységek és egyéb perifériák.

Mindent egybevetve, jóllehet a rendszer rendkívül nagy, egyúttal egészen egyszerű is, kevés új technológia van benne, kivéve a nagy sűrűségű csomagolást. Nem véletlenül döntöttek az egyszerűség mellett, hiszen az egyik nagy cél az volt, hogy nagy megbízhatóságú és hosszú rendelkezésre állási idejű legyen. Következésképpen nagyon körültekintően tervezték meg a tápellátást, a hűtőventillátorokat és vezetékelést azzal a céllal, hogy a meghibásodások közötti átlagos idő 10 nap legyen.

Az összes lapka összekapcsolásához egy skálázható, nagy teljesítményű kommunikációs topológia és technológia szükséges. A tervezők egy 64 × 32 × 32 méretű

tóruszt alkalmaztak. Emiatt minden CPU-nak hat csatlakozója van, kettő a logikailag alatta és felette lévő CPU-k felé, kettő északra és délre, illetve kettő keletre és nyugatra, ahogy a 3.36. ábrán is látható. Fizikailag az 1024 elemet tartalmazó szekrények mindegyike egy 8 × 8 × 16-os tórusz. Szomszédos szekrények 8 × 8 × 32-es tóruszt alkotnak. Ugyanabban a sorban lévő négy pár szekrény egy 8 × 32 × 32-es tóruszt alkot. Végül a 8 sor együtt alakítja ki a 64 × 32 × 32-es tóruszt.

Minden összeköttetés pont-pont kapcsolat, és 1,4 Gbps sebességgel üzemel. Mivel mind a 65 536 elemnek 3 összeköttetése van „nagyobb" sorszámú elemek felé (minden dimenzióban egy), ezért a rendszer összesített sávszélessége 275 terabit/másodpere. Ennek a könyvnek az információtartalma körülbelül 300 millió bit, beleértve a Postscript formátumú illusztrációkat is, így a BlueGene/L ebből a könyvből 900 000 példányt tudna megmozgatni másodpercenként. Hogy hova és kinek, annak eldöntését a kedves olvasóra bízzuk.

A háromdimenziós tóruszban a kommunikáció egy ún. **virtual cut through** útvonalválasztási módszerrel történik. Ez valamelyest hasonlít a tárold és továbbítsd csomagkapcsolás elvére, kivéve hogy továbbítás előtt nem tárolják az egész csomagot. Amint egy bájt megérkezett a csomópontba, azonnal továbbítható a következőhöz, még mielőtt az egész csomag megérkezett volna. Dinamikus (adaptív) és determinisztikus (statikus) útvonalválasztás is elképzelhető. Egy kevés speciális célú hardver valósítja meg a lapkán.

Az adatkommunikációra használt fő 3D tórusz mellett négy másik kommunikációs hálózat is van. A második egy fa alakú kombinációs hálózat. A BlueGene/L-hez hasonló erősen párhuzamos rendszerekben sok olyan műveletet kell végezni, amelyben az összes feldolgozóegységnek részt kell vennie. Például képzeljük el, hogy 65 536 szám közül kell kiválasztanunk a legkisebbet. Kezdetben minden elem egy számot tárol, és amikor ketten elküldik az értékeiket egy magasabb szintű elemnek, akkor az kiválasztja a kisebbiket, és azt továbbítja a felsőbb szintek felé. Ilyen módon sokkal kevesebb üzenet érkezik a gyökérelemhez, mintha mind a 65 536 elem közvetlenül oda küldte volna az üzenetét.

A harmadik hálózat globális sorompókhoz és megszakításokhoz tartozik. Némelyik algoritmus meneteket hajt végre, és minden elemnek meg kell várnia, hogy az összes többi is befejezze a menetet, mielőtt a következőt elkezdhetné. A sorompóhálózat lehetővé teszi a programnak, hogy definiálja a meneteket, és módot ad arra, hogy a menetek végén a számítást végző CPU-kat felfüggessze, amíg mindegyik el nem éri a menet végét, és akkor az összeset továbbengedi. A megszakítások is ezt a hálózatot használják.

A negyedik és az ötödik hálózat gigabit Ethernetet használ. Az egyik a B/K csomópontokat köti össze a fájlszerverekkel (ezek nem a BlueGene/L részei), illetve az internettel. A másik a rendszerhibák keresésében jut szerephez.

A számításokat végző és a kommunikációs elemek is egy erre a célra tervezett, kisméretű, kis erőforrás-igényű operációsrendszer-magot futtatnak, amely egyetlen felhasználót és egyetlen folyamatot támogat. Ennek a folyamatnak két szála van, egy-egy a csomópont mindkét CPU-jára. A nagy teljesítmény és a nagy megbízhatóság érdekében alakították ki ezt az egyszerű szerkezetet.

A megbízhatóság további növelése érdekében az alkalmazások meghívhatnak egy könyvtári eljárást, amellyel ellenőrzőpontot hozhatnak létre. Amint az összes úton lévő üzenet átjutott a hálózaton, egy globális ellenőrzőpontot lehet létrehozni, így rendszerhiba esetén a munka az ellenőrzőponttól folytatható, és nem kell elölről kezdeni. A B/K csomópontokon hagyományos Linux operációs rendszer fut, és egyszerre több folyamat is aktív lehet. A BlueGene/L-ről további információ található (Adiga és társai, 2002; Almasi és társai, 2003a; 2003b; Blumrich és társai, 2005).

Red Storm

Második MPP-példánk a Sandia National Laboratory Red Storm nevű gépe lesz (Thor's Hammer néven is emlegetik). A Sandiát a Lockheed Martin üzemelteti, titkos és kevésbé titkos munkákat is végeznek az Egyesült Államok Energiaügyi Minisztériuma számára. A titkos munkák egy része nukleáris fegyverek tervezésével és szimulációjával kapcsolatos, ami rendkívül számításigényes. A Sandia már régóta tevékenykedik ezen a területen, és sok fejlett szuperszámítógépük volt az évek során. Évtizedekig a vektoros szuperszámítógépeket részesítették előnyben, de végül technológiai és gazdasági okokból az MPP-k előnybe kerültek. 2002-re az akkor ASCI Rednek nevezett MPP egy kicsit kezdett csikorogni. Habár 9460 processzora, 1,2 TB RAM-ja és 12,5 TB lemezterülete volt, alig 3 teraflop/másodpercet tudott. Ezért 2002 nyarán a Sandia megbízására a szuperszámítógépek készítésében nagy múlttal rendelkező Cray Research kapott megbízást az ASCI Red utódjának megépítésére.

Az utód 2004 augusztusában készült el; figyelemre méltóan rövid idő alatt terveztek meg és készítettek el egy ilyen nagy gépet. A gyors tervezést és megépítést az tette lehetővé, hogy a Red Storm egy egyedi útvonalválasztó lapka kivételével csupa kereskedelmi forgalomban kapható alkatrészekből áll.

A Red Storm számára az AMD Opteron processzort választották ki. Az Opteronnak sok olyan jellemzője van, ami miatt a legalkalmasabb erre a célra. Először is három működési üzemmódja van. Legacy üzemmódban szabványos, módosítás nélküli Pentium bináris programokat futtat. Compatibility üzemmódban az operációs rendszer 64 bites módban fut, és 2^{64} bájtnyi memóriát tud megcímezni, de az alkalmazói programok 32 bites módban futnak. Végül 64 bites üzemmódban az egész gép 64 bites, és minden program használhatja a teljes 64 bites címtartományt. 64 bites üzemmódban lehet keverni a programokat: 32 bitesek és 64 bitesek is futhatnak egyszerre, megkönnyítve az áttérést.

Az Opteron második fontos jellemzője, hogy tervezésénél nagyon figyeltek a memória-sávszélesség problémájára. Az elmúlt években a CPU-k egyre gyorsabbak lettek, de a memóriák nem tartottak lépést velük, emiatt nagy árat kell fizetni, ha a második szintű gyorsítótárban sincs a keresett adat. Az AMD a memóriavezérlőt az Opteronba integrálta, így az a rendszersín sebessége helyett a processzorsín sebességével működhet, ez pedig megnöveli a teljesítményt. A vezérlő nyolc darab 4 GB-os DIMM kezelésére képes, összesen tehát egy Opteronnak 32 GB memóriája lehet. A Red Storm rendszerben az Opteronoknak csak 2–4 GB

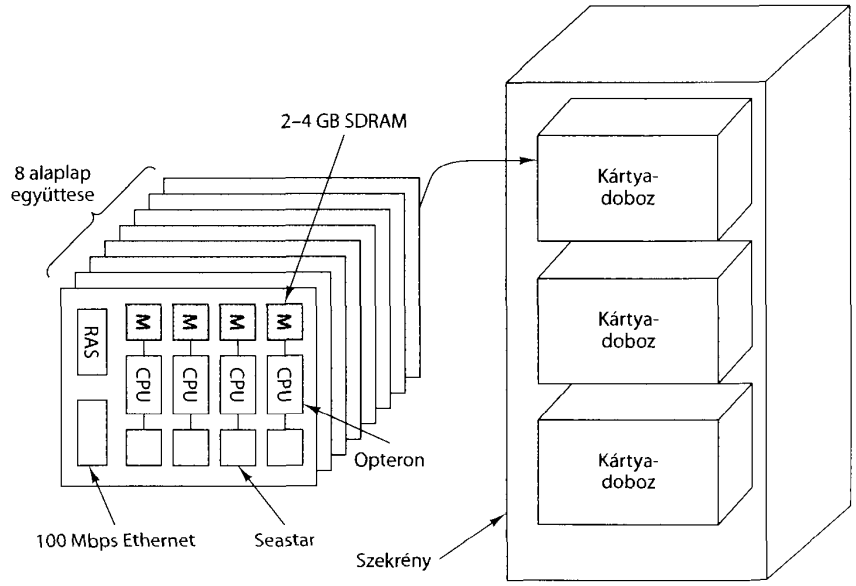
memóriája van. Ahogy azonban a memória egyre olcsóbb lesz, ezt biztosan növelni fogják a jövőben. Az Opteronokat kétmagosakra is le lehet cserélni, ezáltal a nyers számítási kapacitás megkétszereződne.

Minden Opteronnak egy saját **Seastar** nevű hálózati processzora van; ezeket az IBM gyártja. A Seastar kritikus komponens, mert szinte a teljes processzorok közötti adatforgalom a Seastar hálózaton halad keresztül. A rendszer pillanatok alatt telítődne adatokkal, ha ezeknek az egyedi lapkákból kialakított hálózatnak nem lenne elég nagy a sebessége.

Jóllehet Opteron kapható a boltokban is, a Red Stormhoz egyedi foglalattal készültek. Minden Red Storm alaplapon négy Opteron, 2-4 GB RAM, 4 Seastar, egy RAS (Reliability, Availability, Service) processzor és egy 100 Mbps Ethernet-lapka van, ahogy az a 8.38. ábrán is látható.

Az alaplapokat nyolcasával helyezik kártyadobozokba. Minden szekrényben három doboz van összesen 96 Opteronnal, és szükségképpen helyet kapnak a tápegységek és ventillátorok is. A teljes rendszer 108 darab számításokat végző processzorokat tartalmazó szekrényből áll, összesen 10368 Opteron 10 TB SDRAM-mal. Minden CPU csak a saját SDRAM-jához fér hozzá, nincs közös memória. A rendszer elméleti számítási kapacitása 41 teraflop/másodperc.

Az Opteron CPU-k közötti összeköttetést az egyedi tervezésű Seastar útvonalválasztók biztosítják, CPU-nként egy. Egy 27 × 16 × 24-es 3D tórusz minden csomópontjában el van helyezve egy Seastar. Mindegyiknek 7 darab kétirányú, 24 Gbps sebességű csatlakozója van, ezek északra, keletre, délre, nyugatra, fel-

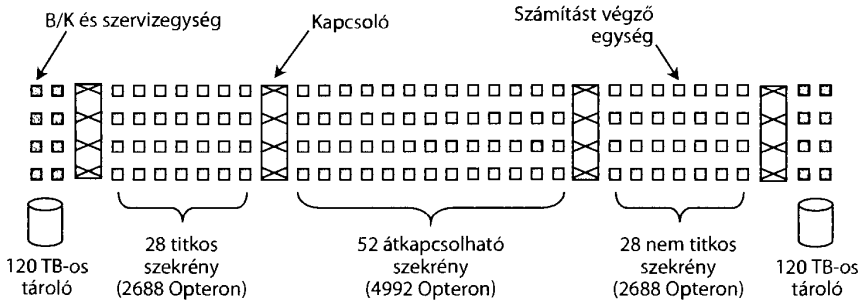


8.38. ábra. *A Red Storm komponenseinek elhelyezése*

felé, lefelé és az Opteronhoz vezetnek. Két szomszédos hálópont közötti átviteli idő 2 μs. A teljes hálózat egyik végéből a másikba csak 5 μs. Egy másik 100 Mbps Ethernet-hálózatot szervizelésre és karbantartásra használnak.

Az előbb említett 108 szekrény mellett van 16 darab, B/K és szervizprocesszorokat tartalmazó szekrény is. Minden ilyen szekrényben 32 Opteron található. Ennek az 512 CPU-nak fele B/K célokra használatos, másik fele pedig szervizprocesszor. A fennmaradó helyet RAID 3 és RAID 5 szervezésű lemezegységek foglalják el, mindegyik esetében egy paritás lemezegységgel, és egy üzem közben cserélhető tartalékkal. A teljes lemezkapacitás 240 TB. A tartható lemezsávszélesség 50 GB/másodperc.

A rendszer titkos és nem titkos részre osztott, a részegységeket mechanikus kapcsolókkal lehet összekapcsolni vagy szétválasztani. 2688 darab számítást végző CPU mindig a titkos részben van, másik ugyanilyen 2688 pedig mindig a nem titkos részben. A maradék 4992 CPU-t bármelyik részhez hozzá lehet kapcsolni, ahogy a 8.39. ábrán látható. A 2688 titkos Opteronnak 4 GB RAM-ja van, a többinek mind 2 GB-ja. A B/K és szervizprocesszorok fel vannak osztva a két rész között.



8.39. ábra. *A Red Storm rendszer felülnézeti képe*

Mindent egy új, 2000 m^2-es épületben helyeztek el. Az épületet és a környezetet úgy tervezték, hogy a rendszert szükség esetén 30 000 CPU-ra lehessen bővíteni a jövőben. A számítást végző processzorokat tartalmazó részrendszer 1,6 megawatt energiát fogyaszt; a lemezegységek még egy megawattot. Ventillátorokkal és légkondicionálókkal együtt 3,5 megawatt a teljes energiaszükséglet.

A hardver és a szoftver 90 millió dollárba került. Az épület és a hűtés további 9 millió dollárba, tehát a teljes összeg valamivel 100 millió dollár alatt maradt, habár ennek egy része egyszeri mérnöki munkára ment el. Ha szeretne valaki egy pontosan ugyanilyet, akkor úgy 60 millió dollárt kellene rászánnia. A Cray Research ennek kisebb verzióit kormányzati és piaci vásárlóknak szeretné értékesíteni X3T név alatt.

A számításokat végző csomópontokban egy **catamount** nevű könnyű súlyú operációsrendszer-mag fut. A B/K és a szervizcsomópontokban sima mezei Linux fut egy kis MPI-támogatással (az MPI-ról később még szó lesz). A RAS processzorokon egyszerűsített Linux fut. Az ASCI Redről sok minden rendelkezésre áll a Red Stormhoz, CPU allokátorok, ütemezők, MPI könyvtárak, matematikai könyvtárak és az alkalmazói programok.

Egy ilyen nagy rendszerben a nagy megbízhatóság alapvető. Minden alaplapon van egy RAS processzor karbantartási célokból, illetve más speciális hardvermegoldásokat is alkalmaztak. A cél 50 óra MTBF (Mean Time Between Failures, meghibásodások között eltelt átlagos idő) volt. Az ASCI Red hardverének 900 órás MTBF értékét kellemetlenül lerontották az operációs rendszer 40 óránként bekövetkező leállásai. Az új hardver sokkal megbízhatóbb ugyan, mint a régi, a gyenge pont azonban a szoftver maradt.

A Red Stormról további információ található (Brightwell és társai, 2005).

A BlueGene/L és a RedStorm összehasonlítása

A Red Storm és a BlueGene/L bizonyos szempontokból hasonló, más szempontokból pedig eléggé különböznek, de mindenesetre érdekes egymás mellé tenni néhány kulcsfontosságú paramétert (lásd 8.40. ábra).

Mindkét gép nagyjából ugyanabban az időszakban épült meg, tehát különbségeik nem technológiai eredetűek, hanem tervezőik hozzáállásában, illetve bizonyos mértékig az előállító cégek, az IBM és a Cray közötti különbségben keresendők. A BlueGene/L-t a kezdetektől fogva kereskedelmi céllal tervezték, az IBM sokat szeretne eladni belőle biotechnológiai, gyógyszeripari és más cégeknek.

Elem	BlueGene/L	Red Storm
CPU	32 bites PowerPC	64 bites Opteron
Órajel	700 MHz	2 GHz
Kiszámítást végző CPU-k száma	65 536	10 368
CPU-k száma alaplaponként	32	4
CPU-k száma szekrényenként	1024	96
Számítást végző szekrények száma	64	108
Teraflop/másodperc	71	41
Memória CPU-nként	512 MB	2–4 GB
Teljes memória	32 TB	10 TB
Útvonalválasztó	PowerPC	Seastar
Útvonalválasztók száma	65 536	10 368
Hálózati topológia	3D tórusz 64 × 32 × 32	3D tórusz 27 × 16 × 24
Egyéb hálózatok	Gigabit Ethernet	Fast Ethernet
Particionálható	Nem	Igen
Számítások operációs rendszere	Egyedi	Egyedi
B/K operációs rendszere	Linux	Linux
Gyártó	IBM	Cray Research
Drága?	Igen	Igen

8.40. ábra. *A BlueGene/L és a Red Storm összehasonlítása*

A Red Storm a Sandiával kötött speciális szerződés keretében készült, habár a Cray is szeretne kereskedni kisebb verzióival.

Az IBM elképzelése egyértelmű: meglévő magokból hozzunk létre egy egyedi lapkát, amely nagy mennyiségben olcsón előállítható, üzemeltessük alacsony sebességen, és kapcsoljunk össze nagyon sokat egy közepes sebességű kommunikációs hálózattal. A Sandia elképzelése ugyanilyen egyértelmű, de más: vegyünk egy boltban kapható erős 64 bites CPU-t, tervezzünk hozzá egy nagyon gyors útvonalválasztó lapkát, és adjunk hozzá jó sok memóriát, így a BlueGene/L-énél sokkal erősebb építőelemet kapunk, kevesebbre van szükség, és a kommunikáció gyorsabb közöttük.

Ezeknek a döntéseknek következményei lettek a csomagolásra nézve. Az IBM egyedi lapkát épített processzorral és útvonalválasztóval, így nagyobb sűrűséget ért el: 1024 CPU/szekrény. A Sandia készen kapható CPU-kat használt fel 2–4 GB RAM-mal társítva, így csak 96 CPU-t tudott elhelyezni egy szekrényben. Az eredmény az, hogy a Red Storm több helyet foglal, és többet fogyaszt, mint a BlueGene/L.

A nemzeti laboratóriumok számítóközpontjainak egzotikus világában a végső cél a teljesítmény. Ebben a vonatkozásban a BlueGene/L a győztes, 71 TF/másodperc a 41 TF/másodperccel szemben, de a Red Stormot bővíthetőre tervezték, tehát még 10 368 Opteron bevetése árán (például kétmagos lapkákra való átállással) a Sandia valószínűleg fel tudja tornászni magát 82 TF/másodpercre. Az IBM válasza az órajel kis megnövelése lehet (700 MHz nem nagyon feszegeti napjaink technológiai korlátait). Röviden, az MPP szuperszámítógépek nem értek még fizikai korlátaik közelébe, és a következő években egyre növekedni fognak.

8.4.3. Klaszterszámítógépek

A multiszámítógépek másik fajtája a **klaszterszámítógép** (Anderson és társai, 1995; Martin és társai, 1997). Rendszerint néhány száz vagy ezer személyi számítógépből (PC) vagy munkaállomásból áll, amelyek a kereskedelemben elérhető hálózati kártyákkal csatlakoznak egymáshoz. Egy MPP és egy COW (Cluster of Workstations, munkaállomások klasztere) közötti különbség hasonló egy nagyszámítógép és egy személyi számítógép közötti különbséghez. Mindkettőnek van CPU-ja, RAM memóriája, lemezeik, operációs rendszerük és így tovább. A nagyszámítógépben minden sokkal gyorsabb (kivéve talán az operációs rendszert). Mégis minőségileg mások, és másképpen kell használni és üzemeltetni. Ugyanez a különbözőség megvan az MPP-k és a COW-k között.

A kulcselem, amely az MPP-ket egyedivé tette, a nagy sebességű összekötő hálózat volt, azonban ma már a kereskedelemben is kaphatók nagy sebességű összekötő hálózatok, emiatt a különbség egyre csökken. Mindent egybevetve a klaszterek valószínűleg egyre szűkebb területre fogják visszaszorítani az MPP-ket, ahogy a PC-k a nagyszámítógépeket különleges ritkaságokká tették. MPP-ket főleg nagy költségvetésű szuperszámítógépek között találunk, ahol a csúcsteljesítmény a legfontosabb. Ha az árat meg kell kérdeznünk, akkor nem engedhetjük meg magunknak.

Sokfajta COW létezik, azonban két csoport meghatározó: a központosított és a széttagolt. Egy központosított COW munkaállomásoknak vagy PC-knek egy olyan csoportja, amelyet egyetlen helyiségben szerelnek össze. Néha a szokásosnál sokkal sűrűbben helyezik el ezeket, hogy a fizikai méret és a kábelek hossza csökkenjen. Általában a gépek homogének, a hálózati kártyán és esetleg a lemezeken kívül nincs más perifériájuk. Gordon Bell, a PDP–11 és a VAX tervezője ezeket a gépeket **fej nélküli munkaállomás**oknak nevezte (mivel a gépeknek nincs gazdájuk). Megtehetnénk, hogy fej nélküli teheneknek (COW, tehén) hívnánk ezeket, de félő, hogy egy ilyen elnevezés túl sok szent tehenet sértene, ezért inkább tartózkodunk tőle.

A széttagolt klaszterek olyan munkaállomásokból vagy PC-kből állnak, amelyek egy épület vagy egy egész intézmény területén találhatók. Ezek legtöbbje naponta több órán keresztül tétlen, elsősorban éjszakánként. Gyakran egy LAN-ban vannak összekapcsolva. Jellemző, hogy a gépek heterogének, és a perifériák teljes sorával rendelkeznek, bár egy 1024 egérrel rendelkező COW nem sokkal jobb, mint egy egér nélküli COW. A legfontosabb, hogy a gépek legtöbbjének van gazdája, akiket érzelmi szálak fűznek a gépükhöz, és rossz néven veszik, ha azokon mindenféle csillagászok az ősrobbanást szimulálják. A tétlen munkaállomások felhasználásával olyan COW alakítható csak ki, amelyben lehetőség van a feladatok elvándoroltatására, amikor a tulajdonosa visszaköveteli a gépét. A feladat vándoroltatás lehetséges, de a szoftvert tovább bonyolítja.

A klaszterek gyakran kicsik, tucatnyitól esetleg 500 PC-ig terjed a méretük. Nagyon nagyokat is lehet azonban építeni bolti PC-kből. A Google meg is tette; nézzük meg, hogyan.

Google

A Google egy népszerű információkereső rendszer az interneten. Népszerűségét részben egyszerűségének és rövid válaszidejének köszönheti, de szerkezete éppenséggel nem egyszerű. A Google nézőpontjából a probléma az, hogy keresni, indexelni és tárolni kell az egész World Wide Webet (több mint 8 milliárd lap és 1 milliárd kép), át kell fésülni az egészet fél másodperc alatt, és kezelni kell a világ minden tájáról másodpercenként érkező több ezer kérést a nap 24 órájában. Ráadásul soha nem állhat le, még földrengés, áramszünet, vonalszakadás, hardver- vagy szoftverhiba miatt sem. És természetesen mindezt olyan olcsón kell elérnie, amennyire csak lehetséges. Egy Google-klón megépítése határozottan nem az a feladat, amit a könyvekben adnak az olvasóknak.

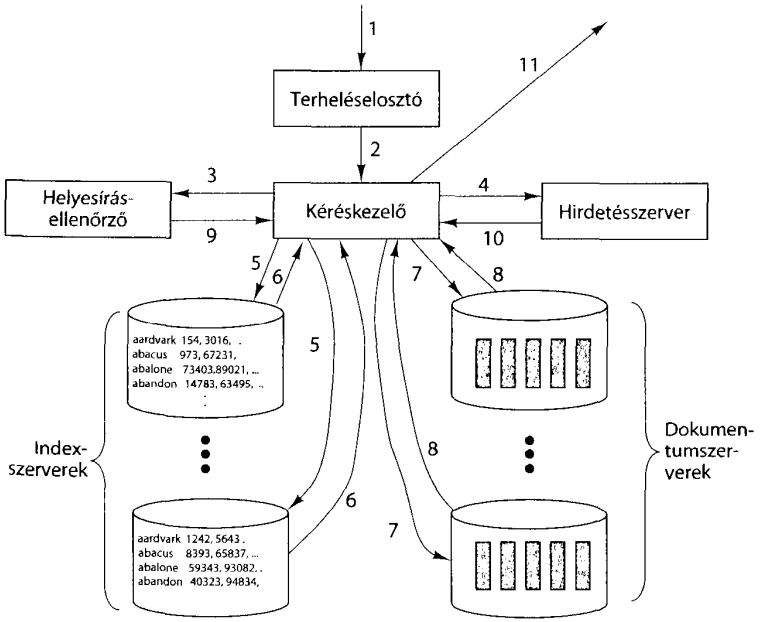
Hogy csinálja a Google? Kezdetnek például sok adatközpontot üzemeltet a világ különböző pontjain. Ez nemcsak azért jó, mert ha földrengés esetén az egyik megsemmisül, akkor van másolat, hanem azért is, mert amikor a *www.google.com* kell valakinek, akkor az illetőt IP címe alapján a legközelebbi adatközponthoz irányítják. A böngésző aztán azzal kommunikál.

Minden adatközpontnak legalább egy OC-48-as (2,488 Gbps) üvegszálas kapcsolata van az internettel, ezen fogadja a kéréseket, és küldi a válaszokat.

Mindegyiknek van még egy OC-12-es (622 Mbps) másodlagos kapcsolata is egy másik szolgáltatón keresztül arra az esetre, ha az elsődlegessel valami történne. Szünetmentes áramforrások és vésztartaléknak dízelmotoros generátorok vannak minden adatközpontban, hogy áramszünet esetén is folytatódhasson a műsor. Egy nagyobb természeti katasztrófa esetén a teljesítmény csökken ugyan, de a Google működőképes marad.

Hogy jobban megérthessük, miért választotta a Google éppen azt az architektúrát, amit választott, hasznos áttekintenünk egy kérés feldolgozásának menetét onnan kezdve, hogy beérkezik egy adatközpontba. Az adatközpontba való beérkezés után (lásd 1. lépés a 8.41. ábrán) a terheléselosztó a kezelők (2) egyikéhez irányítja a kérést, illetve egyidejűleg a helyesírás-ellenőrzőhöz (3) és a reklámszerverhez (4) is. Ezután a keresőszavakat párhuzamosan megkeresi az indexszervereken (5). Ezek a szerverek a weben található minden szóhoz tartalmaznak egy bejegyzést. Minden bejegyzés mögött rangszám szerinti sorrendben listára van fűzve az összes olyan dokumentum (weboldal, PDF fájl, PowerPoint prezentáció stb.), amelyben a szó megtalálható. A rangszámot egy bonyolult (és titkos) képlettel számítják ki, de az egy lapra hivatkozó oldalak száma és ezek rangszáma nagy szerepet játszik.

A teljesítmény növelése érdekében az indexek párhuzamosan kereshető **szeletekre** vannak osztva. Elvileg az 1. rész az index minden szavát tartalmazza, mindegyik után az n legnagyobb rangszámú olyan dokumentum azonosítójával, amely tartalmazza a szót. A 2. szelet is tartalmazza az összes szót, és mindegyikhez az n



8.41. ábra. *Egy Google-kérés feldolgozása*

következő legnagyobb rangszámú dokumentumazonosítót, és így tovább. Ahogy a web növekszik, a szeleteket fel lehet bontani, az első k szó kerülne az első szelethalmazba, a második k szó a második szelethalmazba és így tovább, hogy még több párhuzamosság lehessen a keresésekben.

Az indexszerverek egy dokumentumazonosító halmazt adnak vissza (6), amelyeket aztán a kérés Boolean-műveleteinek megfelelően kombinálnak össze. Például, ha a +digital +capybara +dance volt a keresendő kifejezés, akkor a következő lépésben csak azok a dokumentumazonosítók vesznek részt, amelyek mind a három halmazban benne voltak. A következő lépésben (7) beleolvasnak a dokumentumokba, hogy a cím, URL és a keresendő kifejezést tartalmazó szövegkörnyezet rendelkezésre álljon. A dokumentumszerverek a web több példányát is tárolják minden adatközpontban, ez jelenleg több száz terabájtnyi adat. A dokumentumok szintén szeletekre vannak osztva, hogy a párhuzamos keresés lehetőségét megteremtsék. Jóllehet egy kérés feldolgozása során nem kell az egész webet végignézni (még csak a több tíz terabájtot sem az indexszervereken), de kérésenként 100 MB feldolgozása teljesen szokványos.

Amikor az eredmények a kezelőhöz visszaérkeztek, a lapokat rangszám szerint sorba rendezik. Ha elírást lehet gyanítani (9), akkor beillesztenek egy figyelmeztetést, majd elhelyeznek még egy-két hirdetést (10). A Google azzal keres pénzt, hogy meghatározott keresési kifejezéseket (például „hotel" vagy „camcorder") elad az az iránt érdeklődő hirdetőknek, majd ezek hirdetéseit megjeleníti. Végül az eredményeket HTML (HyperText Markup Language) formátumra hozzák, és weblapként a felhasználóhoz küldik.

Ezzel a háttérrel most megvizsgálhatjuk a Google-architektúrát. A legtöbb cég, amikor azzal a szembesül, hogy egy hatalmas adatbázison megbízhatóan kell egységnyi idő alatt rengeteg tranzakciót végrehajtani, akkor általában megveszi a legnagyobb, leggyorsabb és legmegbízhatóbb berendezést, amit csak be lehet szerezni. A Google éppen az ellenkezőjét tette. Olcsó, közepes teljesítményű PC-ket vett. Nagyon sokat. Ezekből aztán megépítették a világ legnagyobb készen kapható alkatrészekből álló klaszterét. E döntés mögött egy egyszerű alapelv húzódik meg: optimalizáljuk az ár/teljesítmény arányt.

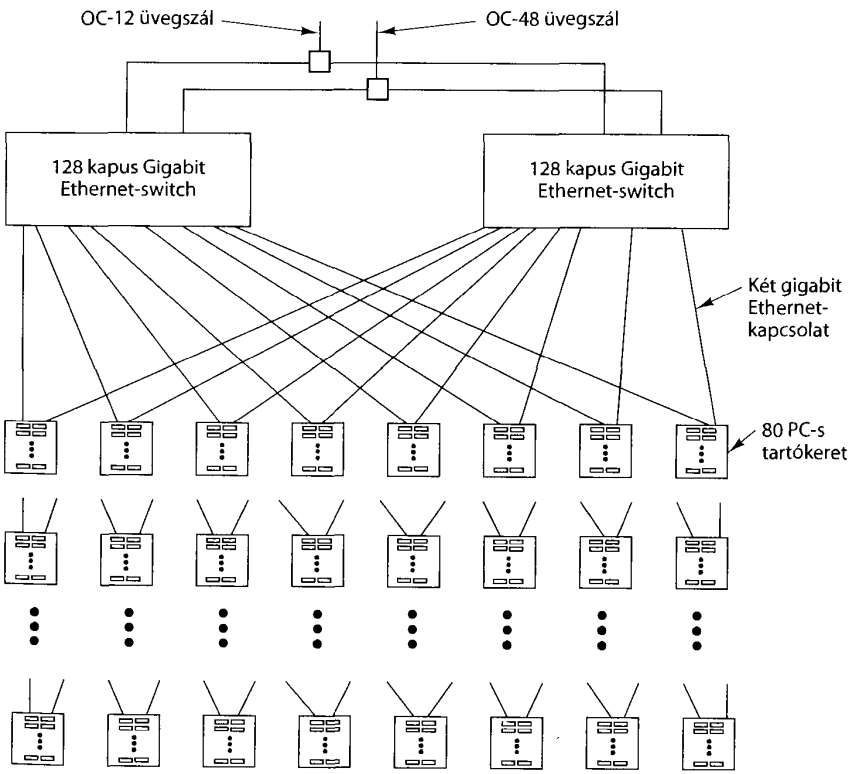
A döntés mögötti logika gazdasági jellegű: a PC-k nagyon olcsók. A nagy teljesítményű szerverek nem azok, a nagy multiprocesszorok pedig még kevésbé. Így míg az erős szerverek teljesítménye ugyan kétszer vagy háromszor nagyobb, mint a közepes asztali PC-ké, de közben 5-10-szer többe kerülnek, ami nem költséghatékony.

Természetesen az olcsó PC-k gyakrabban hibásodnak meg, mint a minőségi szerverek, de az utóbbiak is meghibásodnak néha, ezért a Google-szoftvert mindenképpen fel kell készíteni a hardve hibákra, akármilyen berendezéseket is használnának. Ha a hibatűrő szoftver már elkészült, akkor nem igazán számít, hogy a hibaarány évenként 0,5% vagy 2%, mindenképpen kezelni kell. A Google tapasztalata szerint a PC-k 2%-a hibásodik meg minden évben. A hibák több mint a fele hibás lemezegységek miatt következik be, ezután a tápegységek és a RAM-ok következnek. Bejáratódás után a processzorok már sohasem hibásodnak meg. Valójában a lefagyások leggyakoribb forrása egyáltalán nem a hardver, hanem a

szoftver. A lefagyásra adott leggyakoribb reakció az újratöltés, ami gyakran meg is oldja a problémát (ez az elektronikus megfelelője annak, amikor az orvos azt mondja: „Vegyen be két aszpirint és feküdjön le!").

Egy tipikus modern Google PC-ben egy 2 MHz-es Pentium, 512 MB RAM és körülbelül 80 GB-os lemezegység van, ez nagyjából olyan gép, amit a nagymamák vesznek maguknak emailezéshez. Az egyedüli speciális tétel egy Ethernet-lapka. Nem éppen a legmodernebb, de nagyon olcsó. A PC-k szabványos (kb. 5 cm) magasságú házban vannak, negyvenesével egymásra pakolva 19 inch-es tartókeretekben, két oszlopban egymás mögött, tehát összesen 80 PC van egy tartókeretben. A 80 PC össze van kötve kapcsolt Ethernettel, maga a switch is a tartókeretben van. Az adatközpont egyes tartókeretei szintén össze vannak kötve kapcsolt Ethernettel, sőt, központonként két tartalék switch is van, hogy a switch-meghibásodások is túlélhetők legyenek.

Egy tipikus Google-adatközpont elrendezése látható a 8.42. ábrán. A bejövő nagy sávszélességű OC-48-as optikai kábel hozzá van kötve a két 128 kapus Ethernet-switch-hez. Hasonlóan a tartalék OC-12-es optikai kábel is mindkettőhöz el van vezetve. A bejövő optikai szálak speciális bemeneti kártyákat használnak, nem a 128 Ethernet-kapu egyikéhez csatlakoznak. Minden tartókeretből 4 Ethernet-csatlakozó jön ki, kettő a bal oldali switch-hez, kettő pedig a jobb oldali switch-hez. Ebben a konfigurációban a rendszer bármelyik switch meghibásodását ki tudja heverni. Mivel minden tartókeretnek négy csatlakozása van a switch-ek felé (kettő az elülső 40 PC-től, kettő pedig a hátsóktól), ezért négy csatlakozóhiba, illetve két csatlakozóhiba és egy switch-hiba kell ahhoz, hogy egy tartókeret leszakadjon a hálózatról. Két 128 kapus switch-csel, és tartókeretenként négy csatlakozóval maximálisan 64 tartókeretet lehet kiszolgálni. Tartókeretenként 80 PC-vel számolva 5120 PC fér el egy adatközpontban. Természetesen egy tartókeretben nem kell pontosan 80 PC-nek lennie, és a switch-eknek is lehet több vagy kevesebb kapujuk; ezek csak tipikus értékek egy Google-klaszterben.



8.42. ábra. *Egy tipikus Google-klaszter*

Az energiasűrűség is kulcsfontosságú. Egy tipikus PC nagyjából 120 wattot fogyaszt, ez körülbelül 10 kW tartókeretenként. Egy tartókeretnek nagyjából 3 m^2 hely kell, hogy a karbantartó személyzet PC-t tudjon cserélni, illetve hogy a légkondicionálás hatásos legyen. Ezekkel a paraméterekkel az energiasűrűségre 3000 watt/m^2 feletti érték jön ki. A legtöbb adatközpontot 600–1200 watt/m^2-re tervezik, így speciális intézkedésekre van szükség a hűtéshez.

A Google három alapvető dolgot tanult meg a komoly webszerverek üzemeltetése során, ezeket nem árt megismételni:

1. Alkatrészek meg fognak hibásodni, ezért erre fel kell készülni.
2. Mindent meg kell többszörözni az átbocsátóképesség és a rendelkezésre állás érdekében.
3. Optimalizáljuk az ár/teljesítmény arányt.

Az első pont azt mondja, hogy a hibatűrő szoftverre van szükség. Még a legjobb berendezések esetén is, ha nagyon sok van belőlük, lesz olyan, amelyik meghibásodik, és a szoftvernek ezt kezelnie kell. Akár egy hiba van hetente, akár kettő, ekkora rendszerben a szoftvernek kezelnie kell a hibákat.

A második pont arra hívja fel a figyelmet, hogy a hardvernek és a szoftvernek is többszörösen redundánsnak kell lennie. Ha ez teljesül, akkor nem csak a hibatűrés lesz nagyobb, de az átbocsátóképesség is nő. A Google esetében a PC-k, a lemezegységek, a vezetékek és a switch-ek mind többszörözve vannak. Továbbá az indexek és a dokumentumok szeletekre vannak osztva, a szeletek pedig minden adatközpontban többszörözve vannak, sőt még az adatközpontok is többszörözve vannak.

A harmadik pont az első kettő következménye. Ha a rendszer hibatűrőre van tervezve, akkor hiba lenne drága eszközöket, például RAID-szervezésű SCSI lemezegységet venni. Ezek is meg fognak hibásodni, így rossz ötlet tízszer annyit költeni csak azért, hogy fele annyi hiba forduljon elő. Sokkal jobb tízszer annyi hardvert venni, és kezelni a hibákat, amikor fellépnek. Ha más előnye nincs is, több hardverrel a teljesítmény nagyobb lesz, amikor minden rendben működik.

A Google-ről további információk találhatók (Barosso és társai, 2003; Ghemawat és társai, 2003).

8.4.4. A multiszámítógépek kommunikációs szoftvere

Egy multiszámítógép programozásához speciális szoftverre, rendszerint eljáráskönyvtárakra van szükség a belső folyamatok közötti kommunikáció és a szinkronizáció kezelésére. Ebben a részben röviden ezeket a szoftvereket tárgyaljuk. Többségében ugyanaz a szoftvercsomag az MPP-ken és a klasztereken is fut, így az alkalmazások a platformok között könnyen mozgathatók.

Az üzenetátadásos rendszerekben két vagy több folyamat egymástól függetlenül fut. Például, az egyik folyamat olyan adatokat állíthat elő, amelyet egy vagy több másik folyamat bemeneti adatként használ fel. Semmi garancia nincs arra, hogy amikor a küldő már elkészült egy adattal, akkor a fogadó(k) készen áll(nak) annak a fogadására, mivel mindegyik a saját programját futtatja.

A legtöbb üzenetátadásos rendszer két primitívvel (rendszerint könyvtári hívások) rendelkezik, ezek a send és a receive, de ezeknek számos különböző megvalósítása létezik. A három fő változat a következő:

1. Szinkron üzenetátadás.
2. Pufferelt üzenetátadás.
3. Nem blokkoló üzenetátadás.

A **szinkron üzenetátadás**nál, ha a küldő végrehajt egy send utasítást, és a fogadó addigra még nem hajtott végre egy receive utasítást, akkor a küldő blokkolódik (felfüggesztődik), amíg a fogadó egy receive utasításhoz ér, majd megtörténik az üzenetátadás. Amikor a küldő a hívást követően visszakapja a vezérlést, biztos lehet benne, hogy az üzenet elküldésre került, és a fogadóhoz hibátlanul megérkezett. Ez a módszer a legegyszerűbb, és nem igényel semmiféle pufferezést. Komoly hátránya viszont, hogy a küldő mindaddig blokkolt helyzetben van, amíg a fogadó megkapja és nyugtázza az üzenetet.

A **puffereit üzenetátadás**nál, ha egy üzenet elküldése azt megelőzően történik, hogy a fogadó készen állna, az üzenet valahol tárolódik, például egy postaládában, amíg a fogadó ki nem veszi onnan. Így a pufferelt üzenetátadásnál a küldő a send végrehajtása után folytathatja működését, még akkor is, ha a fogadó valami mással van elfoglalva. Mivel az üzenet ténylegesen elküldésre kerül, ezért a küldőnek lehetősége van arra, hogy az üzenetpuffert azonnal újra felhasználja. Ez a megvalósítás a küldő várakozási idejét csökkenti. Tulajdonképpen amint a rendszer elküldi az üzenetet, a küldő folytathatja a működését. Azonban most a küldő nem lehet biztos abban, hogy az üzenet hibátlanul megérkezett. Még ha a kommunikáció megbízható is, előfordulhat, hogy a fogadó összeomlik, mielőtt megkapja az üzenetet.

A nem blokkoló üzenetküldésnél a küldő a hívás után azonnal folytathatja működését. A könyvtári eljárás csak annyit tesz, hogy az operációs rendszert megkéri, hogy amikor ideje van, csinálja meg a hívást. Ennek következtében a küldő alig blokkolódik. Ennek a módszernek az a hátránya, hogy amikor a küldő folytatja a működését a send után, lehetséges, hogy az üzenetpuffer még nem használható, mert az üzenet még nincs elküldve. Valahogyan meg kell tudnia a küldőnek, hogy

mikor használhatja újra a puffert. Az egyik lehetőség, hogy bizonyos időközönként megkérdezi a rendszert. A másik lehetőség, hogy egy megszakítást kap, amikor a puffer elérhető. Ezek egyike sem teszi egyszerűbbé a szoftvert.

A következő részben egy népszerű, sok multiszámítógépen alkalmazott üzenetátadási rendszert vizsgálunk meg röviden, az MPI-t.

MPI – Üzenetátadás interfész

Hosszú évekig a multiszámítógépek legnépszerűbb kommunikációs csomagja a **PVM (Parallel Virtual Machine, virtuális párhuzamos számítógép)** volt (Geist és társai, 1994; Sunderram, 1990). Az utóbbi években azonban egyre inkább felváltja az **MPI (Message-Passing Interface, üzenetátadás interfész)**. Az MPI a PVM-nél gazdagabb és sokkal összetettebb, sokkal több könyvtári függvénnyel, opciókkal és hívásonkénti paraméterekkel rendelkezik. A manapság MPI–1-nek hívott eredeti MPI-változatot fejlesztették tovább 1997-ben, aminek eredményét MPI–2-nek nevezték el. A következőben vázlatosan bemutatjuk az MPI–1-et (amiben minden alapvető benne van), majd röviden megnézzük, hogy mennyiben több az MPI–2. Az MPI-1-re vonatkozó további információt lásd (Gropp és társai, 1994; Snir és társai, 1996).

Az MPI–1 a PVM-mel ellentétben nem foglalkozik a folyamatok létrehozásával és kezelésével. A felhasználó dolga, hogy helyi rendszerhívások segítségével a folyamatokat létrehozza. Létrejöttük után statikus, később nem változó folyamatcsoportokba szerveződnek. Az MPI ezekkel a csoportokkal dolgozik.

Az MPI négy meghatározó jellegű fogalomra épül: kommunikátorok, üzenetek adattípusai, kommunikációs műveletek és virtuális topológiák. Egy **kommunikátor** egy folyamatcsoportból és egy kontextusból áll. A kontextus valaminek az azonosítására szolgáló címke, például ilyen lehet egy végrehajtási fázis. Az üzenetküldésnél és -fogadásnál a kontextus alkalmazásával érik el, hogy az össze nem tartozó üzenetek ne zavarják egymást.

Az üzenetek típusoltak, sok támogatott adattípus van, beleértve a karaktereket, a rövid, normál és hosszú egészeket, a szimpla és dupla pontosságú lebegőpontos számokat és így tovább. Ezekből származtatva más típusok is létrehozhatók.

Az MPI a kommunikációs műveletek széles körét támogatja. A legalapvetőbb művelet az üzenetküldés, ez a következőképpen történhet:

```
MPI_Send(puffer, darab, adattípus, címzett, címke, kommunikátor)
```

Ez a hívás elküldi a pufferből a megadott darabszámú és adattípusú elemet a címzetthez. A címke mező szerint megcímkézi az üzenetet, így a fogadónak lehetősége van arra, hogy csak bizonyos címkéjű üzeneteket fogadjon. Az utolsó mező azt adja meg, hogy a címzett melyik folyamatcsoportban van (a címzett mező tartalma egy index, amely a folyamatcsoporthoz tartozó listában a folyamatot jelöli). Üzenet fogadásához a megfelelő hívás a következő:

MPI_Recv(&puffer, darab, adattípus, forrás, címke, kommunikátor, &állapot)

Ez azt jelenti, hogy a fogadó a forrás mezőben szereplő folyamattól a megadott adattípusú elemekből álló, megadott címkéjű üzenetre vár.

Az MPI négy alapvető kommunikációs módot támogat. Az 1-es mód a szinkron, amelynél a küldő nem indíthatja a küldést, amíg a fogadó egy MPI_Recv hívást végre nem hajt. A 2-es mód a pufferelt, amelynél nincs ez a korlátozás. A 3-as mód a szabványos, ez függ a megvalósítástól, és lehet akár szinkron, akár pufferelt. A 4-es mód a készenléti, amelynél a küldő tudja, hogy a fogadó már várja az üzenetet (mint a szinkronnál), de nem végez semmilyen ellenőrzést. Ezen primitívek mindegyikének van blokkoló és nem blokkoló változata, vagyis ez összesen nyolc primitívet eredményez. A fogadásnak csak két változata van: blokkoló és nem blokkoló.

Az MPI támogatja a kollektív kommunikációt, beleértve az üzenetszórást, a széttagoló/gyűjtő, a teljes cserélő, a felhalmozó és a sorompó módszereket. Minden kollektív kommunikációs formánál egy csoport összes folyamatának meg kell hívnia a megfelelő függvényt, mégpedig kompatibilis argumentumokkal. Ha nem így történik, hiba lép fel. A kollektív kommunikáció gyakori formája a fastruktúrába rendezett folyamatoknál fordul elő, amikor az értékek a levelektől a gyökér felé haladva adódnak át úgy, hogy minden lépésnél bizonyos feldolgozáson mennek át, például összeadás vagy maximum képzése.

Az MPI-ben a negyedik alapfogalom a **virtuális topológia**, amelyben a folyamatok egy fába, gyűrűbe, rácsba, tóruszba vagy más topológiába szerveződhetnek. Az ilyen elrendezések módot adnak arra, hogy elnevezzünk kommunikációs útvonalakat, és ez megkönnyíti a kommunikációt.

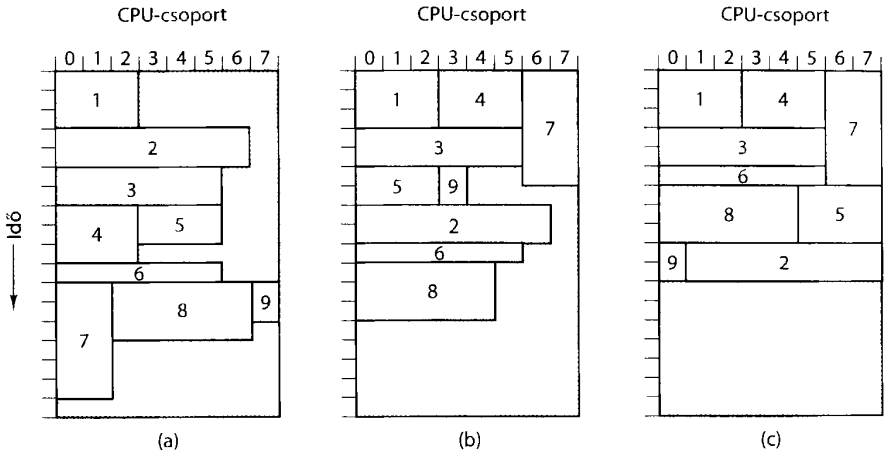
Az MPI–2 bővítései a dinamikus folyamatok, a távoli memória elérése, a nem blokkoló kollektív kommunikáció, a skálázható B/K támogatás, a valós idejű feldolgozás és sok egyéb újdonság, amelyek meghaladják ennek a könyvnek a kereteit. Tudományos körökben az MPI és a PVM pártolói között éveken keresztül csata dúlt. A PVM szószólói azt mondták, hogy a PVM-et könnyebb megtanulni és egyszerűbb használni. Az MPI pártolói azzal érveltek, hogy az MPI többet tud, és hogy az MPI szabványosító testülettel és hivatalos dokumentációval rendelkező szabvány. A PVM-oldal ezzel egyetértett, de azt mondták, hogy a szabványosítási bürokrácia hiánya nem feltétlenül hátrány. Miután mindez lezajlott, úgy tűnik, az MPI került ki győztesen.

8.4.5. Ütemezés

Az MPI-programozók könnyedén létrehozhatnak olyan programokat, amelyeknek sok CPU-ra van szükségük, és hosszú ideig futnak. Amikor több felhasználó egymástól függetlenül különböző hosszúságú időre kér CPU-kat, akkor a klaszterben egy munkaütemezőre van szükség, amely meghatározza, hogy mikor mi fusson.

A legegyszerűbb modellben a munkaütemező megköveteli, hogy mindegyik program tudassa vele a számára szükséges processzorok számát. A programok ezt

követően szigorú elsőként be, elsőként ki sorrendben futnak, ahogyan a 8.43. (a) ábrán látható. Ebben a modellben egy program elindulása után az ütemező megvizsgálja, hogy van-e elegendő számú CPU a bemeneti sorban levő következő feladat elindításához. Ha igen, elindul a feladat és így tovább. Különben a rendszer addig vár, amíg a megfelelő számú CPU elérhetővé válik. Mellékesen szólva, bár a 8.43. ábra azt sugallja, hogy a klaszterben nyolc CPU van, de ez lehet 128 CPU is, amelyek kiosztása 16-os egységekben (nyolc CPU alkot egy csoportot) történik, de más felosztás is lehetséges.



8.43. ábra. *Klaszterütemező. (a) Elsőként be, elsőként ki (FIFO) sorrend. (b) Sor eleji blokkolás nélkül. (c) Lefedő ütemezés. A sötétebb tónusú területek a tétlen CPU-kat jelölik*

Egy jobb ütemező algoritmus elkerüli a sor eleji blokkolást, átugorja azokat a programokat, amelyek az adott pillanatban nem elégíthetők ki, és az első kielégíthetőt választja. Amikor egy program befejeződik, a megmaradt programok sorát mindig elsőként be, elsőként ki sorrendben vizsgálja. Ez az algoritmus a példánkra a 8.43. (b) ábrán látható eredményt adja.

Egy még az eddigieknél is kifinomultabb ütemező algoritmus megköveteli minden jelentkező programtól, hogy adja meg az alakját, azaz határozza meg a számára szükséges CPU-k számát, és hogy mennyi ideig igényli ezeket. Ezen információk ismeretében a munkaütemező megpróbálja lefedni a CPU-idő téglalapot. A lefedéses ütemezés különösen akkor hatékony, amikor nappal küldik el a programokat éjszakai futtatásra, ilyenkor a munkaütemezőnek minden információ előre rendelkezésére áll a programokról, és így azokat optimális sorrendben tudja futtatni, ahogy a 8.43. (c) ábrán látható.

8.4.6. Alkalmazásszintű közös memória

Példáinkból kiderült, hogy a multiszámítógépek nagyobb méretűre skálázhatók, mint a multiprocesszorok. Ez a tény vezetett az MPI-hoz hasonló üzenetküldéses rendszerek kifejlesztéséhez. Sok programozó nem kedveli ezt a modellt, és azt szeretné, ha legalább a közös memória illúziója meglenne, még ha ténylegesen nem is létezik. Ennek megvalósítása a két megközelítés ideális kombinációja lenne: olcsó hardver (legalábbis feldolgozóegységenként) és egyszerű programozhatóság. Ez a párhuzamos számítások területének fő célja.

Sok kutató arra a következtetésre jutott, hogy ha a közös memória architekturális szinten történő megvalósítása nem is skálázható jól, más módszerekkel talán el lehet érni ugyanazt a célt. A 8.19. ábrán láthatjuk, hogy vannak más szintek is, amelyeknél a közös memória bevezethető. A következőkben megnézünk néhány módszert, amelyek segítségével a multiszámítógépek programozási modelljébe a közös memória bevezethető anélkül, hogy hardverszinten jelen lenne.

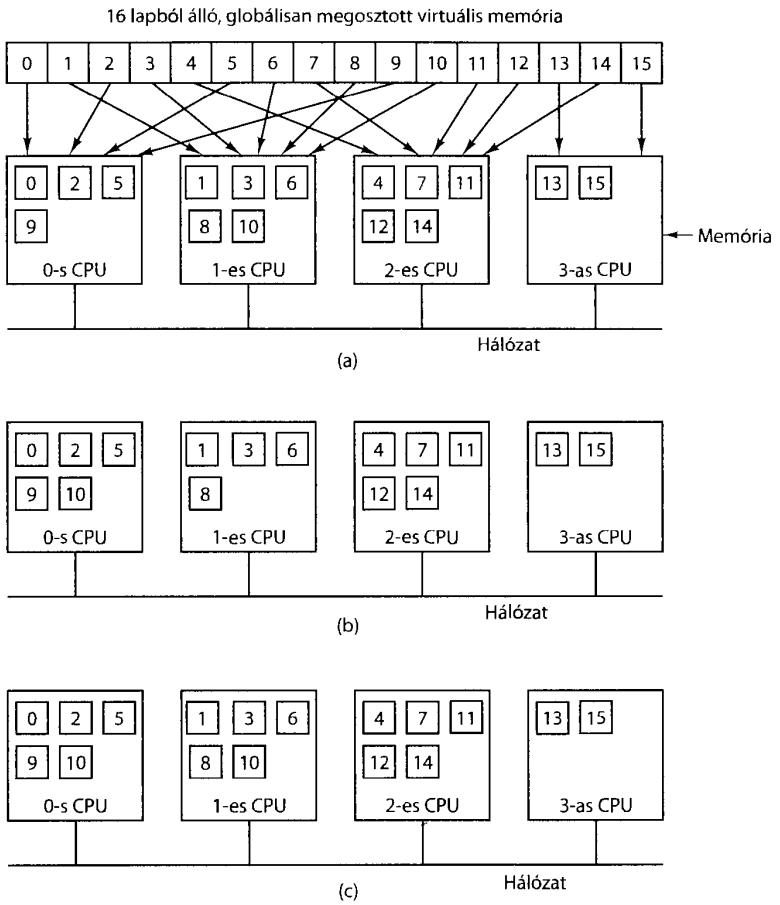
Osztott közös memória

Az alkalmazás szintű közös memóriarendszer megvalósításának egyik lehetősége a memórialapokon alapuló rendszer, amelyet **DSM**-nek (**Distributed Shared Memory, osztott közös memória**) neveztek el. Az alapötlet egyszerű: egy multiszámítógép CPU-inak egy halmaza osztozik egy közös, lapokból álló virtuális címterületen. A legegyszerűbb változatban mindegyik lap pontosan egy CPU RAM-jában található meg. A 8.44. (a) ábrán egy 16 lapból álló közös virtuális címterület látható négy CPU között széttagolva.

Ha a CPU a saját RAM-jában levő lapra hivatkozik, akkor az olvasás vagy írás késedelem nélkül megtörténik. Ha azonban a CPU egy távoli memóriában levő lapra hivatkozik, akkor laphiány lép fel. A hiányzó lap lemezről történő betöltése helyett a futó rendszer vagy az operációs rendszer egy üzenetet küld a lapot birtokló csomópontnak, hogy szabadítsa fel és küldje el. Miután a lap megérkezik, a CPU RAM-jába kerül, és a hiányt kiváltó utasítás végrehajtása újraindul, éppen úgy, mint egy közönséges laphiánynál. A 8.44. (b) ábra azt a helyzetet mutatja, amely azt követően áll elő, hogy a 0-s CPU 10-es lapra történő hivatkozása hiányt okozott, és ezért az 1-es CPU-tól a 0-s CPU-hoz került a kért lap.

Az alapötlet elsőként az IVY-ban (Li és Hudak, 1986; 1989) valósult meg. Ez teljesen megosztott, sorosan konzisztens memóriát biztosított egy multiszámítógépen. Mindamellett a teljesítmény fokozása érdekében számos optimalizálásra van lehetőség. Az első ilyen optimalizáció az IVY-ban, hogy a csak olvashatóként megjelölt lapok egyszerre több csomópontnál is jelen legyenek. Így amikor laphiány lép fel, akkor a hiányt okozó géphez a lap egy másolata kerül elküldésre, az eredeti lap a helyén marad, mivel ilyenkor nincs konfliktusveszély. A 8.44. (c) ábrán látható az a helyzet, amikor két CPU osztozik egy csak olvasható lapon (10-es lap).

A teljesítmény gyakran még ezzel az optimalizációval együtt sem kielégítő, különösen mikor egy folyamat egy lap tetejére, egy másik CPU-n futó másik folya-



8.44. ábra. *Egy multiszámítógép négy csomópontja között szétosztott 16 lapból álló virtuális címtartomány. (a) Kiindulási helyzet. (b) Miután a 0-s CPU a 10-es lapra hivatkozott. (c) Miután az 1-es CPU a 10-es lapra hivatkozott, feltételezve, hogy a lap csak olvasható*

mat pedig ugyanennek a lapnak az aljára ír folyamatosan. Mivel a lapnak csak egy másolata létezik, ezért a lapnak állandóan oda-vissza kell pattognia. Ez a helyzet **téves megosztás**ként ismert.

A téves megosztás elhárítására számos módszer alkalmazható. A Treadmarks rendszernél például a soros konzisztenciájú memóriáról lemondanak az elengedési konzisztencia kedvéért (Amza, 1996). Itt egyszerre több csomópontnál is jelen lehet egy potenciálisan írható lap, de a folyamatoknak írás előtt jelezniük kell a szándékukat egy acquire kérő művelet formájában. Ekkor a legutoljára írt másolat

kivételével az összes többi másolat érvénytelenítődik. Új másolat mindaddig nem készül, amíg egy megfelelő release művelet végre nem hajtódik, ekkor a lap újra megosztottá válhat.

A Treadmarksban van egy másik optimalizáció is, amely kezdetben minden írható lapot csak olvashatóra állít be. A lapra történő első íráskor védelmi hiba lép fel, és a rendszer elkészíti a lap egy **iker**nek nevezett másolatát. Ekkor a lap állapota olvasható-írható, és az egymást követő írások teljes sebességgel hajtódhatnak végre. Ha később egy távoli laphiány fordul elő, és a lapot oda kell szállítani, akkor az aktuális lap és az iker szavanként összehasonlításra kerül. Csak a megváltozott szavak kerülnek elküldésre, hogy ezáltal az üzenet mérete csökkenjen.

Laphiány esetén a hiányzó lapot meg kell keresni. Több megoldás is lehetséges, beleértve a NUMA és a COMA gépeken alkalmazottakat, mint például a (tulajdonos szerinti) katalógusok. Valójában a DSM-nél használt megoldások közül sok alkalmazható a NUMA és a COMA esetében is, mivel a DSM valójában a NUMA vagy COMA szoftveres megvalósítása, ahol mindegyik lap gyorsítósorként van kezelve.

A DSM jelenleg is a kutatás előterében van. A következő rendszerek igen érdekesek: CASHMERE (Kontothanassis és társai, 1997), CRL (Johnson és társai, 1995), Shasta (Scales és társai, 1996) és a Treadmarks (Amza, 1996; Lu és társai, 1997).

Linda

Az IVY-hoz és a Treadmarks-hoz hasonló lapozási technikájú DSM-rendszerek az MMU-hardvert használják, hogy programmegszakítással a hiányzó lapok elérhetővé váljanak. Bár segítséget jelent, ha teljes lapok helyett csupán az eltéréseket küldjük át, de a helyzet valójában az, hogy a lapok a megosztás egy erőltetett formáját jelentik, ezért ennek a problémának más megközelítései is vannak.

A Linda például úgy közelíti meg ezt a kérdést, hogy a különböző gépeken futó folyamatok számára egy erősen strukturált, megosztott közös memóriát biztosít. (Carriero és Gelernter, 1986; 1989). Ennek a memóriának az eléréséhez kisszámú primitív műveletet használ, amelyekkel egy létező nyelv is bővíthető, mint például a C és a Fortran, így párhuzamos nyelvek jönnek létre, esetükben a C-Linda és a Fortran-Linda.

A Linda mögötti egységesítő elv az absztrakt **adategységtér**, amely a teljes rendszer számára globális, és az összes folyamata képes elérni. Ez az adategységtér egy globális közös memóriához hasonló, ráadásul egy bizonyos beépített struktúrával rendelkezik. Az adategységtér olyan **adategység**ekből áll, amelyek mindegyike egy vagy több mezőt tartalmaz. A C-Lindában vannak egész, hosszú egész, lebegőpontos szám típusú mezők, de lehet a mezők típusa összetett, például tömbök (bele-

("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is sister", Carolyn, Elinor)

8.45. ábra. *Három Linda adategység*

értve a karakterláncokat is) és struktúrák (kivéve az adategységet). Példaként három adategység látható a 8.45. ábrán.

Az adategységeken négy művelet végezhető. Az egyik művelet az out, amely betesz egy adategységet az adategységtérbe. Például

```
out("abc", 2, 5);
```

az („abc", 2, 5) adategységet a térbe teszi. Az out mezői általában konstansok, változók vagy kifejezések, mint a következő példában:

```
out("matrix-1", i, j, 3.14);
```

beteszi a térbe a négy mezőből álló adategységet, amelyben a második és a harmadik mező értékét az i és j változók aktuális értéke határozza meg.

Az adategységek az in primitívvel kereshetők ki az adategységtérből. Név vagy cím helyett itt tartalommal történik a hivatkozás. Az in mezői kifejezések vagy formális paraméterek lehetnek. Nézzük példaként a következőt:

```
in("abc", 2, ? i);
```

Ez a művelet kikeres az adategységtérből egy olyan adategységet, amelynek első mezője az „abc" karakterlánc, a második mező a 2 egész szám, a harmadik mező pedig tetszőleges egész szám (feltéve, hogy az i egész típusú). Ha talál ilyet, akkor ezt az adategységet a térből eltávolítja, és az i változó felveszi a harmadik mező értékét. Az illesztés és az eltávolítás atomi tevékenység, ezért ha két folyamat párhuzamosan ugyanazt az in műveletet hajtja végre, akkor a kettő közül csak az egyiknek sikerül ezt megtenni, kivéve, amikor kettő vagy több illeszkedő adategység is van. Az adategységtér ugyanannak az adategységnek több példányát is tartalmazhatja.

Az in műveletnél használt illesztő algoritmus nagyon egyszerű. Az in primitív **mintá**nak nevezett mezői kerülnek összehasonlításra az adategységtér minden egyes adategységének megfelelő mezőjével. Illeszkedés akkor fordul elő, ha a következő három feltétel mindegyike teljesül:

1. A minta és az adategység mezőinek a száma megegyezik.
2. A megfelelő mezők típusa azonos.
3. A minta mindegyik konstansa vagy változója illeszkedik az adategységben lévő, neki megfelelő mezőre.

A formális paraméterek, amelyeket egy kérdőjel és az azt követő változónév vagy típus jelez, nem vesznek részt az illesztésben (kivéve a típusellenőrzést), de sikeres illeszkedés esetén a változók értéket kapnak.

Ha nincs illeszkedő adategység, akkor a hívó folyamat felfüggesztődik, amíg egy másik folyamat a szükséges adategységet el nem helyezi, ebben a pillanatban a hívó automatikusan feléled és megkapja az új adategységet. Mivel a folyama-

tok felfüggesztése és feléledése automatikusan történik, ezért ha egy folyamat egy adategységet készül küldeni, egy másik folyamat pedig ennek a fogadására készül, akkor lényegtelen, hogy melyik művelet előzi meg a másikat.

Az out és az in műveleteken túl a Linda harmadik primitívje a read, amely az in primitívtől csak annyiban tér el, hogy az adategységtérből nem távolítja el az adategységet. Egy másik primitív az eval, amelynek paraméterei párhuzamosan értékelődnek ki, és az eredményül kapott adategység az adategységtérbe kerül elhelyezésre. Ennek alkalmazásával tetszőleges kiszámítások végezhetők el. A Lindában ennek segítségével lehet párhuzamos folyamatokat létrehozni.

A Lindában gyakori programozási paradigma a **többszörözött munkás modell**. Ez a modell az elvégzendő munkákkal teletömött **feladatzsák** ötletén alapszik. A fő folyamat egy ciklus végrehajtásával kezdődik, amely tartalmazza az:

```
out("feladat_zsák", munka);
```

utasítást. Ezáltal minden iterációban egy újabb munkaleírás kerül be az adategységtérbe. Mindegyik munkás azzal kezdi, hogy kér egy munkaleírást tartalmazó adategységet az alábbi utasítással:

```
in("feladat_zsák", ?munka);
```

amelyet aztán elvégez. Mikor elkészült, kér egy másikat. Végrehajtás alatt is elhelyezhető új munka a feladatzsákba. Ezzel az egyszerű módszerrel csekély többletráfordítás árán a munka dinamikusan oszthatő ki a munkások között, és az összes munkás állandóan foglalkoztatva van.

Multiszámítógépes rendszerekre a Linda többféle megvalósítása létezik. Mindegyikükben az a kulcskérdés, hogyan osszák szét az adategységeket a gépek között, és amikor szükséges, hogyan találják meg azokat. Több lehetőség is van, például az üzenetszórás és a katalógusok használata. A többszöröződés szintén fontos kérdés. Ezekről a kérdésekről lásd (Bjornson, 1993).

Orca

A multiszámítógépeken az alkalmazásszintű közös memória valamelyest más megközelítése adategységek helyett teljes objektumokat használ a megosztás egységeként. Az objektumok belső (rejtett) állapottal, és ezeken az állapotokon működő eljárásokkal rendelkeznek. Azáltal, hogy a programozónak nincs lehetősége az állapot közvetlen elérésére, számos lehetőség nyílik meg a fizikailag közös memória nélküli gépek közötti adatmegosztásra.

Egy ilyen, a multiszámítógépes rendszereken a közös memória illúzióját keltő objektumalapú rendszer az Orca (Bal, 1991; Bal és társai, 1992; Bal és Tanenbaum, 1988). Az Orca egy hagyományos programozási nyelv (a Modula 2-n alapszik), két új lehetőséggel kibővítve: az objektumokkal és új folyamatok létrehozásának képességével. Egy Orca-objektum olyan jellegű absztrakt adattípus, mint egy Java-

objektum vagy egy Ada-csomag. Belső adatstruktúrákat és a felhasználó által írt eljárásokat zár egységbe, az utóbbit **műveleteknek** hívják. Az objektumok passzívak abban az értelemben, hogy nincs olyan száljuk, amelynek üzenetek küldhetők. Ehelyett a folyamatok egy objektum belső adatait az objektumhoz tartozó eljárások meghívásával érhetik el.

Mindegyik Orca-eljárás egy (őr, utasításblokk) párokból álló lista. Az őr egy mellékhatás nélküli logikai kifejezés, az üres őrnek az *igaz* logikai érték felel meg. Egy művelet hívásakor az összes őr valamilyen nem meghatározott sorrendben kiértékelődik. Ha mindegyik értéke *hamis*, a hívó eljárásnak addig kell várnia, amíg valamelyik *igaz* értéket nem vesz fel. Amikor egy őr értéke *igaz* lesz, az azt követő utasításblokk végrehajtódik. A 8.46. ábra egy kétműveletes (*push* és *pop*) verem- *(stack)* objektumot mutat be.

```
Object implementation stack;
 top:integer;                          # a verem tárolója
 stack: array [integer 0..N-1] of integer;

 operation push(item: integer);        # a művelet nem ad vissza semmit
 begin
  guard top < N - 1 do
   stack[top] := item;                 # beletesz egy elemet a verembe
   top := top + 1;                     # növeli a veremmutatót
  od;
 end;

 operation pop( ): integer;            # a művelet egy egészet ad vissza
 begin
  guard top > 0 do                     # üres verem esetén felfüggesztődik
   top := top - 1;                     # csökkenti a veremmutatót
   return stack[top];                  # visszaadja a verem tetején levő elemet
  od;
 end;

begin
 top :=0;                              # kezdeti beállítás
end;
```

8.46. ábra. *Egyszerűsített Orca-veremobjektum belső adatokkal és két művelettel*

A *stack* objektum definiálása után definiálhatók ilyen típusú változók, mint például:

```
s, t: stack;
```

ami két stack objektumot hoz létre, mindegyikben a *top* változó értékét 0-ra állítva. Egy *k* egész típusú változó az alábbi utasítással tehető bele az *s* verembe:

```
s$push(k);
```

A *pop* műveletben van őr (guard), így ha egy hívó üres veremből akar értéket kiolvasni, akkor egész addig felfüggesztődik, míg egy másik folyamat valamit a verembe nem tesz.

Az Orcában egy felhasználó által meghatározott processzoron új folyamatok indíthatók a **fork** utasítással. Az új folyamat a **fork** utasításban megnevezett eljárást futtatja. Az új folyamatnak átadhatók paraméterek, beleértve az objektumokat is. Ennek révén az objektumok megosztottá tehetők a gépek között. Például a következő utasítás az 1-es géptől az *n*-es gépig mindegyiken egy új folyamatot hoz létre, ezek mindegyike a *foobar* programot futtatja.

```
for i in 1 .. n do fork foobar(s) on i od;
```

Mivel ez az *n* darab új folyamat (és a szülő) párhuzamosan fut, mindegyik úgy tehet be és vehet ki elemeket a közös *s* veremből, mintha egy közös memóriájú multiprocesszoron futnának. A futtató rendszer feladata a valójában nem létező közös memória illúziójának fenntartása.

Az osztott objektumok műveletei atomiak, és sorosan konzisztensek. A rendszer garantálja, hogy ha több folyamat csaknem párhuzamosan ugyanazon a megosztott objektumon végez műveletet, akkor ugyanazt a rendszer által választott sorrendet látja mindegyik folyamat.

Az Orca az adatok megosztását és a szinkronizációt a lapozásos DSM-től eltérő módon egyesíti. A párhuzamos programokban kétféle szinkronizációra van szükség. Az egyik a kölcsönös kizárás, amely megakadályozza, hogy két folyamat egyszerre kritikus szakaszban lévő kódot futtasson. Az Orcában megosztott objektumon végzett minden művelet valójában egy kritikus szakasznak felel meg, mert a rendszer biztosítja, hogy a végső eredmény ugyanaz legyen, mintha az összes kritikus tartomány egyesével (azaz szekvenciálisan) lenne futtatva. Ebben az értelemben egy Orca-objektum a monitor (Hoare, 1975) egy osztott formája.

A szinkronizáció másik fajtája a feltételes szinkronizáció, amelyben egy folyamat bizonyos feltételek teljesülésére várakozva blokkolódik. Az Orcában a feltételes szinkronizációt az őrök végzik. A 8.46. ábrában levő példánál egy üres veremből olvasni akaró folyamat felfüggesztődik, amíg a verem üres.

Az Orca futtató rendszere kezeli az objektumok többszöröződését, vándoroltatását, a konzisztenciát és a művelethívásokat. Mindegyik objektum két állapot egyikében lehet: egyedüli vagy többszörözött. Egy egyedüli állapotú objektum csak egyetlen gépen létezik, ezért minden rá vonatkozó kérés ide kell, hogy érkezzen. Egy többszörözött objektum az összes olyan gépen jelen van, amelynek folyamata használja azt, ezáltal az olvasás művelet könnyebbé válik (mivel helyileg végrehajtható), de a frissítés sokkal költségesebb. Többszörözött objektumon végrehajtandó módosító műveletnek először egy sorszámot kell kapnia egy ezek kiosztásával foglalkozó központi folyamattól. Ezután az objektumot tároló összes géphez egy üzenet kerül elküldésre, amely a művelet végrehajtását kéri. Mivel az összes ilyen frissítés sorszámmal rendelkezik, ezért a műveletek végrehajtási sorrendjét az összes gépen ezek a sorszámok határozzák meg, így biztosítva a soros konzisztenciát.

Globe

A legtöbb DSM-, Linda- és Orca-rendszer helyi rendszereken fut, amely vagy egyetlen épületen belül van, vagy egy szűkebb területen helyezkedik el. Azonban lehetséges az egész világra kiterjedő alkalmazás szintű közös memóriát használó multiszámítógép megépítése is. A Globe-rendszerben egy objektum több folyamat címterületén is elhelyezkedhet ugyanabban az időben, akár különböző földrészeken (Kermarrec és társai, 1998; Van Steen és társai, 1999). Egy megosztott objektum adatait a felhasználói folyamatok annak eljárásain keresztül érhetik el, amelyek implementációja a különböző objektumoknál más-más lehet. Például az egyik lehetőség az, hogy egy dinamikusan használt adatnak csak egyetlen példánya van (ez egy jó megoldás az olyan adatnál, amelyet az egyedüli tulajdonos módosít gyakran). Egy másik lehetőség, hogy az objektum mindegyik példányába bekerül az adat, és a példányok frissítése egy megbízható üzenetszóró protokollal történik.

A Globe céljaiban nagyon becsvágyó, mert az a fő törekvése, hogy skálázható legyen akár milliárdnyi felhasználó és ennél egy nagyságrenddel több (potenciálisan hordozható) objektum számára. Az objektumok helyének meghatározása, kezelése és a skálázás döntő fontosságú. A Globe ehhez egy általános keretrendszert alkalmaz, amelyben minden egyes objektum többek között megadja a saját többszörözési stratégiáját, biztonsági stratégiáját és így tovább. Ezzel elkerülhetők azok a problémák, amelyek más rendszerekben a kényszerűen egységes kezelésből adódnak, miközben a közös memória révén megőrzi a programozás egyszerűségét.

További nagy területű osztott rendszerek a Globus (Foster és Kesselman, 1998a; Foster és Kesselman, 1998b) és a Legion (Grimshaw és Wulf, 1996; Grimshaw és Wulf, 1997), de ezek a Globe-bal ellentétben nem nyújtják a közös memória illúzióját.

8.4.7. Teljesítmény

A párhuzamos számítógépek építésénél a fő szempont, hogy egy egyprocesszoros gépnél gyorsabb gépet állítsanak elő. Ha az eredmény nem felel meg ennek az egyszerű célkitűzésnek, akkor értéktelen az egész. Továbbá a célkitűzést költséghatékony módon kell elérni. Valószínűleg nem nagyon lehet eladni egy olyan gépet, amelyik egy egyprocesszorosnál kétszer gyorsabb, de annál 50-szer drágább. Ebben a részben a párhuzamos számítógép-architektúrák néhány teljesítménnyel kapcsolatos kérdését tanulmányozzuk.

Hardvermértékek

Hardverszempontból nézve teljesítmény mértékként a CPU-k sebessége, a B/K egységek sebessége, valamint az összekötő hálózat teljesítménye érdekel bennünket. A CPU-k és a B/K egységek sebessége itt is ugyanannyi, mint az egyprocesz-

szoros esetben, ezért a párhuzamos rendszereknél az összekapcsolás paraméterei kulcsfontosságúak. A két jellemző paraméter: a késleltetési idő és a sávszélesség; ezeket most közelebbről is megvizsgáljuk.

A menettérti késleltetés egy CPU esetében az az időtartam, amely egy csomag elküldése és a válasz megérkezése között eltelik. Ha a csomagot a memóriának küldi, akkor a késleltetési idő egy szó vagy blokk kiolvasásához, illetve beírásához szükséges idővel azonos. Ha egy másik CPU-nak küldi a csomagot, akkor a késleltetési idő a processzorok kommunikációs ideje az adott méretű csomag esetében. Általában a minimális csomagokhoz – gyakran egyetlen szó vagy egy kis gyorsítósor – tartozó késleltetési idő az érdekes.

A késleltetési idő több tényezőből áll össze, és ezek eltérők a vonalkapcsolásos, a tárold és továbbítsd csomagkapcsolásos, a virtuális vágásos és a féreglyuk módszert alkalmazó hálózatoknál. A vonalkapcsolásnál a késleltetési idő a kapcsolat felépítésére és az átvitelre fordított idők összege. Egy kapcsolat felépítéséhez egy próbacsomagot kell kiküldeni, amelynek a feladata az erőforrások lefoglalása, ezt követően jön a visszaigazolás. Miután ez megtörtént, az adatcsomagot kell összerakni. Mikor ez kész, a bitek teljes sebességgel áramolhatnak, így ha T_s a kapcsolat felépítéséhez szükséges idő, p bitből áll a csomag, és a sávszélesség b bit/s, akkor az egyirányú késleltetési idő $T_s + p/b$. Ha teljes duplex összekötés van, akkor a válaszhoz nem kell a kapcsolatot ismét felépíteni, így a minimális késleltetési idő egy p bites csomag kiküldéséhez és p bites visszaigazoláshoz $T_s + 2p/b$.

A tárold és továbbítsd csomagkapcsolásnál nincs szükség arra, hogy előzetesen próbacsomagot küldjünk a célállomáshoz, viszont a csomag összerakásához bizonyos előkészítési idő kell, ezt jelölje T_a. Ekkor ahhoz, hogy a csomag az első switch-ig eljusson, $T_a + p/b$ idő kell. Mondjuk a switch-ben T_d időt kell a csomagnak várakozni, és ez a késleltetés ismétlődik minden ezt követő switch-nél. A T_d késleltetési idő a feldolgozási időből és abból a sorbaállási időből tevődik össze, amelyet a kimeneti kapu szabaddá válásáig eltelik. n switch esetében az egyirányú teljes késleltetési idő $T_a + n\,(p/b + T_d) + p/b$, ahol az utolsó tag az az idő, amely a csomagnak az utolsó switch-ből a címzettnek való átadásához kell.

A virtuális vágásos forgalomirányításoknál és a féreglyuk módszernél az egyirányú késleltetési idők a legjobb esetben a $T_a + p/b$ időhöz közeliek, ugyanis ezeknél nincs szükség se egy próbacsomag kiküldésére a kapcsolat kiépítéséhez, se pedig tárolási időt nem igényelnek a switch-eknél. Alapvetően csak a csomagok összeállítása és a bitek kimenetre helyezése igényel időt. Valójában még a terjedési sebesség végességéből adódó késést is figyelembe kellene venni, de ez rendszerint elhanyagolhatóan kicsi.

A másik hardvermérték a sávszélesség. Sok párhuzamosan futó program, különösen a természettudomány területén, nagy mennyiségű adat mozgatásával jár, ezért a teljesítmény szempontjából fontos jellemző, hogy a rendszer másodpercenként hány bájt mozgatására képes. Többféle sávszélességmérték létezik. Ezek közül egyet, a kettévágott sávszélességet már láttunk. Egy másik az **összesített sávszélesség**, amely egyszerűen az összes kapcsolat kapacitásának az összege. Ez a szám az egyszerre úton lévő bitek számának maximumát jelenti. Szintén fontos mérték a CPU-k átlagos kimeneti sávszélessége. Ha mindegyik CPU kimeneti kapacitása

1 MB/s, akkor nem sokat segít, ha a hálózat kettévágott sávszélessége 100 GB/s. Az egyes CPU-k kibocsátotta adatok mennyisége korlátozza a kommunikációt.

Gyakorlatilag az elvi sávszélességhez közeli érték elérése nagyon nehéz feladat. A kapacitás csökkenését okozó többletmunkának sok forrása van. Például, mindig megjelennek a csomagokkal kapcsolatos teendők: összeállítás, a fejlécek kialakítása és útnak indítása. 1024 darab 4 bájtos csomag elküldésével sohasem érhető el az egy darab 4096 bájtos csomagküldési sávszélesség. Sajnos az alacsony késleltetési idő eléréséhez a kisebb csomagok alkalmazása célszerűbb, mivel a nagyméretű csomagok hosszabb ideig foglalják a kapcsolatokat és a switch-eket. Így az alacsony átlagos késleltetési idő elérése és a magas sávszélesség kihasználása egymással ellentétes törekvések. Néhány alkalmazás számára az egyik sokkal fontosabb, míg más alkalmazások esetében a másik. Érdemes megjegyezni azonban, hogy nagyobb sávszélesség mindig elérhető (több vagy szélesebb vezeték behelyezésével), de alacsony késleltetési idő nem. Ezért általában az a kedvezőbb számunkra, ha a késleltetési időt a lehető legkisebbre csökkentjük, a sávszélességgel ráérünk azután foglalkozni.

Szoftvermértékek

A késleltetési időhöz és a sávszélességhez hasonló hardvermértékek azt mutatják meg, hogy a hardver mire képes. A felhasználónak azonban más szempontjai is lehetnek. Azt akarja tudni, hogy a programjai mennyivel lesznek gyorsabbak egy párhuzamos számítógépen az egyprocesszoros géphez képest. Számukra a legfontosabb mutató a gyorsulás, amely azt fejezi ki, hogy hányszor gyorsabb egy program egy n processzoros rendszeren, mint egy egyprocesszoroson. Szokásos diagrammal megadni ezeket az eredményeket, ahogy az a 8.47. ábrán is látható. Itt különböző párhuzamos programokat látunk egy 64 darab Pentium Pro CPU-t tartalmazó párhuzamos számítógépen futtatva. Mindegyik görbe egy program k darab CPU melletti gyorsulási értékét mutatja k függvényében. A pontozott vonal a tökéletes gyorsulást mutatja, vagyis azt, hogy ha tetszőleges k értékre teljesül, hogy k db CPU k-szoros gyorsulást eredményez. Kevés programnál érhető el ez az optimális gyorsulás, de néhány azért megközelíti. Az N-test probléma párhuzamos futtatása rendkívül jó eredményt mutat, az Awari (egy afrikai táblajáték) esetén elfogadható az eredmény, míg egy nagyméretű mátrix invertálásakor sohasem éri el az ötszörös gyorsítást, akárhány CPU áll is rendelkezésre. A programok és az eredmények vizsgálatáról lásd (Bal és társai, 1998).

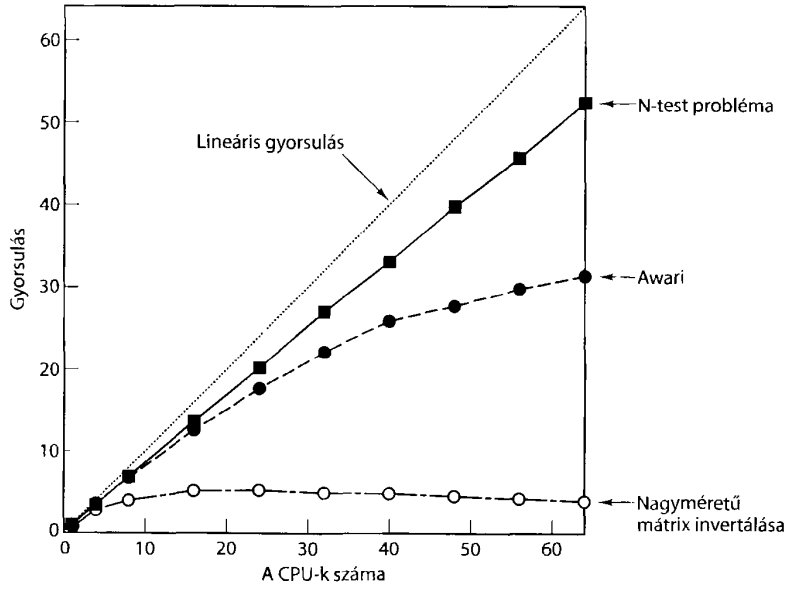
Az optimális gyorsulás azért valósítható meg nagyon nehezen, mert majdnem minden programnak vannak szekvenciális részei, például a kezdeti értékadások, adatbeolvasások vagy az eredmények összegyűjtése. Itt a több CPU nem jelent segítséget. Tételezzük fel, hogy egy program egy egyprocesszoros gépen T ideig fut, és jelölje f azt a hányadot, amelyben szekvenciális kódot hajt végre, míg a többi $(1 - f)$ részben párhuzamosítható kódot, mint ahogy a 8.48. (a) ábra mutatja. Ha ez az utóbbi programrész n CPU-s gépen futna, akkor ennek a résznek a futási ideje $(1 - f)\,T$-ről $(1 - f)\,T/n$-re csökkenhetne a legjobb esetben. Így a szekven-

ciális és a párhuzamos részek teljes futási idejére $fT + (1-f)\,T/n$ adódna. A gyorsítás mértékét az eredeti program T végrehajtási idejének és az új futási időnek a hányadosa adja:

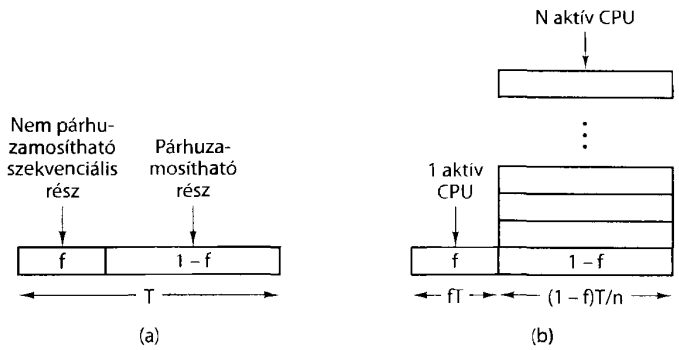
$$Gyorsulás = \frac{n}{1 + (n-1)f}$$

Az $f = 0$ esetben lineáris gyorsulást kapunk, de ha $f > 0$, akkor van szekvenciális rész, és optimális gyorsulás nem lehetséges. Ez az eredmény **Amdahl törvényeként** ismert.

Nemcsak az Amdahl-törvény az oka, hogy szinte lehetetlen az optimális gyorsulás elérése. Szerepet játszik ebben a kommunikációs késleltetési idő, a kommunikáció véges sávszélessége és az algoritmusok hatékonysága. Még ha 1000 CPU is állna rendelkezésünkre, akkor sem lehet minden programot úgy megírni, hogy ilyen sok CPU-t használjon, és az elindításuk is jelentős többletköltséggel jár. Továbbá, gyakran a legjobb algoritmus nem párhuzamosítható jól, így egy kevésbé jó algoritmust kell a párhuzamos esetben használni. Mindezek ellenére sok program esetében kívánatos az n-szeres gyorsulás még úgy is, ha ehhez $2n$ CPU kell. Végülis a CPU-k nem drágák, és sok cég elboldogul 100%-nál jóval kisebb hatékonysággal más területeken is.



8.47. ábra. *A valódi programoknál elérhető gyorsulások a pontozott vonallal jelölt optimális értékek alatt maradnak*



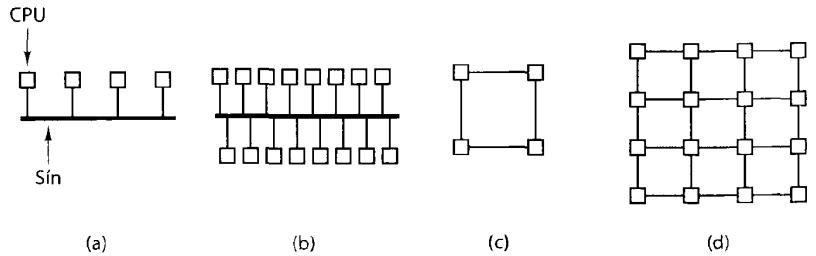
8.48. ábra. *(a) A programnak van szekvenciális és párhuzamosítható része is. (b) Annak hatása, ha a program egy részét párhuzamosan futtatjuk*

Nagy teljesítmény elérése

A teljesítménynövelés legnyilvánvalóbb módja a CPU-k számának növelése a rendszerben. Azonban ennek a bővítésnek oly módon kell megvalósulnia, hogy közben sehol ne keletkezzen szűk keresztmetszet. **Skálázható**nak nevezik az olyan rendszereket, melyekben a CPU-k számának növelésével arányosan növekszik a gép számítási teljesítménye is.

A skálázhatóság néhány következményének vizsgálatához tekintsünk egy sínnel összekötött 4 CPU-s rendszert; ez látható a 8.49. (a) ábrán. Most képzeljük el, hogy ezt a rendszert 16 CPU-sra skálázzuk 12 CPU hozzávételével, amint ezt a 8.49. (b) ábra mutatja. Ha a sín sávszélessége b MB/s, akkor a CPU-k számának négyszerezésével a CPU-ként elérhető sávszélesség $b/4$ MB/s-ról $b/16$ MB/s-ra csökken. Vagyis az ilyen rendszerek nem skálázhatók.

Most egy rácsalapú rendszerrel tegyük ugyanezt; a 8.49. (c) és 8.49. (d) ábrán ez látható. Ennél a topológiánál a CPU-kkal való bővítés a kapcsolatok számának



8.49. ábra. *(a) Egy 4 CPU-s rendszersínnel. (b) Egy 16 CPU-s rendszersínnel. (c) Egy 4 CPU-s rács topológiájú rendszer. (d) Egy 16 CPU-s rács topológiájú rendszer*

növekedésével jár együtt, és így a rendszer skálázása nem csökkenti a CPU-nkénti összesített sávszélességet, mint ahogy ez a sínrendszernél történt. Példánkban a kapcsolatok CPU-nkénti száma a 4 CPU-s rácshoz tartozó 1-ről a 16 CPU-s rácshoz tartozó (16 CPU és 24 kapcsolat) 1,5-re nő, vagyis a CPU-kkal való bővítés javítja a CPU-nkénti összesített sávszélességet.

Természetesen a sávszélesség nem az egyedüli szempont. Sínrendszernél a CPU-k hozzáadása nem növeli az összekötő hálózat átmérőjét vagy forgalommentes időszakban a késleltetési időt, míg a rács topológiánál ezek növekednek. Egy $n \times n$-es rács átmérője $2(n-1)$, így a legrosszabb (és az átlag) esetben a késleltetési idő közelítőleg a CPU-k számának négyzetgyökével arányosan nő. 400 CPU-nál az átmérő 38, azonban 1600 CPU-nál ez a szám 78, vagyis a CPU-k számának négyszereződése közel duplájára növeli az átmérőt, és ezzel együtt az átlagos késleltetési időt is.

Ideális esetben egy skálázható rendszernél újabb CPU-k hozzáadása mellett sem szabadna megváltozni se a CPU-nkénti átlagos sávszélességnek, se pedig a konstans értékű átlagos késleltetési időnek. A gyakorlatban a CPU-nkénti megfelelő sávszélesség elérhető, de az összes gyakorlati példa azt mutatja, hogy a késleltetési idő a mérettel együtt nő. A logaritmikus növekedés nagyjából a legjobb, ami ezzel kapcsolatban elérhető, mint a hiperkockánál.

A rendszerek skálázásából származó késleltetési idő növekedés azért probléma, mert gyakran a finom és közepes szemcsézettségű alkalmazások hatékonyságát drasztikusan lerontja. Amikor egy programnak olyan adatra van szüksége, amely nincs a lokális memóriában, akkor ezek elérése gyakran tekintélyes időt igényel, és ahogy láttuk, nagyobb rendszereknél hosszabb késleltetéssel jár együtt. Ez a probléma ugyanúgy jelentkezik a multiprocesszoroknál, mint a multiszámítógépeknél, mivel mindkettőnél a fizikai memória mindig egymástól távol eső modulokra osztott.

Ezekből a megfigyelésekből kiindulva a rendszertervezők különböző módszerek alkalmazásával nagy gondot fordítanak arra, hogy a késleltetési idő csökkenjen vagy rejtve maradjon. Most ezekre a módszerekre térünk rá. Az első késleltetési időt elrejtő technika az adat többszörözés. Ha egy adatblokk másolatait több helyen is őrizzük, akkor ezeken a helyeken az adatelérés felgyorsul. Egy ilyen többszörözési technika a gyorsítótár alkalmazása, amelynél az adatból a „rendes" helye mellett egy vagy több másolatot is tárolnak a felhasználási helyekhez közel. Más módszert kell alkalmazni egyenrangú többszörös másolatok fenntartásához – ekkor a másolatok státusza megegyezik –, mint amikor aszimmetrikus elsődleges/másodlagos viszonyt használnak a gyorsítótárazásnál. Amikor valamilyen formában több másolat is létezik, akkor a legfontosabb kérdések, hogy az adatblokkokat mikor, ki és hol helyezi el. A válaszok kiterjednek a hardver által igény szerint végrehajtott dinamikus helyfoglalástól a fordítási direktívákkal vezérelt betöltési idejű helyfoglalásáig. Minden esetben lényeges kérdés a konzisztencia fenntartása.

A második késleltetési időt elrejtő technika az **előolvasás**. Ha egy adatelemet azt megelőzően betöltünk, hogy szükség lenne rá, akkor a betöltő folyamat egybeeshet a normál végrehajtással, és így mikor az adatelemre szükség lesz, akkor rendelkezésre fog állni. Az előolvasás lehet automatikus és program által vezérelt

is. Amikor egy gyorsítótár nemcsak a kért szót, hanem a szót tartalmazó egész sort betölti, akkor arra számít, hogy a szomszédos szavakra hamarosan szükség lesz.

Az előolvasást lehet közvetlenül is vezérelni. Mikor a fordító észleli, hogy bizonyos adatokra szükség lesz, elhelyezhet egy közvetlen betöltő parancsot az utasítássorozatban úgy, hogy a szükséges időben az adatok már elérhetők legyenek. Ennek a stratégiának az a feltétele, hogy a fordítónak teljes körű információja legyen a hardverről, az időzítésekről, valamint vezérelni tudja az adatok elhelyezését. Az ilyen fordító által kezdeményezett LOAD utasítások akkor dolgoznak a leghatékonyabban, ha biztosan tudják, hogy az adatra szükség lesz. Nagyon költséges egy olyan LOAD miatti laphiány, amelyet egy rosszul jövendölt ugrás miatt kezdeményeztünk.

A harmadik késleltetési időt elrejtő technika a többszálú végrehajtás, ahogy azt már korábban láttuk. Ha a folyamatok között elég gyors a váltás, például azáltal, hogy mindegyiknek adunk saját memórialeképezőt és hardverregisztereket, akkor abban az esetben, amikor az egyik folyamat távoli adat érkezésére várva blokkolódik, a hardver gyorsan átkapcsolhat egy futtathatóra. Kivételes esetben az is előfordulhat, hogy a CPU végrehajt az első szálon egy utasítást, majd egy másik szálon egyet és így tovább. Ilyen módon a CPU folyamatos terhelése biztosítható még az egyes szálak hosszú memóriakésleltetései esetén is.

A negyedik késleltetési időt elrejtő technika a nem blokkoló írás. Alapesetben egy STORE utasítás végrehajtásakor a CPU addig vár a folytatással, amíg a STORE be nem fejeződik. Nem blokkoló írás esetén a memóriaműveletek alatt a program folytatódhat tovább. A LOAD utasítás kiadása utáni folytatás nehezebb, de soron kívüli végrehajtással még ez is lehetséges.

8.5. Grid számítások

Napjainkban tudományos, mérnöki, környezetvédelmi és más területeken olyan kihívásokkal nézünk szembe, amelyek közül sok nagyléptékű és interdiszciplináris. Megoldásukhoz sok, gyakran különböző országbeli szervezet szakértelmére, gyakorlottságára, tudására, szoftverére és adataira van szükség. Néhány példa:

1. Marsi küldetésen dolgozó tudósok.
2. Összetett terméket (például gátat vagy repülőgépet) építő konzorcium.
3. Természeti katasztrófa után a segélyezést összehangoló nemzetközi segélyszervezet.

Ezek némelyike hosszú távú együttműködés, mások rövidebbek, de közös bennük, hogy a különböző erőforrásokkal és eljárásokkal rendelkező szervezeteknek együtt kell működniük egy közös cél elérése érdekében.

A legutóbbi időkig az erőforrások és adatok megosztása nagyon nehezen volt megoldható különböző számítógépekkel, operációs rendszerrel, adatbázisokkal és protokollokkal rendelkező szervezetek között. Az intézmények közötti szoros

együttműködés iránti egyre növekvő igény azonban oda vezetett, hogy kifejlesztették azokat a rendszereket és technológiákat, amelyekkel nagy távolságban lévő számítógépeket össze lehet kapcsolni a **grid**nek nevezett egységbe. Bizonyos értelemben a grid a következő lépés a 8.1. ábra tengelyén. Úgy tekinthetünk rá, mint egy nagyon nagy, nemzetközi, lazán kapcsolt heterogén klaszterre.

A grid célja, hogy megteremtse a technológiai alapjait annak, hogy egy közös cél érdekében dolgozó szervezetek csoportjai **virtuális szervezetet** alkothassanak. Ennek a virtuális szervezetnek rugalmasnak kell lennie, a tagoknak együtt kell tudniuk dolgozni a célterületeken, de ugyanakkor az általuk szükségesnek tartott mértékig meg kell tudniuk tartani az ellenőrzést saját erőforrásaik felett. Ezért a griddel foglalkozó kutatók szolgáltatásokat, eszközöket és protokollokat fejlesztenek ki, hogy a virtuális szervezetek működését lehetővé tegyék.

A grid jellemzően sokoldalú, egyenrangú felek részvételével. Összehasonlíthatjuk az ismert számítógépes rendszerekkel. A kliens-szerver modellben egy tranzakció két fél, a kliens és a szerver között zajlik. A szerver valamilyen szolgáltatást nyújt, a kliens pedig valamilyen szolgáltatást akar igénybe venni. Egy tipikus kliens-szerver példa a web, ahol a felhasználók webszerverekhez fordulnak információért. A grid különbözik azoktól az alkalmazásoktól is, amelyek két személy között tesznek lehetővé fájlcserét. Mivel a grid más, új protokollokra és technológiákra van szükség.

A gridben sokféle erőforráshoz kell hozzáférést biztosítani. Minden erőforráshoz tartozik egy jól meghatározott rendszer és szervezet; ez az erőforrás tulajdonosa. A tulajdonos dönti el, hogy az erőforrásból mennyit, kinek és mikor bocsát rendelkezésére. Bizonyos értelemben a grid tulajdonképpen az erőforrásokhoz való hozzáférés és azok kezelése.

A grid egyik lehetséges modellje a 8.50. ábrán látható. A **grid szerkezet** legalul azokat a komponenseket foglalja magában, amelyekből a grid felépül. Tartalmazza a CPU-kat, lemezegységeket, hálózatokat és érzékelőket a hardveroldalon, valamint a programokat és az adatokat a szoftveroldalon. Ezek azok az erőforrások, amelyeket a grid szabályozott formában elérhetővé tesz.

Egy szinttel feljebb az **erőforrásréteg** található, amely az egyes konkrét erőforrások kezelését végzi. Sok esetben a gridben használható erőforrásnak van egy helyi kezelő folyamata, amely szabályozott hozzáférést biztosít a távoli felhasználók számára. Ez a réteg egységes és biztonságos felületet nyújt a magasabb szintek felé az erőforrások jellemzőinek és állapotának lekérdezéséhez, illetve az erőforrások nyomon követéséhez és használatához.

Szint	Funkció
Alkalmazási réteg	Alkalmazások, amelyek a kezelt erőforrásokat a szabályozott módon megosztják
Kollektív szolgáltató réteg	Erőforráscsoportok felfedezése, figyelése, vezérlése; brókerszolgáltatás
Erőforrásréteg	Szabályozott és biztonságos hozzáférés konkrét erőforrásokhoz
Szerkezeti réteg	Fizikai erőforrások: számítógépek, tárolók, hálózatok, érzékelők, programok és adatok

8.50. ábra. *A grid szintjei*

A következő a **kollektív szolgáltató réteg**, amely erőforráscsoportokat kezel. Egy funkciója az erőforrás-felfedezés, amelynek segítségével a felhasználó szabad számítási kapacitást, lemezterületet vagy meghatározott adatokat kereshet meg. Ez a réteg katalógusok vagy más adatbázisok alapján adja meg a kívánt információkat. Egyfajta brókerszolgáltatást is nyújthat, amelynek során a szolgáltatásokat nyújtók és az azokat igénybe vevők egymáshoz rendelése megtörténik, közben az esetleg a szükségesnél kisebb mennyiségben rendelkezésre álló erőforrásokat a versengő felhasználók között fel kell osztani. A kollektív szolgáltatások rétege felelős még az adatok többszörözéséért, új erőforrások és felhasználók beléptetéséért a gridbe, az elszámolási információkért, illetve a hozzáférést biztosító jogok adatbázisainak kezeléséért.

Még eggyel feljebb az **alkalmazási réteg** helyezkedik el, ahol a felhasználói alkalmazások találhatók. Az alsóbb rétegekre támaszkodva jogokat szereznek az erőforrásokhoz való hozzáféréshez, felhasználási kéréseket nyújtanak be, nyomon követik a kérések teljesítésének alakulását, kezelik a hibákat és közlik az eredményeket a felhasználóval.

A biztonság kulcsfontosságú a grid sikere szempontjából. Az erőforrás-tulajdonosok szinte mindig szoros felügyelet alatt akarják tudni az erőforrásaikat, és meg akarják határozni, hogy ki mennyit és mikor kap belőlük. Megfelelő biztonság nélkül egyetlen szervezet sem adná az erőforrásait a gridhez. Másfelől, ha a felhasználónak azonosítóra és jelszóra lenne szüksége minden olyan számítógépen, amelyhez hozzá akar férni, akkor a grid használata elviselhetetlenül körülményes lenne. Következésképpen a gridhez egy olyan biztonsági modellt kell kidolgozni, amely figyelembe veszi a fenti szempontokat.

A biztonsági modell alaptulajdonsága az egyszeri feliratkozás. A grid használatának első lépése azonosítás után egy jogosítvány megszerzése, amely egy digitálisan aláírt dokumentum arról, hogy kinek a nevében történik a munkavégzés. A jogosítvány továbbadható, így amikor egy számítási feladat részekre bomlik, a leszármazott folyamatok is azonosíthatók lesznek. Ha a jogosítványt egy távoli gépen akarnak használni, akkor azt le kell képezni a helyi biztonsági mechanizmusokra. A UNIX-gépeken például a felhasználóknak 16 bites azonosítójuk van, de más rendszerek más módszereket használnak. Végül, a gridben szükség van olyan mechanizmusokra, amelyek segítségével a hozzáférési jogok megadhatók, karbantarthatók és módosíthatók.

A különböző szervezetek és gépek közötti együttműködés eléréséhez szabványokra van szükség, mind a szolgáltatások, mind az elérésüket lehetővé tevő protokollok szintjén. A griddel foglalkozó közösség által létrehozott szervezet, a Global Grid Forum kezeli a szabványosítási folyamatot. Elkészítettek egy **OGSA (Open Grid Services Architecture, nyílt grid szolgáltatások architektúrája)** nevű keretrendszert, hogy a különböző, fejlesztés alatt álló szabványokat összefogja. Ahol csak lehetséges, a javasolt szabványok felhasználják a már létező szabvá-

nyokat, például a WSDL-t (Web Services Definition Language) az OGSA-szolgáltatások leírására. A szabványosítás alatt álló szolgáltatások a következő nyolc általános kategóriába esnek, de egyértelmű, hogy ezek később még újabbakkal bővülnek.

1. Infrastrukturális szolgáltatások (erőforrások közötti kommunikációt tesz lehetővé).
2. Erőforrás-kezelő szolgáltatások (erőforrások lefoglalása és felhasználása).
3. Adatszolgáltatások (adatok mozgatása és többszörözése a felhasználás helyén).
4. Kontextusszolgáltatások (igényelt erőforrások és használati jogok leírása).
5. Információs szolgáltatások (információ az erőforrások rendelkezésre állásáról).
6. Önkezelő szolgáltatások (megadott minőségű szolgáltatás támogatása).
7. Biztonsági szolgáltatások (biztonsági előírások betartatása).
8. Végrehajtás-kezelési szolgáltatások (munkavégzés kezelése).

Sokkal többet lehetne még mondani a gridről, de hely hiányában nem folytathatjuk ezt a témát. További információ található (Berman és társai, 2003; Foster és Kesselman, 2003; Foster és társai, 2002).

8.6. Összefoglalás

A számítógépek sebességét az órajel-frekvencia emelésével – hőelvezetési és egyéb problémák miatt – egyre nehezebb növelni. Ehelyett a tervezők a párhuzamossággal próbálnak gyorsulást elérni. A párhuzamosság több különböző szinten is bevezethető, a legalsó szintek szorosan kapcsolt feldolgozó elemeitől kezdve a legfelső szinteken lazán kapcsolt elemekig.

Legalul a lapkaszintű párhuzamosság azt jelenti, hogy egy lapkán belül történnek egyszerre az események. Ennek egyik formája az utasításszintű párhuzamosság, amikor egy utasítás vagy utasítások sorozata több funkcionális egységet kihasználva egymással párhuzamosan végrehajtható műveleteket kezdeményez. A lapkaszintű párhuzamosság másik formája a többszálúság, amikor a processzor több szál között váltogatva hajt végre utasításokat, ezáltal egy virtuális multiprocesszort hozva létre. A lapkaszintű párhuzamosság harmadik formája az egylapkás multiprocesszor, amelyben kettő vagy több, egyszerre működni képes magot helyeznek el a lapkán.

Egy szinttel feljebb a koprocesszorokat találjuk; ezek tipikusan külön kártyán helyezkednek el, és a rendszer feldolgozási sebességét speciális funkciókat ellátva növelhetik, mint például a hálózati protokollok vagy a multimédia területén. A kiegészítő processzorok speciális feladatokat vállalnak át a fő processzortól, amely az így felszabadult időben mással foglalkozhat.

A következő szinten a közös memóriás multiprocesszorokat találjuk. Ezek a rendszerek kettő vagy több teljes értékű CPU-t tartalmaznak, amelyek közösen használnak egy mindannyiuk számára elérhető memóriát. Az UMA-multiprocesszorok közös (szimatoló) sínen vagy crossbar switch-en, esetleg többszintű switch-hálózaton keresztül kommunikálnak. Közös tulajdonságuk, hogy minden memóriacellát egységesen ugyanannyi idő alatt érhetnek el. Ezzel szemben a NUMA-multiprocesszorok ugyan szintén egy közös címterületet biztosítanak a folyamatok számára, de itt a távoli hozzáférések jelentősen több időt igényelnek, mint a helyiek. Végül még egy variáció a COMA-multiprocesszorok, amelyekben a gyorsítósorok igény szerint mozognak a gépben, és az előzőkkel ellentétben nincs tényleges tulajdonosuk, azaz egy olyan hely, ahová tartoznának.

A multiszámítógépek olyan rendszerek, amelyekben a CPU-k nem osztoznak a memórián. Mindegyiknek külön memóriája van, egymással üzenetküldés segítségével kommunikálnak. Az MPP-k egyedi kommunikációs hálózattal rendelkező nagy multiszámítógépek, mint például az IBM által gyártott BlueGene/L. A klaszterek kereskedelmi forgalomban kapható elemekből felépített egyszerűbb rendszerek; ilyen adja a Google hátterét is.

A multiszámítógépeket gyakran üzenetküldést használó szoftvercsomagokkal együtt használják; ilyen például az MPI. Egy alternatív lehetőség az alkalmazásszintű közös memória, mint amilyen egy lapozásos DSM-rendszer, a Linda-adategységtér, az Orca- vagy Globe-objektumok. A DSM lapszinten szimulálja a közös memóriát, emiatt hasonlít a NUMA-rendszerhez, de a távoli eléréseknél sokkal nagyobbak a késések.

Végül a legfelső szinten a leglazábban kapcsolt a grid. Ezek olyan rendszerek, amelyekben egész szervezetek vannak összekapcsolva az interneten keresztül, hogy számítási kapacitásukat, adataikat és más erőforrásaikat megosszák.

8.7. Feladatok

1. A Pentium utasítása akár 17 bájt hosszúságú is lehet. VLIW CPU-e a Pentium?
2. Mit kapunk eredményül, ha a 96, –9, 300 és 256 értékekre vágást végzünk a 0–255 vágási tartományban?
3. Megengedettek-e a következő TriMedia utasítások? Ha nem, miért nem?
   a) Egész összeadás, egész kivonás, betöltés, lebegőpontos összeadás, közvetlen betöltés
   b) Egész kivonás, egész szorzás, közvetlen betöltés, léptetés, léptetés
   c) Közvetlen betöltés, lebegőpontos összeadás, lebegőpontos szorzás, elágazás, közvetlen betöltés
4. A 8.7. (d)–(e) ábra 12 cikluson keresztül mutatja az utasításokat. Mindkettőre határozzuk meg, mi történik a következő 3 ciklusban.
5. Egy bizonyos CPU-nál összesen k ciklust igényel egy olyan utasítás, amelyik az adatot az első szintű gyorsítótárban nem, de a második szintűben megtalálja. Ha finom szemcsézettségű többszálúságot alkalmazunk az első szintű gyorsítótár hiányainak az elfedésére, akkor hány szál egyidejű futtatására van szükség az üres ciklusok kiküszöbölésére?

6. Egy reggelen egy méhkaptárban a méhkirálynő összehívja a dolgozókat és azt az utasítást adja, hogy az aznapi feladat körömvirágnektár gyűjtése lesz. A dolgozók ezután minden irányba szétrepülnek körömvirágot keresni. Ez SIMD- vagy MIMD-rendszer?
7. A memória konzisztencia modellek tárgyalása során azt mondtuk, hogy a konzisztenciamodell egyfajta szerződés a szoftver és a memória között. Miért van szükség ilyen szerződésre?
8. Tekintsünk egy megosztott sínt használó multiprocesszort. Mi történik, ha két processzor pontosan ugyanabban a pillanatban próbál meg hozzáférni a globális memóriához?
9. Tegyük fel, hogy technikai okok miatt egy szimatoló gyorsítótár csak a címvonalakat tudja figyelni, az adatvonalakat nem. Befolyásolná-e ez a változás az írásátcresztő protokoll működését?
10. Egy gyorsítótár nélküli, sínalapú multiprocesszoros rendszer egyszerű modelljeként tegyük fel, hogy minden négy utasításból egy fordul a memóriához, és hogy egy memóriahivatkozás lefoglalja a sínt az egész utasítás időtartamára. Ha a sín foglalt, a további kéréseket kiadó CPU-k egy FIFO várakozósorba kerülnek. Mennyivel lesz gyorsabb egy 64 CPU-s rendszer, mint egy 1 CPU-s?
11. A MESI gyorsítótár-koherencia protokollnak négy állapota van. Más késleltetett írású gyorsítótár-koherencia protokolloknak csak három állapota van. A négy MESI-állapot közül melyiket lehetne feláldozni, és mi lenne a döntések következménye? Ha csak három állapotot választhatnánk, melyek lennének azok?
12. Van-e olyan helyzet a MESI gyorsítótár-koherencia protokollnál, amikor egy sor benne van a gyorsítótárban, de mégis síntranzakciót igényel? Ha igen, mi az a helyzet?
13. Tegyük fel, hogy n CPU van egy közös sínen. Legyen p annak a valószínűsége, hogy bármelyik CPU használni akarja a sínt egy adott ciklusban. Mi a valószínűsége, hogy:
    a) A sín tétlen (0 kérés).
    b) Pontosan egy kérés érkezik.
    c) Egynél több kérés érkezik.
14. Hány crossbar kapcsoló van egy teljesen kiépített Sun Fire E25K gépben?
15. Tegyük fel, hogy az omega hálózat 2A és 3B kapcsolója közötti vezeték elszakad. Kik között szakad meg a kapcsolat?
16. Világos, hogy a forró pontok (sűrűn hivatkozott memóriarekeszek) súlyos problémát jelentenek a többszintű kapcsoló hálózatok számára. Ugyanez-e a helyzet a sínalapú rendszereknél?
17. Egy omega kapcsoló hálózat 4096 darab, 60 ns ciklusidejű RISC CPU-t kapcsol 4096 darab végtelenül gyors memóriamodulhoz. A kapcsolóelemek mindegyikének 5 ns a késleltetése. Hány késleltető ütem kell egy LOAD utasításhoz?
18. Tekintsünk egy gépet, amely a 8.29. ábrán látható omega kapcsoló hálózatot használja. Tételezzük fel, hogy az i. processzor programja és veremtára az i. memóriamodulban van elhelyezve. Tegyünk javaslatot a topológia olyan kismértékű módosítására, amely nagy különbséget eredményez a teljesítményben (az IBM RP3 és a BBN Butterfly ezt a módosított topológiát használja). Mi az új topológia hátránya az eredetihez képest?
19. Egy bizonyos NUMA-multiprocesszorban a lokális memóriahivatkozások időigénye 20 ns, a távoli hivatkozásoké pedig 120 ns. Egy program végrehajtása alatt összesen N memóriahivatkozást végez, amelyek 1%-a vonatkozik a P lapra. Ez a lap először távoli, és C ns-ig tart helyi másolatot készíteni róla. Milyen körülmények között érdemes másolatot készíteni róla, ha más processzor nem használja sokat?
20. Tekintsünk egy, a 8.31. ábrán láthatóhoz hasonló CC-NUMA-multiprocesszort, ez esetben 512 feldolgozóegységgel és mindegyikben 8 MB memóriával. Ha a gyorsítósorok 64 bájtosak, százalékosan mennyivel nagyobb katalógusra van szükség? A feldolgozóegységek számának növelése hogyan befolyásolja a katalógus méretét? Kisebb lesz, ugyanakkora vagy nagyobb?
21. Számítsuk ki a 8.35. ábrán látható topológiák mindegyikének átmérőjét.
22. A 8.35. ábrán látható topológiák mindegyikére határozzuk meg a hibatűrés mértékét. A hibatűrés az a legnagyobb szám, ahány élet törölhetünk anélkül, hogy a hálózat két diszjunkt részre esne.
23. Tekintsük a 8.35. (f) ábra kettős tórusz topológiáját, de $k \times k$ méretben. Mi a hálózat átmérője? *Tipp:* vizsgáljuk páros és páratlan k-ra külön.
24. Tekintsünk egy hálózatot $8 \times 8 \times 8$ formátumú kocka topológiával. Minden kapcsolat teljes duplex, 1 GB/s sávszélességgel. Mekkora a hálózat kettévágott sávszélessége?
25. Amdahl törvénye szerint a párhuzamos számítógéppel elérhető sebességnövekedés korlátozott. Határozzuk meg f függvényében a legnagyobb elérhető sebességnövekedést, ahogy a CPU-k száma végtelenhez közelít. Mik a következményei ennek a határértéknek az $f = 0{,}1$ értékre?
26. A 8.49. ábra bemutatja, hogy a sín nem skálázható, a rács viszont igen. Tegyük fel, hogy minden sín, illetve kapcsolat sávszélessége b, és számítsuk ki a CPU-nkénti átlagos sávszélességet mind a négy esetre. Ezután bővítsük a rendszert 64 CPU-ra, és végezzük el újra a számítást. Mi a határérték, ha a CPU-k száma végtelenhez közelít?
27. A könyvben a send háromféle variációját tárgyaltuk: szinkron, blokkoló és nemblokkoló. Adjunk egy negyedik módszert, amely a blokkoló send-hez hasonlít, de kissé eltérő tulajdonságai vannak. Határozzuk meg az új módszer a blokkoló send-hez képesti előnyeit és hátrányait.
28. Tekintsünk egy Ethernethez hasonló üzenetszórásos hálózattal rendelkező multiszámítógépet. Miért lényeges az olvasó és az írási műveletek számának aránya? (Az olvasó műveletek nem változtatják meg a belső állapotot tároló változókat, az írási műveletek azonban igen.)

9. Ajánlott olvasmányok és irodalomjegyzék

Az előző nyolc fejezetben nagyon sok témát részleteztünk különböző mélységben. Ez a fejezet arra szolgál, hogy segítse azokat az olvasóinkat, akik szeretnék elmélyíteni a számítógép-architektúrákban eddig megszerzett tudásukat. A 9.1. alfejezet az ajánlott irodalmakat sorolja fel a könyv fejezeteihez igazodó csoportosításban. A 9.2. alfejezet a könyvünkben hivatkozott összes könyv és cikk jegyzékét tartalmazza szerzők szerinti ábécérendben.

9.1. Javasolt további olvasmányok

9.1.1. Bevezető és általános művek

Borkar: *Getting Gigascale Chips.*
Moore szabálya várhatóan még legalább egy évtizedig érvényben marad, és akár a milliárdtranzisztoros lapkák is megjelenhetnek. Ezek a lapkák egyaránt jelentenek kihívásokat és lehetőségeket is. Ebben a közleményben az Intel egyik vezető kutatója tárgyalja egyebek között az olyan jövőbeli kihívásokat, mint a hőenergia-elvezetés, valamint az egyre kisebb és sűrűbben elhelyezett, ennek megfelelően nagyobb ellenállású és kapacitású vezetékek problémája. Úgy véli, hogy a fejlődés nem csak egyszerűen a magasabb órajelsebességben, hanem a többszálúságban, a többprocesszoros lapkákban és jobb memóriaszervezésben rejlik.

Colwell: *The Pentium Chronicles.*
Robert Colwell vezette a Pentium processzor tervezői csapatát. Ebben a könyvben a lapka mögött megbúvó emberekről, szenvedélyekről és politikai megfontolásokról ír.

Hamacher és társai: *Computer Organization.* 5. kiadás
Hagyományos tankönyv a számítógépek, azon belül a központi egység, memória, B/K, aritmetikai egységek és perifériák felépítéséről. Fő példái a 68 000-es és a PowerPC.

Heath: *Embedded Systems Design.*
Napjainkban gyakorlatilag minden 50 dollárnál drágább elektromos berendezésben található egy számítógép. Ezek a beágyazott rendszerek képezik e könyv tárgyát. A beágyazott rendszerek, memóriák és perifériák alapjaitól indulva jut el az interfészek, valós idejű operációs rendszerek, szoftver és hibakeresés témakörökig.

Hennessy–Patterson: *Computer Architecture: A Quantitative Approach.* 3. kiadás
Ez a vaskos egyetemi tankönyv mélyremenő részletekig vizsgálja a processzor és memóriájának tervezési elveit. A hangsúlyt a nagy teljesítményre, kiváltképp a párhuzamosság és a csővezetékek kiaknázásával történő elérésére fekteti. Ha mindent tudni szeretne a nagy teljesítményű CPU-k tervezéséről, ebben a könyvben keresse.

Null–Lobur: *The Essentials of Computer Organization and Architecture.*
Egy újabb tankönyv a számítógép-architektúrákról, amely könyvünk számos témakörét tárgyalja, de kevésbé részletesen.

Patterson–Hennessy: *Computer Organization and Design.* 3. kiadás
Ez a kiadás már nincs 1000 oldalas, mint a 2. kiadás, mert a szöveg nagy része átkerült a CD-ROM-mellékletre. A könyvben megtartott részek lefedik a számítógép-architektúra számos aspektusát, beleértve az aritmetikai egységet, teljesítményt, adatutat, csővezetékeket, memóriát, perifériákat és klasztereket. Bár a Pentium 4 is több helyütt előfordul, az elveket főként a MIPS processzoron keresztül magyarázzák. A Hennessy tervezte MIPS processzor volt az első kereskedelmi RISC gép.

Price: *A History of Calculating Machines.*
Bár a modern számítógépek kora a XIX. században Babbage-dzsel kezdődött, az emberek a civilizáció hajnala óta számolnak. Ez a gazdagon illusztrált írás nyomon követi a számolás, a matematika, a naptárak és a számítás fejlődését Kr. e. 3000-től a XX. század kezdetéig.

Slater: *Portraits in Silicon.*
Miért nem adta be Dennis Ritchie a PhD-disszertációját a Harvardra? Miért lett Steve Jobs vegetáriánus? A válaszokat ebben a lebilincselően érdekes könyvben találjuk, amely 34 ember rövid életrajzát tartalmazza, azokét, akik a számítógépipart formálták, Charles Babbage-től Donald Knuth-ig.

Stallings: *Computer Organization and Architecture.* 6. kiadás
Általános leírás a számítógép-architektúrákról. Könyvünk néhány témáját a Stallings-könyvben is megtaláljuk.

Wilkes: *Computers Then and Now.*
Maurice Wilkes-nek, az úttörő számítógép-tervezőnek, valamint a mikroprogramozás feltalálójának személyes története a számítógépekről 1946-tól 1968-ig. Elmeséli a kezdeti harcokat, ami az űrkadétok (space cadets) – akik az automatikus programozás hívei voltak (elő-FORTRAN fordítók) –, valamint a hagyományos gondolkodásúk – akik jobban szerettek nyolcas számrendszerben programozni – között dúlt.

9.1.2. Számítógéprendszerek felépítése

Buchanan–Wilson: *Advanced PC Architecture.*
Kissé rendezetlenül ugyan, de a könyv a PC alkatrészeinek széles sorát bemutatja, egyebek mellett a processzorokat, síneket (PCI, SCSI és USB) és portokat (játék, párhuzamos és soros).

Ng: *Advances in Disk Technology: Performance Issues.*
Egyesek már legalább 20 éve jövendölik a mágneslemezek korának végét, de azok még mindig velünk vannak. E cikk szerint a mágneslemez-technológia olyan rohamosan fejlődik, hogy valószínűleg még évekig fogjuk használni őket.

Messmer: *The Indispensable PC Hardware Book.* 4. kiadás
Ez a könyv a maga 1296 oldalával (amely 37 fejezetre és 7 függelékre oszlik) lehet, hogy nélkülözhetetlen, de az is lehet, hogy nem; egy biztos: vastag. Csaknem minden megtalálható benne kimerítő részletességgel, amit tudni kell a 80x86-os processzorokról, memóriákról, sínekről, segédlapkákról és perifériákról. Ha olvasta és megemésztette Norton és Goodman könyvét (lásd alább), és a technikai részletek eggyel magasabb szintjére szeretne eljutni, akkor itt kell kezdenie.

Norton Goodman: *Inside the PC.* 8. kiadás
A legtöbb PC hardverről írt könyv villamosmérnökök számára készült, így a szoftveres beállítottságúaknak meglehetősen nehéz olvasmány. Ez a könyv azonban más. Szakszerűen, de ennek ellenére közérthető módon magyarázza el a PC hardver működését. Témái között megtalálható a központi egység, memória, sínek, diszkek, megjelenítők, B/K eszközök, mobil PC-k, számítógép-hálózat és még sok más. Ritka és nagyon értékes könyv.

Robinson: *Toward the Age of Smarter Storage.*
A tárolók nagyon sokat fejlődtek a mágnesgyűrűs memóriák és a lyukkártyák kora óta. Ez a rövid írás áttekinti a tárolástechnika múltját, jelenét és a jövőbeli irányzatokat.

Scheible: *A Survey of Storage Options.*
A memóriatechnológia újabb összefoglalása, amely csak a jelenlegi helyzetre koncentrál. Tárgyalásra kerülnek a RAM különböző fajtái, a flash memória, mágnesszalag, merevlemez, hajlékony lemez, CD és DVD.

Stan–Skadron: *Power-Aware Computing.*
A számítógépek kezdenek (szó szerint) túl nagy teljesítményűvé válni. Túl sok energiát fogyasztanak, ami – egyre mobilabb világunkban – egyre komolyabb problémát jelent. Ez a cikk a vendégszerkesztők bevezetője az IEEE Computer Magazinenak a számítástechnika energetikai problémáiról szóló különkiadásához.

Triebel: *The 80386, 80486, and Pentium Processor.*
Kissé nehéz ezt a könyvet besorolni, mert hardverrel, szoftverrel és interfészekkel is foglalkozik. Mivel a szerző az Intel munkatársa, hívjuk inkább hardverkönyvnek. Mindent elmond a processzorokról, memóriákról, B/K eszközökről, és a 80x86 processzorok illesztéseiről, valamint azok assembly nyelvű programozásáról. Mindössze 915 oldalon tartalmazza mindazt, amit Messmer könyve, mivel az oldalak nagyobbak.

9.1.3. Digitális logika szintje

Floyd: *Digital Fundamentals.* 8. kiadás
A hardverbeállítottságú olvasóknak, akik többet szeretnének tudni a digitális logika szintjéről, ez a hatalmas, bőségesen illusztrált négyszínű könyv valódi gyöngyszem. A fejezetek sok egyéb mellett lefedik a kombinációs logikát, programozható logikájú eszközöket, flip-flopokat, léptető regisztereket, memóriákat, interfészeket.

Mano–Kime: *Logic and Computer Design Fundamentals.* 3. kiadás
Bár ez a könyv nem olyan elegáns kivitelű, mint a Floyd-könyv, a digitális logika szintjéhez szintén jó információforrás. Lefedi a kombinációs és szekvenciális áramköröket, regisztereket, memóriákat, a központi egység és B/K tervezését.

máj.hew–Krishnan: *PCI Express and Advanced Switching.*
A PCI Express várhatóan a közeljövőben felváltja a PCI síneket. Ez a tanulmány bemutatja a PCI Express rétegeit, folyamatvezérlését, a virtuális csatornákat, kapcsolást és útvonalválasztást.

Mazidi–Mazidi: *The 80x86 IBM PC and Compatible Computers.* 4. kiadás
Mindazok számára, akik szeretnék megérteni a PC-kben található összes lapkát, ez a könyv teljes fejezeteket szán a legfontosabb lapkáknak, és hasonlóan bőséges információt tartalmaz az IBM PC-hardverről és az assembly nyelvű programozásról is.

Roth: *Fundamentals of Logic Design.*
A digitális logikai tervezés alapjairól olvashatunk ebben a tankönyvben a Boole-algebrától a kapukon, számlálókon, összeadókon, flip-flopokon át egyéb kombinációs és szekvenciális áramkörökig.

9.1.4. A mikroarchitektúra szintje

Burger–Goodman: *Billion-Transistor Architectures: There and Back Again.*
Tegyük fel, hogy 1997-ben valaki adott volna nekünk egy milliárd tranzisztort, és azt mondta volna: Tervezz egy lapkát! Milyen mikroarchitektúrát terveztünk volna? 1997 szeptemberében hét komoly architektúrakutató, akiknek feltették ezt a kérdést, közölte nézetét az IEEE Computer Magazine oldalain. Hét évvel később összehasonlították jóslataikat az aktuális állapottal.

Handy: *The Cache Memory Book.* 2. kiadás
A gyorsítótár tervezése önmagában olyan fontos, hogy teljes könyveket szentelnek neki. Ez a könyv tárgyalja a logikai és fizikai gyorsítótárak, a write-through (írásáteresztés) és write-back (késleltetett írás, visszaírás) vezérelvek, a közös és osztott gyorsítótárak összehasonlítását, valamint a szoftver kérdéseit is. Külön fejezet szól a multiprocesszorok gyorsítótárainak összefüggéséről is.

Johnson: *Superscalar Microprocessor Design.*
Azoknak az olvasóknak, akik a szuperskaláris CPU tervezésének részletei iránt érdeklődnek, jó kiindulási alap ez a könyv. Tartalmazza az utasítások betöltését és dekódolását, a sorrenden kívüli végrehajtást, a regiszterátnevezést, állomások lefoglalását, elágazások előrejelzését és még sok egyéb technikát.

Shriver–Smith: *The Anatomy of a High-Performance Microprocessor.*
Nagyon jó választás egy modern processzort a mikroarchitektúra szintjén tanulmányozni vágyók számára. Részletesen vizsgálja az AMD K6 lapkát, egy Pentium klónt, kiemelve a csővezetékek használatát, az utasítás ütemezést és a teljesítmény optimalizálást.

Sima: *Superscalar Instruction Issue.*
A modern központi egységben egyre fontosabb kérdéskört jelentenek a szuperskaláris utasításkiosztások. Könyvünkben is érintettünk néhány ilyen kérdést, mint például az átnevezést és a spekulatív végrehajtást. Ebben a cikkben ezeket és még számos más kérdést is vizsgálnak.

Wilson: *Challenges and Trends in Processor Design.*
A processzortervezés valóban kihalt? Semmiképpen. Hat menő CPU-tervező, a Sun, Cyrix, Motorola, Mips, Intel és a Digital munkatársai elmondják, hogyan képzelik a CPU-fejlődés irányvonalát a következő években. Élmény lesz ezt 2008-ban olvasni (de most is érdemes).

9.1.5 Az utasításrendszer-architektúra szintje

Antonakos: *The Pentium Microprocessor*
Ennek a könyvnek az első kilenc fejezete azzal foglalkozik, hogy hogyan kell a Pentiumot assembly nyelven programozni. Az utolsó kettő a Pentium-hardvert mutatja be. Számos kódrészletet tartalmaz, valamint foglalkozik a BIOS-szal is.

Ayala: *The 8051 Microcontroller.* 3. kiadás
A 8051-es programozása iránt érdeklődők számára ez a könyv értelmes kiindulópont.

Bryant–O'Hallaron: *Computer Systems: A Programmer's Perspective.*
A könyv kissé szervezetlen ugyan, de az ISA-szint nagy területét fedi le, beleértve az aritmetikát, a különféle utasításokat, a vezérlési szerkezeteket és a programoptimalizálás témakörét.

Paul: *SPARC Architecture, Assembly Language, Programming, and C.*
Csodák csodájára itt egy könyv az assembly nyelvű programozásról, amely nem az Intel 80x86 sorozatról szól. Ehelyett a SPARC és programozása a téma.

Weaver–Germond: *The SPARC Architecture Manual.*
A számítógépipar nemzetközivé válásával egyre fontosabbak a szabványok, tehát nagyon lényeges, hogy megismerkedjünk velük. Ez a SPARC 9-es verziójának definícióját tartalmazza, és nagyon jó képet ad arról, hogyan is néz ki egy szabvány, továbbá a 64 bites SPARC-ok működését is nagyon jól mutatja.

9.1.6. Az operációs rendszer gép szintje

Hart: *Win32 System Programming.*
A Windowsról szóló sok más könyvvel ellentétben ez a mű nem ragad le a grafikus felhasználói interfésznél, sőt nem is foglalkozik vele. Középpontban a Windows felkínálta rendszerhívások állnak, valamint használatuk módja a fájlelérésben, memóriakezelésben, processzusok kezelésében, processzusok közti kommunikációban, fonalak készítésében, B/K-ben és egyéb témákban.

Jacob–Mudge: *Virtual Memory: Issues of Implementation.*
Ha egy jó és modern bevezetésre van szüksége a virtuális memóriák világába, akkor itt keresse. Elmagyarázza a különböző laptáblákat és TLB-struktúrákat, ezen túlmenően a MIPS, a PowerPC és a Pentium processzorokon keresztül illusztrálja is az ötleteket.

McKusick és társai: *The Design and Implementation of the 4.4 BSD Operating System.*
A UNIX-ról szóló könyvek többségétől eltérően, a könyv a négy szerzőnek egy USENIX-konferencián készült fotójával kezdődik, hárman közülük írták a 4.4-es

BSD nagy részét, így kiválóan alkalmasak arra, hogy annak belső működését elmagyarázzák. Ez a könyv lefedi a rendszerhívásokat, folyamatokat, B/K műveleteket, és tartalmaz egy különösen értékes részt a hálózatokról.

Ritchie–Thompson: *The UNIX Time-Sharing System.*
Ez az eredeti cikk a UNIX-ról. Még most is érdemes olvasgatni. Ebből a kis magból nőtt ki egy nagy operációs rendszer.

Russinovich–Solomon: *Inside Microsoft Windows.* 4. kiadás
Azok számára, akik a Windows belső működésére kíváncsiak, ez a legjobb választás. A témakörök között találhatók egyebek között a rendszer architektúrája, a rendszer mechanizmusa, processzusok, fonalak, memóriakezelés, biztonsági kérdések, B/K lehetőségek, gyorsítók és a fájlrendszer. A nagyon alapos könyv informatika szakos hallgatók és informatikai szakemberek számára készült.

Tanenbaum–Woodhull: *Operating Systems: Design and Implementation.* 2. kiadás
Az operációs rendszerekről szóló könyvek többségével ellentétben, amelyek csak az elmélettel foglalkoznak, ez a könyv az összes lényeges elmélet tárgyalása mellett illusztrálja is azokat az IBM PC-n és egyéb számítógépeken futtatható MINIX, egy UNIX-szerű operációs rendszer tényleges kódjain keresztül. A megjegyzésekkel bőségesen ellátott forráskód megtalálható a függelékben.

9.1.7. Assembly nyelv szintje

Levine: *Linkers and Loaders.*
Ha szerkesztőkkel és betöltőkkel foglalkozik, és kedvét leli a különféle tárgykódformátumokban, a dinamikus és statikus szerkesztés közötti különbségekben, valamint a különféle könyvtárformátumokban, ez a könyv Önnek való.

Saloman: *Assemblers and Loaders.*
Itt minden megtalálható, ami érdekes lehet az egy- és kétmenetes assembler, továbbá a szerkesztők és betöltők működéséről. A könyv tárgyalja továbbá a makrókat és a feltételes assembly fordítást.

9.1.8. Párhuzamos számítógép-architektúrák

Adve–Gharachorloo: *Shared Memory Consistency Models: A Tutorial.*
Nagyon sok modern számítógép – különösen a multiprocesszorok – csak a szekvenciálisan konzisztensnél gyengébb memóriamodellt támogatja. Ez az összefoglaló tárgyalja a különböző modelleket és elmagyarázza a működésüket. Ezen kívül kimond és cáfol számos mítoszt a gyengén konzisztens memóriákról.

Comer: *Network Systems Design.*
A könyv első része a hagyományos hálózati csomagfeldolgozásról szól, a második rész azonban bemutatja a hálózati processzorokat és leírja azok célját, architektúráját és tervezési kompromisszumait. A harmadik rész esettanulmányként az Agere hálózati processzorral foglalkozik.

Dally–Towles: *Principles and Practices of Interconnection Networks.*
Az összekapcsolódó hálózatok iránt érdeklődőknek ez a megfelelő olvasmány. Egy kis topológiai bevezető után tárgyalja a pillangóhálózatokat, a tóruszhálózatokat és a nemblokkoló hálózatokat is. Az ezután következő jó néhány fejezet az útvonalválasztásról, folyamatvezérlésről, pufferelésről, holtpontkezelésről és kapcsolódó kérdéskörökről szól.

Dongarra és társai: *The Sourcebook of Parallel Computing.*
A multiprocesszorok és klaszterek programozása lényegesen eltér az egyprocesszoros rendszerek programozásától. Ebben a könyvben a párhuzamos programozás hét vezető szakértője tárgyalja a párhuzamos programozás különféle aspektusait, beleértve a párhuzamos architektúrákat, szoftvertechnológiákat, párhuzamos algoritmusokat és néhány alkalmazást is.

Hill: *Multiprocessors Should Support Simple Memory-Consistency Models.*
A gyengített memóriaszemantika a multiprocesszorok memóriatervezésének nagyon aktuális és vitatott témája. A gyengébb modellek ugyan megengednek olyan hardveroptimalizációkat, mint a nem sorrendben történő memóriahivatkozások, a programozást azonban nehézkesebbé teszik. Ebben a cikkben a szerző a memóriakonzisztencia számos fontos kérdését tárgyalja, majd arra a következtetésre jut, hogy a gyengített memóriával több a baj, mint a haszon.

Hwang–Xu: *Scalable Parallel Computing.*
A hardver és a szoftver együttes tárgyalásával a szerzőknek sikerült a párhuzamos számításról egy minden részletre kitérő, mégis olvasmányos összefoglalást készíteniük. A témák között megtalálhatók az UMA és NUMA multiprocesszorok, MPP-k és COW-k, az üzenettovábbítás és az adatpárhuzamos programozás.

Lawton: *Will Network Processors Units Live up to Their Promise?*
Bár a hálózati processzorok gyorsabb csomagfeldolgozást ígérnek, a sikerük nem garantált. Ebben a cikkben a szerző áttekinti a technológiát és néhány olyan tényezőt, amelyek meghatározók lehetnek annak sikerében vagy bukásában.

McKnight és társai: *Wireless Grids.*
Alig születtek meg a rácshálózatok máris egy új generációs rácshálózat (a vezeték nélküli rács) van feltűnőben. A hagyományos rácshálózatokhoz hasonlóan ezek is a szervezetek közötti erőforrás-megosztást célozzák meg, hogy virtuális szervezeteket hozzanak létre, azonban ezek vezeték nélküli technológiát alkalmaznak,

hogy a vándorló felhasználók számára is elérhetővé tegyék az erőforrásokat. A két ezt követő cikk is vezeték nélküli rácshálózatokról szól.

Pfister: *In Search of Clusters.* 2. kiadás
Annak ellenére, hogy a klaszter fogalmát egészen a 72. oldalig nem definiálja (ez együttműködő számítógépek csoportját jelenti), a klaszter kétségtelenül magába foglalja az összes szokásos többprocesszoros és többszámítógépes rendszert. A könyv részletesen tanulmányozza ezek hardverét, szoftverét, teljesítményét és elérhetőségét. Meg kell jegyeznünk azonban, hogy a szerző eleinte szórakoztató aranyos stílusa az 500. oldalra fokozatosan elveszti újszerűségét.

Snir és társai: *MPI: The Complete Reference Manual.*
A cím mindent elmond. Ha meg akar tanulni MPI-ben programozni, ezt a könyvet keresse. A könyv egyebek között tárgyalja a pont-pont közötti és a kollektív kommunikációt, kommunikátorokat, környezetmenedzsmentet és a profilok készítését.

Stenstrom és társai: *Trends in Shared Memory Multiprocessing.*
Bár a megosztott memóriás multiprocesszorokról gyakran azt gondolják, hogy ezek nagymennyiségű tudományos számításokra szolgáló szuperszámítógépek, a valóságban ez csak egy nagyon vékony rétege ennek a piacnak. Ebben a cikkben a szerzők bemutatják, hogy valójában hol is van e gépek igazi piaca, és ennek milyen kihatása van architektúrájukra.

Ungerer és társai: *A Survey of Processors with Explicit Multithreading.*
A cikk elmagyarázza a többszálúság minden fontosabb fajtájának — finoman tagolt, durván tagolt és egyidejű — működését, és számos példát ad az adott technikát használó tudományos és kereskedelmi számítógépek közül.

Wolf: *The Future of Multiprocessor Systems-on-Chips.*
Három jelenlegi lapkára integrált rendszer terveinek bemutatása után a szerző tovább vizsgálja a jövőbeli rendszerek hardver- és szoftverkihívásait. A hardverproblémák között a valós idejűség, a hőtermelés és hőelvezetés gondja, míg a szoftveroldalon az operációs rendszer kérdései és a lehetséges biztonsági problémák találhatók.

9.1.9. Bináris és lebegőpontos számok

Cody: *Analysis of Proposals for the Floating-Point Standard.*
Évekkel ezelőtt az IEEE megtervezett egy lebegőpontos architektúrát, amely de facto szabvánnyá vált a modern processzorlapkák számára. Cody a szabványosítás folyamata során felmerülő különböző kérdéseket, javaslatokat és ellentmondásokat tárgyalja.

Koren: *Computer Arithmetic Algorithms.*
Az egész könyv az aritmetikáról szól, hangsúlyt fektetve az összeadás, szorzás és osztás gyors algoritmusaira. Mindenkinek melegen ajánljuk, aki úgy gondolja, hogy a számtanról már hatodik osztályos korában mindent megtanult.

IEEE: *Proc. of the n-th Symposium on Computer Arithmetic.*
Más véleményekkel ellentétben az aritmetika élő kutatási terület. Nagyon sok a tudományos cikk, amelyeket aritmetikai specialisták, illetve nekik írtak. Ezen a szimpóziumsorozaton többek között a nagy sebességű összeadás és szorzás fejlődési trendjeit, a VLSI aritmetikai hardvert, a társprocesszorokat, a hibakezelést és a kerekítést mutatták be.

Knuth: *Seminumerical Algorithms.* 3. kiadás
Gazdag anyag a pozicionális számrendszerekről, lebegőpontos aritmetikáról, többszörös pontosságú aritmetikáról és a véletlen számokról. Ez az anyag megkívánja és megérdemli a gondos tanulmányozást.

Wilson: *Floating-Point Survival Kit.*
Ez a szép bevezetés a lebegőpontos számok témakörébe és szabványokba azoknak szól, akik úgy gondolják, hogy a világ 65 535-nél befejeződik. Néhány népszerű és mérvadó lebegőpontos rendszert, mint például a *Linpack*ot is bemutatják.

9.1.10. Assembly nyelvű programozás

Blum: *Professional Assembly Language.*
Útmutató a Pentium assembly nyelvű programozásához profiknak. Mivel a könyv elsősorban szakértőknek készült, feltételezi, hogy az olvasó számítógépén Linux operációs rendszert futtat, és a Linuxos assemblerre és a GNU-eszközökre összpontosít, valamint a Linux rendszerhívásait tárgyalja.

Irvine: *Assembly Language for Intel-Based Computers.* 4. kiadás
A könyv témája az Intel központi egységének assembly nyelven való programozása. Sok egyéb kapcsolódó téma mellett lefedi továbbá a B/K programozás, a makrók, a fájlok, a szerkesztés és a megszakítások kérdéskörét.

9.2. Bibliográfia

ADAMS, M.–DULCHINOS, D.: OpenCable. *IEEE Commun. Magazine*, 39. évf., pp. 98–105., 2001. jún.

ADIGA, N.R. és társai: An Overview of the BlueGene/L Supercomputer. *Proc. Supercomputing 2002*, ACM, pp. 1–22., 2002.

ADVE, S.V.–CHARACHORLOO, K.: Shared Memory Consistency Models: A Tutorial. *IEEE Computer Magazine*, 29. évf., pp. 66–76., 1996. dec.

ADVE, S.V.–HILL, M.: Weak Ordering: A New Definition. *Proc. 17th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 2–14., 1990.

AGERWALA, T.–COCKE, J.: High Performance Reduced Instruction Set Processors. IBM T.J. Watson Research Center Technical Report RC12434, 1987.

ALAMELDEEN, A. R.–Wood, D. A.: Adaptive Cache Compression for High-Performance Processors. *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 212–223., 2004.

ALMASI, G. S. és társai: System Management in the BlueGene/L Supercomputer. *Proc. 17th Int'l Parallel and Distr. Proc. Symp.*, IEEE, 2003a.

ALMASI, G.S. és társai: An Overview Of The BlueGene/L System Software Organization. *Par. Proc. Letters*, 13. évf., pp. 561–574., 2003b. ápr.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., ZWAENEPOEL, W: TreadMarks: Shared Memory Computing on a Network of Workstations. *IEEE Computer Magazine*, 29. évf., pp. 18–28., 1996. febr.

ANDERSON, D.: *Universal Serial Bus System Architecture*. Reading, MA: Addison-Wesley, 1997.

ANDERSON, D., BUDRUK, R.–SHANLEY, T.: *PCI Express System Architecture*. Reading, MA: Addison-Wesley, 2004.

ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A.–the NOW team: A Case for NOW (Networks of Workstations). *IEEE Micro Magazine*, 15. évf., pp. 54–64., 1995. jan.

ANTONAKOS, J.L.: *The Pentium Microprocessor*. Upper Saddle River, NJ: Prentice Hall, 1997.

AUGUST, D. I, CONNORS, D. A., MSHLKE, S. A., SIAS, J. W., CROZIER, K. M., CHENG, B.-C., EATON, P. R., OLANIRAN, Q. B.–HWU, W.-M.: Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. *Proc. 25th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 227–237., 1998.

AYALA, K: *The 8051 Microcontroller*. 3. kiad., Clifton Park, NY: Thomson Delmar Learning, 2004.

BAL, H. E.: *Programming Distributed Systems*. Hemel Hempstead, England: Prentice Hall Int'l, 1991.

BAL, H.E., BHOEDJANG, R., HOFMAN, R, JACOBS, C., LANGENDOEN, K., RUHL, T.–KAASHOEK, M. E: Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16. évf., pp. 1–40., 1998. jan.–febr.

BAL, H. E., KAASHOEK, M. F.–TANENBAUM, A. S.: Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. on Software Engineering*, 18. évf., pp. 190–205., 1992. márc.

BAL, H. E.–TANENBAUM, A.S.: Distributed Programming with Shared Data. *Proc. 1988 Int'l Conf. on Computer Languages*, IEEE, pp. 82–91., 1988.

BARROSO, L. A., DEAN, J., HOLZLE, U.: Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro Magazine*, 23. évf., pp. 22–28., 2003. márc.–ápr.

BECHINI, A., CONTE, T. M.–PRETE, C. A.: Opportunities and Challenges in Embedded Systems. *IEEE Micro Magazine*, 24. évf., pp. 8–9., 2004. júl.–aug.

BENINI, L.–DE MICHELI, G.: Networks on Chips: A New SoC Paradigm. *IEEE Computer Magazine*, 35. évf., pp. 70–78., 2002. jan.

BERMAN, E, FOX, G.–HEY, A.J.G.: *Grid Computing: Making the Global Infrastructure a Reality*. Hoboken, NJ: John Wiley. 2003.

BJORNSON, R. D.: Linda on Distributed Memory Multiprocessors. Ph.D. Thesis, Yale Univ., 1993.

BLUM, R.: *Professional Assembly Language*. New York: Wiley, 2005.

BLUMRICH, M., CHEN, D., CHIU, G., COTEUS, P., GARA, A., GIAMPAPA, M. E., HARING, R. A., HEIDELBERGER, P., HOENICKE, D., KOPCSAY, G. V., OHMACHT, M., STEINMACHER-BUROW, B. D., TAKKEN, T., VRANSAS, P.–LIEBSCH, T.: An Overview of the BlueGene/L System. *IBM J. Research and Devel.*, 49. évf., 2005. márc.–máj.

BORKAR, S.: Getting Gigascale Chips. *Queue*, pp. 26–33., 2003. okt.

BOSE, P.: Computer architecture research: Shifting priorities and newer challenges. *IEEE Micro Magazine*, 24. évf., p. 5., 2004. nov.–dec.

BOUKNIGHT, W. J., DENENBERG, S. A., McINTYRE, D. E., RANDALL, J. M., SAMEH, A. H.–SLOTNICK, D. L.: The Illiac IV System. *Proc. IEEE*, pp. 369–388., 1972. ápr.

BRIGHTWELL, R., CAMP, W., COLE, B., DeBENEDICTIS, E., LELAND, R., TOMPKINS, H–MacCABE: Architectural Specification for Massively Parallel Supercomputers: An Experience and Measurement-Based Approach. *Concurrency and Computation: Practice and Experience*, 17. évf., pp. 1–46., 2005.

BRYANT, R. E.–O'HALLARON, D.: *Computer Systems: A Programmer's Perspective*. Upper Saddle River, NJ: Prentice Hall, 2003.

BUCHANAN, W., WILSON, A.: *Advanced PC Architecture*. Reading, MA: Addison-Wesley, 2001.

BURGER, D.–GOODMAN, J. R.: Billion-Transistor Architectures: There and Back Again. *IEEE Computer Magazine*, 37. évf., pp. 22–28., 2004. márc.

BURKHARDT, H., FRANK, S., KNOBE, B.–ROTHNIE, J.: Overview of the KSR-1 Computer System, Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.

CAIN, H.–LIPASTI, M.: Memory Ordering: A Value-Based Approach. *Proc. 31th Ann. Int'l Symp. on Computer Arch.* ACW. pp. 90–101., 2004.

CALCUTT, D., COWAN, F.–PARCHIZADEH, H.: *8051 Microcontrollers: An Applications Based Introduction*. Oxford: Newnes, 2004.

CARRIERO, N.–GELERNTER, D.: Linda in Context. *Commun. of the ACM*, 32. évf., pp. 444–458., 1989. ápr.

CHARLESWORTH, A.: The Sun Fireplane Interconnect. *IEEE Micro Magazine*, 22. évf., pp. 36–45., 2002. jan.–febr.

CHARLESWORTH, A.: The Sun Fireplane Interconnect. *Proc. Conf: on High Perf. Networking and Computing*, ACM, 2001.

CHARLESWORTH, A., PHELPS, A., WILLIAMS, R.–GILBERT, G.: Gigaplane-XB: Extending the Ultra Enterprise Family. *Proc. Hot Interconnects V*, IEEE, 1998.

CHEN, L., DROPSHO, S., ALBONESI, D.H.: Dynamic Data Dependence Tracking and its Application to Branch Prediction. *Proc. Ninth Int'l Symp. on High-Performance Computer Arch.*, IEEE, pp. 65–78., 2003.

CHOU, Y., FAHS, B., ABRAHAM, S.: Microarchitecture Optimizations for Exploiting Memory-Level Parallelism.|*(CQ *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM. pp. 76–77., 2004.

CLAASEN, T. A. C. M.: System on a Chip: Changing IC Design Today and in the Future. *IEEE Micro Magazine*, 23. évf., pp. 20–26., 2003. máj.–jún.

CODY, W. J.: Analysis of Proposals for the Floating-Point Standard. *IEEE Computer Magazine*, 14. évf., pp. 63–68., 1981. márc.

COHEN, D.: On Holy Wars and a Plea for Peace. *IEEE Computer Magazine*, 14. évf., pp. 48–54., 1981. okt.

COLWELL, R: *The Pentium Chronicles*. New York: Wiley, 2005.

COMER, D. E.: *Network Systems Design Using Network Processors, Agere Version*. Upper Saddle River, NJ: Prentice Hall, 2005.

CORBATO, F. J.: PL/I as a Tool for System Programming, *Datamation*, 15. évf., pp. 68–76., 1969. máj.

CORBATO, F. J.–VYSSOTSKY, V. A.: Introduction and Overview of the MULTICS System. *Proc. FJCC*, pp. 185–196., 1965.

CROWLEY, P., FRANKLIN, M.A., HADIMIOGLU, H.–ONUFRYK, P. Z.: *Network Processor Design: Issues and Practices. Vol. 1.* San Francisco: Morgan Kaufmann, 2002.

DALLY, W. J.–TOWLES, B. P.: *Principles and Practices of Interconnection Networks.* San Francisco: Morgan Kaufmann, 2004.

DANESHBEH, A. K.–HASAN, M. A.: Area Efficient High Speed Elliptic Curve Cryptoprocessor for Random Curves. *Proc. Int'l Conf. on Inf. Tech.: Coding and Computing*, IEEE, pp. 588–593., 2004.

DEAN, A. G.: Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers. *IEEE Micro Magazine*, 24. évf., pp. 10–22., 2004. júl.–aug.

DENNING, P. J.: The Working Set Model for Program Behavior. *Commun. of the ACM*, 11. évf., pp. 323–333., 1968. máj.

DIJKSTRA, E. W.: GOTO Statement Considered Harmful. *Commun. of the ACM*, 11. évf., pp. 147–148., 1968a. márc.

DIJKSTRA, E. W.: Co-operating Sequential Processes. in *Programming Languages*, F. Genuys (szerk.), New York: Academic Press, 1968b.

DONALDSON, G., und JONES, D.: Cable Television Broadband Network Architectures. *IEEE Commun. Magazine*, 39. évf., pp. 122–126., 2001. jún.

DONGARRA, J., FOSTER, I., FOX, G., GROPP, W., KENNEDY, K., TORCZON, L.–WHITE, A.: *The Sourcebook of Parallel Computing*. San Francisco: Morgan Kaufman, 2003.

DUBOIS, M., SCHEURICH, C.–BRIGGS, F. A.: Memory Access Buffering in Multiprocessors. *Proc. 13th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 434–442., 1986.

DULONG, C.: The IA-64 Architecture at Work. *IEEE Computer Magazine*, 31. évf., pp. 24–32., 1998. júl.

DUTTA-ROY, A.: An Overview of Cable Modem Technology and Market Perspectives. *IEEE Commun. Magazine*, 39. évf., pp. 81–88., 2001. jún.

ESCHMANN, E., KLAUER, B., MOORE, R.–WALDSCHMIDT, K.: SDAARC: An Extended Cache-Only Memory Architecture. *IEEE Micro Magazine*, 22. évf., pp. 62–70., 2002. máj.–jún.

FAGGIN, F., HOFF, M. E., Jr., MAZOR, S.–SHIMA, M.: The History of the 4004. *IEEE Micro Magazine*, 16. évf., pp. 10–20., 1996. nov.

FALCON, A., STARK, J., RAMIREZ, A., LAI, K.–VALERO, M.: Prophet/Critic Hybrid Branch Prediction. *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 250–261., 2004.

FISHER, J. A.–FREUDENBERGER, S. M.: Predicting Conditional Branch Directions from Previous Runs of a Program. *Proc. Fifth Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85–95., 1992.

FLOYD, T. L.: *Digital Fundamentals.* 8. kiad., Upper Saddle River, NJ: Prentice Hall, 2002.

FLYNN, D.: AMBA: Enabling Reusable On-Chip Designs. *IEEE Micro Magazine*, 17. évf., pp. 20–27., 1997. júl.

FLYNN, M. J.: Some Computer Organizations and Their Effectiveness, *IEEE Trans. on Computers*, C-21. évf., pp. 948–960., 1972. szept.

FOSTER, I.–KESSELMAN, C.: *The Grid 2: Blueprint for a New Computing Infrastructure.* San Francisco: Morgan Kaufman, 2003.

FOSTER, I., KESSELMAN, C., NICK, J. M.–TUECKE, S.: Grid Services for Distributed Systems Integration. *IEEE Computer Magazine*, 35. évf., pp. 37–46., 2002. jún.

FOSTER, I.–KESSELMAN, C.: Globus: A Metacomputing Infrastructure Toolkit. *Int'l J. of Supercomputer Applications*, 11. évf., pp. 115–128., 1998a.

FOSTER, I.–KESSELMAN, C.: The Globus Project: A Status Report. *IPPS/SPDP '98 Heterogeneous Computing Workshop*, IEEE, pp. 4–18., 1998b.

FOTHERINGHAM, J.: Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store. *Commun. of the ACM*, 4. évf., pp. 435–436., 1961. okt.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R.–SUNDERRAM, V.: *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* Cambridge, MA: M.I.T. Press, 1994.

GERBER, R.–BINSTOCK, A.: *Programming with Hyper-Threading Technology.* Santa Clara, CA: Intel Press, 2004.

GHEMAWAT, S., GOBIOFF, H.–LEUNG, S.-T.: The Google File System. *Proc. 19th Symp. on Operating Systems Principles.* ACM, pp. 29–43., 2003.

GOODMAN, J. R.: Using Cache Memory to Reduce Processor Memory Traffic. *Proc. 10th Ann. Int'l Symp. on Computer Arch.* ACM, pp. 124–131., 1983.

GOODMAN, J. R.: Cache Consistency and Sequential Consistency. Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GRIMSHAW, A. S.–WULF, W.: Legion: A View from 50,000 Feet. *Proc. Fifth Int'l Symp. on High-Performance Distributed Computing,* IEEE, pp. 89–99., 1996. aug.

GRIMSHAW, A. S.–WULF, W.: The Legion Vision of a Worldwide Virtual Computer. *Commun. of the ACM*, 40. évf., pp. 39–45.,1997. jan.

GROPP, W., LUSK, E–SKJELLUM, A: *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Cambridge, MA: M.I.T Press, 1994.

GURUMTURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M.–FRANKE, H.: Reducing Disk Power Consumption in Servers with DRPM. *IEEE Computer Magazine*, 36. évf., pp. 59–66., 2003. dec.

HAGERSTEN, E., LANDIN, A., HARIDI, S.: DDM—A Cache-Only Memory Architecture. *IEEE Computer Magazine*, 25. évf., pp. 44–54., 1992. szept.

HAMACHER, V. V., VRANESIC, Z. G.–ZAKY, S. G.: *Computer Organization.* 5. kiad., New York: McGraw-Hill, 2001.

HAMMING, R. W.: Error Detecting and Error Correcting Codes. *Bell Syst. Tech. J.*, 29. évf., pp. 147–160., 1950. ápr.

HAMMOND, L., WONG, V., CHEN, M., HERTZBERG, B, DAVIS, J., CARLSTROM, B., PRABHU, M., WIJAYA, H., KOZYRAKIS, C.–OLUKOTUN, K.: Transactional Memory Coherence and Consistency. *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 102–113., 2004.

HANDY, J.: *The Cache Memory Book.* 2. kiad., Orlando, FL: Academic Press, 1998.

HART, J. M.: *Win32 System Programming.* Reading, MA: Addison-Wesley. 1997.

HEATH, S.: *Embedded Systems Design.* Oxford: Newnes, 2003.

HENKEL, J., HU, X. S.–BHATTACHARYYA, S. S.: Taking on the Embedded System Challenge. *IEEE Computer Magazine*, 36. évf., pp. 35–37., 2003. ápr.

HENNESSY, J. L.: VLSI Processor Architecture. *IEEE Trans. on Computers*, C-33. évf., pp. 1221–1246., 1984. dec.

HENNESSY, J. L.–PATTERSON, D. A.: *Computer Architecture: A Quantitative Approach.* 3. kiad., San Francisco: Morgan Kaufmann, 2003.

HILL, M.: Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer Magazine*, 31. évf., pp. 28–34., 1998. aug.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., ROUSSEL, P.: The Microarchitecture of the Pentium 4. *Intel Technology Journal*, 5. évf., pp. 1–12., 2001. jan.–márc.

HOARE, C. A. R.: Monitors, An Operating System Structuring Concept. *Commun. of the ACM*, évf. 17, pp. 549–557, 1974. okt.; Erratum in *Commun. of the ACM*, 18. évf., p. 95., 1975. febr.

HUH, J., BURGER, D., CHANG, J.–SOHI, G. S.: Speculative Incoherent Cache Protocols. *IEEE Micro Magazine*, 24. évf., pp. 104–109., 2004. nov.–dec.

HWANG, K.–XU, Z.: *Scalable Parallel Computing.* New York: McGraw-Hill, 1998.

HWU, W.-M.: Introduction to Predicated Execution. *IEEE Computer Magazine*, 31. évf., pp. 49–50.,1998. jan.

IRVINE, K: *Assembly Language for Intel-Based Computers.* 4. kiad., Upper Saddle River, NJ: Prentice Hall, 2002.

JACOB, B.–MUDGE, T.: Virtual Memory: Issues of Implementation. *IEEE Computer Magazine*, 31. évf., pp. 33–41., 1998a. jún.

JACOB, B.–MUDGE, T.: Virtual Memory in Contemporary Microprocessors. *IEEE Micro Magazine*, 18. évf., pp. 60–75., 1998b. júl.–aug.

JERRAYA, A. A.–WOLF, W.: *Multiprocessor Systems-on-a-Chip.* San Francisco: Morgan Kaufmann, 2005.

JIMENEZ, D. A.: Fast Path-Based Neural Branch Prediction. *Proc. 36th Int'l Symp. on Microarchitecture*, IEEE, pp. 243–252., 2003.

JOHNSON, K. L., KAASHOEK, M. F.–WALLACH, D. A.: CRL: High-Performance All-Software Distributed Shared Memory. *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213–228., 1995.

JOHNSON, M.: *Superscalar Microprocessor Design.* Upper Saddle River, NJ: Prentice Hall, 1991.

KALLA, R., SINHAROY, B.–TENDLER, J. M.: IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro Magazine*, 24. évf., pp. 40–47., 2004. márc.–ápr.

KAPASI, U. J., RIXNER, S., DALLY, W. J., KHAILANY, B., AHN, J. H., MATTSON, P.–OWENS, J. D.: Programmable Stream Processors. *IEEE Computer Magazine*, 36. évf., pp. 54–62., 2003. aug.

KAPIL, S., McGHAN, H.–LAWRENDRA, J.: A Chip Multithreaded Processor for Network-Facing Workloads. *IEEE Micro Magazine*, 24. évf., pp. 20–30., 2004. márc.–ápr.

KATZ, R. H.–BORRIELLO, G.: *Contemporary Logic Design.* Upper Saddle River, NJ: Prentice Hall, 2004.

KAUFMAN, C., PERLMAN, R.–SPECINER, M.: *Network Security.* 2. kiad., Upper Saddle River, NJ: Prentice Hall, 2002.

KERMARREC, A.-M., KUZ, I, VAN STEEN, M.–TANENBAUM, A. S.: A Framework for Consistent Replicated Web Objects. *Proc. 78th Int'l Conf. on Distr. Computing Syst.*, IEEE, pp. 276–284., 1998.

KIM, N. S., AUSTIN, T., BLAAUW, D., MUDGE, T., FLAUTNER, K., HU, J. S., IRWIN, M. J., KANDEMIR, M.–NARAYANAN, V.: Leakage Current: Moore's Law Meets Static Power. *IEEE Computer Magazine*, 36. évf., pp. 68–75., 2003. dec.

KNUTH, D.E.: An Empirical Study of FORTRAN Programs. *Software—Practice & Experience*, 1. évf., pp. 105–133., 1971.

KNUTH, D. E.: *The Art of Computer Programming: Fundamental Algorithms.* 3. kiad., Reading, MA: Addison-Wesley, 1997.

KNUTH, D. E.: *The Art of Computer Programming: Seminumerical Algorithms.* 3. kiad., Reading, MA: Addison-Wesley, 1998.

KOGEL, T.–MYER, H.: Heterogeneous MP-SoC: the solution to energy-efficient signal processing. *Proc. 41st Ann. Conf. on Design Automation*, IEEE, pp. 686–691., 2004.

KONTOTHANASSIS, L., HUNT, G., STETS, R., HARDAVELLAS, N., CIERNIAD, M., PARTHASARATHY, S., MEIRA, W., DWARKADAS, S.–SCOTT, M.: VM-Based Shared Memory on Low Latency Remote Memory Access Networks. *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 157–169., 1997.

KOREN, I.: *Computer Arithmetic Algorithms.* Natick, MA: A.K. Peters, 2002.

KOUFATY, D.–MARR, D. T.: Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro Magazine*, 23. évf., pp. 56–65., 2003. márc.–ápr.

KUMAR, R., JOUPPI, N. P.–TULLSEN, D. M.: Conjoined-Core Chip Multiprocessing. *Proc. 37th Int'l Symp. on Microarchitecture*, IEEE, pp. 195–206., 2004.

LAMPORT, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28. évf., pp. 690–691., 1979. szept.

LAROWE, R. P.–ELLIS, C. S.: Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Trans. on Computer Systems*, 9. évf., pp. 319–363., 1991. nov.

LAVAGNO, L.: Systems on a Chip: The Next Electronic Frontier. *IEEE Micro Magazine*, 22. évf., pp. 14–15., 2002. szept.–okt.

LAWTON, G.: Will Network Processor Units Live up to Their Promise? *IEEE Computer Magazine*, 37. évf., pp. 13–15., 2004. ápr.

LEKKAS, P. C.: *Network Processors: Architectures, Protocols, and Platforms.* New York: McGraw-Hill, 2003.

LEVINE, J. R.: *Linkers and Loaders.* San Francisco: Morgan Kaufmann, 2000.

LI, K.–HUDAK, P.: Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7. évf., pp. 321–359., 1989. nov.

LIMA, F., CARRO, L., VELAZCO, R.–REIS, R.: Injecting Multiple Upsets in a SEU Tolerant 8051 Micro-Controller. *Proc. Eighth IEEE Int'l On-Line Testing Workshop*, IEEE, p. 194., 2002. júl.

LINES, A.: Asynchronous Interconnect for Synchronous SoC Design. *IEEE Micro Magazine*, 24. évf., pp. 32–41., 2004. jan.–febr.

LU, H., COX, A. L., DWARKADAS, S., RAJAMONY, R.–ZWAENEPOEL, W.: Software Distributed Shared Memory Support for Irregular Applications. *Proc. Sixth Conf. on Prin. and Practice of Parallel Progr.*, pp. 48–56., 1997. jún.

LUKASIEWICZ, J.: *Aristotle's Syllogistic.* 2. kiad., Oxford: Oxford University Press, 1958.

LUTZ, J.–HASAN, A.: High Performance FPGA based Elliptic Curve Cryptographic Co-Processor. *Proc. Int'l Conf. on Inf. Tech.: Coding and Computing*, IEEE, pp. 486–492., 2004.

LYYTINEN, K.–YOO, Y.: Issues and Challenger in Ubiquitous Computing. *Commun. of the ACM*, 45. évf., pp. 63–65., 2002. dec.

MACKENZIE, I. S., PHAN, R.: *The 8051 Microcontroller.* 4. kiad., Upper Saddle River, NJ: Prentice Hall, 2005.

MANO, M. M.–KIME, C. R.: *Logic and Computer Design Fundamentals.* 3. kiad., Upper Saddle River, NJ: Prentice Hall, 2003.

MARTIN, A. J., NYSTROM, M., PAPADANTONAKIS, K., PENZES, P. L, PRAKASH, P., WONG, C. G., CHANG, J., KO, K.S., LEE, B., OU, E., PUGH, J, TALVALA, E-V., TONG, J. T.–TURA, A.: The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. *Proc. Ninth Int'l Symp. on Asynchronous Circuits and Systems*, IEEE, pp. 14–23., 2003.

MARTIN, R. P., VAHDAT, A. M., CULLER, D. E.–ANDERSON, T. E.: Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 85–97., 1997.

MAYHEW, D.–KRISHNAN, V: PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects. *Proc. 11th Symp. on High Perf. Interconnects*, IEEE, pp. 21–29., 2003. aug.

MAZIDI, M. A., MCKINLAY–MAZIDI, J. G.: *8051 Microcontroller and Embedded Systems.* Upper Saddle River, NJ: Prentice Hall, 2005.

MAZIDI, M. A.–MAZIDI, J. G.: *The 80x86 IBM PC and Compatible Computers.* 4. kiad., Upper Saddle River, NJ: Prentice Hall, 2002.

MCKNIGHT, L. W., HOWISON, J.–BRADNER, S: Wireless Grids. *IEEE Internet Computing*, 8. évf. pp. 24–31., 2004. júl.–aug.

MCKUSICK, M. K., BOSTIC, K., KARELS, M.–QUARTERMAN, J. S.: *The Design and Implementation of the 4.4 BSD Operating System.* Reading, MA: Addison-Wesley, 1996.

MCKUSICK, M. K., JOY, W.N., LEFFLER, S. J.–FABRY, R. S.: A Fast File System for UNIX. *ACM Trans. on Computer Systems*, 2. évf., pp. 181–197., 1984. aug.

MCNAIRY, C.–SOLTIS, D.: Itanium 2 Processor Microarchitecture. *IEEE Micro Magazine*, 23. évf., pp. 44–55., 2003. márc.–ápr.

MIN, R., W.-Ben, J.–HU, Y.: Location Cache: A Low-Power L2 Cache System. *Proc. 2004 Int'l Symp. on Low Power Electronics and Design*, IEEE, pp. 120–125., 2004. aug.

MESSMER, H.-P.: *The Indispensible PC Hardware Book.* 4. kiad., Reading, MA: Addison-Wesley, 2001.

MOUDGILL, M.–VASSILIADIS, S.: Precise Interrupts. *IEEE Micro Magazine*, 16. évf., pp. 58–67.,1996. jan.

MULLENDER, S. J.–TANENBAUM, A. S.: Immediate Files. *Software-Practice and Experience*, 14. évf., pp. 365–368., 1984.

NESBIT, K. J.–SMITH, J. E.: Data Cache Prefetching Using a Global History Buffer. *Proc. 10th Int'l Symp. on High Perf. Computer Arch.*, IEEE, pp. 96–106., 2004.

NG, S. W.: Advances in Disk Technology: Performance Issues. *IEEE Computer Magazine*, 31. évf., pp. 75–81., 1998. máj.

NICKOLLS, J., MADAR, L.J. III, JOHNSON, S., RUSTAGI, V., UNGER, K.–CHOUDHURY, M.: Calisto: A Low-Power Single-Chip Multiprocessor Communications Platform. *IEEE Micro Magazine*, 23. évf., pp. 29–43., 2003. márc.

NORTON, P.–GOODMAN, J.: *Inside the PC.* 8. kiad., Indianapolis, IN: Sams, 1999.

NULL, L.–LOBUR, J.: *The Essentials of Computer Organization and Architecture.* Sudbury, MA: Jones and Bartlett, 2003.

O'CONNOR, J. M.–TREMBLAY, M.: PicoJava-I: The Java Virtual Machine in Hardware. *IEEE Micro Magazine*, 17. évf., pp. 45–53., 1997. márc.–ápr.

ORGANICK, E.: *The MULTICS System.* Cambridge, MA: M.I.T. Press, 1972.

OSKIN, M., CHONG, F. T.–CHUANG, I. L.: A Practical Architecture for Reliable Quantum Computers. *IEEE Computer Magazine*, 35. évf., pp. 79–87., 2002. jan.

OUADJAOUT, S.–HOUZET, D.: Easy SoC Design with VCI SystemC Adapters. *Proc. Digital System Design*, IEEE, pp. 316–323., 2004.

PAPAEFSTATHIOU, I., NIKOLAOU, N. A., DOSHI, B.–GROSSE, E.: Network Processors for Future High-End Systems and Applications. *IEEE Micro Magazine*, 24. évf., pp. 7–9., 2004. szept.–okt.

PAPAMARCOS, M.–PATEL, J.: A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proc. 11th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 348–354., 1984.

PARIKH, D., SKADRON, K., ZHANG, Y.–STAN, M.: Power-Aware Branch Prediction: Characterization and Design. *IEEE Trans. on Computers*, 53. évf., pp. 168–186., 2004. febr.

PATTERSON, D. A.: Reduced Instruction Set Computers. *Commun. of the ACM*, 28. évf., pp. 8–21.,1985. jan.

PATTERSON, D. A., GIBSON, G.–KATZ, R.: A case for redundant arrays of inexpensive disks (RAID). *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109–166., 1988.

PATTERSON, D. A.–HENNESSY, J. L.: *Computer Organization and Design.* 3. kiad., San Francisco: Morgan Kaufmann, 2005.

PATTERSON, D. A.–SEQUIN, C. H.: A VLSI RISC, *IEEE Computer Magazine*, évf. 15, pp. 8–22., 1982. szept.

PAUL, R. P.: *SPARC Architecture, Assembly Language, Programming, and C.* Upper Saddle River. NJ: Prentice Hall, 1994.

PFISTER, G. E: *In Search of Clusters.* 2. kiad., Upper Saddle River, NJ: Prentice Hall, 1998.

POPESCU, B. C., STEEN, M. VAN–TANENBAUM, A. S.: A Security Architecture for Object-Based Distributed Systems. *Proc. 18th Annual Computer Security Appl. Conf.*, ACM, pp. 161–171., 2002.

POUNTAIN, D.: Pentium: More RISC than CISC. *Byte*, 18. évf., pp. 195–204., 1993. szept.

PRICE, D.: A History of Calculating Machines. *IEEE Micro Magazine*, 4. évf., pp. 22–52.,1984. jan.

RADIN, G.: The 801 Minicomputer. *Computer Arch. News*, 10. évf., pp. 39–47., 1982. márc.

RAMAN, S. K., PENTKOVSKI, V.–KESHAVA, J.: Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro Magazine*, 20. évf., pp. 47–57., 2000. júl.–aug.

RAVIKUMAR, C. P.: Multiprocessor Architectures for Embedded System-on-a-Chip Applications. *Proc. 17th Int'l Conf. on VLSI Design*, IEEE, pp. 512–519., 2004. jan.

RITCHIE, D. M.–THOMPSON, K.: The UNIX Time-Sharing System. *Commun. of the ACM*, 17. évf., pp. 365–375., 1974. júl.

ROBINSON, G. S.: Toward the Age of Smarter Storage. *IEEE Computer Magazine*, 35. évf., pp. 35–41., 2002. dec.

ROSENBLUM, M.–OUSTERHOUT, J. K.: The Design and Implementation of a Log-Structured File System. *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1–15., 1991.

ROTH, C. H.: *Fundamentals of Logic Design.* 5. kiad., Florence, KY: Thomson Engineering, 2003.

RUSSINOVICH, M. E.–SOLOMON, D. A.: *Microsoft Windows Internals.* 4. kiad., Redmond, WA: Microsoft Press. 2005.

RUSU, S., MULJONO, H.–CHERKAUER, B.: Itanium 2 Processor 6M. *IEEE Micro Magazine*, 24. évf., pp. 10–18., 2004. márc.–ápr.

SAHA, D.–MUKHERJEE, A.: Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer Magazine*, 36. évf., pp. 25–31., 2003. márc.

SAKAMURA, K: Making Computers Invisible. *IEEE Micro Magazine*, 22. évf., pp. 7–11., 2002.

SALOMAN, D.: *Assemblers and Loaders.* Upper Saddle River, NJ: Prentice Hall, 1993.

SCALES, D. J., GHARACHORLOO, K.–THEKKATH, C. A.: Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proc. Seventh Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM. pp. 174–185., 1996.

SCHEIBLE, J. P.: A Survey of Storage Options. *IEEE Computer Magazine*, 35. évf., pp. 42–46., 2002. dec.

SELTZER, M., BOSTIC, K., McKUSICK, M. K.–STAELIN, C.: An Implementation of a Log-Structured File System for UNIX. *Proc. Winter 1993 USENIX Technical Conf.*, pp. 307–326., 1993.

SHANLEY, T.–ANDERSON, D.: *PCI System Architecture.* 4. kiad., Reading, MA: Addison-Wesley. 1999.

SHRIVER, B.–SMITH, B.: *The Anatomy of a High-Performance Microprocessor: A Systems Perspective.* Los Alamitos, CA: IEEE Computer Society, 1998.

SIMA, D: Superscalar Instruction Issue. *IEEE Micro Magazine*, 17. évf., pp. 28–39., 1997. szept.–okt

SIMA, D., FOUNTAIN, T.–KACSUK, P.: *Advanced Computer Architectures: A Design Space Approach.* Reading, MA: Addison-Wesley, 1997.

SLATER, R.: *Portraits in Silicon*, Cambridge, MA: M. I. T. Press, 1987.

SOHI, G. S.–ROTH, A.: Speculative Multithreaded Processors. *IEEE Computer Magazine*, 34. évf., pp. 66–73., 2001. ápr.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D. W.–DONGARRA, J.: *MPI: The Complete Reference Manual.* Cambridge, MA: M.I.T. Press, 1996.

SOLARI, E.–CONGDON, B.: *PCI Express Design & System Architecture.* Research Tech. Inc., 2005.

SOLARI, E.–WILLSE, G.: *PCI and PCI-X Hardware and Software.* 6. kiad., San Diego, CA: Annabooks, 2004.

STALLINGS, W.: *Computer Organization and Architecture.* 6. kiad., Upper Saddle River, NJ: Prentice Hall. 2003.

STENSTROM, P., HAGERSTEN, E., LILJA. D.J., MARTONOSI, M.–VENUGOPAL, M.: Trends in Shared Memory Multiprocessing. *IEEE Computer Magazine*, 30. évf., pp. 44–50., 1997. dec.

STETS, R., DWARKADAS. S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S.–SCOTT, M.: CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks. *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170–183., 1997.

SUH, T., LEE, H.-H. S., BLOUGH, D. M.: Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems. Part 1, *IEEE Micro Magazine*, 24. évf., pp. 33–41., 2004. júl.

SUMMERS, C. K.: *ADSL: Standards, Implementation, and Architecture.* Boca Raton, FL: CRC Press, 1999.

SUNDERRAM, V. B.: PVM: A Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 2. évf., pp. 315–339., 1990. dec.

SWAN, R. J., FULLER, S. H.–SIEWIOREK, D. P.: Cm* – A Modular Multiprocessor. *Proc. NCC*, pp. 645–655., 1977.

TAN, W. M.: *Developing USB PC Peripherals.* San Diego, CA: Annabooks, 1997.

TANENBAUM, A. S.: *Computer Networks.* Upper Saddle River, NJ: Prentice Hall, 2003.

TANENBAUM, A. S.: Implications of Structured Programming for Machine Architecture. *Commun. of the ACM*, 21. évf., pp. 237–246., 1978. márc.

TANENBAUM, A. S.: *Operating Systems: Design and Implementation.* Upper Saddle River, NJ: Prentice Hall, 1987.

TANENBAUM, A. S–WOODHULL, A. W.: *Operating Systems: Design and Implementation.* 2. kiad., Upper Saddle River, NJ: Prentice Hall, 1997.

THOMPSON, K.: UNIX Implementation, Bell Syst. Tech. J., 57. évf., pp. 1931–1946., 1978. júl.–aug.

TRELEAVEN, P.: Control-Driven, Data-Driven, and Demand-Driven Computer Architecture. *Parallel Computing*, 2 évf., 1985.

TREMBLAY, M.–O'CONNOR, J. M.: UltraSPARC I: A Four-Issue Processor Supporting Multimedia. *IEEE Micro Magazine*, 16. évf., pp. 42–50., 1996. márc.

TRIEBEL, W. A.: *The 80386, 80486, and Pentium Processor.* Upper Saddle River, NJ: Prentice Hall, 1998.

TUCK, N.–TULLSEN, D. M.: Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. *Proc. 12th Int'l Conf. on Parallel Arch. and Compilation Techniques*, IEEE, pp. 26–35., 2003.

UNGER, S. H.: A Computer Oriented Toward Spatial Problems. *Proc. IRE*, 46. évf., pp. 1744–1750., 1958.

VAHALIA, U.: *UNIX Internals.* Upper Saddle River, NJ: Prentice Hall, 1996.

VAHID, F.: The Softening of Hardware. *IEEE Computer Magazine*, 36. évf., pp. 27–34., 2003. ápr.

VAN STEEN, M., HOMBURG, P.C.–TANENBAUM, A. S.: The Architectural Design of Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7. évf., pp. 70–78., 1999. jan.–márc.

VETTER, P., GODERIS, D., VERPOOTEN, L.–GRANGER, A.: Systems Aspects of APON/VDSL Deployment. *IEEE Commun. Magazine*, 38. évf., pp. 66–72., 2000. máj.

WEAVER, D. L.–GERMOND, T.: *The SPARC Architecture Manual. Version 9*, Upper Saddle River, NJ: Prentice Hall, 1994.

WEISER, M.: The Computer for the 21st Century, *IEEE Pervasive Computing*, 1. évf., pp. 19–25., 2002. jan.–márc.; eredetileg in *Scientific American*, 1991. szept.

WILKES, M. V.: Computers Then and Now. *J. ACM*, 15. évf., pp. 1–7., 1968. jan.

WILKES, M. V.: The Best Way to Design an Automatic Calculating Machine. *Proc. Manchester Univ. Computer Inaugural Conf.*, 1951.

WILSON, J.: Challenger and Trends in Processor Design. *IEEE Computer Magazine*, 31. évf., pp. 39–48., 1998. jan.

WILSON, P.: Floating-Point Survival Kit. *Byte*, 13. évf., pp. 217–226., 1988. márc.

WOLF, W.: The Future of Multiprocessor Systems-on-Chips. *Proc. 41st Ann. Conf. on Design Automation*, IEEE, pp. 681–685., 2004.

A) Bináris számok

Az a számolási mód, amelyet a számítógépek használnak, bizonyos tekintetben különbözik az emberek által használt számítási módtól. A legnagyobb különbség az, hogy a számítógépek olyan számokon végeznek műveleteket, amelyek pontossága véges és rögzített. A másik különbség az, hogy a legtöbb számítógép kettes (bináris) számrendszerbeli ábrázolást használ a tízes (decimális) helyett. A melléklet ezeket a témákat tárgyalja.

A.1. Véges pontosságú számok

Számoláskor nagyon kevés figyelmet fordítunk arra, hogy hány decimális jegygyel ábrázoljuk a számot. A fizikusok 10^{78} elektronnal számolnak az univerzumban, talán anélkül hogy végiggondolnák, hogy ennek a teljes leírása 79 decimális jegy kiírását igényeli. Annak, aki papíron ceruzával számol, és 6 jegyre pontosan kell megadnia az eredményt, a közbenső számításokat 7 vagy 8 vagy még több jegy pontossággal számolja. Az a probléma soha nem fordul elő, hogy a papír nem elég széles a hétjegyű számokra.

A számítógépekkel a dolog egészen máshogy áll. A legtöbb számítógépen egy szám tárolására szolgáló memória fix méretű, rögzítették a számítógép tervezésekor. Bizonyos erőfeszítésekkel a programozó tudja a számokat ábrázolni az eredeti fix méretnél kétszer, háromszor, sőt többször nagyobb helyen is, de ezzel nem változtatja meg a probléma természetét. A számítógépes erőforrásnak ez a véges természete csak azokkal a számokkal enged foglalkozni, amelyek fix számú számjeggyel ábrázolhatók. Ezeket a számokat **véges pontosságú számok**nak (**finite-precision numbers**) hívjuk.

A véges pontosságú számok tanulmányozásához vizsgáljuk meg a pozitív számok egy halmazát, amelyeket három jeggyel ábrázolunk, tizedespont és előjel nélkül. Ez pontosan ezer elemet jelent: a 000, 001, 002, 003, ..., 999. Ezzel a megszorítással lehetetlen bizonyos számokat kifejezni:

1. a 999-nél nagyobbakat;
2. a negatív számokat;
3. a törteket;
4. az irracionális számokat,;
5. a komplex számokat.

Egyik legfontosabb sajátsága az összes egész szám halmazán történő számolásnak az, hogy ez a halmaz **zárt (closure)** az összeadás, kivonás és a szorzás műveletére nézve. Más szóval, bármelyik i és j egész számra az $i + j$, $i - j$ és az $i \times j$ szintén egész szám. Az egész számok halmaza nem zárt az osztásra nézve, mert létezik olyan i és j, amelyek esetében az i/j nem fejezhető ki egész számmal (például 7/2 és 1/0).

Véges pontosságú számok halmaza nem zárt a négy alapműveletre nézve. nézzük például a három számjegyű decimális számokat:

600 + 600 = 1200	(túl nagy)
003 – 005 = –2	(negatív)
050 × 050 = 2500	(túl nagy)
007 / 002 = 3,5	(nem egész)

A zártságot megszüntető okok két osztályba sorolhatók: azokra a műveletekre, amelyeknél az eredmény nagyobb, mint a legnagyobb halmazbeli szám (túlcsordulási hiba, overflow error), vagy kisebb, mint a legkisebb szám a halmazban (alulcsordulási hiba, underflow error), és azokra a műveletekre, amelyeknél az eredmény se nem túl nagy, se nem túl kicsi, de nem eleme a halmaznak. A fenti négy példa közül az első három az első osztályra példa, az utolsó a másodikra.

Mivel a számítógépek memóriája véges, és emiatt a számítást szükségképpen véges pontosságú számokon kell végrehajtani, az eredmény bizonyos számításoknál nem lesz helyes a klasszikus matematika szabályai szerint. Különösnek tűnhet első látásra, hogy létezik olyan számoló berendezés, amely még tökéletes működési körülmények között is hibás eredményt ad, de a hiba a véges aritmetika természetének logikus következménye. Néhány számítógépnek speciális hardvere van, amely észreveszi a túlcsordulási hibákat.

A véges pontosságú számok algebrája különbözik a normál algebrától. Példaként tekintsük az asszociativitási törvényt:

$$a + (b - c) = (a + b) - c$$

Számítsuk ki mindkét oldalt $a = 700$, $b = 400$, $c = 300$ esetén. A bal oldalon először kiszámoljuk $(b - c)$-t, ami 100, és hozzáadjuk a-t, így 800-at kapunk. A jobb oldalon először $(a + b)$-t számoljuk, amely túlcsordulást eredményez a három számjegyes egészek véges aritmetikájában, és az eredmény függ a géptől, de nem lehet 1100. Ha ebből levonnánk a 300-at, nem kaphatjuk meg a 800-at. Az asszociativitási törvény nem működik. A műveletek sorrendje is számít.

Egy másik példa, a disztributivitási törvény:

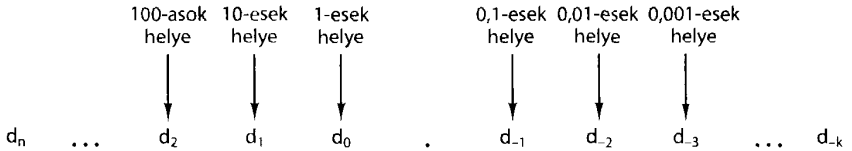
$$a \times (b - c) = a \times b - a \times c$$

Értékeljük ki mindkét oldalt $a = 5$, $b = 210$, $c = 195$ esetén. A bal oldal 5×15, egyenlő 75-tel. A jobb oldal viszont nem lesz 75, mivel az $a \times b$ túlcsordul.

Megvizsgálva ezeket a példákat arra a következtetésre juthatunk, hogy bár a számítógépek általános célú eszközök, mégis véges természetű rendszereik alkalmatlanná teszik őket számítások végrehajtására. Ez a következtetés természetesen helytelen, de rávilágít arra, hogy milyen fontos, hogy megértsük, hogyan működnek a számítógépek, és milyen korlátokkal rendelkeznek.

A.2. Számrendszerek alapszámai

A közönséges számok, amelyeket mindenki ismer, decimális jegyekből álló sorozatok, melyek tizedesvesszőt is tartalmazhatnak. Az általános formát és annak hagyományos értelmezését mutatja be az A.1. ábra. A hatvány kifejezésben a 10-est választottuk **alapszám**nak (**radix**), mivel decimális, más szóval tízes alapú számrendszert használunk. Számítógépek esetében kényelmesebb nem tízes alapú, hanem más rendszereket használni. A legfontosabb alapszámok a 2, a 8 és a 16. Ezen



$$\text{Szám} = \sum_{i=-k}^{n} d_i \times 10^i$$

A.1. ábra. *Decimális szám általános formája*

alapszámokra épülő számrendszereket sorrendben **kettes (bináris)**, **nyolcas (oktális)** és **tizenhatos (hexadecimális)** számrendszernek nevezzük.

A k alapszámú számrendszerek k különböző szimbólumot igényelnek, hogy a számjegyeket 0-tól $(k - 1)$-ig tudjuk ábrázolni. A decimális számok 10-féle decimális jegyből épülnek fel:

0 1 2 3 4 5 6 7 8 9

Ezzel szemben a bináris számok nem tíz jegyet használnak. Ezeket kétféle jeggyel írhatjuk fel:

0 1

Az oktális számok 8 oktális jegyből épülnek fel:

0 1 2 3 4 5 6 7

A hexadecimális számoknál 16-féle számjegy szükséges, így hat új szimbólumra van szükség. Kényelmes bevezetni a nagybetűket A-tól F-ig, amelyek a 9-et követő hat számjegy számára szolgálnak. Így a hexadecimális számok a következő jegyekből épülnek fel:

0 1 2 3 4 5 6 7 8 9 A B C D E F

A „bináris számjegy" kifejezést, amely egy 1-est vagy egy 0-t jelent, **bit**nek hívjuk. Az A.2. ábra a 2001 decimális számot mutatja be bináris, oktális, decimális és hexadecimális formában. A 7B9 szám természetesen hexadecimális, mert a B szimbólum csak hexadecimális számokban fordulhat elő. A 111 azonban előfordulhat bármelyik rendszerben a négy közül. Azért hogy egyértelművé tegyék az alapszámot, alsó indexben a 2, 8, 10 vagy 16 számokat használják jelzésre, amikor ez nem nyilvánvaló a környezetből.

Bináris 1 1 1 1 1 0 1 0 0 0 1

$1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 1

Oktális 3 7 2 1

$3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0$

1536 + 448 + 16 + 1

Decimális 2 0 0 1

$2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$

2000 + 0 + 0 + 1

Hexadecimális 7 D 1

$7 \times 16^2 + 13 \times 16^1 + 1 \times 16^0$

1792 + 208 + 1

A.2. ábra. *2001 bináris, oktális, decimális és hexadecimális számrendszerben*

Példaként szolgál a bináris, oktális, hexadecimális jelölésekre az A.3. ábra, amely nem negatív számok egy részét mutatja a négy különböző számrendszerben. Talán néhány régész évezredek múlva lel majd rá e táblázatra, és a késő XX. és a kora XXI. századi számrendszerek Rosetta kövének fogja tartani.

Decimális	Bináris	Oktális	Hexa-decimális
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

A.3. ábra. *Decimális számok és bináris, oktális és hexadecimális számrendszerbeli megfelelőik*

A.3. Konverzió egyik alapról a másik alapra

Az egész részt a törtrésztől elválasztó jelet tizedes pontnak nevezzük, még abban az esetben is, ha nem tízes számrendszerben dolgozunk. Ezt gyakran bináris pontnak is hívjuk. A hexadecimális vagy oktális számok konverziója bináris számokra nagyon könnyű. Egy bináris szám oktálisra konvertálásához osszuk hárombites csoportokra a számot. A tizedes ponttól közvetlenül balra (vagy jobbra) az első három bit alkot egy csoportot, a tőlük közvetlenül balra (vagy jobbra) eső három bit a következő csoportot és így tovább. Minden egyes ilyen hárombites csoportot direkt módon konvertálhatunk egy oktális számjegyre, 0-tól 7-ig aszerint, hogy a

konverziókban az A.3. ábra mely adott sorával egyezik meg. Lehet, hogy szükséges egy vagy két kezdő vagy záró 0-val kitölteni a hárombites sorozatokat. A konverzió az oktális számrendszerből a bináris számrendszerbe is hasonlóan egyszerű. Minden oktális jegyet egyszerűen cseréljünk le egy vele ekvivalens hárombites bináris számra. A konverzió hexadecimálisból binárisba lényegében hasonló, mint az oktálisból binárisba, csak minden hexadecimális jegy megfelel egy négybites csoportnak az előzőkben használt három bit helyett. Az A.4. ábra néhány példát tartalmaz a konverziókról.

A decimális számrendszerbeli számok konverziója bináris számrendszerbe két különböző módszerrel oldható meg. Az első módszer közvetlen módon követi a bináris számok definícióját. A legnagyobb kettes kitevővel rendelkező számot, amely nem nagyobb, mint maga a szám, le kell vonni az eredeti decimális számból. Ezután a különbségre ismételjük az eljárást. Ha egyszer az adott számot szétbontottuk kettő hatványaira, ezután a bináris szám összeállítható: 1-esek lesznek azokon a bitpozíciókon, amelyeket a szétbontásban szereplő kettő hatványok kitevői határoznak meg, máshol pedig 0-k.

A másik módszer csak egészek esetén alkalmazható, 2-vel való osztást tartalmaz. A hányadost az eredeti szám alá írjuk, a maradék 0 vagy 1, ezt a hányados mellé, a következő oszlopba írjuk. Ezután a hányadossal addig folytatjuk ezt az eljárást, amíg hányadosként 0-t nem kapunk. Ennek az eljárásnak az eredménye két oszlopot tartalmaz, a hányadosokét és a maradékokét. A bináris számot most már közvetlenül a maradék oszlopból olvashatjuk le úgy, hogy elindulunk az oszlop aljától felfelé a leolvasással. Az A.5. ábra egy példát mutat decimális számrendszerből binárisba történő konverzióra.

A bináris számokat szintén két módszerrel konvertálhatjuk decimális számra. Az egyik módszer, amikor összegezzük kettő azon hatványait, ahol 1-es volt a kitevőben. Például:

$$10110 \text{ egyenlő } 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22\text{-vel}$$

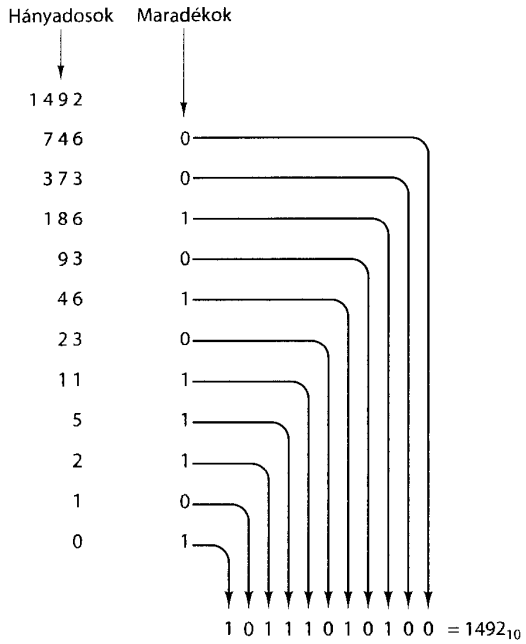
1. példa

Hexadecimális		1		9		4		8	.	B		6
Bináris	0	0	0	1	1	0	0	1	0	1	0	0	0	.	1	0	1	1	0	1	1	0	0
Oktális		1		4		5		1	0	.	5		5		4								

2. példa

Hexadecimális		7			B			A			3		.	B			C			4			
Bináris	0	1	1	1	1	0	1	1	1	0	1	0	0	0	1	1	.	1	0	1	1	1	1	0	0	0	1	0	0
Oktális		7			5			6			4		3	.	5			7			0			4				

A.4. ábra. *Konverziós példák oktálisból binárisba és hexadecimálisból binárisba*



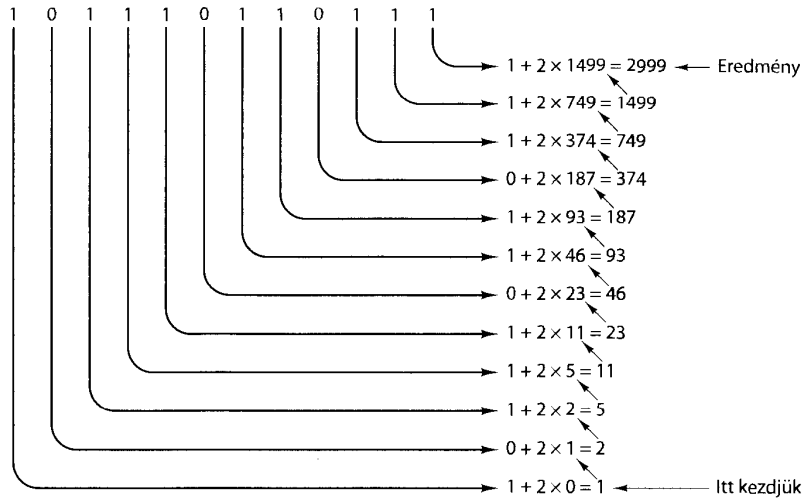
A.5. ábra. *1492 decimális szám konverziója bináris számmá egymást követő felezéssel, felülről lefelé. Például 93-at osztjuk 2-vel, a hányados 46 és a maradék 1, egy sorral lejjebb írjuk őket*

A másik módszer szerint, a bináris számot függőlegesen írjuk le, soronként egy bitet, és a legaljára írjuk a bal szélső bitet. Az alsó sort egyes sornak, az utána következőt kettes sornak hívjuk és így tovább. A decimális számot felépíthetjük egy ezzel a függőlegesen írt bináris számmal párhuzamos oszlopban. Azzal kezdjük, hogy 1-et írunk az 1-es sorba. Az n. sor eredménye az előző sor (n – 1.) kétszerese plusz az n. sor bitje (0 vagy 1). Az eredmény a legfelső sorban képződik. Az A.6. ábra ennek a módszernek az alkalmazását mutatja be binárisból decimálisba történő konverzió esetén.

A decimálisból oktálisba és hexadecimálisba történő konverziót úgy kell végrehajtanunk, hogy először bináris számrendszerbe konvertálunk, és azután a kívánt számrendszerbe, vagy pedig 8, illetve 16 hatványainak kivonogatásával.

A.4. Negatív bináris számok

A negatív számok megjelenítésére négy különböző módszert (kódot) használtak a digitális számítógépek történetében, újra és újra elővéve őket. Az elsőt **előjeles**



A.6. ábra. *101110110111 bináris szám konverziója decimális számmá sorozatos duplázással, a legalsó sortól kezdve. Mindegyik sor az alatta lévő kétszerese plusz a megfelelő sor bitje. Például 749 kétszerese 374-nek plusz az 1-es bit ebben a sorban*

abszolút értéknek (signed magnitude) a módszernél a bal oldali első bit az előjelbit, (a 0 a +, az 1 pdig a –) és a maradék bitek jelzik a szám abszolút értékét.

A második rendszernek, amelyet **egyes komplemens (one's complement)** rendszernek hívunk, szintén van egy előjelbitje, 0 jelenti a pozitív, 1 a negatív számot. Egy szám negatívját úgy kapjuk meg, hogy minden 1-est 0-ra, és minden 0-t 1-esre cserélünk. Ez természetesen az előjelbitre is vonatkozik. Az egyes komplemens képzése elavult rendszer.

A harmadik rendszer, amelyet **kettes komplemens**nek **(two's complement)** nevezünk, szintén tartalmaz előjelbitet, amely 0 a pozitív, 1 pedig a negatív számok esetében. Egy szám negatívját kétlépéses eljárással kapjuk meg. Az első lépésben, minden 1-est 0-ra cserélünk, és minden 0-t 1-esre, ugyanúgy, mint az egyes komplemens képzésnél. A második lépésben 1-et hozzáadunk az eredményhez. A bináris összeadás hasonló a decimális összeadáshoz, de átvitel akkor képződik, ha az összeg nagyobb, mint 1, decimális esetben akkor, ha nagyobb, mint 9. Példaként állítsuk elő 6 kettes komplemensét két lépésben:

00000110 (+6)
11111001 (–6 egyes komplemens kódja)
11111010 (–6 kettes komplemens kódja)

Ha átvitel áll elő a bal szélső bitnél, akkor ezt eldobjuk.

A negyedik rendszer, amelyet az m bites szám esetében 2^{m-1} **többletes**nek (**excess** 2^{m-1}) hívunk, egy szám helyett a számnak és 2^{m-1}-nek az összegét tárolja. Például egy 8 bites számra $m = 8$, a rendszert 128 többletesnek nevezzük, és egy szám helyett a valódi szám plusz 128-at tároljuk. Így a –3 ábrázolása –3 + 128 = 125, azaz –3 nyolcbites számként ábrázolva 125 (01111101). A –128-tól + 127-ig elhelyezkedő számokat így meg tudjuk feleltetni 0-tól 255-ig terjedő számoknak, amelyeket 8 bites pozitív számként fejezhetünk ki. Érdekességképpen megjegyezzük, hogy ez a rendszer azonos a kettes komplemenssel fordított előjelbittel. Az A.7. ábra példákat ad negatív számok ábrázolására mind a négy rendszerben.

Mind az előjeles abszolút értéknek, mind az egyes komplemensnek két reprezentációja van 0-ra: a pozitív 0 és a negatív 0. Ez a szituáció nem kívánatos. A kettes komplemensnél nincs meg ez a probléma, mivel a pozitív 0 kettes komplemense pozitív 0. A kettes komplemens azonban más jellegzetességgel rendelkezik. Az a bitminta, amely egy 1-esből áll és utána csupa 0, az saját magának a komplemense. Ez a tény azt jelenti, hogy a pozitív és negatív számok aszimmetrikusak, azaz van olyan negatív szám, amelynek nincs pozitív párja.

N decimális	N bináris	–N előjeles abszolút érték	–N 1-es komplemens	–N 2-es komplemens	–N 128 többletes
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nem ábrázolható	Nem ábrázolható	Nem ábrázolható	10000000	00000000

A.7. ábra. *8 bites negatív számok a négy rendszerben*

Ezeknek a problémáknak nem nehéz megtalálni az okát, egy kódoló rendszernél két sajátosságot kívánunk teljesíteni:

1. A nullát csak egyféleképpen ábrázoljuk.
2. Pontosan annyi pozitív számot ábrázoljunk, ahány negatívot.

A probléma az, hogy bármelyik halmaznak, amely ugyanannyi pozitív és negatív számot és egy 0-t tartalmaz, páratlan számú eleme van, ellenben m bit páros számú bitmintát enged meg. Ez mindig azt jelenti, hogy egy bitmintával több vagy kevesebb lesz, függetlenül attól, hogy milyen reprezentációt választnk. Ezt az extra bitmintát tudjuk használni –0-nak vagy egy nagy negatív számnak, vagy bármi másnak, de függetlenül attól, hogy mire használjuk, mindig kényelmetlenséget okoz.

A.5. Bináris aritmetika

Az A.8. ábra a bináris számokra vonatkozó összeadó táblát tartalmazza.

Első tag	0	0	1	1
Második tag	+0	+1	+0	+1
Összeg	0	1	1	0
Átvitel	0	0	0	1

A.8. ábra. *Bináris számok összeadó táblája*

Két bináris számot úgy tudunk összeadni, hogy az első és második tag jobb szélső bitjével kezdjük az összeadást. Ha átvitel keletkezik, akkor ezt egy pozícióval balra visszük, hasonlóan, mint a decimális aritmetikánál. Az egyes komplemens számolási módszernél az összeadásnál a bal oldalon keletkezett átvitelbitet a jobb szélső bithez adjuk hozzá. Ezt az eljárást körbejárásos (end-around) átvitelnek nevezzük. A kettes komplemens számolási módszernél az átvitelt a bal szélső bitnél csupán eldobjuk. Az A.9. ábra példát tartalmaz a bináris aritmetikára.

Decimális	1-es komplemens	2-es komplemens
10	00001010	00001010
+ (–3)	11111100	11111101
+7	1 00000110	1 00000111
	átvitel 1	eldobott
	00000111	

A.9. ábra. *Összeadás 1-es, illetve 2-es komplemens esetén*

Ha az összeadandó és a bővítendő szám különböző előjelű, nem fordulhat elő túlcsordulás. Ha azonos előjelűek, és az eredmény ettől különböző előjelű, akkor túlcsordulás következett be, és az eredmény rossz. Mind az egyes, mind a kettes komplemens számolási módszernél túlcsordulás akkor és csak akkor fordulhat elő, ha az előjelbithez bejövő átvitelbit különbözik az előjelbitnél keletkező továbbmenő átvitelbittől. A legtöbb számítógép megtartja az előjelbitnél keletkező átvitelbitet, de az előjelbitnél bejövő átvitelbit nem látható az eredményből. Emiatt ezt általában egy speciális túlcsordulás biten jelzik.

A.6. Feladatok

1. Konvertáljuk a következő számokat bináris számokká: 1984, 4000, 8192.
2. Mi az 1001101001 (bináris szám) decimális, oktális és hexadecimális megfelelője?
3. Melyek az érvényes hexadecimális számok a következők közül? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.
4. Fejezzük ki a decimális 100-at a kettes alaptól a kilencesig minden alapban.
5. Hány különböző pozitív egész szám fejezhető ki k számjegy segítségével r alapszám esetén?
6. A legtöbb ember 10-ig tud elszámolni az ujjain, azonban a számítógéptudósok ezt jobban tudják csinálni. Ha mindegyik ujjunkat egy bitnek tekintjük, a kinyújtott ujj legyen 1, a behajlított ujj pedig 0. Meddig tudunk elszámolni, ha mindkét kezünket ilyen módon használjuk? Mindkét kezünkkel és mindkét lábunkkal? Most használjuk a kezünket és a lábunkat úgy, hogy a bal lábunkon a nagy lábujjunk jelenti az előjelbitet kettes komplemens kódban. Mi az ábrázolható számok intervalluma?
7. Hajtsuk végre a következő számításokat 8 bites kettes komplemensű számokon.

00101101	11111111	00000000	11110111
+ 01101111	+ 11111111	- 11111111	- 11110111

8. Ismételjük meg az előző feladat számítását, csak most egyes komplemens kódban.
9. Tekintsük a kettes komplemens kódban levő következő 3 jegyű bináris számok összeadását. Mindegyik összegnél állapítsuk meg:
   a) Vajon az eredmény előjelbitje 1-e?
   b) Vajon az alsó helyértékű három bit 0-e?
   c) Van-e túlcsordulás?

000	000	111	100	100
+ 001	+ 111	+ 110	+ 111	+ 100

10. n jegyet tartalmazó előjeles decimális számokat kifejezhetünk $n + 1$ jeggyel az előjel használata nélkül. A pozitív számok esetén 0 legyen a bal szélső jegy. A negatív számokat jegyenként 9-ből való kivonással ábrázoljuk. Így a negatív 014 725-ből 985 274 lesz. Az ilyen számokat kilences komplemens kódú számoknak hívjuk a bináris számoknál használt egyes komplemens analógiájára. Fejezzük ki a következő számokat háromjegyű kilences komplemensű számként: 6, –2, 100, –14, –1, 0.
11. Határozzuk meg az összeadás szabályát a kilences komplemens kód alkalmazásakor, és hajtsuk végre a következő összeadásokat.

0001	0001	9997	9241
+ 9999	+ 9998	+ 9996	+ 0802

12. A tízes komplemens hasonló a kettes komplemenshez. A tízes komplemensnél a negatív számot alakítjuk ki úgy, hogy 1-et hozzáadunk a megfelelő kilences komplemens számhoz, figyelmen kívül hagyva az átvitelt. Mi a szabálya a tízes komplemens kódú az összeadásnak?
13. Szerkesszünk szorzótáblát hármas alapú számokhoz.
14. Szorozzuk össze a 0111 és a 0011 bináris számokat.
15. Írjunk programot, amely ASCII karaktersorozatot előjeles decimális számként értelmez, és kinyomtatja kettes komplemens kódban bináris, oktális és hexadecimális formában.
16. Írjunk programot, amely két 32 karakteres, csak 0-t és 1-et tartalmazó ASCII karaktersorozatot kettes komplemensű 32 bites bináris számként értelmez. A program írja ki az összegüket, 0 és 1 jegyekből álló 32 karakteres ASCII karaktersorozatként.

B) Lebegőpontos számok

Sok számításnál a használt számok intervalluma nagyon nagy lehet. Például egy csillagászati számítás tartalmazhatja az elektron tömegét: 9×10^{-28} gramm, és a Nap tömegét: 2×10^{33} gramm, ez az intervallum meghaladja a 10^{60}-t. Ezeket a számokat a következő módon ábrázolhatjuk:

000000000000000000000000000000000.00000000000000000000000000009
200000000000000000000000000000000.00000000000000000000000000000

és minden számítást úgy kellene végrehajtani, hogy megmaradjon a 34 jegy a tizedesponttól balra és 28 jegy ettől jobbra. Azért, hogy ezt meg tudjuk tenni, 62 szignifikáns (értékkel bíró, fontos) jegyet kellene tartalmaznia az eredményeknek. Egy bináris számítógép dupla pontosságú számítási módszerrel megfelelő pontosságot érhetne el. A Nap tömege azonban nem ismert 5 szignifikáns jegyre, még ha 62 jegy állna is rendelkezésünkre. Valójában bármilyen jellegű mérést felesleges 62 jegyre elvégeznünk. Bár lehetséges lenne megtartani mind a 62 jegyet a közbülső részeredményeknél, utána pedig eldobni 50-et vagy 60-at, mielőtt kinyomtatnánk a végeredményt. Ha így tennénk, ez nagyon sok CPU-időt és memóriát pazarolna.

Hogyan lehet olyan számábrázolási rendszert készíteni, amelyben az ábrázolható számok tartománya független a szignifikáns jegyek számától? Ebben a függelékben ilyen rendszerről lesz szó. Ez azon a tudományos jelölésmódon alapul, amely széles körben használt a fizikában, kémiában és mérnöki tudományokban.

B.1. A lebegőpontos számábrázolás elvei

A szám nagyságának és a pontosságának szétválasztására az egyik módszer, hogy a számokat a jól ismert tudományos jelöléssel ábrázoljuk:

$$n = f \times 10^e$$

ahol az f a **törtrész (fraction)** vagy **mantissza (mantissa)**, az e pedig egy pozitív vagy negatív szám, amelyet **kitevő**nek vagy **exponens**nek **(exponent)** hívunk. Ennek a jelölésnek a számítógépes verzióját **lebegőpontos szám**nak **(floating point)** hívjuk. Néhány példa az így ábrázolt számokra:

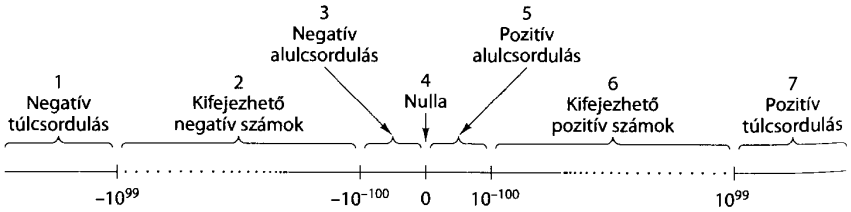
$$\begin{array}{llll} 3{,}14 & = 0{,}314 & \times 10^{1} & = 3{,}14 \times 10^{0} \\ 0{,}000001 & = 0{,}1 & \times 10^{-5} & = 1{,}0 \times 10^{-6} \\ 1941 & = 0{,}1941 & \times 10^{4} & = 1{,}941 \times 10^{3} \end{array}$$

A nagyságrendet a kitevő jegyeinek száma, a pontosságot pedig a törtrész jegyeinek száma határozza meg. Mivel különböző módon fejezhetünk ki egy adott számot, a sok forma közül az egyiket általában szabványosnak nevezzük. Tekintsük a következő ábrázolást, hogy megvizsgáljuk a számábrázolás ezen módszerének sajátosságait: R egy előjeles háromjegyű törtszám, ahol $0{,}1 \leq |f| < 1$ vagy 0, és az exponens kétjegyű. Ezen a számok nagyságrendje $+0{,}100 \times 10^{-99}$-től $+0{,}999 \times 10^{+99}$-ig terjed, lefedik a közel 199-es nagyságrendet, és csak öt számjegy és két előjel szükséges egy szám tárolásához.

A lebegőpontos számokat használhatjuk arra, hogy modellezzük a matematikában használt valós számokat, bár van néhány jelentős különbség. A B.1. ábra a valós számegyenes egy durván elnagyolt sémáját mutatja valós számok 7 régióra oszlanak:

1. Nagy negatív számok, amelyek kisebbek, mint $-0{,}999 \times 10^{99}$.
2. Negatív számok $-0{,}999 \times 10^{99}$ és $-0{,}100 \times 10^{-99}$ között.
3. Kis negatív számok, abszolút értékük kisebb, mint $0{,}100 \times 10^{-99}$.
4. Nulla.
5. Kis pozitív számok, nagyságuk kisebb, mint $0{,}100 \times 10^{-99}$.
6. Pozitív számok $0{,}100 \times 10^{-99}$ és $0{,}999 \times 10^{99}$ között.
7. Nagy pozitív számok, amelyek nagyobbak, mint $0{,}999 \times 10^{99}$.

Nagyon fontos különbség a három tört és két exponens kitevő jeggyel megadott számkészlet és a valós számok között, hogy az előzőnél nem lehet megadni egyetlen számot sem az 1, 3, 5 vagy 7-es régióban. Ha egy számítási eredmény az 1-es vagy 7-es régióban van – például $10^{60} \times 10^{60} = 10^{120}$ – **túlcsordulási hiba** fog előfordulni, és a megoldás nem lesz helyes. Az ok, a számok reprezentációjának



B.1. ábra. *A valós számegyenes 7 régióra osztható*

véges természetéből fakad, és ez elkerülhetetlen. Hasonlóképpen, az eredmény a 3-as vagy 5-ös régióban sem fejezhető ki. Ezt a helyzetet **alulcsordulási hibának** nevezzük. Az alulcsordulási hiba kevésbé súlyos, mint a túlcsordulási hiba, mert a 0 nagyon gyakran jó becslése a 3-as, 5-ös régióban lévő számoknak. A 10^{-102} dollár banki egyenlege alig jobb, mint 0.

Egy másik jelentős különbség a lebegőpontos és a valós számok között a sűrűség. Bármely két valós x és y szám között van másik valós szám, függetlenül attól, hogy milyen közel van x és y. Ez a tulajdonság abból a tényből következik, hogy bármely két nem azonos valós x és y szám között a $z = (x + y)/2$ is valós. A valós számok folytonos számosságot alkotnak.

A lebegőpontos számok számossága ezzel ellentétben nem végtelen. Pontosan 179 100 pozitív szám fejezhető ki az öt számjegyű (3 + 2), két előjeles rendszerben, amelyet fentebb használtunk, 179 100 negatív szám és 0 (amelyet többféleképpen kifejezhetünk), ez összesen 358 201 számot jelent. A végtelen sok -10^{+100} és $+0{,}999 \times 10^{99}$ közötti valós szám közül ezzel a jelöléssel csak 358 201-et tudunk felírni. Ezt szimbolizálják a pontok a B.1. ábrán. Könnyen lehetséges, hogy egy számítás eredménye egy más szám lesz, annak ellenére, hogy a 2-es vagy a 6-os régióban van. Például $+0{,}100 \times 10^3$ harmadát nem tudjuk *pontosan* kifejezni pontosan a mi ábrázolásunkkal. Ha a számítás eredményét nem tudjuk megadni az adott ábrázolásban, akkor kézenfekvő, hogy a legközelebbi ábrázolható számot használjuk helyette. Ezt az eljárást **kerekítésnek** (**rounding**) nevezzük.

A szomszédos ábrázolható számok közötti távolság nem állandó a 2-es és a 6-os régióban. Ez a távolság $+0{,}998 \times 10^{99}$ és $+0{,}999 \times 10^{99}$ között lényegesen nagyobb, mint $+0{,}998 \times 10^{0}$ és $+0{,}999 \times 10^{0}$ között. Azonban, ha a szeparációt egy szám és az őt követő között az adott szám százalékos arányával fejezünk ki, akkor a teljes 2-es és a 6-os régióban nem lesz lényeges eltérés. Más szavakkal kifejezve, a kerekítésnél elkövetett **relatív hiba (relative error)** hozzávetőlegesen ugyanakkora a kis számokra, mint a nagy számokra.

Bár az előző tárgyalás a három jegyű törtrészt és a két jegyű exponenst ábrázoló rendszerben történt, a kapott következtetések érvényesek más rendszerekre is. Ha lecseréljük a törtrész vagy az exponens jegyeinek számát, csupán a 2-es, 6-os régió határait toljuk el, és változik a bennük ábrázolható pontok (számok) száma. Növelve a törtrész számjegyeinek számát, a pontok sűrűsége javulni fog, és ezért javulni fog a közelítés pontossága is. Ha növeljük az exponens számjegyeinek számát, akkor a 2-es és 6-os régió nagysága fog növekedni, és csökken az 1, 3, 5 és 7-es régió. A B.2. ábra mutatja a különböző exponenssel és törtrésszel ábrázolt számok becsült határait.

A fenti ábrázolások különböző variációit használják a számítógépekben. A hatékonyság kedvéért 2, 4, 8 vagy 16, és nem 10 alapú kitevőt használnak, a törtrész pedig kettes, négyes, nyolcas vagy tizenhatos számrendszerbeli számjegyek sorozatát jelenti. Ha a bal szélső számjegy 0, akkor az egész számot balra lehet tolni egy jeggyel, a kitevőt pedig eggyel csökkenteni anélkül, hogy a szám értéke megváltozna (eltekintve az alulcsordulástól). Ha a bal szélső jegy nem nulla, akkor ezt az ábrázolást **normalizáltnak** (**normalized**) nevezzük.

Törtrész számjegyei	Kitevő számjegyei	Alsó korlát	Felső korlát
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

B.2. ábra. *Az ábrázolható (nem normalizált) számok közelítő alsó és felső határa lebegőpontos decimális számok esetén*

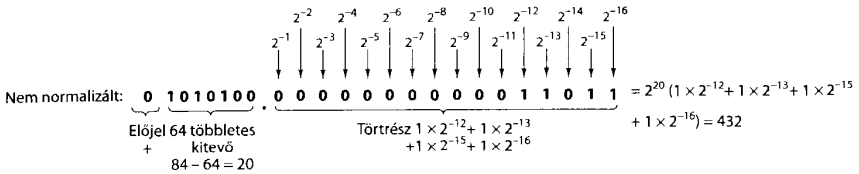
A normalizált számok általában előnyben részesítendők a nem normalizált számokkal szemben, mert csak egyetlen normalizált forma van, szemben a sok nem normalizált formával. A B.3. ábra normalizált számokat mutat be két különböző hatványalappal. Ezekben a példákban a törtrészt 16 bit hosszú (amely az előjelbitet is tartalmazza) és a kitevő 7 bit hosszú, 64 többletes ábrázolásban. Az alapszám pont (tizedes pont) a törtrész bal szélső bitjének bal oldalán van – azaz a kitevőtől jobbra.

B.2. Az IEEE 754-es lebegőpontos szabvány

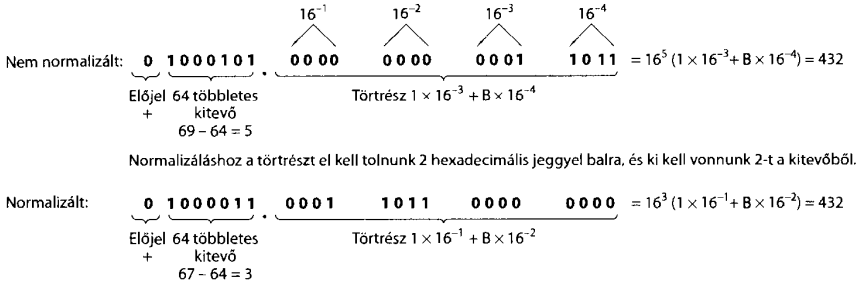
Körülbelül 1980-ig minden számítógépgyártónak saját lebegőpontos formátuma volt. Szükségtelen említeni, hogy természetesen mindegyik más és más. Ennél is rosszabb, hogy néhányan hibás aritmetikát csináltak, mert a lebegőpontos számítási módszernek van néhány finomsága, ami egy átlagos hardvertervező számára nem nyilvánvaló.

A helyzet javítására 1970-es évek végén az IEEE felállított egy bizottságot a lebegőpontos számítások szabványosítására. A cél nemcsak az volt, hogy olyan lebegőpontos adatokat tervezzenek, amelyek kicserélhetők a különböző számítógépek között, hanem az is, hogy egy helyes, jól működő modellt biztosítsanak a hardvertervezők számára. Ez a munka vezetett el az IEEE 754-es szabványhoz (IEEE, 1985). A legtöbb központi egység napjainkban (beleértve az Intel-, SPARC- és JVM-processzorokat, amelyeket ebben a könyvben tanulmányozunk) olyan lebegőpontos utasításokat használ, amelyek IEEE lebegőpontos szabványú adatokkal dolgoznak. Más szabványokkal ellentétben, amelyek beérik „nesze semmi,

1. Példa: 2-es alapú hatvány

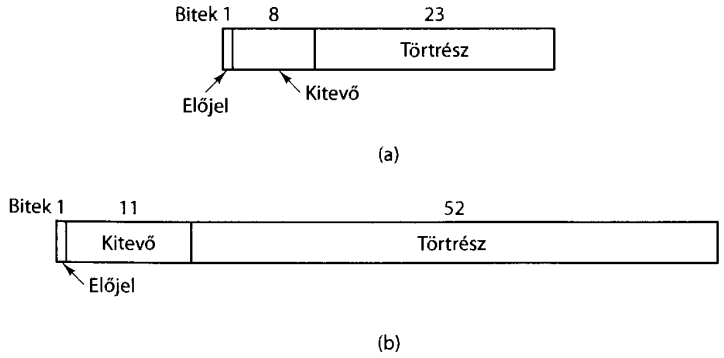


2. Példa: 16-os alapú hatvány



B.3. ábra. *Példák a normalizált lebegőpontos számokra*

fogd meg jól" kompromisszumokkal, amelyek senkinek se tetszenek, ez a szabvány nem számít rossznak, mivel alapvetően egy embernek, William Kahannek, a Berkeley Egyetem professzorának a munkája. A továbbiakban ezt a szabványt mutatjuk be.



B.4. ábra. *IEEE lebegőpontos formátumok: (a) egyszeres pontosságú; (b) dupla pontosságú*

A szabvány három formát definiál: az egyszeres pontosságút (32 bit), a dupla pontosságút (64 bit) és a kiterjesztett pontosságút (80 bit). A kiterjesztett pontosságú forma arra szolgál, hogy csökkentse a kerekítési hibákat. Ezt főleg a lebegőpontos aritmetikai egység belsejében használják, így nem beszélünk róla a továbbiakban. Az egyszeres és a dupla pontosságú forma egyaránt kettes alapot használ a törtrészben, és többletes ábrázolást a kitevőben. Formáját megtaláljuk a B.4. ábrán.

Mindkét forma egy előjelbittel kezdődik, ahol 0 a pozitív, 1 pedig a negatív előjel. Ezután következik a kitevő, 127 többletes a szimpla pontosságú és 1023 többletes dupla pontosságú ábrázolásban. A minimális (0) és a maximális (255, illetve 2047) kitevőket nem használjuk normalizált számok esetén, ezek használata speciális, amelyet a későbbiekben írunk le. Végül a törtrészek következnek 23, illetve 52 biten.

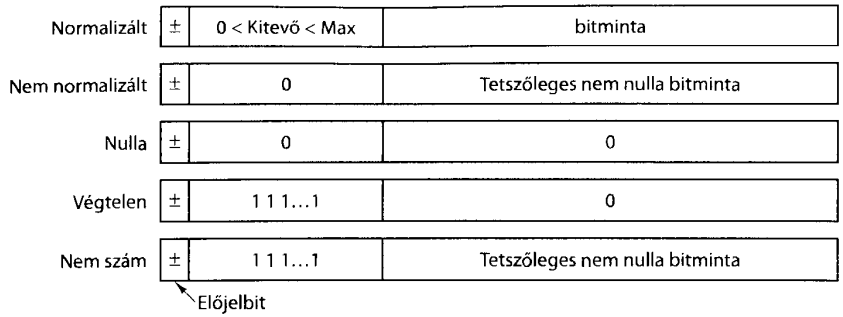
A normalizált törtrész bináris ponttal kezdődik, amelyet 1-es bit követ, és ezután a törtrész további része következik. A PDP–11-nél elkezdett gyakorlatot követve, a szabvány szerzői azt valósították meg, hogy a vezető 1-es bit a törtrészben nincs tárolva, feltételezzük, hogy jelen van. Ennek következtében a szabvány egy kicsit más módon definiálja a törtrészt, mint ahogy szokásos. Tartalmaz egy nem tárolt 1 bitet (implicit, feltételezett 1-es), a nem tárolt (implicit) bináris pontot, azután pedig 23, illetve 52 tetszőleges bitet. Ha a 23, illetve 52 bites törtrész végig nulla, a tört numerikus értéke 1,0; ha az összes 1-es, akkor a törtrész kicsit kevesebb, mint 2,0. Hogy elkerüljük a hagyományos törtrésszel (fraction) való összetévesztést, az implicit 1-es, az implicit bináris pont és a 23, illetve 52 bit együttesét **szignifikáns**nak (**significand**, meghatározó, fontos) nevezzük a törtrész vagy a mantissza megnevezés helyett. Az összes normalizált számnak van egy ilyen s szignifikánsa, ahol $1 \leq s < 2$.

Az IEEE lebegőpontos számok numerikus jellemzőit a B.5. ábrán láthatjuk. Példaként vegyük a 0,5, 1 és 1,5 számok normalizált, egyszeres pontosságú alakját. Ezeket sorrendben hexadecimálisan így ábrázoljuk: 3F000000, 3F800000 és 3FC00000.

A lebegőpontos számábrázolás hagyományos problémája, hogy hogyan kezeljük az alulcsordulást, túlcsordulást és kezdeti érték nélküli (inicializálatlan) számokat. Az IEEE szabvány ezekkel kifejezetten foglalkozik, és a CDC 6600-as szá-

Jellemzők	Egyszeres pontosság	Dupla pontosság
Bitek száma az előjelben	1	1
Bitek száma a kitevőben	8	11
Bitek száma a törtrészben	23	52
Az összes bit	32	64
Kitevő rendszere	127 többletes	1023 többletes
Kitevő kiterjedése	–126-tól +127-ig	–1022-től +1023-ig
Legkisebb normalizált szám	2^{-126}	2^{-1022}
Legnagyobb normalizált szám	kb. 2^{128}	kb. 2^{1024}
Decimális kiterjedés	kb. 10^{-38}-tól 10^{38}	kb. 10^{-308}-tól 10^{308}
Legkisebb nem normalizált szám	kb. 10^{-45}	kb. 10^{-324}

B.5. ábra. *Az IEEE lebegőpontos számok jellemzői*



B.6. ábra. *IEEE numerikus típusok*

mítógép példáját használja ebben a részben. A normalizált számokon túlmenően a szabványban négy további numerikus típus van, ezeket írjuk le a következőkben és mutatjuk be a B.6. ábrán.

Probléma adódik, ha a számítási eredmény abszolút értéke kisebb a legkisebb normalizált lebegőpontos számnál, amelyet az adott rendszerben ábrázolhatunk. Korábban a legtöbb hardver az alábbi két megoldás valamelyikét alkalmazta: vagy beállította az eredményt nullának, és így folytatta, vagy pedig lebegőpontos alulcsordulási csapdát okozott. Egyik sem teljesen kielégítő, így az IEEE bevezette a **normalizálatlan (nem normalizált, denormalized) számok**at. Ezeknek a számoknak az exponense 0, ezt követi a törtrész 23 vagy 52 biten. A bináris pont bal oldalán levő implicit 1-es bit most 0-ra változik. A nem normalizált számokat meg tudjuk különböztetni a normalizáltaktól, mert az utóbbinál nem megengedett, hogy az exponens 0 legyen.

A legkisebb egyszeres pontosságú normalizált szám a kitevő részen 1, a törtrészen pedig 0, így $1{,}0 \times 2^{-126}$-t ábrázolja. A legnagyobb nem normalizált szám 0 kitevőjű, a törtrésze pedig csupa 1-esből áll, így a $0{,}9999999 \times 2^{-126}$-t reprezentálja, ami közel ugyanaz az érték. Egy dolgot azonban jegyezzünk meg, hogy ennek a számnak csak 23 szignifikáns bitje van, ezzel szemben a normalizált számoknak 24.

Mivel a számítások tovább csökkenthetik az eredményt, a kitevő 0-ra állítódik, de az első néhány biten a törtrész is 0-vá válik, ezzel csökken mind az értéke, mind a szignifikáns bitek száma a törtben. A legkisebb nullánál nagyobb nem normalizált szám jobb szélső bitje 1, és az összes többi pedig 0. A kitevő –126, a törtrész pedig 2^{-23}, így az érték 2^{-149}. Ez a megoldás egy elegáns alulcsordulást biztosít ahelyett, hogy 0-ra ugrana, ha az eredmény nem írható fel normalizált számmal.

A 0 két módon ábrázolható ebben a sémában, egy pozitív és egy negatív nulla, amelyet az előjelbit határoz meg. Mind a kettőnek a kitevője és a törtrésze is nulla. Itt is 0 a bináris pont bal oldalán levő implicit bit, és nem 1.

A túlcsordulást nem lehet jól kezelni. Nem maradt (szabad) bitkombináció. Ehelyett, egy speciális ábrázolását adják a végtelennek: a kitevő csupa 1 (amely nem megengedett a normalizált számoknál), a törtrész pedig 0. Ez a szám használható műveletek operandusaként, és ugyanúgy viselkedik, mint a hagyományos

matematikai szabályoknál a végtelen. Például végtelen plusz bármi, az végtelen, és bármelyik véges számot elosztjuk végtelennel, az nulla. Hasonlóan bármely véges szám osztva nullával végtelent eredményez.

Mi van, ha a végtelent osztjuk végtelennel? Az eredmény meghatározatlan. Ennek az esetnek a kezelésére más speciális forma szolgál, amelyet **NaN**-nak (**Not a Number, nem szám**) hívnak. Ez szintén használható operandusként előre megjósolható eredménnyel.

B.3. Feladatok

1. Konvertáljuk a következő számokat IEEE egyszeres pontosságú formába. Az eredményeket nyolc jegyű hexadecimális számként adjuk meg.
   a) 9
   b) 5/32
   c) –5/32
   d) 6,125
2. Konvertáljuk a következő egyszeres pontosságú IEEE lebegőpontos hexadecimális számokat decimális számokká:
   a) 42E48000H
   b) 3F880000H
   c) 00800000H
   d) C7F00000H
3. Az IBM 370-es gépen az egyszeres lebegőpontos számoknak 64 többletes rendszerben megadott 7 bites kitevője van. A törtrész 24 bitet és 1 előjelbitet tartalmaz, a bináris pont a törtrész bal oldalán van. Az alapszám a kitevőben 16. A mezők sorrendje: előjelbit, kitevő, törtrész. Fejezzük ki a 7/64-et mint normalizált számot ebben a rendszerben hexadecimális formában.
4. A következő bináris lebegőpontos számok egy előjelbitet, 64 többletes kitevőt és 16 bites törtrészt tartalmaznak, az exponens alapszáma 2. Normalizáljuk a következő számokat:
   a) 0 1000000 0001010100000001
   b) 0 0111111 0000001111111111
   c) 0 1000011 1000000000000000
5. Két lebegőpontos szám összeadásához (a törtrész eltolásával) az exponenseket azonossá kell tennünk. Ezután tudjuk összeadni a törtrészeket, és normalizálni az eredményt, ha szükséges. Adjuk össze az IEEE egyszeres pontosságú 3EE00000H és 3D800000H számokat, és adjuk meg hexadecimálisan a normalizált eredményt.
6. A Tightwad („fösvény") Computer Company elhatározta, hogy előállít egy számítógépet 16 bites lebegőpontos számokkal. A Model 0.001 lebegőpontos formája: előjelbit, 7 bites 64 többletes exponens és 8 bites törtrész. A Model 0.002-é: előjelbit, 5 bites 16 többletes exponens és 10 bites törtrész. Mindkettő 2-es alapszámot használ. Melyik a legkisebb és a legnagyobb pozitív normali-

zált szám a két modellben? Körülbelül hány decimális jegyű a pontosság mindegyiknél? Meg venné ezek valamelyikét?

7. Előfordul, hogy két lebegőpontos számon végrehajtott műveletnél az eredmény szignifikáns bitjeinek számában drasztikus csökkenés áll be. Mikor?
8. Néhány lebegőpontos lapka (digitális áramkör) beépített gyökvonást tartalmaz. Egy lehetséges algoritmus egy iteratív (például a Newton–Raphson-) algoritmus. Iteratív algoritmusok egy kezdő becslésből indulnak ki, és ezt állandóan javítják. Hogyan kaphatnánk gyors becslést a lebegőpontos számok négyzetgyökére?
9. Írjunk eljárást, amely összead két IEEE egyszeres pontosságú lebegőpontos számot. Mindegyik szám 32 elemű Boole-tömbben van ábrázolva.
10. Írjunk eljárást, amely összead két egyszeres pontosságú számot, amelyek kitevője 16-os alapszámra vonatkozik, törtrésze 2-es számrendszerű, de nincs implicit 1-es bit a bináris pont bal oldalán. A normalizált szám törtrészének baloldali 4 bitje 0001, 0010, ..., 1111, de nem 0000. A számot úgy normalizáljuk, hogy a törtrészt balra toljuk 4 bittel, és 1-et adunk hozzá az exponenshez.

C) Assembly nyelvű programozás

Evert Wattel
Vrije Universiteit
Amszterdam, Hollandia

Minden számítógéphez létezik egy **ISA (Instruction Set Architecture, Utasításrendszer-architektúra)**, amely regiszterek, utasítások és egyéb, az alacsony szinten programozó személy számára látható tulajdonságok együttese. Az ISA-t gyakran **gépi nyelv**nek is nevezik, bár ez a kifejezés nem teljesen pontos. Ezen az absztrakciós szinten a program bináris számoknak egy hosszú sorozata, utasításonként egy szám, amely megmondja, hogy milyen utasítást kell végrehajtani, és mik az operandusok. A bináris számokkal való programozás nagyon nehézkes, ezért minden gép rendelkezik egy **assembly nyelv**vel, amely az utasításrendszer-architektúra szimbolikus reprezentációja, a bináris számok helyett olyan szimbolikus nevekkel, mint az ADD, SUB és MUL. Ez a függelék egy konkrét gép, az Intel 8088 assembly nyelvű programozását mutatja be. Ezt a processzort használták az eredeti IBM PC-ben is, és ebből fejlődött ki a modern Pentium is. A függelék bemutatja néhány, az assembly nyelvű programozás tanulását segítő, letölthető eszköz használatát is.

E függelék célja nem az, hogy tökéletes assembly nyelvű programozókat képezzen, hanem, hogy kézzelfogható tapasztalatokkal segítse az olvasót a számítógép-architektúrák tanulmányozásában. Ezért választottunk példaként egy egyszerű processzort, az Intel 8088-at. Bár a 8088-assal már csak elvétve találkozhatunk, minden Pentium képes a 8088-as programjait futtatni, így az itt megtanultak a modern gépek esetén is alkalmazhatók. Ezen kívül a Pentium alaputasításainak többsége megegyezik a 8088-aséval, csak éppen 32 bites regisztereket használnak 16 bitesek helyett. Így ez a függelék tekinthető egy gyenge bevezetésnek is a Pentium assembly nyelvű programozásába.

Bármely számítógép assembly nyelvű programozásához a programozónak részletesen kell ismernie a gép utasításrendszer-architektúráját. Ennek megfelelően, a C.1-től C.4-ig tartó alfejezeteket a 8088-as architektúrájának, memóriaszervezésének, címzési módjainak, valamint utasításainak szenteljük. A C.5. alfejezet az ebben a függelékben használt, szabadon hozzáférhető (lásd később) assemblert mutatja be. A függelék jelölésrendszere ehhez az assemblerhez igazodik. Más assemblerek más jelölésrendszert használnak, ezért a 8088-as assembly programozásában jártas olvasóink figyeljenek az eltérésekre. A C.6. alfejezetben egy letölthető értelmező/nyomkövető/hibakereső (interpreter/tracer/debugger) eszközt

mutatunk be, amely segíthet a kezdő programozóknak programhibáik felderítésében. A C.7. alfejezetben az eszközök telepítését és használatba vételének módját írjuk le. A C.8. alfejezet programokat, példákat, gyakorlatokat és megoldásokat tartalmaz. A C.9. alfejezetben pedig megvalósítási kérdésekkel, hibákkal és az anyagrész korlátaival foglalkozunk.

C.1. Áttekintés

Az assembly nyelvű programozásban tett utazásunk elején ejtünk néhány szót az assembly nyelvről, és egy rövid példával illusztráljuk is.

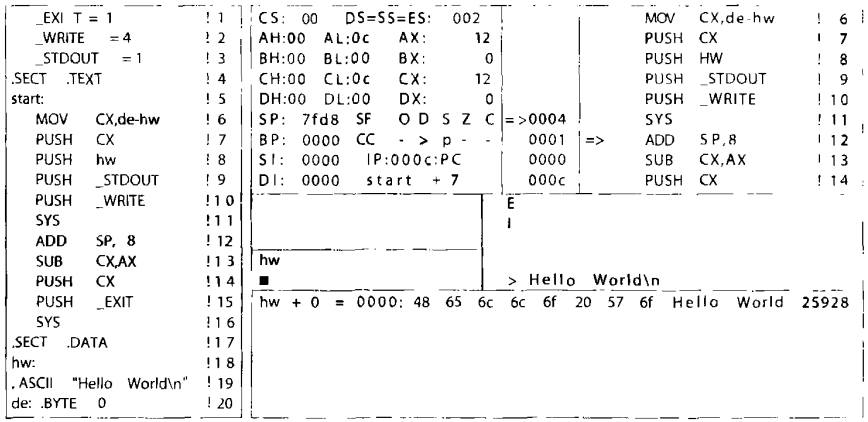
C.1.1. Az assembly nyelv

Minden assembler könnyen megjegyezhető **mnemonik**okat, azaz rövid szavakat használ, ilyenek például az ADD, SUB és MUL, rendre az összeadás, kivonás és szorzás gépi utasításokra. Ezen kívül az assemblerekben **szimbolikus neveket** használhatunk konstansokra, és **címkék**kel jelölhetünk meg utasításokat és memóriarekeszeket. A legtöbb assembler támogatja **pszeudoutasítások** használatát is, amelyekből nem ISA-utasítás keletkezik, hanem az assemblert vezérlik a fordítási folyamat során.

Az **assembler**nek nevezett program a számára átadott assembly nyelvű programot tényleges végrehajtásra alkalmas **bináris program**má alakítja. Az így előállt programot lehet a tényleges hardveren futtatni. A kezdő assembly nyelven programozók azonban gyakran ejtenek hibákat, és ilyenkor a bináris program egyszerűen megáll, a programozónak pedig semmi elképzelése sincs a hibáról. A kezdők életének megkönnyítésére gyakran lehetőség van arra, hogy a bináris programot ne a konkrét hardveren futtassuk, hanem szimulátoron, amely egyszerre egy utasítást hajt végre, és részletes tájékoztatást ad arról, hogy éppen mi is történik. Ily módon sokkal könnyebb a hibakeresés. A programok természetesen nagyon lassan futnak a szimulátoron, de amikor a cél az assembly nyelvű programozás megtanulása, nem pedig a termékfejlesztés, akkor ez a sebességveszteség nem lényeges. Ez a függelék egy olyan eszközre épít, amely tartalmaz egy ilyen szimulátort is. Ezt **értelmező (interpreter)** vagy **nyomkövető (tracer)** néven szokták emlegetni, mivel a szimulátor lépésről lépésre értelmezi, nyomon követi a bináris program futását. A szimulátor, értelmező és nyomkövető elnevezéseket egymással felcserélhetőkként fogjuk használni ebben a függelékben. Általában akkor beszélünk értelmezőről, amikor csak a program futtatásáról van szó. Abban az esetben, ha hibakereső eszközként használjuk, nyomkövető néven beszélünk ugyanarról a programról.

C.1.2. Egy rövid assembly nyelvű program

Azért, hogy ezeket az elvont fogalmakat kicsit konkrétabbá tegyük, tekintsük a C.1. ábrán látható programot és a hozzá tartozó nyomkövető képet. A C.1. (a) ábra egy egyszerű 8088-as assembly nyelvű programot mutat. A felkiáltójelek után található számok a forrásprogrambeli sorok számát jelölik, így könnyebb a program különböző részeire hivatkozni. A program megtalálható a kiegészítő anyagban az *examples* alkönyvtárban a *HlloWrld.s* nevű forrásfájlban. A mellékletben található többi assembly nyelvű programhoz hasonlóan ez is *.s* kiterjesztéssel bír, ami assembly nyelvű forrásprogramot jelöl. A nyomkövető képernyője a C.1. (b) ábrán látható.* Hét ablakot tartalmaz, mindegyik más jellegű információt mutat az éppen futtatás alatt álló bináris program állapotáról.



C.1. ábra. *(a) Egy assembly nyelvű program. (b) A hozzá tartozó nyomkövető képernyő*

Most röviden vizsgáljuk meg a C.1. (b) ábrán látható hét ablakot. Felül három ablak található, két nagyobb és közöttük egy kisebb harmadik. A bal felső ablak a processzor tartalmát mutatja, azaz a CS, DS, SS és ES szegmensregiszterek, az AH, AL, AX aritmetikai regiszterek és egyéb regiszterek tartalmát.

A felső sor középső ablakában látható a verem, azaz az átmeneti értékek tárolására használt memória.

A jobb felső ablak az assembly nyelvű program egy részletét mutatja, a nyíl pedig az éppen végrehajtás alatt álló utasítást jelöli. A program futása közben, ahogy változik az aktuális utasítás, a nyíl elmozdul, hogy mutassa azt. A nyomkövető erőssége abban rejlik, hogy a RETURN (PC-billentyűzeten többnyire ENTER felirattal

* Az ábrák néhol kissé eltérnek a CD-mellékletben lévő programok futtatásával nyerhető ábráktól. *(A lektor)*

jelzett) billentyűt leütve végrehajtódik egy utasítás, és minden ablak tartalma frissül, ezzel lehetővé téve a program lassított végrehajtását.

A bal oldali ablak alatt található a szubrutin hívások verme, amely esetünkben üres. Ez alatt a nyomkövetőnek szóló parancsok láthatók. Ezektől jobbra helyezkedik el egy ablak beviteli és kiviteli célokra, valamint hibaüzenetek megjelenítésére.

Az alsó ablak a memória egy részletét mutatja. Ezekkel az ablakokkal később részletesen foglalkozunk, de már most is nyilvánvaló az alapötlet: a nyomkövető mutatja a forrásprogramot, a gép regisztereit és még jó néhány információt a futtatás alatt álló program állapotáról. Ahogy az egyes utasítások végrehajtódnak, az információ is frissítésre kerül, így a felhasználó elég részletesen láthatja, hogy mit is csinál a program.

C.2. A 8088-as processzor

Minden processzor, beleértve a 8088-ast is, rendelkezik egy belső állapottal, ahol néhány lényeges információt tárol. Erre a célra a processzor egy **regiszter**készlettel rendelkezik, amelyben ez az információ tárolható és feldolgozható. Valószínűleg a legfontosabb ezek közül a **PC** (**program counter, programszámláló** vagy **utasításszámláló**), amelyik a következőként végrehajtandó utasítás memóriahelyét, azaz **címét** tartalmazza. Erre a regiszterre IP (instruction pointer, utasításmutató) néven is szoktak hivatkozni. Ez az utasítás a központi memória egy részében, a **kódszegmens**ben helyezkedik el. A 8088-as központi memóriája akár 1 MB is lehet, az aktuális kódszegmens mérete azonban csak 64 KB. A CS regiszter megmutatja, hogy hol kezdődik a 64 KB-os kódszegmens az 1 MB-os memóriában (C.1. ábra). A CS regiszter egyszerű módosításával lehet új kódszegmenst kiválasztani. Hasonlóan, létezik egy 64 KB-os adatszegmens is, amely megmutatja, hol kezdődnek az adatok. Ennek a kezdetét a DS regiszter adja meg, amely szükség szerint szintén megváltoztatható, ha az aktuális adatszegmensen kívüli adathoz kell hozzáférni. A CS és DS regiszterekre azért van szükség, mert a 8088-as 16 bites regiszterekkel dolgozik, így azok nem tudják közvetlenül tárolni az 1 MB-os memória címzéséhez szükséges 20 bites címeket. Ezért vezették be a kód- és adatszegmensregisztereket.

A többi regiszter adatot vagy mutatót tartalmaz, amely valamely adatnak a központi memóriában elfoglalt helyére mutat. Assembly nyelvű programokban ezek a regiszterek közvetlenül elérhetők. E regisztereken kívül a processzorban megtalálható még az utasítások végrehajtásához szükséges minden egyéb dolog is, de ezek a részek a programozó számára csak az utasításokon keresztül érhetők el.

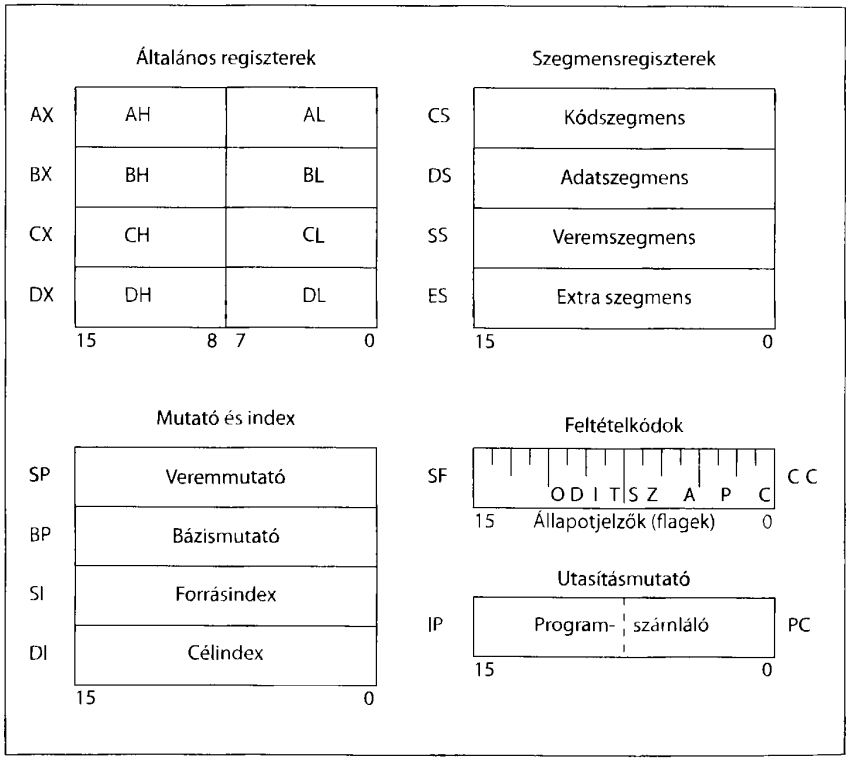
C.2.1. A processzorciklus

A 8088-as (és minden más számítógép) működése utasítások egymás utáni végrehajtásából áll. Egy utasítás végrehajtása az alábbi lépésekre bontható:

1. Az utasítás betöltése a memóriából a kódszegmensből a PC alkalmazásával.
2. A programszámláló növelése.
3. A betöltött utasítás dekódolása.
4. A szükséges adatok betöltése a memóriából és/vagy a processzor regisztereiből.
5. Az utasítás végrehajtása.
6. Az utasítás eredményének tárolása a memóriában vagy regiszterekben.
7. Vissza az 1. lépésre a következő utasítás megkezdéséhez.

Egy utasítás végrehajtása kicsit hasonlít egy kis program futtatása. Bizonyos gépek tényleg egy kicsi programmal, az ún. **mikroprogram**mal, hajtják végre az utasításaikat. A mikroprogramokat a 4. fejezet tárgyalja részletesen.

Az assembly programozó szempontjából a 8088-as 14 regiszterrel rendelkezik. Ezek a regiszterek bizonyos értelemben egy jegyzettömböt alkotnak, ahol az utasítások működnek, és ezért folyamatos használatban vannak, az itt tárolt eredmények azonban nagyon illékonyak. Ennek a 14 regiszternek az áttekintése látható a C.2. ábrán. Nyilvánvaló, hogy ez az ábra és a C.1. ábrán a nyomkövető regiszterablaka nagyon hasonló, hiszen ugyanazt az információt jeleníti meg.



C.2. ábra. *A 8088-as regiszterei*

A 8088-as regiszterei 16 bit szélesek. Nincs két funkcionálisan teljesen ekvivalens regiszter, de vannak közöttük hasonló tulajdonságokkal bírók, ezért vannak a C.2. ábrán csoportokba osztva. Most nézzük meg a különböző csoportokat.

C.2.2. Az általános regiszterek

Az első csoport regiszterei, az AX, BX, CX, és DX az általános regiszterek. A csoport első tagját, az AX regisztert **akkumulátorregiszter**nek is nevezik. Számítási eredmények tárolására használjuk, és számos utasítás eredménye itt képződik. Jóllehet minden regiszter sokféle művelet operandusa lehet, néhány utasításnál, például a szorzásnál, az AX az implicit céloperandus.

A csoport második regisztere a BX, a **bázisregiszter**. Az AX-hez hasonlóan a BX sokféle célra használható, de van egy olyan erőssége, ami az AX-nek nincs. A BX-ben elhelyezhetünk egy memóriacímet, és utána olyan utasítást hajthatunk végre, amelynek az operandusa a BX-ben tárolt memóriacímről jön. Más szóval, a BX tárolhat egy memóriamutatót, az AX nem. Ennek szemléltetésére összehasonlítunk két utasítást. Először a

```
MOV AX,BX
```

amely a BX tartalmát az AX-be másolja. Másodszor a

```
MOV AX,(BX)
```

amely annak a memóriaszónak a tartalmát másolja az AX-be, amelynek címét a BX tartalmazza. Az első példában a BX a forrásoperandust tartalmazza, míg a másodikban a forrásoperandusra mutat.

A következő általános regiszter a CX, a **számlálóregiszter (counter)**. Számos egyéb feladat elvégzése mellett ezt a regisztert használjuk ciklusok számlálóinak tárolására. Értéke a LOOP utasítás hatására automatikusan csökken eggyel, és a ciklusok általában akkor érnek véget, amikor a CX eléri a 0-t.

Az általános regiszterek csoportjának negyedik tagja a DX, az **adatregiszter (data register)**. Ezt használják az AX regiszterrel együtt a dupla szó (32 bites) hosszúságú utasítások. Ebben az esetben a DX tartalmazza a magas helyértékű 16 bitet, az AX pedig az alacsony helyértékű 16 bitet. Általában a 32 bites egészekre a **hosszú (long)** elnevezést használjuk. A **double (kétszeres, dupla)** elnevezést általában a 64 bites lebegőpontos értékekre tartjuk fent, bár egyesek a „double" elnevezést a 32 bites egészekre is használják. Ebben a leírásban nem lesz félreértés, mert egyáltalán nem foglalkozunk a lebegőpontos számokkal.

Mindegyik általános regiszter tekinthető egy 16 bites regiszternek vagy két 8 bites regiszterből álló párnak. Így a 8088-as pontosan nyolc különböző 8 bites regiszterrel rendelkezik, amelyeket bájt- és karakterutasításokban használhatunk. A többi regiszter nem osztható 8 bites részekre. Néhány utasítás az egész regisztert használja, például AX-et, míg mások csak a regiszter felét, például AL-t vagy AH-t.

Általában az aritmetikai utasítások a teljes 16 bites regisztereket használják, a karakterkezelő utasítások azonban a 8 biteseket. Fontos azonban tisztában lenni azzal, hogy az AL és AH csak az AX két felének elnevezései. Amikor az AX-be új érték töltődik, az AL és AH is megváltozik az AX-be töltött 16 bites szám alsó és felső felére. Hogy lássuk az AX, AH és AL kölcsönhatását, tekintsük a

```
MOV AX,258
```

utasítást, amely az AX regiszterbe tölti a 258 decimális értéket. Az utasítás után az AH bájtregiszter az 1 értéket, az AL bájtregiszter pedig a 2 értéket tartalmazza. Ha ezt az

```
ADDB AH,AL
```

bájtösszeadás utasítás követi, akkor az AH bájtregiszter az AL értékével (2) nő, azaz a tartalma 3 lesz. Ennek az a hatása az AX regiszterre, hogy annak értéke most 770, amely ekvivalens a 00000011 00000010 bináris, vagy a 0x03 0x02 hexadecimális jelölésű számmal. A nyolc bájtszélességű regiszterek majdnem felcserélhetők, azzal a kivétellel, hogy a MULB utasításban az egyik operandust mindig az AL tartalmazza, és az AH-val együtt ez ennek az utasításnak az implicit céloperandusa is. A DIVB utasítás az osztandóra szintén az AH : AL párost használja. Eltoló és forgató utasításokban a lépések száma a számlálóregiszter alsó bájtjában, a CL-ben tárolható.

A C.8. fejezet 2. példája a *GenReg.s* program tárgyalásán keresztül mutatja be az általános regiszterek néhány tulajdonságát.

C.2.3. Mutatóregiszterek

A regiszterek második csoportja a **mutató- és indexregiszterek**. Ezek közül a legfontosabb a **veremmutató (stack pointer)**, amelynek jele SP. A vermek a legtöbb programozási nyelvben fontos szerepet töltenek be. A memória veremszegmense a futó program állapotáról tárol bizonyos információkat. Általában, amikor egy eljárást meghívnak, a verem egy részét fenntartják az eljárás lokális változóinak, az eljárás befejezése utáni visszatérési címnek és egyéb vezérlő információknak. A veremnek egy adott eljárással kapcsolatos részét az eljárás **veremkeret**ének **(stack frame)** hívjuk. Ha egy eljárás egy másik eljárást hív meg, egy újabb veremkeret jön létre, általában közvetlenül az aktuális alatt. További hívások újabb veremkereteket hoznak létre az eddigiek alatt. Bár nem kötelező érvénnyel, a verem szinte mindig lefelé nő, azaz a magas címektől az alacsonyabbak felé. Ennek ellenére, a veremben elfoglalt legalacsonyabb címet mindig a verem tetejének hívjuk.

A lokális változók tárolásán kívül a verem ideiglenes eredményeket is tárolhat. A 8088-as PUSH utasítása egy 16 bites szót helyez el a verem tetejére. Ez az utasítás először 2-vel csökkenti SP-t, majd az operandusát az SP által mutatott (új) címre helyezi. Hasonlóan, a POP eltávolít egy 16 bites szót a verem tetejéről úgy, hogy először kiolvassa a verem tetején található értéket majd 2-vel növeli SP-t. Az SP re-

giszter a verem tetejére mutat, és a PUSH, a POP, valamint a CALL utasítások módosítják: a PUSH csökkenti, a POP növeli, a CALL pedig szintén csökkenti.

A csoport következő regisztere a BP, a **bázismutató (base pointer)**. Ez többnyire a verem egy címét tartalmazza. SP-vel szemben, amely mindig a verem tetejére mutat, BP a vermen belül bárhova mutathat. Gyakorlatban általában BP az aktuális eljárás veremkeretének kezdetére mutat, ezáltal az eljárás lokális változói könnyen elérhetővé válnak. Így BP gyakran mutat az aktuális veremkeret aljára (a veremkeret legmagasabb értékű helyére), SP pedig a tetejére (a veremkeret legalacsonyabb értékű helyére). Az aktuális veremkeretet tehát BP és SP értéke határolja.

Ebben a regisztercsoportban található még két indexregiszter: az SI, a **forrásindex (source index)**, és a DI, a **célindex (destination index)**. Ezeket gyakran BP-vel kombinálva verembeli, vagy BX-szel kombinálva memóriabeli adatok címzésére használjuk. Ezeket a regisztereket a címzési módokról szóló fejezetben tárgyaljuk részletesebben.

Az egyik legfontosabb regiszter, amely önmaga alkot egy külön csoportot, az **utasításmutató (instruction pointer)**, amely az Intel elnevezése a programszámlálóra (program counter, PC). Ez a regiszter egy, a program kódszegmensében található címet tartalmaz, de utasításokkal nem lehet rá közvetlenül hivatkozni. A processzor utasításciklusa a PC által mutatott utasítás betöltésével kezdődik. Ezt a regisztert az utasítás további végrehajtása előtt növelik, így ezután a programszámláló az aktuálist követő első utasításra mutat.

A **flag-regiszter** vagy **feltételkód- (condition code)** vagy PSW **(Program Status Word, programállapotszó) regiszter** valójában egybites regiszterek halmaza. Néhány bitet aritmetikai utasítások állítanak be, és az eredménnyel kapcsolatosak:

Z – az eredmény nulla (zero);
S – az eredmény negatív (előjel, sign bit);
V – az eredmény túlcsordulást okozott (overflow);
C – az eredmény átvitelt okozott (carry);
A – kiegészítő átvitel (a 3-as bitről) (auxilliary carry);
P – az eredmény paritása (parity).

A regiszter többi bitje a processzor működésének bizonyos aspektusait szabályozza. Az I bit a megszakításokat (interrupt) engedélyezi. A T bit a nyomkövető (trace) módot engedélyezi, amely hibakeresésre használatos. Végül a D bit vezérli a string műveletek irányát (direction). A flag-regiszter 16 bitje közül nem mindegyik van használatban, a használaton kívüliek mindig nullák.

A **szegmensregiszter-csoport**ban négy regiszter található. Emlékezzünk vissza, hogy a verem, az adat és az utasításkódok mind a központi memóriában, de annak általában különböző részeiben helyezkednek el. A szegmensregiszterek irányítják a memóriának ezeket a különböző részeit, a **szegmens**eket. A regiszterek neve CS, a kódszegmens (code segment), DS, az adatszegmens (data segment), SS, a veremszegmens (stack segment), és ES, az extra szegmens (extra segment) regiszter esetében. Az idő nagy részében ezek értékei nem változnak. Gyakorlatban

az adatszegmens és a veremszegmens ugyanazt a memóriaterületet használja, az adatszegmens a terület alján, veremszegmens a tetején kezdődik. Ezekről a regiszterekről részletesebben beszélünk a C.3.1. fejezetben.

C.3. Memória és címzés

A 8088-as kissé esetlen memória szervezése az 1 MB-os memória és a 16 bites regiszterek keveréséből adódik. 1 MB memória esetén egy memóriacímet 20 bittel ábrázolhatunk. Következésképpen lehetetlen egy memóriamutatót egyetlen 16 bites regiszterben tárolni. A probléma megkerülésére a memóriát 64 KB-os szegmensekre tagolják, így a szegmensen belüli címek elférnek 16 biten. Most nézzük meg részletesebben a 8088-as memória felépítését.

C.3.1. Memóriaszervezés és szegmensek

A 8088-as memóriája, mely egyszerűen csak megcímezhető 8bites bájtok tömbje, utasítások és adatok tárolására, valamint a verem számára használatos. Ezekre az eltérő célokra használt memóriarészek a szegmensek. A 8088-asban egy ilyen szegmens 65 536 egymást követő bájtból áll. A négy elkülönített szegmens:

1. A kódszegmens (code segment);
2. Az adatszegmens (data segment);
3. A veremszegmens (stack segment);
4. Az extra szegmens (extra segment).

A kódszegmens tárolja a program utasításait. A PC regiszter tartalmát mindig a kódszegmensen belüli memóriacímként értelmezzük. Ha a PC értéke 0, az a kódszegmens legalacsonyabb címét jelöli, nem pedig a valódi 0 memóriacímet. Az adatszegmens a program inicializált és inicializálatlan adatait tartalmazza. Amikor a BX egy mutatót tartalmaz, akkor azt ebben az adatszegmensben kell értelmezni. A veremszegmens lokális változókat és a verembe tárolt részeredményeket tárolja. Az SP és a BP regiszterekben tárolt címek mindig ebben a veremszegmensben helyezkednek el. Az extra szegmens egy tartalék szegmens, amely szükség szerint a memória bármely részén elhelyezkedhet.

Minden egyes szegmenshez tartozik egy szegmensregiszter: a 16 bites CS, DS, SS, és ES regiszterek. A szegmens kezdőcíme az a 20 bites előjel nélküli egész, amely úgy áll elő, hogy a szegmensregisztert balra léptetjük 4 bittel, majd a jobb szélső négy pozícióra 0-kat pótolunk. A szegmensregiszter a szegmens kezdetére (bázisára, alapjára) mutat. A szegmensen belüli címek úgy képezhetők, hogy a szegmensregiszter értékét a végén négy 0 bittel megtoldva valódi 20 bites címmé alakítjuk, majd hozzáadjuk az eltolás értéket (offset). Gyakorlatilag az abszolút memóriacímet a szegmensregiszter 16-szorosának és az eltolás értékének összege ad-

ja. Például, ha DS = 7, BX = 12, akkor a BX által jelölt cím 7 × 16 + 12 = 124. Más szóval, a DS = 7 által jelölt 20 bites bináris cím 00000000000001110000. Ehhez hozzáadva a 16 bites 0000000000001100 (decimális 12) eltolást a szegmens kezdetének címe 00000000000001111100 (decimális 124).

Minden memóriahivatkozásnál a tényleges cím képzéséhez felhasználjuk valamelyik szegmensregisztert. Ha egy utasítás direkt címet tartalmaz regiszterhivatkozás nélkül, akkor a cím automatikusan az adatszegmensben van, és a DS-t használjuk a szegmens kezdetének meghatározásához. A fizikai címet úgy találjuk meg, hogy ezt az alapot hozzáadjuk az utasításban szereplő címhez. A következő utasítás memóriabeli fizikai címét a CS regisztert 4 bináris hellyel léptetve, majd a programszámlálót hozzáadva kapjuk meg. Más szóval, először kiszámítjuk a 16 bites CS regiszter által kijelölt valódi 20 bites címet, majd ehhez hozzáadjuk a 16-bites PC értékét, hogy így előálljon a 20 bites abszolút memóriacím.

A veremszegmens 2 bájtos szavakból áll, tehát a veremmutató, az SP, mindig páros számot kell tartalmazzon. A verem a magas címektől az alacsonyak felé telítődik. Így tehát a PUSH utasítás 2-vel csökkenti a veremmutatót, majd az operandusát az SS és SP regiszterekből képezett memóriacímre tárolja. A POP utasítás visszatölti az értéket, majd 2-vel növeli az SP-t. A veremszegmensben szabadnak tekintendők azok a címek, amelyek az SP által jelöltnél kisebbek. A verem takarítása így elintézhető pusztán az SP növelésével. Gyakorlatban a DS és SS mindig azonosak, tehát egy 16 bites mutatóval hivatkozhatunk az osztott adat-/veremszegmensben tárolt változóra. Ha a DS és az SS különböznének, egy 17. bitre lenne szükség minden mutatóhoz, hogy megkülönböztessük az adatszegmensbe mutatókat a veremszegmensbe mutatóktól. Visszagondolva, valószínűleg hiba volt külön veremszegmenst egyáltalán bevezetni is.*

Ha a négy szegmensregiszterben található címet nagyon távolinak választjuk, akkor a szegmensek szétválnak, de ha a rendelkezésre álló memória korlátos, akkor nem szükséges szétválasztani. Fordítás után ismert a programkód mérete. Ilyenkor érdemes az adat és verem szegmenseket az utolsó utasítást követő első 16-tal osztható címre helyezni. Ezzel feltételezzük, hogy a kód- és adatszegmens soha nem használja ugyanazt a fizikai címet.

C.3.2. Címzés

Majdnem minden utasításnak szüksége van adatokra, vagy a memóriából, vagy a regiszterekből. Az adatok megadásához címzési módok viszonylag változatos gyűjteményét kínálja a 8088-as. Számos utasítás két operandust tartalmaz, amelyeket általában **céloperandus**nak **(destination)** és **forrásoperandus**nak **(source)** nevezünk. Gondoljunk például a másoló utasításra, vagy az összeadásra:

* A megvalósítás 64 KB adat- és 64 KB veremterület kijelölését teszi lehetővé. A szerző javasolta megoldásban a két terület együtt sem haladhatná meg a 64 KB-ot. *(A lektor)*


```
MOV AX,BX
```

vagy

```
ADD CX,20
```

Ezekben az utasításokban az első operandus a cél, a második pedig a forrás. (Önkényes a választás, hogy melyik van elöl. A fordított sorrend is elfogadható lett volna.) Természetesen ilyenkor a céloperandusnak **balérték**nek **(left value)** kell lennie, azaz olyan helynek, ahol valami tárolható. Ez azt jelenti, hogy konstans lehet forrásoperandus, de céloperandus nem.

Az eredeti terv szerint a 8088-asban a két operandus közül legalább az egyiknek regiszternek kell lennie. Ezzel az volt a cél, hogy a **szavas utasítások (word instructions)** és a **bájtos utasítások (byte instructions)** megkülönböztethetők legyenek azáltal, hogy a megadott regiszter **szavas regiszter** vagy **bájtos regiszter**. A processzor első kiadásában ezt oly mértékben megkövetelték, hogy egyenesen lehetetlen volt konstanst a veremre helyezni, mert abban az utasításban sem a forrásoperandus sem a céloperandus nem regiszter. A későbbi változatok kevésbé voltak szigorúak, de az ötlet mindenesetre kihatással volt az egész tervre. Néhány esetben az egyik operandust nem is említjük külön. Például a MULB utasításban csak az AX regiszter képes céloperandusként viselkedni.

Jó néhány egyoperandusú művelet is létezik, mint például a növelések, eltolások, negálások stb. Ilyen esetekben a regiszter nem követelmény, és a szavas és bájtos műveletek közötti különbséget csak a műveleti kódból (az utasítás típusából) lehet kikövetkeztetni.

A 8088-as négy alap adattípust támogat: az 1 bájtos **bájt (byte)**, a 2 bájtos **szó (word)**, a 4 bájtos **hosszú szó (long)** és a **binárisan kódolt decimális (binary coded decimal, BCD)**, amelyben egy szó két decimálist számjegyet tárol. Az utóbbi típust az értelmező nem támogatja.

A memóriacímek mindig bájtokra mutatnak, de szavak vagy hosszú szavak implicit módon mutatnak a megcímzett bájt feletti memóriahelyekre is. A 20-as címen található szó a 20-as és 21-es memóriahelyeket jelenti. A 24-es címen lévő hosszú szó a 24, 25, 26, és 27-es címeket foglalja el. A 8088-as **kis endián (little endian)** szervezésű, ami azt jelenti, hogy a szó alacsony helyértékű része van az alacsonyabb címen tárolva. A veremszegmensben a szavakat mindig páros címekre kell helyezni. A processzor regisztereiben a hosszú szavakra vonatkozó egyetlen intézkedés az DX : AX összetétel, amikor is az AX tárolja az alacsony helyértékű szót.

A C.3. ábra táblázata bemutatja a 8088-as címzési módjait. Tekintsük át ezeket röviden. A táblázat legfelső vízszintes blokkja a regisztereket sorolja fel. Ezek szinte minden utasításban használhatók akár forrás-, akár céloperandusként. Nyolc szavas és nyolc bájtos regiszter van.

A második vízszintes blokk az adatszegmens címzésének módjait tartalmazza. Az ilyen típusú címek mindig tartalmaznak egy zárójelpárt, ami azt jelöli, hogy a címen található tartalmat, nem pedig az értékét jelöljük. A legegyszerűbb ilyen

Mód	Operandus	Példák
Regisztercímzés		
Bájtos regiszter	Bájtos regiszter	AH,AL,BH,BL,CH,CL,DH,DL
Szavas regiszter	Szavas regiszter	AX,BX,CX,DX,SP,BP,Si,DI
Adatszegmenscímzés		
Direkt cím	A cím a műveleti kódot követi	(#)
Regiszter-indirekt	A cím regiszterben van	(SI), (DI), (BX)
Indexelt	A cím regiszter+eltolás	#(SI), #(DI), #(BX)
Bázis relatív	A cím BX+SI/DI	(BX)(SI), (BX)(DI)
Bázis relatív eltolással	A cím BX+SI/DI+eltolás	#(BX)(SI), #(BX)(DI)
Veremszegmenscímzés		
Bázismutató indirekt	A cím regiszterben van	(BP)
Bázismutató eltolás	A cím BP+eltolás	#(BP)
Stack relatív	A cím BP+SI/DI	(BP)(SI), (BP)(DI)
Stack relatív eltolással	A cím BP+SI/DI+eltolás	#(BP)(SI), #(BP)(DI)
Közvetlen adat		
Közvetlen bájt/szó	Az adat az utasítás része	#
Implicit címzés		
Push/pop utasítás	(SP) indirekt cím	PUSH, POP, PUSHF, POPF
Flagek betöltése/tárolása	állapotjelző (flag) regiszter	LAHF, STC, CLC, CMC
XLAT-átalakítás	AL, BX	XLAT
Ismétlődő string utasítások	(SI), (DI), (CX)	MOVS, CMPS, SCAS
In/out utasítások	AX, AL	IN #, OUT #
Bájt- és szókonverziók	AL, AX, DX	CBW, CWD

C.3. ábra. *Operandusok címzési módjai. A # szimbólum numerikus értéket vagy címkét jelöl*

címzésmód a **direkt címzés (direct addressing)**, amikor az operandus címe van megadva az utasításban. Például:

```
ADD CX,(20)
```

amelyben a 20-as és 21-es címeken található memóriaszó tartalma adódik CX-hez. Az assembly nyelvben a memóriacímeket általában címkékkel jelöljük a numerikus értékek helyett, a konverzió pedig a fordítási időben történik. Még a CALL és a JMP utasításoknál is tárolható a folytatás címe a memória egy címkézett helyén. A címkét közrefogó zárójelek fontosak (az általunk használt assembler számára), mert az

```
ADD CX,20
```

szintén érvényes utasítás, csak ez azt jelenti, hogy a 20 konstanst kell CX-hez hozzáadni, nem pedig a 20-as memóriaszó tartalmát. A C.3. ábrán a # szimbólum jelöli a numerikus konstanst, címkét vagy címkét használó konstans kifejezést.

Regiszter-indirekt címzés (register indirect addressing) esetén az operandus címe a BX, SI vagy DI regiszterek valamelyikében van tárolva. Mindhárom esetben az operandus az adatszegmensben található. Ha a regiszter elé egy konstanst írunk, akkor a cím a regiszter és a konstans összeadásával alakul ki. Ez az ún. **indexelt**

(register displacement) címzés nagyon kényelmes tömbök esetén. Ha például az SI tartalma 4, akkor a FORMAT címkénél található string ötödik karakterét AL-be töltjük az alábbi utasítással:

```
MOVB AL,FORMAT(SI)
```

Az egész string végigjárható a regiszter lépésenkénti növelésével vagy csökkentésével. Amikor szavas operandust használunk, a regisztert mindig kettővel kell változtatni.

Lehetőség van arra is, hogy a tömb báziscímét (azaz a legalacsonyabb numerikus címét) a BX regiszterbe tegyük, az SI vagy DI regisztert pedig számlálásra használjuk. Ez a **bázis relatív (register with index)** címzés. Például:

```
PUSH (BX)(DI)
```

kiolvassa az adatszegmens azon helyének tartalmát, melynek címe a BX és DI regiszterek összege. Ez az érték utána a verem tetejére kerül. Az utóbbi két címtípus kombinálható, így kapjuk a **bázis relatív eltolással (register with index and displacement)** címzést, mint például az alábbi utasításban:

```
NOT 20(BX)(DI)
```

amely a BX + DI + 20 és BX + DI + 21 címeken található memóriaszó komplemensét képezi.

Az adatszegmensben használható minden indirekt címzési mód a veremszegmensben is létezik, ilyen esetben a BP bázismutatót használjuk a BX bázisregiszter helyett. A (BP) az egyetlen regiszter-indirekt veremcímzési mód, de az összetettebb változatok szintén léteznek, mint például a stack relatív címzés eltolással: -1(BP)(SI). Ezek nagyon értékesek lokális változók és függvényparaméterek címzésénél, amelyeket a szubrutinok veremcímeken tárolnak. Ezzel az elrendezéssel a C.4.5. részben részletesebben foglalkozunk.

Minden eddig tárgyalt címzési módnak megfelelő cím használható a műveletekben forrás- és céloperandusként is. Ezeket együtt **effektív címek**nek **(effective address)** hívjuk. A két további blokk címzési módjai nem használhatók céloperandusként, és azokat nem is hívjuk effektív címzésnek. Ezek csak forrásoperandusként használhatók.

Amikor az operandus egy az utasításban megadott konstans bájt vagy szó, **közvetlen címzés**ről **(immediate addressing)** beszélünk. Így például a:

```
CMP AX,50
```

összehasonlítja AX-et az 50 konstanssal, és a flag-regiszterben az eredménytől függően beállít bizonyos biteket.

Végül, néhány utasítás **implicit címzés**t **(implied addressing)** használ. Ezeknek az utasításoknak az operandusa(i) az utasításból magából következik. Például a

```
PUSH AX
```

utasítás AX tartalmát a verem tetejére helyezi úgy, hogy először csökkenti az SP-t, majd AX-et az új SP által mutatott helyre másolja. Az SP-t külön nem nevezzük meg az utasításban, hanem magából a tényből – hogy egy PUSH utasításról van szó – következik, hogy az SP-t kell használni. Hasonlóképpen, a flag-manipuláló utasítások külön megnevezés nélkül, implicit módon hivatkoznak az állapotjelző regiszterre. Több más utasításnak is van implicit operandusa.

A 8088-asnak speciális utasításai vannak stringek mozgatására (MOVS), összehasonlítására (CMPS) és bejárására (SCAS). Ezeknél a string utasításoknál az SI és a DI indexregiszterek automatikusan változnak a művelet után. Ezt a viselkedés hívjuk **automatikus növelés (auto increment)** vagy **automatikus csökkentés (auto decrement)** módnak. Hogy az SI és a DI regiszterek növekednek vagy csökkennek, az az állapotjelző regiszter **irány-flag**jétől **(direction flag)** függ. Az irány-flag 0 értéke növelést, az 1 érték csökkentést jelent. Bájtos utasítások esetén a változás 1, míg szavas utasítások esetén 2. Bizonyos szempontból a veremmutató is automatikus növelés és automatikus csökkentés módon viselkedik: 2-vel csökken a PUSH utasítás elején és 2-vel nő a POP utasítás végén.

C.4. A 8088 utasításrendszere

Minden számítógép lelke az általa végrehajtható utasítások rendszere. Ahhoz, hogy valóban ismerjünk egy számítógépet, annak utasításrendszerét kell igazán jól ismernünk. A következő fejezetekben a 8088-as legfontosabb utasításait tárgyaljuk. A C.4. ábra ezek közül 10 csoportba osztva mutat be néhányat.

C.4.1. Adatmozgatás, -másolás és aritmetika

Az utasítások első csoportja az adatmásoló és -mozgató utasítások. Messze a leggyakoribb utasítás a MOV, amely explicit forrás- és céloperandussal rendelkezik. Ha a forrásoperandus regiszter, akkor a céloperandus lehet effektív cím. A táblázatban a regiszteroperandust *r*, míg az effektív címet *e* jelöli, tehát ezt az operanduskombinációt *e*←*r* jelöli. Ez a MOV utasításnál az *Operandusok* oszlop első bejegyzése. Mivel az utasítás szintaxisa szerint a céloperandus az első és a forrásoperandus a második, a balra mutató ← nyíllal jelöljük az operandusokat. Az *e*←*r* jelentése tehát az, hogy a regisztert másoljuk az effektív címre.

A MOV utasításnál a forrásoperandus szintén lehet effektív cím, a céloperandus pedig regiszter, amelyet az utasítás *Operandusok* oszlopának második, *r*←*e*, bejegyzése jelöl. A harmadik lehetőség, amikor a forrásoperandus közvetlen adat a céloperandus pedig effektív cím, azaz *e*←#. A közvetlen adatot a táblázatban a kettős kereszt (#) szimbólum jelzi. Mivel létezik a szót mozgató MOV és a bájtot

Mnemonik	Angol leírás	Leírás	Operandusok	Állapot-flagek			
				O	S	Z	C
MOV(B)	Move word/byte	Szó (bájt) mozgatása	r←e, e←r, e←#	–	–	–	–
XCHG(B)	Exchange word	Szó (bájt) cseréje	r↔e	–	–	–	–
LEA	Load effective address	Effektív cím betöltése	r←#e	–	–	–	–
PUSH	Push onto stack	Verem tetejére	e, #	–	–	–	–
POP	Pop from stack	Verem tetejéről	e	–	–	–	–
PUSHF	Push flags	Flagek verembe	–	–	–	–	–
POPF	Pop flags	Flagek veremből	–	–	–	–	–
XLAT	Translate AL	AL átalakítása	–	–	–	–	–
ADD(B)	Add word	Összeadás	r←e, e←r, e←#	*	*	*	*
ADC(B)	Add word with carry	Összeadás átvitellel	r←e, e←r, e←#	*	*	*	*
SUB(B)	Subtract word	Kivonás	r←e, e←r, e←#	*	*	*	*
SBB(B)	Subtract word with borrow	Kivonás átvitellel	r←e, e←r, e←#	*	*	*	*
IMUL(B)	Multiply signed	Előjeles szorzás	e	*	U	U	*
MUL(B)	Multiply unsigned	Előjel nélküli szorzás	e	*	U	U	*
IDIV(B)	Divide signed	Előjeles osztás	e	U	U	U	U
DIV(B)	Divide unsigned	Előjel nélküli osztas	e	U	U	U	U
CBW	Sign extend byte-word	Előjel kiterjesztése bájtról szóra	–	–	–	–	–
CWD	Sign extend word-double	Előjel kiterjesztése szóról hosszú szóra	–	–	–	–	–
NEG(B)	Negate binary	Szó negatívja	e	*	*	*	*
NOT(B)	Logical complement	Logikai komplemens	e	–	–	–	–
INC(B)	Increment destination	Növelés 1-gyel	e	*	*	*	*
DEC(B)	Decrement destination	Csokkentés 1-gyel	e	*	*	*	*
AND(B)	Logical and	Logikai és	r←e, e←r, e←#	0	*	*	0
OR(B)	Logical or	Logikai vagy	r←e, e←r, e←#	0	*	*	0
XOR(B)	Logical exclusive or	Logikai kizáró vagy	r←e, e←r, e←#	0	*	*	0
SHR(B)	Logical shift right	Logikai léptetés jobbra	e←1, e←CL	*	*	*	*
SAR(B)	Arithmetic shift right	Aritmetikai léptetés jobbra	e←1, e←CL	*	*	*	*
SAL(B) (=SHL(B))	Shift left	Léptetés balra	e←1, e←CL	*	*	*	*
ROL(B)	Rotate left	Forgatás balra	e←1, e←CL	*	–	–	*
ROR(B)	Rotate right	Forgatás jobbra	e←1, e←CL	*	–	–	*
RCL(B)	Rotate left with carry	Forgatás balra a carryn keresztül	e←1, e←CL	*	–	–	*
RCR(B)	Rotate right with carry	Forgatás jobbra a carryn keresztül	e←1, e←CL	*	–	–	*
TEST(B)	Test operands	Ellenőrzés logikai és alapján	e↔r, e↔#	0	*	*	0
CMP(B)	Compare operands	Operandusok összehasonlítása	e↔r, e↔#	*	*	*	*
STD	Set direction flag (↓)	Irány-flag beállítása (↓)	–	–	–	–	–
CLD	Clear direction flag (↑)	Irány-flag torlése (↑)	–	–	–	–	–
STC	Set carry flag	Carry beállítása	–	–	–	–	1
CLC	Clear carry flag	Carry torlése	–	–	–	–	0
CMC	Complement carry	Carry komplementálasa	–	–	–	–	*
LOOP	Jump back if decremented CX≠0	Ugrás, ha a csökkentett CX≠0	címke	–	–	–	–
LOOPZ LOOPE	Back if Z=1 and DEC(CX)≠0	Ugrás, ha Z=1 és DEC(CX)≠0	címke	–	–	–	–
LOOPNZ LOOPNE	Back if Z=0 and DEC(CX)≠0	Ugrás, ha Z=0 és DEC(CX)≠0	címke	–	–	–	–
REP REPZ REPNZ	Repeat string instruction	String utasítás ismétlése	string utasítás	–	–	–	–
MOVS(B)	Move word string	String mozgatása	–	–	–	–	–
LODS(B)	Load word string	String betöltése	–	–	–	–	–
STOS(B)	Store word string	String tárolása	–	–	–	–	–
SACS(B)	Scan word string	String bejárása	–	*	*	*	*
CMPS(B)	Compare word string	Stringek összehasonlítása	–	*	*	*	*
JCC	Jump according conditions	Feltételek szerinti ugrás	címke	–	–	–	–
JMP	Jump to label	Címkére ugrás	e, címke	–	–	–	–
CALL	Jump to subroutine	Szubrutinra ugrás	e, címke	–	–	–	–
RET	Return from subroutine	Visszatérés szubrutinból	–, #	–	–	–	–
SYS	System call trap	Rendszerhívás csapda	–	–	–	–	–

C.4. ábra. *A 8088-as legfontosabb utasításai*

mozgató MOVB is, az utasítás mnemonikjának végén a *B* zárójelek között található. Így ez az egy sor valójában hat különböző utasítást jelent.

Mivel adatmozgató utasítás hatására az állapotkód regiszter egyetlen flagje sem változik, az utolsó négy oszlopban a „–" bejegyzést találjuk. Megjegyezzük, hogy a mozgató utasítások igazából nem mozgatják az adatot. Másolatot készítenek, azaz a forrásoperandus nem változik, mint ahogy az egy valódi mozgatás esetén történne.

A táblázat második utasítása az XCHG, amely egy regiszter és egy effektív cím tartalmát cseréli fel. A csere jelölésére a táblázatban a ↔ szimbólumot használjuk. Így az XCHG utasítás *Operandusok* oszlopában $r \leftrightarrow e$ jelölést találunk. Itt is létezik bájtos és szavas változat. A következő utasítás a LEA, amely a Load Effective Address (effektív cím betöltése) rövidítése. Ez kiszámítja az effektív cím numerikus értékét, és tárolja azt egy regiszterben.

A következő a PUSH, amely az operandusát a verem tetejére helyezi. Az explicit operandus lehet konstans (*#* az *Operansudok* oszlopban) vagy egy effektív cím (*e* az *Operandusok* oszlopban). Van azonban egy implicit operandusa is, az SP, amelyet az utasítás szintaxisában külön nem említünk. Az utasítás 2-vel csökkenti SP-t, majd az operandust az új SP által mutatott helyre tárolja.

A POP következik, amely a verem tetejéről eltávolít egy operandust, majd azt egy effektív címre tárolja. A PUSHF és a POPF utasítás szintén rendelkezik implicit operandusokkal, ez a flag-regisztert teszi a verembe, illetve veszi ki onnan. Ez a helyzet az XLAT-tal is, amely az AL bájtos regiszterbe tölti az AL+BX cím tartalmát. Ezzel az utasítással 256 bájtos táblázatokban lehet gyorsan keresni.

Hivatalosan a 8088-asban definiáltak, az értelmezőben azonban nincsenek megvalósítva (ezért a C.4. ábrán sincsenek feltüntetve) az IN és OUT utasítások. Ezek valójában mozgató utasítások B/K eszközök felé és felől. Az implicit cím mindig az AX regiszter, a második operandus pedig a kiválasztott eszközregiszter portszáma.

A C.4. ábra második csoportjába az összeadó és kivonó utasítások tartoznak. Ezek mindegyike ugyanazt a három operanduskombinációt használja, mint a MOV: effektív címről regiszterbe, regiszterből effektív címre és konstanst effektív címre. Ezért a táblázat *Operandusok* oszlopának tartalma $r \leftarrow e$, $e \leftarrow r$ és $e \leftarrow \#$. Mind a négy utasításnál az O túlcsordulás-flag, az S előjel-flag, a Z zéró flag és a C átvitel- (carry) flag az utasítás eredményétől függően állítódik be. Eszerint például az O akkor kerül beállításra, ha az eredmény a rendelkezésre álló számú biten nem ábrázolható helyesen, ha viszont ábrázolható, akkor az O törlődik. Amikor a legnagyobb 16 bites (előjeles) számot, 0x7fff (decimális 32 767), hozzáadjuk önmagához, az eredmény nem ábrázolható 16 bites előjeles számként, ezért a hibát az O beállítása jelzi. Ezeknél az utasításoknál a többi állapot-flaggel is hasonló dolgok történnek. Ha egy utasítás hatással van egy állapot-flagre, akkor azt a megfelelő oszlopban egy csillag (*) jelzi. Az ADC és SBB utasításokban az utasítás végrehajtása előtti átvitel-flag egy extra 1 (vagy 0) értékként jelentkezik, amit az előző műveletből történő átvitelnek tekinthetünk. Ez az eszköz különösen hasznos 32 bites vagy még hosszabb egészek több szón történő ábrázolására. Az összeadásnak és a kivonásnak is léteznek bájtos változatai.

A következő blokk tartalmazza a szorzás és osztás utasításokat. Előjeles egész operandusokkal az IMUL és IDIV utasításokat, míg előjel nélküliekkel a MUL és DIV utasításokat kell használni. Ezen utasítások bájtos változatainak implicit céloperandusa az AH : AL regiszter összetétel. A szavas változatban az implicit cél az DX : AX regiszter összetétel. A DX vagy AH regiszterek még akkor is felülíródnak, ha a szorzás eredménye csak egy bájt vagy egy szó. A szorzás mindig lehetséges, hiszen a céloperandus rendelkezik elegendő számú bittel. Ha a szorzat nem ábrázolható egy szón vagy bájton, akkor beállítódnak a túlcsordulás és átvitel bitek. A zéró és a negatív flagek szorzás után nem definiáltak.

Az osztás szintén a DX : AX vagy az AH : AL regiszter összetételek valamelyikét használja céloperandusként. A hányados az AX-be vagy az AL-be kerül, a maradék pedig a DX-be vagy az AH-ba. Az osztás művelet után a négy flag, az átvitel-, a túlcsordulás-, a zéró és a negatív flag közül egyik sem definiált. Ha az osztó 0, vagy ha a hányados nem fér el a regiszterben, a művelet végrehajt egy **csapdát** (**trap**), amely leállítja a programot, kivéve, ha meg van adva egy külön csapdakezelő (trap handler) rutin. Ezen felül, érdemes a negatív előjelre figyelni a programban osztás előtt és után, mert a 8088-asban a maradék előjele mindig megegyezik az osztandó előjelével, míg a matematikában a maradék mindig nemnegatív.

A binárisan kódolt decimális számokra vonatkozó olyan utasítások, mint például az ascii korrekció összeadásra (Ascii Adjust for Addition, AAA) és a decimális korrekció összeadásra (Decimal Adjust for Addition, DAA), az értelmezőben nincsenek megvalósítva, így a C.4. ábrán sincsenek felsorolva.

C.4.2. Logikai, bit- és léptető műveletek

A következő blokk az előjel-kiterjesztés, negálás, logikai komplemens, növelés és csökkentés utasításokat tartalmazza. Az előjel-kiterjesztő utasításoknak nincs explicit operandusuk, hanem a DX : AX vagy az AH : AL regiszter összetételekkel dolgoznak. A csoport többi utasításának egyetlen operandusa bármely effektív címen lehet. A NEG, az INC és a DEC utasítások az elvárt módon hatnak a flagekre, kivéve, hogy a növelő és csökkentő utasítás nincs hatással az átvitel-flagre, ami elég váratlan, és amit néhányan tervezési hibának is tartanak.

A következő utasításblokk a két operandusú logikai csoport, amelyek közül minden utasítás az elvárt módon működik. Az eltoló és forgató csoportban minden művelet céloperandusa egy effektív cím, a forrásoperandus azonban vagy a CL bájtregiszter, vagy az 1 szám. Az eltolások mind a négy flaget módosítják, míg a forgatások csak az átvitelt és a túlcsordulást. Az átvitel-flagbe mindig az a bit kerül, amely az eltolás vagy forgatás irányától függően a legmagasabb vagy a legalacsonyabb helyértékű bitről kilép vagy kiforgatásra kerül. A carryn keresztül történő forgatások, azaz az RCR, RCL, RCRB és RCLB esetén az átvitelbit az effektív címen található operandussal együtt egy 17 bites vagy 9 bites körkörös léptető regiszter összetételt alkot, ami lehetővé teszi a többszavas léptetéseket és forgatásokat.

Az utasítások következő csoportja a feltételbiteket (jelző, flag) manipulálja. Ezeknek az elsődleges célja feltételes ugrások előkészítése. A kétirányú nyíl (↔)

jelöli az összehasonlító vagy ellenőrző utasítások két operandusát, melyek a művelet során nem változnak. A TEST műveletben az operandusok közötti logikai ÉS kerül kiszámításra, így állítva be vagy törölve a zero és az előjel-flageket. A kiszámított érték maga sehol sem tárolódik, az operandusok nem változnak. A CMP utasításban az operandusok különbsége kerül kiszámításra, és az összehasonlítás eredményeként mind a négy flag beállítódik vagy törlődik. Az irány-flag – ami meghatározza, hogy az SI és DI regiszterek növekedjenek vagy csökkenjenek string utasítások hatására – az STD utasítással állítható be, és a CLD utasítással törölhető.

A 8088-asban van még egy **paritás-flag (parity flag)** és egy **kiegészítő átvitel-flag (auxiliary carry flag)** is. A paritás-flag az eredmény paritását adja meg (páratlan vagy páros). A kiegészítő átvitel-flag azt mondja meg, hogy a cél alsó (4 bites) bájtrészéről (nibble) történt-e átvitel. Léteznek még továbbá az LAHF és SAHF utasítások, amelyek a flag-regiszter alacsony helyértékű bájtját az AH-ba, illetve onnan visszatöltik. A túlcsordulás-flag a feltételkód-regiszter magas helyértékű bájtjában van, így ezek az utasítások ezt nem másolják. Ezek az utasítások és flagek főként a 8080-as és 8085-ös processzorokkal való visszafelé kompatibilitás biztosítására szolgálnak.

C.4.3. Ciklusszervezés és ismétlődő string műveletek

A következő blokk tartalmazza ciklusszervező utasításokat. A LOOP utasítás eggyel csökkenti a CX regisztert, majd ha az eredmény nem 0, elugrik (általában visszaugrik) a megjelölt címkére. A LOOPZ, LOOPE, LOOPNZ és LOOPNE utasítások a zéró flaget is ellenőrzik annak eldöntésére, hogy vajon a ciklust a CX regiszter 0-vá válása előtt meg kell-e szakítani.

A LOOP utasítások céloperandusának az utasításszámláló aktuális pozíciójától számított 128 bájton belül kell lennie, mert az utasítás csak egy 8 bites előjeles eltolást tartalmaz. Az átugorható *utasítások* száma (eltérően a bájtoktól) pontosan nem adható meg, mert a különböző utasítások hossza különböző. Általában az első bájt definiálja az utasítás típusát, és vannak is olyan utasítások, amelyek csak egyetlen bájtot foglalnak el a kódszegmensben. Gyakran azonban a második bájt adja meg az utasítás regisztereit és regisztermódjait, ha pedig az utasítás eltolást vagy közvetlen adatot is tartalmaz, az utasítás hossza akár négy vagy hat bájtra is nőhet. Az átlagos utasításhossz tipikusan kb. 2,5 bájt utasításonként, tehát a LOOP nem tud kb. 50 utasításnál távolabbra visszaugrani.

String utasítások számára létezik néhány speciális ismétlő mechanizmus. Ezek a REP, REPZ és a REPNZ. A C.4. ábra következő blokkjának öt string utasítása mind implicit címekkel dolgozik, és mindegyik az automatikus növelés vagy csökkentés módban használja az indexregisztereket. Ezen utasítások mindegyikében az SI regiszter az **adatszegmens**re (**data segment**), a DI regiszter viszont az **extra szegmens**re (**extra segment**) mutat, amelynek a bázisa ES. A MOVSB a REP utasításokkal kombinálva teljes stringek egyetlen utasítással történő mozgatására használható. A string hosszát a CX regiszterben kell megadni. Mivel a MOVSB utasítás nincs hatással a flagekre, így a másoló művelet közben a REPNZ-vel nem lehet az ASCII nulla bájtot ellenőrizni. Ez azonban kiküszöbölhető, ha először a REPNZ SCASB utasítást használjuk arra, hogy a CX-be a megfelelő érték kerüljön, és utána hajtunk végre egy REP MOVSB-t. Ezt a C.8. fejezetben a string másoló példában szemléltetjük. Ezen utasítások mindegyikénél külön figyelmet kell fordítani az ES szegmensregiszterre, hacsak az ES és a DS nem azonosak. Az értelmezőben a kis (small) memória modellt használjuk, így ES = DS = SS.

C.4.4. Ugró és eljáráshívó utasítások

Az utolsó blokk a feltételes és feltétel nélküli ugrásokról, szubrutin hívásokról és az azokból történő visszatérésekről szól. Itt a legegyszerűbb művelet a JMP utasítás. Ez egy címkét vagy egy bármely effektív címen található tartalmat kaphat céloperandusként. Megkülönböztetjük a **közeli ugrás**okat (**near jump**) és a **távoli ugrás**okat (**far jump**). Közeli ugrásban a cél az aktuális kódszegmensben található, amely nem változik a művelet során. A direkt (címkés) változatban a kódszegmens-regiszter új értéke a címke utáni utasítás címe. Az effektív címes változatban a memóriából töltünk be egy hosszú szőt (long), ebből az alsó szó a célcímkének, míg a felső az új kódszegmens-regiszter értékének felel meg.

Ez a megkülönböztetés természetesen nem meglepő. A 20 bites címtartomány tetszőleges címére ugráshoz 16 bitnél több megadásáról is gondoskodni kell. Ennek módja a CS és PC regiszterek új értékkel való feltöltése.

Feltételes ugrások

A 8088-as 15 feltételes ugrással rendelkezik, ezek közül néhánynak több neve is van (például az „UGRÁS HA NAGYOBB VAGY EGYENLŐ" ugyanaz az utasítás, mint az „UGRÁS HA NEM KISEBB"). Ezeket a C.5. ábrán soroljuk fel. Mindegyikük az adott utasítástól legfeljebb 128 bájtos távolságra képes elugrani. Ha a cél ezen a tartományon kívülre esik, akkor egy átugrásos konstrukciót kell alkalmazni. Ilyen esetben egy ellentétes feltételű ugróutasítást használunk arra, hogy a következő utasítást átugorjunk. Ha a következő utasítás egy feltétel nélküli ugrás a tervezett utasításra, akkor ennek a két utasításnak az együttes hatása egy a tervezett típusú, de hosszabb távú ugrás. Például a:

```
    JB TAVOLICIMKE
```

helyett a

```
    JNB 1f
    JMP TAVOLICIMKE
1:
```

kódrészletet használjuk. Más szóval, ha nem lehetséges egy JUMP BELOW (UGRÁS HA ALATTA VAN) végrehajtása, akkor elhelyezünk egy JUMP NOT BELOW (UGRÁS HA NINCS ALATTA) utasítást egy közeli *1*-es címkére, majd ezt követi egy feltétel nélküli ugrás

a TAVOLICIMKE-re. A hatás ugyanaz, csak egy kicsit több helyet és időt igényel. Az assembler automatikusan legenerálja ezeket az átugrásos konstrukciókat, ha a cél várhatóan túl távoli. A pontos számítás azonban kicsit ravasz. Tegyük fel, hogy a távolság a határ közelében van, de néhány köztes utasítás szintén feltételes ugrás. A külső nem oldható fel addig, amíg a belsők mérete nem ismert stb. A biztonság kedvéért az assembler inkább az óvatosság oldalán vétkezik. Néha akkor is átugrásos konstrukciót generál, amikor az nem lenne feltétlenül szükséges. Csak akkor generál közvetlen feltételes ugrást, ha teljesen biztos abban, hogy a cél megfelelő távolságon belül van.

A legtöbb feltételes ugrás az állapot-flagektől függ, és egy összehasonlító vagy ellenőrző utasítás előzi meg. A CMP utasítás kivonja a forrást a céloperandusból, majd beállítja a feltételkódokat, és eldobja az eredményt. Egyik operandus sem változik. Ha az eredmény nulla vagy az előjelbitje be van állítva (azaz negatív), a megfelelő flag-bitek beállítódnak. Ha az eredmény nem ábrázolható a rendelkezésre álló számú biten, a túlcsordulás-flag kerül beállításra. Ha a legmagasabb helyértékű biten átvitel keletkezik, azt az átvitel-flag jelzi. A feltételes ugrások ezen bitek mindegyikét ellenőrizhetik.

Ha az operandusokat előjelesként kezeljük, akkor a „NAGYOBB MINT" (GREATER THAN) és a „KISEBB MINT" (LESS THAN) utasításokat kell használnunk. Ha előjel nélküliek, akkor a „FELETTE" (ABOVE) és „ALATTA" (BELOW) változatokat kell használni.

Utasítás	Angol leírás	Leírás	Mikor ugrik
JNA, JBE	Not Above, Below or Equal	Nem nagyobb, Alatta vagy egyenlő	CF=1 vagy ZF=1
JNB, JAE, JNC	Not Below, Above or Equal, Not Carry	Nem alatta, Felette vagy egyenlő, Nincs átvitel	CF=0
JE, JZ	Equal, Zero	Egyenlő, Nulla	ZF=1
JNLE, JG	Not Less than or Equal, Greater than	Nem kisebb vagy egyenlő, Nagyobb	SF=OF és ZF=0
JGE, JNL	Greater than or Equal, Not Less than	Nagyobb vagy egyenlő, Nem kisebb	SF=OF
JO	Overflow	Túlcsordulás	OF=1
JS	Sign negative	Negatív előjel	SF=1
JCXZ	CX is Zero	A CX nulla	CX=0
JB, JNAE, JC	Below, Not Above or Equal, Carry	Alatta, Nem felette vagy egyenlő, Van átvitel	CF=1
JNBE, JA	Not Below or Equal, Above	Nem alatta vagy egyenlő, Felette	CF=0 és ZF=0
JNE, JNZ	NonEqual, NonZero	Nem egyenlő, Nem nulla	ZF=0
JL, JNGE	Less than, Not Greater or Equal	Kisebb, Nem nagyobb vagy egyenlő	SF≠OF
JLE, JNG	Less than or Equal, Not Greater than	Kisebb vagy egyenlő, Nem nagyobb	SF≠OF vagy ZF=1
JNO	NonOverflow	Nincs túlcsordulás	OF=0
JNS	Nonnegative	Nem negatív	SF=0

C.5. ábra. *Feltételes ugrások*

C.4.5. Szubrutin hívások

A 8088-as külön utasítást használ, az assembly nyelvben általában **szubrutin**nak nevezett, eljárások hívására. Hasonlóan az ugró utasításokhoz, léteznek **közeli hívás (near call)** és **távoli hívás (far call)** utasítások. Az értelmezőben csak a közeli hívás van megvalósítva. A céloperandus vagy egy címke, vagy egy effektív címen található. A szubrutin hívásához szükséges paramétereket fordított sorrendben helyezzük a veremre, ahogy azt a C.6. ábra is mutatja. Az assembly nyelvben a paramétereket általában **argumentumok**nak hívjuk, de ezek felcserélhető elnevezések. Ezeket a verembe írásokat követi a CALL utasítás végrehajtása. Az utasítás először elhelyezi a verem tetejére az aktuális utasításszámlálót, így kerül elmentésre a visszatérési cím. A visszatérési cím az a cím, amelyen a hívó rutinnak folytatódnia kell, miután a szubrutin visszatér.

Ezután vagy a címkéből, vagy az effektív címről betöltődik az új utasításszámláló. Ha a hívás távoli, akkor a CS regiszter még a PC előtt kerül a verembe, a közvetlen adatból vagy az effektív címről az utasításszámlálóba és a kódszegmens-regiszterbe is új érték töltődik. Ezzel fejeződik be a CALL utasítás.

A RET visszatérő utasítás egyszerűen csak kiveszi a veremből a visszatérési címet, azt eltárolja az utasításszámlálóban, és a program a CALL utasítást közvetlenül követő utasítással folytatódik. Néha a RET utasítás közvetlen adatként egy pozitív számot is tartalmaz. Ezt a számot úgy tekintjük, mint a hívás előtt a verembe tárolt argumentumok bájtjainak számát, és ezt az SP-hez hozzáadva történik meg a verem takarítása. A távoli változatban, a RETF-ben, a kódszegmens-regiszter, ahogy az elvárható, a veremből az utasításszámláló után kerül kiolvasásra.

A szubrutin belsejében elérhetőnek kell lenni az argumentumoknak. Ezért a szubrutin gyakran azzal kezdődik, hogy vermeli a bázismutatót, és az SP aktuális értékét bemásolja a BP-be. Ez azt jelenti, hogy a bázismutató a saját előző értékét tartalmazó címre mutat. Ekkor a visszatérési érték a BP+2 címen, az első és második argumentum pedig rendre a BP+4 és BP+6 címeken találhatók. Ha az eljárásnak szüksége van lokális változókra, akkor a szükséges bájtok számát levonjuk a veremmutatóból, és a változók a bázismutatóval és negatív eltolással címezhetők. A C.6. ábrán látható példában három egyszavas lokális változó van, rendre a BP−2, BP−4 és BP−6 címeken. Így az aktuális argumentumok és lokális változók teljes halmaza elérhető a BP regiszteren keresztül.

A verem a szokásos módon használható részeredmények mentésére vagy a következő hívás argumentumainak előkészítésére. A szubrutin által használt veremméret külön kiszámítása nélkül, a visszatérés előtt a verem helyreállítható úgy, hogy a bázismutatót a veremmutatóba töltjük, majd a régi BP értéket visszaállítjuk a veremből, és végül végrehajtjuk a RET utasítást.

Szubrutinhívás közben a processzor regisztereinek értékei általában megváltoznak. A helyes gyakorlat az, ha ragaszkodunk bizonyos szabályokhoz, mint például ahhoz, hogy a hívó eljárásnak nem kell tudnia a hívott rutin által használt regiszterekről. Feltételezzük azt, hogy az AX és a DX a hívott rutinban megváltozhat. Ha

BP+8	...	
BP+6	2. argumentum	
BP+4	1. argumentum	
BP+2	Visszatérési cím	
BP	régi BP	←BP
BP–2	1. lokális változó	
BP–4	2. lokális változó	
BP–6	3. lokális változó	
BP–8	részeredmény	←SP

C.6. ábra. *Példa a veremre*

ezek közül valamelyik regiszter értékes információt tartalmaz, akkor érdemes azt a hívó rutinban a verembe helyezni még az argumentumok vermelése előtt. Ha a szubrutin más regisztereket is használ, akkor azokat rögtön a szubrutin elején a veremre tehetjük, és a RET utasítás előtt visszatölthetjük. Más szóval, jó szabály az, hogy a hívó menti az AX és DX regisztereket, ha azok valami fontosat tartalmaznak, a hívott pedig menti az összes többi regisztert, amelyet felülír.

C.4.6. Rendszerhívások és rendszerszubrutinok

A programokat egy operációs rendszer felett futtatjuk azért, hogy a fájlok megnyitását, lezárását, olvasását és írását elkülöníthessük az assembly programozástól. Az értelmező hét rendszerhívást és öt függvényt biztosít, hogy több platformon is futhasson. Ezeket sorolja fel a C.7. ábra.

Sorszám	Név	Argumentumok	Visszatérési érték	Leírás
5	_OPEN	*name, 0/1/2	fájlleíró	Fájl megnyitása
8	_CREAT	*name, *mode	fájlleíró	Fájl létrehozása
3	_READ	fd, buf, nbytes	bájtok száma	nbytes bájt beolvasása a buf pufferbe
4	_WRITE	fd, buf, nbytes	bájtok száma	nbytes bájt kiírása a buf pufferből
6	_CLOSE	fd	0, ha sikeres	Az fd fájlleíróval adott fájl lezárása
19	_LSEEK	fd, offset(long), 0/1/2	pozíció (long)	Fájlmutató mozgatása
1	_EXIT	status		Fájlok lezárása és a processzus leállítása
117	_GETCHAR		beolvasott karakter	Karakter beolvasása a szabványos bemenetről
122	_PUTCHAR	char	kiírt bájt	Karakter kiírása a szabványos kimenetre
127	_PRINTF	*format, arg		Formázott kiírás a szabványos kimenetre
121	_SPRINTF	buf, *format, arg		Formázott kiírás a buf pufferbe
125	_SSCANF	buf, *format, arg		Argumentumok beolvasása a buf pufferből

C.7. ábra. *Az értelmezőben elérhető néhány UNIX-rendszerhívás és -szubrutin*

Ezt a tizenkét rutint a szabványos hívási szekvenciával lehet aktiválni. Először a szükséges argumentumokat kell fordított sorrendben a veremre helyezni, aztán a hívás sorszámát kell a veremre tenni, és végül az operandusok nélküli SYS rendszercsapda (system trap) utasítást kell végrehajtani. A rendszerrutin az összes szükséges információt a veremben találja, beleértve az igényelt rendszerszolgáltatás hívási sorszámát is. A visszatérési érték vagy az AX regiszterbe vagy – ha az eredmény túl hosszú – a DX : AX regiszter-összetételbe kerül.

A többi regiszter garantáltan megőrzi értékét a SYS utasítás alatt. Ezen túlmenően a hívás után az argumentumok még mindig a veremben lesznek. A veremmutatót a hívás után meg kell igazítani (a hívónak), hacsak egy következő híváshoz nincs rájuk szükség.

Kényelmi szempontból a rendszerhívások neveit az assembler program elején konstansokként definiálhatjuk, így azokat számok helyett nevekkel hívhatjuk. A példákban majd számos rendszerhívást tárgyalunk, úgyhogy ebben a fejezetben csak a legszükségesebb részleteket írjuk le.

Ezekben a rendszerhívásokban a fájlokat vagy az OPEN vagy a CREAT hívással nyithatjuk meg. Mindkét esetben az első argumentum a fájl nevét tartalmazó string kezdetének címe. Az OPEN hívás második argumentuma 0 (ha a fájlt olvasásra kell megnyitni), 1 (ha írásra kell megnyitni) vagy 2 (ha mindkettőre). Ha a fájlra az írást is engedélyezni kell, és az még nem létezik, akkor a hívás létrehozza. A CREAT hívás egy üres fájlt hoz létre a második argumentumnak megfelelő jogosultságokkal. Mind az OPEN, mind a CREAT hívás egy kicsi egész értékkel tér vissza az AX regiszterben, amelyet **fájlleírónak (file descriptor)** hívunk, és amely a fájl olvasásához, írásához és lezárásához használható. A negatív visszatérési érték a hívás sikertelenségét jelzi. A program indításakor három fájl már nyitott állapotban van az alábbi fájlleírókkal: 0 a szabványos bemenet, 1 a szabványos kimenet és 2 a szabványos hibakimenet.

A READ és WRITE hívás ugyanazt a három argumentumot várja: a fájlleírót, az adatokat tároló puffert, valamint az átviendő bájtok számát. Mivel az argumentumok fordított sorrendben kerülnek a veremre, először a bájtok számát, majd a puffer kezdetének címét, aztán a fájlleírót, és végül a hívás sorszámát (READ vagy WRITE) kell a veremre helyezni. Az argumentumoknak ezt a vermelési sorrendjét a szabványos C nyelvű hívásokkal megegyezőre választottuk, mint például ahogy a

```
read(fd, buffer, bytes);
```

megvalósítása a paramétereket a *bytes*, *buffer* és végül az *fd* sorrendben vermeli.

A CLOSE hívásnak csak a fájlleíróra van szüksége, és az AX-ben 0-val tér vissza, ha a fájlt sikeresen lezárta. Az EXIT hívás egy kilépési kódot vár a veremben, és nem tér vissza.

Az LSEEK hívás módosítja a nyitott fájlban az **olvasás/írás mutatót**. Az első argumentum a fájlleíró. Mivel a második argumentum egy hosszú egész (long), annak először a magas, majd az alacsony helyértékű szavát kell a veremre tenni, még akkor is, ha az eltolás elfér egyetlen szóban. A harmadik argumentum jelzi, hogy az új olvasás/írás mutatót a fájl kezdetéhez képest (0), az aktuális pozícióhoz képest

(1), vagy pedig a fájl végéhez képest (2) kell kiszámítani. A visszatérési érték a mutatónak a fájl kezdetéhez viszonyított új pozíciója, amely egy hosszú egészként a DX:AX regiszter összetételben található.

Ezzel elérkeztünk azokhoz a függvényekhez, amelyek nem rendszerhívások. A GETCHAR függvény beolvas egy karaktert a szabványos bemenetről, és elhelyezi AL-ben, AH-t pedig kinullázza. Hiba esetén az egész AX szó értéke –1 lesz. A PUTCHAR függvény egy bájtot ír a szabványos kimenetre. Sikeres írás esetén a visszatérési érték a kiírt bájt, sikertelenség esetén –1.

A PRINTF függvény formázott információt nyomtat. Az első argumentum a kimenet formázásának módját megadó formátum-string címe. A „%d" azt jelzi, hogy a veremben a következő argumentum egy egész szám, amelyet decimális jelölésre konvertálva kell kinyomtatni. Hasonlóan a „%x" hexadecimálisra, a „%o" pedig oktálisra konvertál. Ezen felül a „%s" azt jelzi, hogy a következő argumentum egy null-végű string, amelyet a string kezdőcímének verembe helyezésével adunk át a hívásnak. A vermen található további argumentumok számának illeszkednie kell a formátum-stringben található konverziós jelzésekhez. Például a

```
printf("x=%d és y=%d\n", x, y);
```

hívás a stringet a formátum-stringben a „%d" helyére az *x* és *y* numerikus értékeit helyettesítve nyomtatja ki. Ismét, a C nyelvvel való kompatibilitás érdekében az argumentumok verembe kerülési sorrendje „y", „x" és végül a formátum-string. Ennek az oka az, hogy a *printf* változó számú paraméterrel hívható, és azokat fordított sorrendben a verembe helyezve a formátum-string mindig az utolsó, és így mindig beazonosítható. Ha a paramétereket balról jobbra vermelnénk, a formátum-string valahol mélyen a veremben helyezkedne el, és a *printf* eljárás nem tudná, hogy hol keresse.

Az SPRINTF hívásban az első argumentum a puffer, amelybe a kimeneti string kerül a szabványos kimenet helyett. A többi argumentum a PRINTF-ével megegyező. Az SSCANF hívás az SPRINTF fordítottja abban az értelemben, hogy az első argumentum egy string, amely egész számokat tartalmaz decimális, oktális, vagy hexadecimális jelöléssel, a következő argumentum pedig egy formátum-string, amelyben a konverziós jelzők találhatók. A többi argumentum a konvertált információt fogadó memóriaszavak címe. Ezek a rendszerszubrutinok nagyon változatosak, és lehetőségeiknek részletes tárgyalása messze túlmutat e függelék témakörén. A C.8. fejezetben több példában látható, hogy különböző helyzetekben hogyan használhatók.

C.4.7. Záró megjegyzések az utasításrendszerről

A 8088-as hivatalos definíciójában létezik egy **szegmensregiszter-választó (segment override) prefix**, amely lehetővé teszi, hogy különféle szegmensekből képezzünk effektív címeket. Azaz, a választást követő első memóriacím kiszámítása a megjelölt szegmensregiszterrel történik. Például az


```
ESEG MOV DX,(BX)
```

utasítás először az extra szegmens használatával kiszámítja a BX címet, majd annak a tartalmát tölti a DX-be. Nem bírálható felül azonban a veremszegmens az SP-t használó címek esetén, valamint az extra szegmens a DI regisztert használó string utasítások esetén. A MOV utasításban ugyan használhatók az SS, DS, és ES szegmensregiszterek, szegmensregiszterbe azonban nem lehet közvetlen adatot tölteni, és azok az XCHG műveletben sem használhatók. Szegmensregiszterek változtatásával, illetve szegmensregiszter-választással programozni elég bonyolult, és amikor csak lehet, kerülendő. Az értelmező rögzített szegmensregisztereket használ, ezért itt ezek a problémák nem jönnek elő.

A legtöbb számítógépben rendelkezésre állnak lebegőpontos utasítások is, néha közvetlenül a processzorban, néha egy külön társprocesszorban, néha pedig csak szoftverben egy speciális lebegőpontos csapdán keresztül értelmezve. Ezeknek a tárgyalása szintén nem része e függeléknek.

C.5. Az assembler

A 8088-as architektúra tárgyalásának végére értünk. Következő témánk a 8088-as assembly nyelvű programozását lehetővé tevő szoftver, konkrétan az assembly nyelvű programozás tanulására általunk biztosított eszközök. Először az assemblert, majd a nyomkövetőt mutatjuk be, majd végül használatukhoz néhány gyakorlati tudnivalót.

C.5.1. Bevezetés

Egészen mostanáig az utasításokra **mnemonikok**kal hivatkoztunk, azaz olyan rövid, könnyen megjegyezhető szimbolikus nevekkel, mint az ADD és a CMP. A regisztereket is szimbolikus neveken neveztük, mint például az AX és a BP. Az olyan program, amelyben az utasításokat és a regisztereket szimbolikus nevekkel írjuk le, **assembly nyelvű program**. Az ilyen program futtatásához azt először át kell alakítani a CPU által értelmezhető bináris számokká. Az assembly nyelvű programot bináris számokká alakító program az **assembler**. Az assembler kimenetét **tárgymodul**nak (**object file**) nevezzük. Sok program hív olyan szubrutinokat, amelyeket korábban fordítottak le és tárgymodul könyvtárakban (library) tároltak. Ezeknek a programoknak a futtatásához az újonnan fordított tárgymodult és az általa használt könyvtári szubrutinokat (amelyek szintén tárgymodulok) egy másik programmal, a **szerkesztő**vel (**linker**) egyetlen végrehajtható bináris fájllá kell összeszerkeszteni. Az átalakítás csak akkor van készen, mikor a szerkesztő egy vagy több tárgymodulból elkészítette a futtatható bináris fájlt. Az operációs rendszer ezután már beolvashatja a memóriába, és végrehajthatja a futtatható bináris fájlt.

Az assembler első feladata egy **szimbólumtábla** (**symbol table**) felépítése, amely segítségével a szimbolikus konstansok és címkék nevei leképezhetők az

általuk jelölt bináris számokra. A programban közvetlenül definiált konstansok minden feldolgozás nélkül elhelyezhetők a szimbólumtáblában. Ezzel szemben a címkék olyan címeket reprezentálnak, amelyek értéke nem azonnal nyilvánvaló. Ezek értékeinek kiszámításához az assembler sorról sorra megvizsgálja a programot. Ezt hívjuk **első menet**nek **(first pass)**. Eközben nyomon követi az **ún. helyszámlálót (location counter)**, amelyet általában a „.” (pont) szimbólummal jelölünk és **pont**nak **(dot)** ejtünk. Ebben a menetben minden egyes megtalált utasítás és memóriafoglalás esetén a helyszámláló a talált elem memóriabeli tárolásához szükséges mérettel növekszik. Így, ha az első két utasítás rendre 2 és 3 bájtos, akkor a harmadik utasításon elhelyezett címke a numerikus 5 értéket kapja. Például, ha az alábbi kódrészlet egy program kezdete, akkor az *L* értéke 5 lesz.

```
    MOV AX,6
    MOV BX,500
L:
```

A **második menet (second pass)** kezdetén minden szimbólum numerikus értéke ismert. Mivel az utasítás mnemonikok numerikus értékei szintén konstansok, elkezdődhet a **kódgenerálás (code generation)**. Az utasítások egyesével újra beolvasásra kerülnek, és a bináris értékük a tárgymodulba íródik. A tárgymodul akkor készül el, amikor az utolsó utasítás is lefordult.

C.5.2. Az ACK-alapú assembler, az *as88*

Ebben a fejezetben részletesen bemutatjuk az *as88* assembler/szerkesztő programot, amely elérhető a CD-ROM-mellékleten, és amely együttműködik a nyomkövetővel. Ez az Amsterdam Compiler Kit (ACK) assembler, amely MS-DOS- vagy Windows-assemblerek helyett UNIX-assemblerek mintájára készült. A megjegyzés szimbólum ebben az assemblerben a felkiáltójel (!). A felkiáltójelet követő dolgok a sor végéig megjegyzésnek minősülnek, és nincsen hatásuk a keletkező tárgymodulra. Az assembler megengedi az üres sorokat is, de azokat is ugyanúgy figyelmen kívül hagyja.

Az assembler a lefordított kód és adat tárolására három különböző szegmenst használ. Ezek a szegmensek a gép memóriaszegmenseihez kapcsolódnak. Az első a **TEXT szegmens (TEXT section)**, a processzorutasítások számára. A következő a **DATA szegmens (DATA section)** az adatszegmensbe tartozó memória azon részének inicializálására, amely az eljárás kezdetén ismert. Végül a **BSS (Block Started by Symbol)** szegmens az adatszegmensben tárolt, de nem inicializált (azaz 0-ra inicializált) memória lefoglalására szolgál. Ezek mindegyike saját helyszámlálóval (location counter) rendelkezik. A szegmensek használatának célja az, hogy az assembler felváltva generálhasson utasításokat, majd adatokat, aztán újabb utasításokat, majd megint adatokat stb., és végül a szerkesztő ezeket a darabokat úgy szerkeszthesse össze, hogy az összes utasítás a kódszegmensbe, az adatok pedig mind az adatszegmensbe kerüljenek. Az assembly kód minden sorából csak egy

szegmensbe kerül kimenet, a kódsorok és adatsorok azonban keverten használhatók. Futási időben a TEXT szegmens a kódszegmensbe, a DATA és a BSS szegmensek pedig (egymást követően) az adatszegmensbe kerülnek.

Az assembly nyelvű program bármely utasítása vagy adatszava kezdődhet címkével. Címke önmagában is szerepelhet egy sorban, ilyenkor ezt úgy tekintjük, mintha a következő utasítás vagy adatszó előtt állna. Például a:

```
    CMP AX,ABC
    JE L
    MOV AX,XYZ
L:
```

programrészlet esetén az *L* címke az őt követő utasításra vagy adatszóra hivatkozik. Két címketípus megengedett. Az első csoportba tartoznak a globális címkék, amelyek alfanumerikus azonosítók a végükön egy kettősponttal (:). Ezeknek egyedinek kell lenni, és nem lehetnek azonosak sem kulcsszavakkal, sem utasításmnemonikkal. Másrészről, csak a TEXT szegmensben, használhatunk lokális címkéket is, melyek mindegyike egyetlen számjegyből és az azt követő kettőspontból (:) áll. Egy lokális címke többször is előfordulhat. Ha a program a:

```
JE 2f
```

utasítást tartalmazza, akkor az azt jelenti, hogy UGRÁS HA EGYENLŐ előre (forward) a következő lokális 2-es címkére. Hasonlóan, a

```
JNE 4b
```

jelentése az, hogy UGRÁS HA NEM EGYENLŐ vissza (backward) a legközelebbi 4-es címkére.

Az assembler megengedi, hogy a konstansokra is szimbolikus nevekkel hivatkozzunk az alábbi szintaxis szerint:

```
azonosító = kifejezés
```

ahol az azonosító egy alfanumerikus string, mint például:

```
BLOCKSIZE = 1024
```

Mivel ebben az assembly nyelvben minden azonosítóban, csak az első nyolc karakter lényeges (megkülönböztető szerepű), ezért a *BLOCKSIZE* és a *BLOCKSIZZ* ugyanaz a szimbólum, konkrétan a *BLOCKSIZ*. Konstansokból, numerikus értékekből és műveleti jelekből kifejezéseket képezhetünk. A címkéket konstansoknak tekintjük, hiszen értékük az első menet végén már ismert.

A numerikus érték lehet **oktális** (0-val kezdődő), **decimális** vagy **hexadecimális** (0X vagy 0x jelekkel kezdődő). A hexadecimális számok az a–f vagy A–F betűket

használják a 10–15 értékek jelölésére. Az egész műveleti jelek a +, –, *, / és a %, rendre az összeadásra, kivonásra, szorzásra, osztásra és maradékképzésre. A logikai műveletek a &, ^, ~, rendre a bitenkénti ÉS (AND), bitenkénti VAGY (OR) és logikai komplemens (NEM, NOT) jelölésére. Kifejezésekben a [és] szögletes zárójeleket használhatjuk csoportosításra. A címzési módokkal való keveredés elkerülése végett a kerek zárójeleket *NEM* használhatjuk.

A kifejezésekben a címkéket értelmes módon kell kezelni. Utasításcímkéket nem lehet adatcímkékből kivonni. Két összehasonlítható címke közötti különbség egy numerikus érték, de sem a címkék, sem pedig azok különbsége nem használható konstansként szorzási, osztási vagy logikai kifejezésekben. A konstans definíciókban alkalmazható kifejezések processzorutasításokban is használhatók konstansként. Bizonyos assemblerek rendelkeznek a makrózás lehetőségével, amellyel több utasítás egy csoportba fogva külön névvel látható el, az *as88* azonban ezt nem támogatja.

Minden assembly nyelvben vannak bizonyos direktívák, amelyek nem fordulnak bináris kódra, hanem magát az assembly folyamatot befolyásolják. Ezeket **pszeudoutasítások**nak nevezzük. Az *as88* pszeudoutasításait a C.8. ábrán soroljuk fel.

Utasítás	Leírás
.SECT .TEXT	A következő sorokat a TEXT szegmensben fordítsa
.SECT .DATA	A következő sorokat a DATA szegmensben fordítsa
.SECT .BSS	A következő sorokat a BSS szegmensben fordítsa
.BYTE	Az argumentumokat bájtok sorozataként kezelje
.WORD	Az argumentumokat szavak sorozataként kezelje
.LONG	Az argumentumokat hosszú szavak (long) sorozataként kezelje
.ASCII "str"	str-t ascii-ként tárolja egy stringben záró nulla bájt nélkül
.ASCIZ "str"	str-t ascii-ként tárolja egy stringben záró nulla bájttal
.SPACE n	A helyszámlálót léptesse előre n pozícióval
.ALIGN n	A helyszámlálót léptesse előre a következő n bájtos határra
.EXTERN	Az azonosító egy külső név

C.8. ábra. *Az as88 pszeudoutasításai*

A pszeudoutasítások első blokkja azt a szegmenst határozza meg, amelyikben a következő sorokat az assemblernek fel kell dolgoznia. Általában az ilyen szegmens követelményeket külön sorban adjuk meg, és azok a kódban bárhol elhelyezhetők. Megvalósítási okokból mindig a TEXT szegmens az első, ezt követi a DATA, majd a BSS szegmens. E kezdeti hivatkozások után a szegmensek már tetszőleges sorrendben használhatók. Ezen túlmenően minden szegmens első sorát el kell látni egy globális címkével. Egyéb megkötések nincsenek a szegmensek sorrendiségére.

A második blokkban találhatók az adattípusjelzők az adatszegmens számára. Négy típus van: .BYTE, .WORD, .LONG, és string. Egy opcionális címkét és a pszeudoutasítás kulcsszavát követően az első három típus a sor további részében konstans kifejezéseknek vesszőkkel elválasztott listáját várja. A stringekre vonatkozó két kulcsszó, az .ASCII és az .ASCIZ, csak annyiban különböznek, hogy a második kulcsszó még egy nulla bájtot is elhelyez a string végén. A stringet mindkettő idézőjelek között várja. A string-definíciókban többféle kiterjesztett kód használható. Ezeket a C.9. ábra tartalmazza. Ezeken kívül bármely speciális karakter beilleszthető egy fordított per (backslash) karakterrel oktális ábrázolásban, például a \377 (legfeljebb három oktális számjegy, nem szükséges vezető 0).

Kiterjesztett kód	Leírás
\n	újsor (soremelés)
\t	tabulátor
\\	fordított per (backslash, \)
\b	visszalépés
\f	lapemelés
\r	kocsi vissza
\"	idézőjel

C.9. ábra. *Néhány az as88-ban megengedett kiterjesztett kód*

A .SPACE pszeudoutasításnak csak a helyszámlálót kell az argumentumban megadott számú bájttal növelnie. Ez a kulcsszó különösen hasznos egy címkét követően a BSS szegmensben változók számára memória foglalására. Az .ALIGN kulcsszó a helyszámlálót lépteti a legközelebbi 2, 4 vagy 8 bájtos határra eső memóriahelyre, hogy a szavak, hosszú szavak stb. megfelelő helyre kerülhessenek. Végül az .EXTERN kulcsszó azt jelzi, hogy a megnevezett rutin vagy memóriahely a szerkesztő számára külső hivatkozásként áll majd rendelkezésre. A definíciónak nem kell az aktuális fájlban szerepelnie, az valahol máshol is lehet, csak a szerkesztőnek kell tudnia a hivatkozást lekezelni.

Bár maga az assembler elég általános, ha a nyomkövetővel együtt használjuk, bizonyos dolgokra oda kell figyelnünk. Az assembler a kulcsszavakat akár nagybetűs akár kisbetűs írásmóddal elfogadja, a nyomkövető viszont mindig nagybetűsen jeleníti meg. Hasonlóképpen, az assembler megengedi a „\r" (kocsi vissza) és a „\n" (soremelés) karaktereket is az új sor jelzésére, a nyomkövető viszont mindig az utóbbit használja. Továbbá annak ellenére, hogy az assembler képes kezelni a több fájlra darabolt programokat is, a nyomkövetővel történő használathoz az egész programnak egyetlen, egy „.$" kiterjesztéssel rendelkező fájlban kell elhelyezkedni. Ezen belül fájlok beillesztését a

```
#include fájlnév
```

parancsal lehet kérni. Ebben az esetben a kívánt fájl szintén a kombinált „.$" fájlba íródik a kérés helyétől kezdődően. Az assembler ellenőrzi, hogy a beillesztendő fájlt már feldolgozta-e, és csak egy példányt tölt be. Ez különösen hasznos, ha több fájl ugyanazokat a fejlécfájlokat használja. Ilyenkor a kombinált forrásfájlba csak egy példány kerül beillesztésre. A fájl beillesztéséhez szükséges, hogy a *#include* a sor első tokenje legyen vezető szóközök nélkül, a fájl útvonalát pedig idézőjelek között kell megadnunk.

Ha csak egyetlen forrásfájlunk van, mondjuk a *pr.s*, akkor feltételezzük, hogy a projekt neve *pr*, a kombinált fájl pedig a *pr.$* lesz. Ha egynél több forrásfájlunk van, akkor az assembler az első fájl nevének tövét tekinti a projekt nevének, és használja a *.$* fájl definiálásához, amelyet utána a forrásfájlok összemásolásával állít elő. Ez a viselkedés felülbírálható a parancssoron az első forrásfájl megadását megelőzően megadott „-o projname" kapcsolóval, amikor is a kombinált fájl a *projname.$* lesz.

Megjegyezzük azonban, hogy a beillesztett fájlok, és egynél több forrásfájl használatának vannak hátrányai is. A címkéknek, változóknak és konstansoknak az összes forrásban különbözőnek kell lenni. Ezen kívül, a ténylegesen fordításra kerülő fájl a *projname.$*, tehát arra vonatkoznak az assembler által a hibák és figyelmeztetések jelzésekor kiírt sorszámok. Nagyon kis projektek esetén néha az a legegyszerűbb, ha az egész programot egyetlen fájlban helyezzük, és elkerüljük a *#include* használatát.

C.5.3. Eltérések más 8088-as assemblerektől

Az *as88* assembler a szabványos UNIX assembler mintájára készült, így több tekintetben is eltér a Microsoft Macro Assembler-től (MASM) és a Borland 8088-as assemblerétől (TASM). Ezt a két assemblert kifejezetten az MS-DOS operációs rendszerre tervezték, és helyenként az assembler és az operációs rendszer kérdései szorosan összefonódnak. A MASM és a TASM is támogatja a 8088-asnak az MS-DOS alatt támogatott összes memóriamodelljét. Létezik például **tiny** modell, amelyben az összes kódnak és adatnak el kell férnie 64 KB-on, **small** modell, amelyben a kódszegmens és az adatszegmens külön-külön 64 KB-os lehet, és **large** modell, amely több kód- és adatszegmenst is megenged. A különböző modellek közötti eltérés a szegmensregiszterek használatában van. A large modell megengedi a távoli hívásokat és a DS regiszter módosítását. A szegmensregiszterekkel kapcsolatban a processzor is támaszt feltételeket (például a CS regiszter nem lehet MOV utasítás célja). A nyomkövetés egyszerűbbé tételére az *as88*-ban használt memóriamodell a small modellre hasonlít, bár az assembler a nyomkövető nélkül a szegmensregisztereket egyéb megszorítások nélkül is képes kezelni.

Ezek a másik assemblerek nem rendelkeznek a BSS szegmenssel, és a memóriát csak a DATA szegmensekben inicializálják. Az assembly fájl általában bizonyos fejlécinformációval kezdődik, majd ezt követi a .data kulcsszóval bevezetett DATA szegmens, amelyet a .code kulcsszóval bevezetett programszöveg követ. A fejlécben a program neve a .title kulcsszóval van megadva, a .model kulcsszó jelöli a memóriamodellt, a .stack kulcsszó pedig a veremszegmens számára foglal helyet. Ha a tervezett bináris program egy *.com* fájl, akkor a tiny modellt használjuk, az összes szegmensregiszter értéke megegyezik, és ennek az összetett szegmensnek az első 256 bájtos fejléce az ún. programszegmens-előtét (Program Segment Prefix, PSP) számára van fenntartva.

A .WORD, .BYTE, és .ASCIZ direktívák helyett ezek az assemblerek a DW kulcsszóval definiálnak szavas, a DB kulcsszóval pedig bájtos adatot. Egy string a DB direktíva után adható meg idézőjelek között. Az adatdefiníciókra használt címkéket nem követi kettőspont. Nagy memóriadarabokat a DUP kulcsszóval inicializálhatunk, amelyet a darabszám előz meg, és az iniciális érték követ. Például a:

```
LABEL DB 1000 DUP (0)
```

utasítás a LABEL címkétől kezdődően 1000 bájtnyi memóriát inicializál nulla bájtokkal.

Ezenfelül, a szubrutinok címkéit sem kettőspont zárja, hanem a PROC kulcsszó. A szubrutin végén pedig a címkét az ENDP kulcsszóval követve meg kell ismételni, így az assembler ki tudja következtetni a szubrutin pontos hatókörét. Lokális címkéket nem támogatnak.

Az utasítások kulcsszavai a MASM, a TASM, és az *as88* esetén megegyeznek. A kétoperandusú utasításoknál a forrásoperandus mindegyiknél a céloperandus után következik. Szokásos gyakorlat azonban, hogy a függvények argumentumait regiszterekben adják át, nem pedig a vermen keresztül. Ha a rutinokat C vagy C++ programokban használjuk, akkor javasolt a verem használata, hogy kompatibilisek legyünk a C szubrutin hívási mechanizmusával. Ez azonban nem lényeges eltérés, hiszen az argumentumok átadására az *as88*-ban is használhatunk regisztereket.

A MASM, a TASM, és az *as88* közötti legnagyobb különbség a rendszerhívásokban van. A MASM és a TASM esetén a rendszert az INT rendszermegszakítással hívjuk. A leggyakrabban használt az INT 21H, amely az MS-DOS függvényeinek hívására szolgál. Mivel a hívó sorszámot az AX regiszterbe tesszük, így ismét regiszterben adunk át argumentumot. A különböző eszközökhöz különböző megszakításvektorok és megszakítás számok tartoznak, mint például az INT 16H a BIOS billentyűzetkezelő függvényeihez, az INT 10H pedig a megjelenítéshez. Ezeknek a függvényeknek a programozásához a programozónak jókora adag eszközfüggő ismerettel tisztában kell lennie. Ezzel szemben az *as88*-ban található UNIX-rendszerhívásokat sokkal egyszerűbb használni.

C.6. A nyomkövető

A nyomkövető-hibakereső úgy készült, hogy az egy közönséges (VT100), 24 × 80 karakteres, az ANSI szabványos terminálparancsokat támogató terminálon fusson. UNIX- és Linux-gépeken az X-Window-rendszer terminálemulátora általában megfelel ennek a követelménynek. Windows-gépeken azonban többnyire be kell tölteni az *ansi.sys* meghajtót a rendszerinicializáló fájlokba az alább részletezett módon. A nyomkövető példákban már láttuk a nyomkövető ablakainak elrendezését. A C.10. ábrán látható, hogy a nyomkövető képernyője hét ablakra oszlik.

A bal felső ablak a processzor ablaka, amely az általános regisztereket decimális, a többi regisztert pedig hexadecimális jelöléssel mutatja. Mivel az utasításszámláló numerikus értéke nem mond túl sokat, ezért egy sorral lejjebb az is szerepel, hogy hány utasítással vagyunk a program forráskódjában a megelőző globá-

C.10. ábra. *A nyomkövető ablakai*

lis címke után. Az utasításszámláló mező felett látható az öt feltételkód. A túlcsordulást egy „v", az irány-flaget pedig egy „>" vagy „<" jelöli attól függően, hogy növekvő vagy csökkenő irányról van-e szó. Az előjel-flag a negatívok esetén „n", a nulla és a pozitívok esetén pedig „p". A zéró flaget „z" jelöli, ha be van állítva, az átvitel-flaget pedig „c". A „–" a flag törölt állapotát jelzi.

A felső középső ablakban a verem tartalma látható hexadecimálisan. A veremmutató pozícióját a „=>" nyíl mutatja. A szubrutinok visszatérési címét a hexadecimális érték előtti számjegy jelöli. A jobb felső ablakban a forrásfájlnak a következő végrehajtandó utasítás környezetét mutató részlete olvasható. Az utasításszámláló pozícióját szintén egy „=>" nyíl mutatja.

A processzor alatti ablak jeleníti meg a legutóbbi szubrutinhívások helyeit. Közvetlenül alatta a nyomkövető parancsok ablaka helyezkedik el, amelyben az előzőleg kiadott parancs van felül, alatta pedig a parancs kurzor. Megjegyezzük, hogy minden parancsot a RETURN (PC-billentyűzeten többnyire ENTER felirattal jelzett) billentyűvel kell lezárni.

Az alsó ablak a globális memória hat elemét tartalmazza. Minden elem valamely címkéhez képest megadott pozícióval kezdődik, ezt követi az adatszegmensben elfoglalt abszolút pozíció. A kettőspont után 8 bájt következik hexadecimálisan. A következő 11 pozíciókarakterek számára fenntartott, majd azt követi egy decimális szó. A bájtok, a karakterek és a szavak mind ugyanazt a memóriatartalmat jelenítik meg, de a karakteres ábrázoláshoz van három extrabájt, decimálisan pedig csak az első szó (2 bájt) értéke jelenik meg. Ez azért kényelmes, mert az adat elejéből nem mindig derül ki egyértelműen, hogy az előjeles vagy előjel nélküli egészként, vagy pedig stringként használatos.

A jobb középső ablak a be- és kimenet számára van fenntartva. A nyomkövető ennek első sorába írja ki a hibákat, a második sor a bemeneté, a többi pedig a kimenet számára áll rendelkezésre. A hiba kimenetet egy „E" betű, a bemenetet egy „I" betű, a szabványos kimenetet pedig „>" vezeti be. A bemeneti soron egy „–>" nyíl jelöli a következő beolvasás helyét. Ha a program *read* vagy *getchar* hívást végez, a nyomkövető parancssorából a következő bemenet a bemeneti mezőbe kerül. Ilyenkor is le kell zárni a bemeneti sort RETURN-nel. A sornak a még fel nem dolgozott része a „–>" nyíl után látható.

Általában a nyomkövető a parancsait és a bemenetét is a szabványos bemenetről olvassa. Lehetőség van azonban arra, hogy előkészítsünk egy-egy fájlt a nyomkövető parancsaiból illetve a bemeneti sorokból, amelyekből azután a nyomkövető a szabványos bemenetet megelőzően olvas. A nyomkövető parancsfájljai a *.t*, a bemeneti fájlok pedig az *.i* kiterjesztéssel vannak ellátva. Az assembly nyelvben a kulcsszavakra, a rendszerszubrutinokra és a pszeudoutasításokra a nagybetűs és a kisbetűs írásmód is használható. Az assembly folyamatban keletkezik egy *.$* kiterjesztésű fájl, amelybe a kisbetűs írásmódú kulcsszavak nagybetűsre konvertálva, a kocsivissza karakterek pedig egyáltalán nem kerülnek át. Így egy projekthez, mondjuk a *pr* nevűhöz, akár hat különböző fájl is tartozhat:

1. *pr.s* az assembly forráskódhoz;
2. *pr.$* az összeállított forráskódhoz;
3. *pr.88* a betöltő fájlhoz;
4. *pr.i* az előre összeállított szabványos bemenethez;
5. *pr.t* az előre összeállított nyomkövető parancsokhoz;
6. *pr.#* az assembly ködnak a betöltő fájhoz kapcsolásához.

Az utolsó fájlt a nyomkövető használja a jobb felső ablak kitöltésére és az utasításszámláló mező megjelenítésére. A nyomkövető ellenőrzi továbbá azt is, hogy a betöltő fájl a forrásprogram utolsó módosítása után keletkezett-e, és ha nem, akkor figyelmeztet.

C.6.1. Nyomkövető parancsok

A nyomkövető parancsait C.11. ábra sorolja fel. A legfontosabb közülük a táblázat első sorában található egyszerű return parancs, amely pontosan egy processzorutasítást hajt végre, valamint a táblázat alján található *q* kilépő parancs. Ha parancsként egy számot adunk meg, akkor annyi utasítás hajtódik végre. A *k* szám ekvivalens *k* darab return leütésével. Ugyanez a hatás érhető el akkor, ha a számot egy felkiáltójel (*!*) vagy egy *X* követi.

A *g* parancsot a forrásfájl adott soráig történő futáshoz használhatjuk. A parancsnak három változata létezik. Ha egy sorszám előzi meg, akkor a nyomkövető addig hajtja végre a programot, amíg az adott sorra nem ér. Egy */T* címkével – akár +# eltolással (offset) akár anélkül – annak a sornak a meghatározása, melyen a végrehajtást szeretnénk megállítani. A minden előtét nélküli *g* parancs hatására a nyomkövető az aktuális sor újbóli eléréséig hajtja végre az utasításokat.

A / (*label*) parancs eltérően viselkedik utasításcímkék és adatcímkék esetén. Adatcímke hatására az alsó ablak egy sora a címkénél kezdődő adatokkal lesz kitöltve illetve felülírva. Utasításcímke esetén ekvivalens a *g* parancsal. A címkét követheti egy plusz jel és egy szám (amelyet a C.11. ábrában a # jelez); ez a címkétől számított eltolást (offset) adja meg.

Lehetőség van arra, hogy egy utasításon **töréspontot (breakpoint)** helyezzünk el. Ezt a *b* parancsal tehetjük meg, amelyet opcionálisan egy utasításcímke és

Cím	Parancs	Példa	Leírás
			Egy utasítás végrehajtása
#	, !, X	24	# számú utasítás végrehajtása
/T+#	g, !,	/start+5g	Futtatás a T címkét követő #. sorig
/T+#	b	/start+5b	Töréspont elhelyezése a T címkét követő #. sorra
/T+#	c	/start+5c	Töréspont eltávolítása a T címkét követő #. sorról
#	g	108g	A program végrehajtás a #. sorig
	g	g	A program végrehajtása, míg az aktuális sor újra elő nem fordul
	b	b	Töréspont elhelyezése az aktuális sorra
	c	c	Töréspont eltávolítása az aktuális sorról
	n	n	A program végrehajtása a következő sorig
	r	r	Végrehajtás a következő töréspontig vagy a program végéig
	=	=	A program futtatása, amíg azonos szubrutinszinten marad
	–	–	Futtatás az aktuálisnál eggyel alacsonyabb szubrutinszintig
	+	+	Futtatás az aktuálisnál eggyel magasabb szubrutinszintig
/D+#		/buf+6	Az adatszegmens megjelenítése a D címke+# címtől
/D+#	d, !	/buf+6d	Az adatszegmens megjelenítése a D címke+# címtől
	R, CTRL L	R	Ablakok frissítése
	q	q	Nyomkövetés befejezése, visszatérés a parancsértelmezőbe

C.11. ábra. *A nyomkövető parancsai. Minden parancsot return karakterrel (az ENTER billentyű) kell lezárni. Az üres mező azt jelzi, hogy csak a return szükséges. A fent felsorolt, Cím mező nélküli parancsok nem rendelkeznek címmel. A # szimbólum egy egészértékű eltolást (offset) reprezentál*

esetleg egy eltolás (offset) előzhet meg. Amikor a futtatás a töréspontal ellátott sorra ér, a nyomkövető megáll. A töréspontól kezdve folytatható a futtatás a RETURN billentyűvel vagy a futtató (*r*) paranccsal. Ha a címkét és a számot elhagyjuk, akkor a töréspont az aktuális sorra kerül. A töréspontot a *c* paranccsal lehet eltávolítani, amelyet a *b* parancshoz hasonlóan megelőzhet címke és szám. Az *r* futtató parancs hatására a nyomkövető addig hajtja végre az utasításokat, amíg egy töréspontlhoz, egy exit híváshoz, vagy az utasítások végére nem ér.

A nyomkövető nyilvántartja azt a szubrutin hívási szintet, amelyen éppen fut a program. Ez a processzorablak alatt, valamint a veremablakban megjelenő jelző számokból látszik. Három parancs foglalkozik ezekkel a szintekkel. A – parancs hatására a nyomkövető addig fut, amíg a szubrutinszint eggyel az aktuális szint alá csökken. Ez a parancs valójában addig hajtja végre az utasításokat, amíg az aktuális szubrutin be nem fejeződik. Ennek fordítottja a + parancs, amelynek hatására a nyomkövető addig fut, amíg a következő szubrutinszintre nem ér. Az = parancs addig fut, amíg ugyanazt a szintet el nem érjük, és ezt egy CALL hívásnál a szubrutin végrehajtására használhatjuk. Ha az = parancsot használjuk, a szubrutin részletei nem jelennek meg a nyomkövető ablakban. A hasonló *n* parancs, a következő programsorig fut. Ez a parancs különösen hasznos, ha egy LOOP parancson adjuk ki, ilyenkor a végrehajtás pontosan a ciklus végén áll meg.

C.7. Alapismeretek

Ebben a fejezetben elmagyarázzuk, hogyan is használjuk ezeket az eszközöket. Mindenekelőtt meg kell találnunk a platformunknak megfelelő szoftvert. Előre lefordított változatokat találunk Solaris-, UNIX-, Linux- és Windows-rendszerekre. Az eszközök megtalálhatók a CD-ROM-mellékleten és a *www.prenhall.com/tanenbaum* webcímen is. Az utóbbi helyen kattintsunk a bal oldali menüben ennek a könyvnek a *Companion Web Site* linkjére. A kiválasztott zip fájlt egy *assembler* nevű könyvtárban tömörítsük ki. Ez a könyvtár és alkönyvtárai tartalmaznak minden szükséges anyagot. A CD-ROM-on a fő könyvtárak a *BigendNx*, *LtlendNx*, *MSWindos*, és mindegyikben található egy *assembler* alkönyvtár, amely az anyagot tartalmazza. A legfelső szinten található három könyvtár rendre a Big-Endian UNIX- (például Sun-munkaállomások), a Little-Endian UNIX- (például PC-n futó Linuxok) és a Windows-rendszerek számára.*

Kitömörítés vagy másolás után az assembler könyvtárban az alábbi alkönyvtárakat és fájlokat kell találnunk: *READ_ME*, *bin*, *as_src*, *trce_src*, *examples*, és *exercise*. Az előre lefordított források a *bin* könyvtárban találhatók, de kényelmi okokból egy másolatuk az *examples* könyvtárban is megvan.

Hogy gyors áttekintést kapjunk a rendszer működéséről, váltsunk az *examples* könyvtárba, és gépeljük be az alábbi parancsot:

```
t88 HlloWrld
```

Ez a parancs tartozik a C.8. alfejezet első példájához.

Az assembler forráskódja az *as_src* alkönyvtárban található. A forrásfájlok C nyelvűek, és a *make* parancs használatával lehet őket újrafordítani. POSIX-kompatibilis platformokhoz van a forráskönyvtárban erre egy *Makefile*. Windows-rendszerekhez itt egy *make.bat* kötegfájl található. Szükség lehet arra is, hogy a fordítás után a futtatható állományokat egy program könyvtárba másoljuk, vagy pedig a PATH környezeti változót módosítsuk úgy, hogy az *as88* assembler és a *t88* nyomkövető (tracer) az assembly forráskódot tartalmazó könyvárakból is indítható legyen. Máskülönben a *t88* egyszerű begépelése helyett a teljes útvonal nevét kell használni.

Windows 2000- és XP-rendszereken szükség lehet az *ansi.sys* terminálmeghajtó telepítésére, amelyet a következő sornak a *config.nt* konfigurációs fájlhoz adásával tehetünk meg:

```
device=%systemRoot%\System32\ansi.sys
```

Ezt a fájlt az alábbi helyen találjuk:

```
Windows 2000:    \winnt\system32\config.nt
Windows XP:      \windows\system32\config.nt
```

* A CD-ROM-on az említett fő könyvtárak a *8088_tra* könyvtárban *solaris*, *linux* és *windows* néven találhatók. *(A lektor)*

Windows 95-, 98- és ME-rendszereken a meghajtót a *config.sys* fájlba kell bejegyezni. UNIX- és Linux-rendszereken a meghajtó általában standard módon rendelkezésre áll.

C.8. Példák

A C.2-től a C.4-ig tartó fejezetek a 8088-as processzorról, annak memóriájáról és utasításairól szólt. A C.5. fejezetben az ebben az útmutatóban használt *as88* assembly nyelvet tanulmányoztuk. A C.6. fejezetben a nyomkövetőről ejtettünk szót, és végül a C.7. fejezetben leírtuk az eszközkészlet telepítésének menetét. Elvileg ennyi információ elegendő ahhoz, hogy a rendelkezésre álló eszközökkel assembly programokat tudjunk írni, és azokban hibát keresni. Ennek ellenére sok olvasó biztosan hasznosnak találja, ha bemutatunk néhány részletesebb assembly program példát, és hogy a nyomkövetővel hogyan lehet bennük a hibákat megkeresni. Pontosan ez ennek a fejezetnek a célja. A fejezetben tárgyalt összes példaprogram megtalálható az eszköz *examples* alkönyvtárában.* Arra bíztatjuk az olvasót, hogy az itt tárgyaltak szerint fordítsa le és ellenőrizze a programokat.

C.8.1. Helló Világ példa

Kezdjük a C.12. ábra *HlloWrld.s* példájával. A program listája a bal oldali ablakban látható. Mivel az assembler megjegyzés szimbóluma a felkiáltójel (!), ezt használjuk a programablakban az utasítások és az azokat követő sorszámok elválasztására. Az első három sor konstans definíciókat tartalmaz, amelyek két rendszerhíváshoz és a kimeneti fájlhoz tartozó belső ábrázoláshoz rendelnek szokásos neveket.

A 4. sorban a .SECT pszeudoutasítás azt állítja, hogy a következő sorokat a TEXT szegmens részeként kell tekinteni, azaz ezek processzorutasítások. Hasonlóan, a 17. sor azt mondja, hogy ami következik, azt adatnak kell tekinteni. A 19. sor inicializál egy 12 bájtból álló string-adatot, amelybe a szóköz és a végén található soremelés (\n) is beleértendő.

Az 5., 18. és 20. sor címkéket tartalmaz, amelyeket a kettőspont (:) jelöl. Ezek a címkék a konstansokhoz hasonlóan numerikus értékeket reprezentálnak. Itt azonban az assemblernek kell a numerikus értéket meghatároznia. Mivel a *start* a TEXT szegmens legelején helyezkedik el, az értéke 0 lesz, de a TEXT szegmens további címkéinek értéke (amelyet ez a példa nem tartalmaz) attól függne, hogy hány bájtnyi kód van előttük. Most tekintsük a 6. sort. Ez a sor két címke különbségével végződik, amely numerikusan egy konstans. Így a 6. sor lényegében ugyanaz, mint

```
MOV CX,12
```

* A CD-ROM-melléklet *examples* alkönyvtárában a példák és nyomkövető ablakok néhol kissé eltérnek a könyvbeli példáktól és ábráktól. *(A lektor)*


```
_EXIT = 1                    ! 1
_WRITE = 4                   ! 2
_STDOUT = 1                  ! 3
.SECT .TEXT                  ! 4
start:                       ! 5
    MOV CX,de-hw             ! 6
    PUSH CX                  ! 7
    PUSH hw                  ! 8
    PUSH _STDOUT             ! 9
    PUSH _WRITE              ! 10
    SYS                      ! 11
    ADD SP, 8                ! 12
    SUB CX,AX                ! 13
    PUSH CX                  ! 14
    PUSH _EXIT               ! 15
    SYS                      ! 16
.SECT .DATA                  ! 17
hw:                          ! 18
.ASCII "Hello World\n"       ! 19
de: .BYTE 0                  ! 20
```

(a)

```
CS: 00   DS=SS=ES:  002
AH:00  AL:0c   AX:    12
BH:00  BL:00   BX:     0
CH:00  CL:0c   CX:    12
DH:00  DL:00   DX:     0
SP: 7fd8  SF  O D S Z C   =>0004
BP: 0000  CC  - > p -       0001
SI: 0000   IP:000c:PC       0000
DI: 0000   start + 7        000c

hw
■

hw + 0 = 0000: 48 65 6c 6c 6f 20 57 6f Hello World 25928
```

```
    MOV CX,de-hw             ! 6
    PUSH CX                  ! 7
    PUSH HW                  ! 8
    PUSH _STDOUT             ! 9
    PUSH _WRITE              ! 10
    SYS                      ! 11
=>  ADD SP,8                 ! 12
    SUB CX,AX                ! 13
    PUSH CX                  ! 14

E
|

> Hello World\n
```

(b)

C.12. ábra. *(a) HlloWrld.s. (b) A hozzá tartozó nyomkövető ablak*

kivéve, hogy a programozó helyett az assemblerre bízza a string hosszának meghatározását. Az itt megadott érték annak a helynek a mérete, amelyet a 19. sorban adott string számára foglaltunk le. A MOV a 6. sorban a másoló utasítás, amely a *de–hw* értékét másolja CX-be.

A 7–11. sor azt mutatja, hogy az eszközben hogyan kell rendszerhívást végrehajtani. Ez az öt sor a C nyelvű:

```
write(1, hw, 12);
```

függvényhívás assembly kódú fordítása, ahol az első paraméter a szabványos kimenet fájlleírója (1), a második a kinyomtatandó string (*hw*) címe, a harmadik pedig a string hossza (12). A 7–9. sor ezeket a paramétereket fordított sorrendben a veremre helyezi, amely a nyomkövető által is használt C hívási szekvencia. A 10. sor a write rendszerhívás sorszámát (4) teszi a veremre, majd a 11. sorban történik a tényleges hívás. Bár ez a hívási szekvencia nagyon hasonlít ahhoz, ahogy egy valódi assembly nyelvű program működne egy UNIX- (vagy Linux-) rendszerű PC-n, más operációs rendszeren kicsit módosítani kellene, hogy annak az operációs rendszernek a hívási szabályait használja. Az *as88* assembler és a *t88* nyomkövető azonban a Windows-rendszeren futtatva is a UNIX hívási szabályokat használja.

A 11. sorban található rendszerhívás végzi el a tényleges nyomtatást. A 12. sor takarítja a vermet azáltal, hogy a veremmutatót visszaállítja arra az értékre, amelylyel a négy darab 2 bájtos szó veremre helyezése előtt rendelkezett. Ha a write hívás sikeres, az AX-ben a kinyomtatott bájtok számát kapjuk vissza. A 13. sor kivonja a 11. sor után a rendszerhívás eredményét a CX-ben található eredeti string hosszból, hogy ellenőrizze, valóban minden bájt kinyomtatásra került-e. Így a program kilépési állapota 0 lesz sikeres futás esetén, és valami ettől különböző, ha hiba történt. A 14. és 15. sor a 16. sorban található exit rendszerhívást készíti elő

azzal, hogy a kilépési állapotot és a függvény kódját a veremre teszi az EXIT hívás számára.

Felhívjuk a figyelmet arra, hogy a MOV és a SUB utasítás esetén az első argumentum a céloperandus és a második a forrásoperandus. Assemblerünk ezt a konvenciót követi, más assemblerek ezt a sorrendet felcserélhetik. Semmilyen különös okunk nem volt az egyik sorrend választására a másikkal szemben.

Most próbáljuk meg a *HlloWrld.s* programot lefordítani és futtatni. Utasításokat adunk mind a UNIX-, mind a Windows-platformhoz. Linux-, Solaris-, MacOS X- és egyéb UNIX-változatokon az eljárás lényegében ugyanaz, mint a UNIX-on. Először indítsunk egy parancsértelmező (héj, shell) ablakot. Windowson a kattintási sorozat általában

```
Start > Programok > Kellékek > Parancssor
```

Angol nyelvű Windowson:

```
Start > Programs > Accessories > Command prompt
```

Ezután váltsunk az *examples* alkönyvtárba a *cd* (change directory, munkakönyvtár váltása) paranccsal. E parancs argumentumát attól függően kell megadni, hogy az eszközt hova helyeztük el a fájlrendszerben. Utána ellenőrizzük UNIX-on az *ls*, Windowson a *dir* paranccsal, hogy a könyvtárban megtalálhatók-e az assembler és a nyomkövető bináris állományai. Ezek neve rendre *as88* és *t88*. Windows-rendszereken van még egy *.exe* kiterjesztésük, de azt nem kell begépelni a parancsokban. Ha az assembler és a nyomkövető nincsenek ott, akkor keressük meg, és másoljuk oda.

Most fordítsuk le a teszt programot az alábbi paranccsal:

```
as88 HlloWrld.s
```

Ha az assembler az *examples* könyvtárban van, de a parancs mégis hibát jelez, akkor próbáljuk meg UNIX-rendszeren az alábbit:

```
./as88 Hllowrld.s
```

vagy Windows-rendszeren a következőt:

```
.\as88 Hllowrld.s
```

Ha az assembly folyamat helyesen fejeződik be, a következő üzenet jelenik meg:

```
Project HlloWrld listfile HlloWrld.$
Project HlloWrld num file HlloWrld.#
Project HlloWrld loadfile HlloWrld.88
```


és létrejön a három megfelelő fájl. Ha nincs hibaüzenet, akkor adjuk ki a nyomkövető parancsot:

```
t88 HlloWrld
```

A nyomkövető képernyőben a jobb felső sarokban megjelenő nyíl a 6. sor

```
MOV CX,de-hw
```

utasítására mutat. Most üssük le a RETURN (PC-billentyűzeten ENTER) billentyűt. Vegyük észre, hogy most a mutatott utasítás a:

```
PUSH CX
```

és hogy a CX értéke a bal oldali ablakban most 12. Üssük le újra a RETURN-t és vegyük észre, hogy a felső sorban a középső ablak a 000c értéket tartalmazza, amely a 12 hexadecimális alakja. Ez az ablak a vermet mutatja, amely most egyetlen szót tartalmaz a 12 értékkel. Üssük le a RETURN-t még háromszor, hogy lássuk a 8., 9. és 10. sor PUSH utasításainak végrehajtását. Ezen a ponton a verem már négy elemet tartalmaz, a bal oldali ablakban az utasításszámlálóban pedig a 000b értéket találjuk.

A RETURN következő lenyomására végrehajtódik a rendszerhívás, és a „Hello World\n" string megjelenik a jobb alsó ablakban. Figyeljük meg, hogy most az SP értéke 0x7ff0. A következő RETURN után az SP 8-cal nő és 0x7ff8-ra változik. Négy további RETURN után befejeződik az exit rendszerhívás, és a nyomkövető kilép.

Hogy meggyőződjön arról, valóban minden működését érti, töltse be a *HlloWrld.s* fájlt kedvenc szövegszerkesztőjébe (text editor). Jobb, ha nem szövegfeldolgozót (word processor) használ. UNIX-rendszereken jó választás az *ex*, a *vi* vagy az *emacs*. Windows-rendszeren a Jegyzettömb (*notepad*) egy egyszerű szerkesztő, és általában a:

```
Start > Programok > Kellékek > Jegyzettömb
```

vagy a:

```
Start > Programs > Accessories > Notepad
```

helyen érhető el. Ne használja a *Word*öt, mivel az eredményt helytelenül formázhatja, és így nem lesz jól olvasható.

Módosítsa a 19. sorban a stringet, hogy más üzenet jelenjen meg, majd mentse el a fájlt, fordítsa le, és futtassa a nyomkövetőben. Ezzel elkezdett assembly nyelven programozni.

C.8.2. Példa az általános regiszterekre

A következő példa részletesebben bemutatja a regiszterek megjelenítésének módját és a szorzás művelet buktatóit a 8088-ason. A C.13. ábrán a bal oldalon a *GenReg.s* program részlete, tőle jobbra pedig a program futásának két különböző fázisához tartozó nyomkövető ablak látható. A C.13. (b) ábra mutatja a regiszterek állapotát a 7. sor végrehajtása után. A 4. sor:

```
MOV AX,258
```

utasítása a 258-as értéket tölti AX-be, melynek hatására AH-ba 1, AL-be pedig 2 kerül. Ezután az 5. sor AL-t AH-hoz adja, így AH 3 lesz. A 6. sorban CX-be kerül a *times* változó tartalma (10). A 7. sorban pedig BX-be töltődik a *muldat* változó címe, amely 2, mivel ez a DATA szegmens második szava. Ebben az időpillanatban készült a C.13. (b) ábra képe. Megjegyezzük, hogy AH értéke 3, AL-é 2, és AX így 770, ahogy az várható is, hiszen 3 × 256 + 2 = 770.

```
start:                  ! 3
    MOV   AX,258        ! 4
    ADDB  AH,AL         ! 5
    MOV   CX,(times)    ! 6
    MOV   BX,muldat     ! 7
    MOV   AX,(BX)       ! 8
llp: MUL  2(BX)         ! 9
    LOOP  llp           ! 10
.SECT .DATA             ! 11
times: .WORD 10         ! 12
muldat :. WORD 625,2    ! 13
```

(a)

```
CS:  00   DS=SS=ES:   002
AH:03  AL:02   AX:    770
BH:00  BL:02   BX:      2
CH:00  CL:0a   CX:     10
DH:00  DL:00   DX:      0
SP:  7fe0  SF  O D S Z C
BP:  0000  CC  - > p - -
SI:  0000   IP:0009:PC
DI:  0000   start  +4
```

(b)

```
CS:  00   DS=SS=ES:   002
AH:38  AL:80   AX:  14464
BH:00  BL:02   BX:      2
CH:00  CL:04   CX:      4
DH:00  DL:01   DX:      1
SP:  7fe0  SF  O D S Z C
BP:  0000  CC  v > p - c
SI:  0000   IP:0011:PC
DI:  0000   start  +7
```

(c)

C.13. ábra. *(a) Programrészlet. (b) A nyomkövető regiszter ablaka, miután a 7. sor végrehajtásra került. (c) A nyomkövető ablak a ciklus 7. futásában, a szorzás után*

A következő utasítás (8. sor) a *muldat* tartalmát tölti AX-be. Ezáltal a RETURN billentyű lenyomása után AX értéke 625 lesz.

Most érkeztünk el a ciklusig, amely AX és a 2(BX) által megcímzett szó (azaz *muldat*+2) szorzata, ahol az utóbbi értéke 2. A MUL utasítás implicit céloperandusa a DX : AX hosszú szavas regiszter-összetétel. Az első iterációs lépésben az eredmény elfér egyetlen szóban, így AX tartalmazza az eredményt (1250), DX pedig 0 marad. A regisztereknek 7 szorzás utáni tartalmát mutatja a C.13. (c) ábra.

Mivel az AX kezdőértéke 625 volt, ezt hétszer megszorozva 2-vel az eredmény 80 000. Ez már nem fér el az AX-ben, de az eredmény a DX : AX regiszterek összefűzésével előálló 32 bites regiszterben tárolódik, tehát a DX az 1, az AX pedig a 14 464 értéket kapja. Numerikusan ez az érték 1 × 65 536 + 14 464, amely valóban 80 000. Figyeljük meg, hogy a CX értéke ekkor 4, mivel azt a LOOP utasítás minden egyes iterációs lépésben csökkenti eggyel. Mivel a CX 10-ről indult, hét MUL utasítás (de csak hat LOOP iterációs lépés) után az értéke 4 lesz.

A következő szorzásnál probléma jelentkezik. A szorzás csak az AX-et veszi figyelembe, a DX-et nem, tehát a MUL az AX (14 464) értékét szorozza 2-vel, és így 28 928-at ad eredményként. Ezáltal az AX értéke 28 928 lesz, a DX-é pedig 0, ami numerikusan helytelen.

```
_EXIT = 1                          ! 1   az _EXIT értékének definiálása
_PRINTF = 127                      ! 2   a _PRINTF értékének definiálása
.SECT .TEXT                        ! 3   a TEXT szegmens kezdete
inpstart:                          ! 4   az inpstart címke definiálása
        MOV   BP,SP                ! 5   SP mentése BP-be
        PUSH  vec2                 ! 6   vec2 címe a verembe
        PUSH  vec1                 ! 7   vec1 címe a verembe
        MOV   CX,vec2-vec1         ! 8   CX = a vektorban található bájtok száma
        SHR   CX,1                 ! 9   CX = a vektorban található szavak száma
        PUSH  CX                   ! 10  a szavak száma a verembe
        CALL  vecmul               ! 11  vecmul hívása
        MOV   (inprod),AX          ! 12  AX áttöltése
        PUSH  AX                   ! 13  a nyomtatandó eredmény a verembe
        PUSH  pfmt                 ! 14  a formátum-string cime a verembe
        PUSH  _PRINTF              ! 15  a PRINTF függvény kódja a verembe
        SYS                        ! 16  a PRINTF függvény hívása
        ADD   SP,12                ! 17  a verem takarítása
        PUSH  0                    ! 18  állapotkód a verembe
        PUSH  _EXIT                ! 19  az EXIT függvény kódja a verembe
        SYS                        ! 20  az EXIT függvény hívása

vecmul:                            ! 21  a vecmul(count, vec1, vec2) kezdete
        PUSH  BP                   ! 22  BP mentése a verembe
        MOV   BP,SP                ! 23  SP a BP-be az argumentumok eléréséhez
        MOV   CX,4(BP)             ! 24  a darabszám CX-be a ciklus vezérléséhez
        MOV   SI,6(BP)             ! 25  SI = vec1
        MOV   DI,8(BP)             ! 26  DI = vec2
        PUSH  0                    ! 27  0 a verembe
1:      LODS                       ! 28  (SI) betöltése AX-be
        MUL   (DI)                 ! 29  AX szorzása (DI)-vel
        ADD   -2(BP),AX            ! 30  AX hozzáadása a gyűjtő memóriaszóhoz
        ADD   DI,2                 ! 31  DI növelése a következő elemre
        LOOP  1b                   ! 32  ha CX > 0, vissza az 1b címkére
        POP   AX                   ! 33  verem tetejéről AX-be
        POP   BP                   ! 34  BP helyreállítása
        RET                        ! 35  visszatérés a szubrutinból

.SECT .DATA                        ! 36  DATA szegmens kezdete
pfmt: .ASCIZ "Inner product is: %d!\n"  ! 37  string definiálása
.ALIGN 2                           ! 38  páros címre pozicionálás
vec1: .WORD 3,4,7,11,3             ! 39  1. vektor
vec2: .WORD 2,6,3,1,0              ! 40  2. vektor
.SECT .BSS                         ! 41  BBS szegmens kezdete
inprod: .SPACE 2                   ! 42  helyfoglalás az inprod számára
```

C.14. ábra. *A vecprod.s program*

C.8.3. A call utasítás és a mutató regiszterek

A következő példa, a *vecprod.s*, egy olyan program, amely két vektor, a *vec1* és a *vec2*, belső szorzatát számítja ki. A listája a C.14. ábrán látható.

A program első része előkészíti a *vecmul* hívását az SP BP-be mentésével, majd a *vec2* és a *vec1* címének verembe helyezésével, hogy azokat a *vecmul* elérhesse. Ezután, a 8. sorban, CX-be kerül a vektor bájtokban mért hossza.* Ezt a 9. sorban egy bittel jobbra léptetve, CX most a vektorban tárolt szavak számát tartalmazza, és ez kerül a verembe a 10. sorban. A *vecmul* hívása a 11. sorban történik meg.

Érdemes újra megemlíteni, hogy a szubrutinok argumentumai, megállapodás szerint, fordított sorrendben kerülnek a veremre, hogy összhangban legyen a C hívási konvencióval. Ily módon a *vecmul* C-ből is hívható lenne az alábbi módon:

```
vecmul(count, vec1, vec2)
```

A CALL utasítás végrehajtása közben a visszatérési cím a verem tetejére kerül. Ha a programot nyomon követjük, kiderül, hogy ez a cím a 0x0011.

A szubrutin első utasítása, a BP bázismutató vermelése a 22. sorban a PUSH-sal. A BP-t elmentjük, mert a szubrutinon belül szükségünk van erre a regiszterre az argumentumok és a lokális változók címzéséhez. Következőnek a veremmutatót másoljuk a BP regiszterbe a 23. sorban, így a bázismutató új értéke annak régi értékére mutat.

Ezzel minden készen áll ahhoz, hogy az argumentumokat a regiszterekbe töltsük, és kialakítsuk a lokális változók helyét. A következő három sorban mindegyik argumentumot a veremből egy-egy regiszterbe tesszük. Emlékezzünk rá, hogy a verem szó-orientált, vagyis a veremcímek mindig párosak. A visszatérési cím a régi bázismutató mellett található, tehát annak címe 2(BP). A következő a *count* argumentum a 4(BP) címen. Ezt töltjük CX-be a 24. sorban. A 25. és 26. sorban *vec1* SI-be, *vec2* pedig DI-be kerül. Végül a 27. sorban a verembe tesszük a 0 értéket.

A processzornak közvetlenül a 28. sorban történő ciklusba lépés előtti állapota a C.15. ábrán látható. A felső sor keskeny középső ablaka (a regiszterektől jobbra) mutatja a vermet. A verem alján a *vec2* címe (0x0022), felette a *vec1* címe (0x0018), majd afelett a harmadik argumentum, a vektorokban található elemek száma (0x0005) található. Ezután jön a visszatérési cím (0x0011). A címtől balra elhelyezkedő 1-es szám azt jelzi, hogy ez a visszatérési cím a főprogramtól egy szintre helyezkedik el. A regiszterek alatti ablakban ugyanaz az 1-es szám jelenik meg, ezúttal a szimbolikus címet mutatva. A visszatérési cím felett a veremben a BP régi értéke (0x7fc0), majd a 27. sorban odahelyezett nulla található. Az erre mutató nyíl az SP által jelzett helyet mutatja. A veremtől jobbra lévő ablak a programszöveg egy részletét mutatja, ahol a nyíl a következőként végrehajtandó utasítást jelzi.

* A program kihasználja, hogy *vec2* közvetlenül *vec1* után helyezkedik el a memóriában. Ez általában nem szerencsés. *(A ford.)*


```
   MOV  BP,SP          !5
   PUSH vec2           !6
   PUSH vec1           !7
   MOV  CX,vec2-vec1   !8
   SHR  CX, 1          !9
   PUSH CX             !10
   CALL vecmul         !11
   ---------
vecmul :               !21
   PUSH BP             !22
   MOV  BP,SP          !23
   MOV  CX,4(BP)       !24
   MOV  SI,6(BP)       !25
   MOV  DI,8(BP)       !26
   PUSH 0              !27
1: LODS                !28
   MUL  (DI)           !29
   ADD  -2(BP),A       !30
   ADD  DI,2           !31
   LOOP 1b             !32
```

```
CS: 00   DS=SS=ES: 004
AH:00 AL:00   AX:      0
BH:00 BL:00   BX:      0
CH:00 CL:05   CX:      5   =>0000   PUSH BP          ! 22
DH:00 DL:00   DX:      0     7fc0   MOV  BP,SP       ! 23
SP: 7fb4 SF   O D S Z C  1  0011   MOV  CX,4(BP)     ! 24
BP: 7fb6 CC   - > p z -     0005   =>1: LODS         ! 25
SI: 0018   IP:0031:PC       0018   MUL  (DI)         ! 29
DI: 0022   vecmul+7         0022   ADD  -2(BP),AX    ! 30
```

```
1 <= inpstart + 7
```

```
vec1+0  =0018: 3  0  4  0  7  0  b  0 ..........      3
vec2+0  =0022: 2  0  6  0  3  0  1  0 ..........      2
pfmt+0  =0000:54 68 65 20 69 6e 20 70 The in prod 26708
pfmt+18 =0012:25 64 21  a  0  0  3  0 %d!..... 25637
```

C.15. ábra. *A* vecprod.s *végrehajtása, amint először fut rá a 28. sorra*

Most vizsgáljuk meg a 28. soron kezdődő ciklust. A LODS utasítás az adatszegmensből az SI regiszteren keresztül indirekt módon tölt be a memóriából AX-be. Mivel az irány-flag be van állítva, és a LODS automatikus növelés módban működik, így az utasítás után SI a *vec1* következő komponensére mutat.

Hogy a hatást a képernyőn is lássuk, indítsuk el a nyomkövetőt az alábbi parancsal:

```
t88 vecprod
```

Miután megjelenik a nyomkövető ablak, gépeljük be a:

```
/vecmul+7b
```

parancsot egy RETURN-nel lezárva, hogy a LODS utasítást tartalmazó sorra elhelyezzünk egy töréspontot. Ettől kezdve már nem fogjuk külön hangsúlyozni, hogy minden parancsot a RETURN billentyűvel kell lezárni. Aztán adjuk ki a:

```
g
```

parancsot, amire a nyomkövető egészen a töréspontig végrehajtja az utasításokat. Így a LODS műveletet tartalmazó soron fog megállni.

A 29. sorban AX értéke megszorzódik a forrásoperandussal. A MUL utasításhoz a memóriaszó az adatszegmensből DI-n keresztül, regiszter-indirekt módban kerül betöltésre. A MUL implicit céloperandusa a DX:AX hosszú szavas regiszter-összetétel, amelyet az utasításban nem kell külön megadnunk.

A 30. sorban az eredményt hozzáadjuk a verem -2(BP) címén található lokális változóhoz. Mivel a MUL nem növeli automatikusan az operandusát, ezt explicit módon kell megtenni a 31. sorban. Ezután DI a *vec2* következő komponensére mutat.

Ezt a részt a LOOP utasítás zárja. A CX regiszter csökken, és ha még mindig pozitív, akkor a program visszaugrik a lokális *1* címkére, a 28. sorra. A *1b* lokális címke alkalmazása az aktuális pozíciótól visszafelé haladva a legközelebbi *1*-es címkét jelenti. A ciklus után a szubrutin a visszatérési értéket a veremből AX-be teszi (33. sor), helyreállítja BP-t (34. sor), majd visszatér a hívó programhoz (35. sor).

A főprogram a hívás után a 12. sorban folytatódik a MOV utasítással. Ez az utasítás az első egy öt utasításból álló sorozatból, amelynek feladata az eredmény kinyomtatása. A printf rendszerhívás a szabványos C programkönyvtár *printf* függvénye mintájára készült. Három argumentum kerül a verembe a 13–15. sorban. Ezek az argumentumok a kinyomtatandó egész szám, a formátum-string (*pfmt*) címe és a printf függvény kódja (127). Vegyük észre, hogy a *pfmt* formátum-string egy *%d* bejegyzéssel jelzi, hogy a printf hívás argumentumában egy egész változó található, amely kiegészíti a kimenetet.

A 17. sor takarítja ki a vermet. Mivel a program az 5. sorban azzal kezdődött, hogy a veremmutatót a bázismutatóba töltöttük, a:

```
MOV SP,BP
```

utasítást is használhatnánk a verem takarítására. Az utóbbi megoldásnak az az előnye, hogy a programozónak nem kell a műveletsor alatt a vermet egyensúlyban tartania. A főprogram esetén ez persze nem nagy probléma, de a szubrutinoknál ez az elavult lokális változókhoz hasonló szemét eldobásának egy egyszerű módja.

A *vecmul* szubrutint más programokban is használhatjuk. Ha a *vecprod.s* forrásfájlt más assembly forrásfájlok mögé írjuk a parancssorban, akkor rendelkezésünkre áll a két fix hosszúságú vektor szorzására szolgáló szubrutin. Javasoljuk, hogy először távolítsuk el az _EXIT és _PRINTF konstans definíciókat, hogy elkerüljük azok többszöri definiálását. Ha a *syscalnr.h* fejlécfájlt beillesztjük valahova, akkor már máshol nem szükséges a rendszerhívás konstansokat definiálni.

C.8.4. Hibakeresés egy tömbkiíró programban

Az előző példákban a programok egyszerűek voltak, és hibátlanok. Most megmutatjuk, hogyan segíthet a nyomkövető a hibakeresésben hibás programok esetén. A következő programnak egy egész számokból álló tömböt kellene kinyomtatnia, amelyet a *vec1* címke után adunk meg. A program kezdeti változata azonban három hibát is tartalmaz. Ezeknek a hibáknak a kijavítására az assemblert és a nyomkövetőt használjuk majd, de először nézzük meg a kódot.

Mivel minden programnak szüksége van rendszerhívásokra, és ezért definiálnia kell konstansokat, melyekkel a hívási sorszámokat azonosíthatja, ezeket a konstans definíciókat a *../syscalnr.h* nevű külön fejlécfájlba tettük, amelyet a kód 1. sorában illesztünk be. Ez a fájl az alábbi, a folyamat elején megnyitásra kerülő fájlleírókhoz is definiál konstansokat:


```
STDIN=0
STDOUT=1
STDERR=2
```

valamint a kód- és adatszegmensek fejléccímkéit is definiálja. Érdemes ezt minden assembly forrásfájl elején beilleszteni, mivel ezek gyakran használt definíciók. Ha a forrást több fájlra bontjuk, az assembler csak a fejlécfájl első példányát illeszti be, hogy elkerülje a konstansok többszöri definiálását.

Az *arrayprt* program a C.16. ábrán látható. Itt kihagytuk a megjegyzéseket, mivel az utasítások mostanra már jól ismertek. Ezért használhatjuk a kétoszlopos formátumot is. A 4. sor az üres verem címét teszi a bázismutató regiszterbe, ezzel lehetővé teszi, hogy a 10. sorban a veremtakarítást egyszerűen a bázismutatónak

```
#include "../syscalnr.h"                ! 1

.SECT .TEXT                             ! 2
vecpstrt:                               ! 3
    MOV  BP,SP                          ! 4
    PUSH vec1                           ! 5
    MOV  CX,frmatstr-vec1               ! 6
    SHR  CX                             ! 7
    PUSH CX                             ! 8
    CALL vecprint                       ! 9
    MOV  SP,BP                          !10
    PUSH 0                              !11
    PUSH _EXIT                          !12
    SYS                                 !13

.SECT .DATA                             !14
vec1: .WORD 3,4,7,11,3                  !15
frmatstr: .ASCIZ "%s"                   !16

frmatkop:                               !17
.ASCIZ "The array contains „            !18
frmatint: .ASCIZ "%d"                   !19

.SECT .TEXT                             !20
vecprint:                               !21
    PUSH BP                             !22
    MOV  BP,SP                          !23
    MOV  CX,4(BP)                       !24
    MOV  BX,6(BP)                       !25
    MOV  SI,0                           !26
    PUSH frmatkop                       !27
    PUSH frmatstr                       !28
    PUSH _PRINTF                        !29
    SYS                                 !30
    MOV  -4(BP),frmatint                !31
1:  MOV  DI,(BX)(SI)                    !32
    MOV  -2(BP),DI                      !33
    SYS                                 !34
    INC  SI                             !35
    LOOP 1b                             !36
    PUSH '\n'                           !37
    PUSH _PUTCHAR                       !38
    SYS                                 !39
    MOV  SP,BP                          !40
    RET                                 !41
```

C.16. ábra. *Az arrayprt program hibakeresés előtt*

a veremmutatóba töltésével végezzük el, ahogy azt az előző példánál említettük. Szintén láttuk már az előző példa 5–9. sorában az argumentumok kiszámítását és veremre helyezését a hívás előtt. A 22–25. sor a szubrutin regisztereit tölti fel.

A 27–30. sor azt mutatja, hogyan lehet egy stringet kinyomtatni, a 31–34. sor pedig a printf rendszerhívást mutatja egy egész értékre. Vegyük észre, hogy a string címe kerül a verembe a 27. sorban, a 33. sorban azonban magát az egész értéket tesszük a verembe. Mindkét esetben a PRINTF-nek a formátum-string az első argumentuma. A 37–39. sorok között egyetlen karakter kinyomtatása látható a putchar rendszerhívással.

Most próbáljuk meg lefordítani és futtatni a programot. Miután az:

```
as88 arrayprt.s
```

parancsot begépeljük, operandus hibát kapunk az *arrayprt.$* fájl 28. sorában. Ezt a fájlt az assembler generálja a beillesztett fájlok és a forrásfájl összekombinálásával, hogy egyetlen összetett fájl álljon elő, ami az assembler tényleges bemenetét képezi. Ahhoz, hogy lássuk, a valóságban hol is van a 28. sor, meg kell vizsgálnunk az *arrayprt.$* fájl 28. sorát. Nem nézhetjük az *arrayprt.s* fájlban, mert a két fájl különbözik, mivel a fejlécfájl sorról sorra kerül beillesztésre az *arrayprt.$* fájlba. Az *arrayprt.$* fájl 28. sora az *arrayprt.s* 7-es számú sorának felel meg, mivel a beillesztett fejlécfájl, a *syscalnr.h* 21 sorból áll.

Az arrayprt.$ fájl 28. sorát könnyű megtalálni UNIX-rendszereken az alábbi paranccsal:

```
head -28 arrayprt.$
```

amely a kombinált fájl első 28 sorát jeleníti meg. A lista legalsó sora tartalmazza a hibát. Ily módon (vagy egy szerkesztőt használva abban a 28. sort kiválasztva) láthatjuk, hogy az SHR utasítást tartalmazó 7. sorban van a hiba, Ezt a C.4. ábrán található utasítástáblázattal összevetve már látható is a probléma: a léptetés számát kifelejtettük. A kijavított 7. sor így néz ki:

```
SHR CX,1
```

Nagyon fontos megjegyeznünk, hogy a hibát az eredeti *arrayprt.s* forrásfájlban kell kijavítani, *nem* pedig az *arrayprt.$* kombinált forrásfájlban, hiszen az utóbbit az assembler minden hívásakor újragenerálja.

A forráskód következő fordítási kísérletének sikerülnie kell. Ezután elindíthatjuk a nyomkövetőt a:

```
t88 arrayprt
```

paranccsal. A nyomkövetési folyamat alatt láthatjuk, hogy a kimenet nincs összhangban az adatszegmensben található vektorral. A vektor a 3, 4, 7, 11 és 3 elemeket tartalmazza, a megjelenített adatok viszont a 3, 1024 stb. értékeket mutatják. Nyilvánvaló, hogy valami nincs rendben.

A hiba megkereséséhez újra kell futtatnunk a nyomkövetőt, ezúttal lépésről lépésre, megvizsgálva a gép állapotát a helytelen érték kinyomtatását megelőzően. A kinyomtatandó érték a 32. és 33. sorokban tárolódik a memóriában. Mivel helytelen érték kerül kinyomtatásra, érdemes a hiba okát errefelé keresni. A ciklus második menetében azt látjuk, hogy SI páratlan szám, holott ennek nyilvánvalóan párosnak kellene lennie, hiszen szavakat indexel, nem pedig bájtokat. A probléma a 35. sorban van, amely SI-t 1-gyel növeli, holott 2-vel kellene. A hiba javításához ezt a sort az alábbira kell cserélni:

```
ADD SI,2
```


A javítás után a számok kinyomtatott listája helyes.

Vár ránk azonban még egy hiba. Amikor a *vecprint* befejeződik és visszatér, a nyomkövető panaszkodik a veremmutatóra. Ilyenkor az a természetes, hogy megnézzük, hogy a *vecprint* hívásakor a veremre kerülő érték megegyezik-e a verem tetején levő értékkel akkor, amikor a 41. sorban a RET-et végrehajtjuk. Nem egyezik meg. A megoldás a 40. sor helyettesítése az alábbi kettővel:

```
ADD SP,10
POP BP
```

Az első utasítás eltünteti a *vecprint* alatt a veremre került 5 szót, így felfedve a BP értékét, amelyet a 22. sorban mentettünk el. Ezt a veremből a BP-be töltve már helyreállítjuk a BP hívás előtti értékét, és a helyes visszatérési cím most már a verem tetején van. A program így már hiba nélkül áll le. A hibakeresés assembly kódban sokkal inkább művészet, mint tudomány, de nyomkövető használatával sokkal könnyebb, mint anélkül.

C.8.5. String-kezelés és string-utasítások

Függelékünk legfőbb célja, hogy megmutassa, hogyan kezeljük az ismétlődő string-utasításokat. A C.17. ábrán két egyszerű stringkezelő program látható, a *strngcpy.s* és a *reverspr.s*, mindkettő megtalálható az *examples* könyvtárban. A C.17. (a) ábrán látható a string másolását végző szubrutin. Ez a *stringpr* szubrutint hívja, amely egy külön *stringpr.s* fájlban található. Ennek a listáját nem tartalmazza ez a függelék. A külön forrásfájlokban tárolt szubrutinokat használó programok fordításához csak fel kell sorolni az összes forrásfájlt az *as88* parancsban – elsőként a főprogramét –, amelyből a futtatható és a segédfájlok neve képződik. Például a C.17. (a) ábra programjához a következőt kell írnunk:

```
as88 strngcpy.s stringpr.s
```

A C.17. (b) ábra programja fordított sorrendben nyomtatja ki egy string karaktereit. Nézzük szépen sorban.

Hogy megmutassuk, hogy a sorszámok valóban csak megjegyzések, a C.17. (a) ábrán a sorokat az első címkétől kezdve számoztuk csak, és kihagytuk a számozásból az előtte lévőket. A főprogram a 2–8. sorban először meghívja az *strngcpy* rutint két argumentummal, hogy a forrás stringet a célba másolja, ahol a forrás string a *mesg2*, a cél string pedig a *mesg1*.

Most nézzük a 9. soron kezdődő *strngcpy* rutint. Ez azt várja, hogy a célpuffer és a forrás-string címét a hívás előtt a veremre helyezzük. A 10–13. sorban a használt regisztereket elmentjük a verembe, hogy később a 27–30. sorban helyreállíthassuk azokat. A 14. sorban SP-t BP-be másoljuk a szokásos módon. Így BP használható az argumentumok betöltésére. A 26. sorban, a vermet ismét BP-nek SP-be másolásával takarítjuk.

```
.SECT .TEXT
stcstart:                              ! 1
    PUSH  mesg1                        ! 2
    PUSH  mesg2                        ! 3
    CALL  strngcpy                     ! 4
    ADD   SP,4                         ! 5
    PUSH  0                            ! 6
    PUSH  1                            ! 7
    SYS                                ! 8
strngcpy:                              ! 9
    PUSH  CX                           ! 10
    PUSH  SI                           ! 11
    PUSH  DI                           ! 12
    PUSH  BP                           ! 13
    MOV   BP,SP                        ! 14
    MOV   AX,0                         ! 15
    MOV   DI,10(BP)                    ! 16
    MOV   CX,-1                        ! 17
    REPNZ SCASB                        ! 18
    NEG   CX                           ! 19
    DEC   CX                           ! 20
    MOV   SI,10(BP)                    ! 21
    MOV   DI,12(BP)                    ! 22
    PUSH  DI                           ! 23
    REP MOVSB                          ! 24
    CALL  stringpr                     ! 25
    MOV   SP,BP                        ! 26
    POP   BP                           ! 27
    POP   DI                           ! 28
    POP   SI                           ! 29
    POP   CX                           ! 30
    RET                                ! 31
.SECT .DATA                            ! 32
mesg1: .ASCIZ "Have a look\n"          ! 33
mesg2: .ASCIZ "qrst\n"                 ! 34
.SECT .BSS
```

(a)

```
#include "../syscalnr.h"              ! 1
start: MOV DI,str                     ! 2
    PUSH  AX                          ! 3
    MOV   BP,SP                       ! 4
    PUSH  _PUTCHAR                    ! 5
    MOVB  AL,'\n'                     ! 6
    MOV   CX,-1                       ! 7
    REPNZ SCASB                       ! 8
    NEG   CX                          ! 9
    STD                               ! 10
    DEC   CX                          ! 11
    SUB   DI,2                        ! 12
    MOV   SI,DI                       ! 13
1:  LODSB                             ! 14
    MOV   (BP),AX                     ! 15
    SYS                               ! 16
    LOOP  1b                          ! 17
    MOVB  (BP),'\n'                   ! 18
    SYS                               ! 19
    PUSH  0                           ! 20
    PUSH  _EXIT                       ! 21
    SYS                               ! 22
.SECT .DATA                           ! 23
str: .ASCIZ "reverse\n"               ! 24
```

(b)

C.17. ábra. *(a) Egy string másolása (strngcpy.s). (b) Egy string kiírása visszafelé (reverspr.s)*

A szubrutin lelke a 24. sorban lévő REP MOVSB utasítás. A MOVSB utasítás az SI által mutatott bájtot másolja a DI által mutatott memóriacímre. SI és DI is 1-gyel nő. A REP egy ciklust képez, amelyben ez az utasítás ismétlődik, CX értékét minden bájtra 1-gyel csökkentve. A ciklus akkor ér véget, amikor CX 0 lesz.

Mielőtt azonban futtathatnánk a REP MOVSB ciklust, a regisztereket elő kell készíteni; ezt a 15–22. sor teszi meg. Az SI forrásindex a 21. sorban verembe helyezett argumentumot kapja, a DI célindexet pedig a 22. sorban készítjük elő. CX értékének meghatározása kicsit bonyolultabb. Megjegyezzük, hogy a string végét egy nulla bájt jelzi. A MOVSB utasítás nem állítja a zéró flaget, a SCASB (SCAn String Byte, bájt-string bejárása) utasítás viszont igen. Ez összehasonlítja a DI által mutatott értéket AL értékével, majd menet közben növeli DI-t. Ezen túlmenően, ugyanúgy ismételhető, mint a MOVSB. Tehát a 15. sorban AX, és így AL is ki van ürítve, a 16. sorban DI-be betöltjük a mutatót a veremből, CX pedig a –1 kezdőértéket kapja a 17. sorban.

A 18. sorban van a REPNZ SCASB, amely ciklusban végzi az összehasonlítást, és egyenlőség esetén beállítja a zéró flaget. A ciklus minden lépésében CX csökken, a ciklus pedig akkor áll le, amikor a zéró flag beállítódik, mert a REPNZ a zéró flaget és CX-et is vizsgálja. A MOVSB ciklus számára a lépések számát ezután 19. és 20. sorban CX aktuális értéke és a kezdő –1 közötti különbségből számítjuk.

Egy kicsit nehézkes, hogy két ismétlődő utasításra van szükség, de ez az ára annak a tervezési döntésnek, hogy a mozgató utasítások ne befolyásolják a feltételkódokat. A ciklusok közben növelni kell az indexregisztereket, ehhez pedig az szükséges, hogy az irány-flag törölt állapotú legyen.

A 23. és 25. sorok nyomtatják ki az átmásolt stringet a *stringpr* szubrutin hívásával, mely az *examples* könyvtárban található. Ez nagyon egyszerű, úgyhogy itt nem is tárgyaljuk.

A C.17. (b) ábrán látható fordított sorrendben nyomtató program első sora a szokásos rendszerhívási számokat illeszti be. A 3. sorban egy helyfoglaló (dummy) értéket teszünk a verem tetejére, később a 15. utasítás ide fogja tenni a kiírandó karaktert. A 4. sorban a BP bázismutatót a verem aktuális tetejére irányítjuk. A program egyesével nyomtatja majd az ASCII karaktereket, tehát a _PUTCHAR numerikus értékét tesszük a veremre. Figyeljük meg, hogy a SYS hívás végrehajtásakor BP a kinyomtatandó karakterre mutat.

A 2., 6., és 7. sorok a DI, AL és CX regisztereket készítik elő az ismétlődő SCASB utasítás számára. A számláló regiszter és a célindex a string-másoló rutinhoz hasonlóan kapnak értéket, AL értéke viszont az újsor karakter, nem pedig a 0. Így a SCASB utasítás az *str* string karaktereinek értékét a \n értékkel hasonlítja össze a 0 helyett, a zéró flaget pedig akkor állítja be, ha megtalálta azt.

A REP SCASB növeli a DI regisztert, tehát a találat után a célindex az újsor karaktert követő nulla karakterre mutat. A 12. sorban DI-t kettővel csökkentjük, hogy az a string utolsó betűjére mutasson.

Ha a stringet fordított sorrendben bejárjuk, és karakterenként kinyomtatjuk, akkor pont elérjük célunkat, úgyhogy a 10. sorban beállítjuk az irány-flaget, hogy a string utasításokban az indexregiszterek csökkentése történjen. A 14. sorban a LODSB egy karaktert másol AL-be, majd a 15. sorban ez a karakter a _PUTCHAR mellé kerül a veremben, végül pedig a SYS utasítás kinyomtatja.

A 18. és 19. sorban található utasítások még kinyomtatnak egy újabb soremelést, majd a program a szokásos módon az _EXIT hívással ér véget.

A programnak a jelen verziója hibás. A hibát megtalálhatjuk, ha a programot lépésenként nyomon követjük.

A */str* parancs az *str* stringet a nyomkövető adatmezőjébe teszi. Mivel az adat címének numerikus értéke is adott, kideríthetjük, hogyan futnak végig az indexregiszterek az adaton a string pozíciójához képest.

A hiba azonban csak a return sokszori megnyomása után jelentkezik. A nyomkövető parancsainak használatával gyorsabban rátalálunk a problémára. Indítsuk

el a nyomkövetőt, és adjuk ki a *13* parancsot, hogy a ciklus belsejébe jussunk. Most a *b* paranccsal elhelyezhetünk egy töréspontot a 15. soron. Két újabb soremelés után láthatjuk, hogy az utolsó betű, az e megjelenik a kimeneti mezőben. Az *r* parancs a nyomkövetőt addig futtatja, amíg törésponthoz vagy a program végéhez nem ér. Ily módon az *r* parancs ismételt kiadásával átfuthatunk a betűkön, amíg közel nem jutunk a problémához. Ettől a ponttól kezdve már futtathatjuk lépésenként a nyomkövetőt, amíg meg nem látjuk, mi történik a kritikus utasításokban.

Konkrét sorra is helyezhetünk el töréspontot, de észben kell tartani, hogy a *../syscalnr.h* fájlt is beillesztettük, amitől a sorszámok 21-gyel eltolódnak. Következésképpen a 15. sorra a *36b* paranccsal helyezhetünk el töréspontot. Ez nem szép megoldás, ezért sokkal jobb a 2. soron az utasítás előtt megadott *start* globális címkét használni, és a */start+13b* parancsot kiadni, amely ugyanoda helyezi el a töréspontot anélkül, hogy a beillesztett fájl méretét ismerni kellene.

C.8.6. Ugrótáblák

Számos programozási nyelvben létezik *case* vagy *switch* utasítás, amellyel valamilyen változó numerikus értéke alapján választhatunk több ugrási lehetőség közül. Néha assembly programokban is szükség van ilyen többirányú elágazásra. Gondoljunk például arra, amikor rendszerhívás szubrutinoknak egy halmazát összekapcsoljuk egyetlen SYS csapda rutinba. A C.18. ábrán mutatott *jumptbl.s* program bemutatja, hogyan lehet ilyen több elágazásos kapcsolókat programozni a 8088-as assemblerben.

A program egy string kinyomtatásával kezdődik, amelynek *strt* a címkéje, ezzel felkéri a felhasználót, hogy gépeljen be egy oktális számjegyet (4–7. sor). Ezután a szabványos bemenetről beolvas egy karaktert (8 és 9. sor). Ha AX értéke 5-nél kisebb, a program azt fájlvégjelzőnek értelmezi, és a 8-as címkére, a 22-es sorra ugrik, ahol 0 állapotkóddal kilép.

Ha nem fájlvéget találtunk, akkor az AL-ben lévő beolvasott karaktert megvizsgáljuk. Minden a 0 számjegynél kisebb karaktert szóköznek tekintünk, amelyet figyelmen kívül is hagyunk a 13. sorra történő ugrással, ahol megtörténik a következő karakter beolvasása. Minden a 9-es számjegynél nagyobb karaktert hibás bemenetnek tekintünk. A 16. sorban ezt a 9-es számjegyet követő karakterré, ASCII kettősponttá alakítjuk át.

Így a 17. sorban AX-ben egy a 0 számjegy és a kettőspont közé eső értéket találunk. Ezt másoljuk BX-be. A 18. sorban az AND utasítás kimaszkolja az alsó négy bit kivételével az összes bitet, ami egy 0 és 10 közötti számot eredményez (abból a tényből fakadóan, hogy az ASCII 0 a 0x30). Mivel szavaknak és nem bájtoknak a táblázatában fogunk indexelni, BX értékét megszorozzuk kettővel a balra léptető utasítással a 19. sorban.

A 20. sorban következik egy CALL utasítás. Az effektív címet úgy kapjuk, hogy BX értékét a *tbl* címke numerikus értékéhez adjuk, majd ennek az összetett címnek a tartalmát töltjük a PC utasításszámlálóba.

Ez a program tíz szubrutinból választ egyet annak megfelelően, hogy milyen karaktert olvastunk a szabványos bemenetről. Mindegyik szubrutin valamilyen üze-


```
#include "../syscalnr.h"                    ! 1
.SECT .TEXT                                 ! 2
jumpstrt:                                   ! 3
        PUSH  strt                          ! 4
        MOV   BP,SP                         ! 5
        PUSH  _PRINTF                       ! 6
        SYS                                 ! 7
        PUSH  _GETCHAR                      ! 8
1:      SYS                                 ! 9
        CMP   AX,5                          ! 10
        JL    Bf                            ! 11
        CMPB  AL,'0'                        ! 12
        JL    1b                            ! 13
        CMPB  AL,'9'                        ! 14
        JLE   2f                            ! 15
        MOVB  AL,'9'+1                      ! 16
2:      MOV   BX,AX                         ! 17
        AND   BX,0Xf                        ! 18
        SAL   BX,1                          ! 19
        CALL  tbl(BX)                       ! 20
        JMP   1b                            ! 21
8:      PUSH  0                             ! 22
        PUSH  _EXIT                         ! 23
        SYS                                 ! 24

rout0:  MOV   AX,mes0                       ! 25
        JMP   9f                            ! 26
rout1:  MOV   AX,mes1                       ! 27
        JMP   9f                            ! 28
rout2:  MOV   AX,mes2                       ! 29
        JMP   9f                            ! 30
rout3:  MOV   AX,mes3                       ! 31
        JMP   9f                            ! 32
rout4:  MOV   AX,mes4                       ! 33
        JMP   9f                            ! 34
rout5:  MOV   AX,mes5                       ! 35
        JMP   9f                            ! 36
rout6:  MOV   AX,mes6                       ! 37
        JMP   9f                            ! 38
rout7:  MOV   AX,mes7                       ! 39
        JMP   9f                            ! 40
rout8:  MOV   AX,mes8                       ! 41
        JMP   9f                            ! 42
erout:  MOV   AX,emes                       ! 43
9:      PUSH  AX                            ! 44
        PUSH  _PRINTF                       ! 45
        SYS                                 ! 46
        ADD   SP,4                          ! 47
        RET                                 ! 48

.SECT .DATA                                 ! 49
tbl: .WORD rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout8,erout  ! 50
mes0: .ASCIZ "This is a zero.\n"            ! 51
mes1: .ASCIZ "How about a one.\n"           ! 52
mes2: .ASCIZ "You asked for a two.\n"       ! 53
mes3: .ASCIZ "The digit was a three.\n"     ! 54
mes4: .ASCIZ "You typed a four.\n"          ! 55
mes5: .ASCIZ "You preferred a five.\n"      ! 56
mes6: .ASCIZ "A six was encountered.\n"     ! 57
mes7: .ASCIZ "This is number seven.\n"      ! 58
mes8: .ASCIZ "This digit is not accepted as an octal.\n"  ! 59
emes: .ASCIZ "This is not a digit. Try again.\n"  ! 60
strt: .ASCIZ "Type an octal digit with a return. Stop on end of file.\n"  ! 61
```

C.18. ábra. *Egy többirányú elágazást ugrótábla alkalmazásával bemutató program*

net címét teszi a verembe, és utána egy olyan _PRINTF rendszerszubrutin-hívásra ugrik, amely közös.

Ahhoz, hogy megértsük, mi is történik, tisztában kell lennünk azzal, hogy a JMP és CALL utasítások a PC-be valamilyen kódszegmensbeli címet töltenek. Az ilyen címek is csak bináris számok, és az assembly folyamat közben minden címet a nekik megfelelő bináris értékkel helyettesítünk. Ezekkel a bináris számokkal inicializálhatunk egy adatszegmensbeli tömböt, és pontosan ez történik az 50. sorban. Így a *tbl* címkénél kezdődő tömb a *rout0*, *rout1*, *rout2* stb. kezdőcímeit tartalmazza, címenként két bájton. A címek 2 bájton helyezkednek el, ez a magyarázat arra, hogy

miért van szükség az 1 bites eltolásra a 19. sorban. Az ilyen típusú táblákat gyakran **ugrótáblának (dispatch table)** nevezzük.

A rutinok működése az *erout* címkénél látható a 43–48. sorig. Ez a rész kezeli a tartományon kívüli számjegy esetét. Az üzenet címe először AX-be, majd a verembe kerül a 43. sorban. Azután a _PRINTF rendszerhívás sorszámát tesszük a veremre. Aztán következik a rendszerhívás, majd a verem takarítása, és végül a rutin visszatér. A többi kilenc rutin, a *rout0*, ..., *rout8*, mindegyike betölti a saját üzenetének címét AX-be, majd az *erout* második sorára ugrik, hogy kinyomtassa az üzenetet, és befejezze a szubrutint.

Az ugrótáblák jobb megismerésének érdekében jó, ha a programot több különböző bemenettel is végigkövetjük. Gyakorlásképpen a programot módosíthatjuk úgy, hogy minden karakter hatására valamilyen értelmes művelet történjen. Például az oktális számjegyeken kívül minden más karakter adjon egy hibaüzenetet.

C.8.7. Pufferelt és véletlen fájlelérés

A C.19. ábrán látható *InFilBuf.s* program a véletlen fájl B/K műveleteket szemlélteti. A fájlról feltételezzük, hogy néhány sorból áll, és a sorok hosszúsága különböző lehet. A program először beolvassa a fájlt, majd egy olyan táblázatot épít fel, amelynek az *n*. bejegyzése az *n*. sor kezdetének fájlbeli pozíciója. Ezután, ha keresünk egy sort, akkor annak pozícióját kikereshetjük a táblázatból, majd a sort az lseek és a read rendszerhívásokkal beolvashatjuk. A fájl neve a szabványos bemenet első soraként adott. Ez a program több viszonylag független kódrészletet tartalmaz, melyek egyéb célokra is módosíthatók.

Az első öt sor egyszerűen a rendszerhívások sorszámait és a puffer méretét definiálja, valamint a szokásos módon a verem tetejére állítja a bázismutatót. A 6–13. sor olvassa be a szabványos bemenetről a fájl nevét, majd ezt egy stringként a *linein* címkére tárolja. Ha a fájlnév nincs rendesen lezárva egy újsor karakterrel, akkor hibaüzenet keletkezik, és a folyamat nullától különböző állapottal lép ki. Ez a 38. és a 45. sor között történik. Megjegyezzük, hogy a fájlnév címe a 39., míg a hibaüzenet címe a 40. sorban kerül a veremre. Ha megnézzük magát a hibaüzenetet (a 113. sorban), akkor látjuk a *%s* string jelzést a _PRINTF formátumában. Ide kerül beillesztésre a *linein* string tartalma.

Ha a fájlnév másolása probléma nélkül megtörtént, a fájl megnyitása a 14–20. sorban történik. Ha a megnyitás sikertelen, akkor a visszatérési érték negatív, és a 38. sorban lévő 9-es címkére ugrunk, hogy a hibaüzenetet kiírjuk. Ha a rendszerhívás sikeres, akkor a visszatérési érték a fájlleíró, amelyet a *fildes* változóban tárolunk el. Erre a fájlleíróra szükség lesz a következő read és lseek hívásoknál.

Ezután a fájlt 512 bájtos blokkokban beolvassuk, minden blokkot a buf nevű pufferben tárolunk. A puffert kizárólag azért allokáljuk a szükséges 512 bájtnál két bájttal nagyobbra, hogy bemutassuk, hogyan lehet egy kifejezésben (a 123. sorban) keverni egy szimbolikus konstanst és egy egész számot. Ugyanígy, a 21. sorban SI-be betöltjük a *linh* tömb második elemének címét, ezáltal a tömb elején megmarad a 0 kezdeti értékű szó. A BX regiszterben a fájl első be nem olvasott karakterének fájlcíme lesz, ezért ezt a puffer első feltöltése előtt 0-ra inicializáljuk (22. sor).


```
#include "../syscalnr.h"      ! 1
bufsiz = 512                  ! 2
.SECT .TEXT                   ! 3
infbufst:                     ! 4
    MOV  BP,SP                ! 5
    MOV  DI,linein            ! 6
    PUSH _GETCHAR             ! 7
1:  SYS                       ! 8
    CMPB AL,'\n'              ! 9
    JL   9f                   !10
    JE   1f                   !11
    STOSB                     !12
    JMP  1b                   !13
1:  PUSH 0                    !14
    PUSH linein               !15
    PUSH _OPEN                !16
    SYS                       !17
    CMP  AX,0                 !18
    JL   9f                   !19
    MOV  (fildes),AX          !20
    MOV  SI,linh+2            !21
    MOV  BX,0                 !22
1:  CALL fillbuf              !23
    CMP  CX,0                 !24
    JLE  3f                   !25
2:  MOVB AL,'\n'              !26
    REPNE SCASB               !27
    JNE  1b                   !28
    INC  (count)              !29
    MOV  AX,BX                !30
    SUB  AX,CX                !31
    XCHG SI,DI                !32
    STOS                      !33
    XCHG SI,DI                !34
    CMP  CX,0                 !35
    JNE  2b                   !36
    JMP  1b                   !37
9:  MOV  SP,BP                !38
    PUSH linein               !39
    PUSH errmess              !40
    PUSH _PRINTF              !41
    SYS                       !42
    PUSH _EXIT                !43
    PUSH _EXIT                !44
    SYS                       !45
3:  CALL getnum               !46
    CMP  AX,0                 !47
    JLE  8f                   !48
    MOV  BX,(curlin)          !49
    CMP  BX,0                 !50
    JLE  7f                   !51
    CMP  BX,(count)           !52
    JG   7f                   !53
    SHL  BX,1                 !54
    MOV  AX,linh-2(BX)        !56
    MOV  CX,linh(BX)          !55
    PUSH 0                    !57
    PUSH 0                    !58
    PUSH AX                   !59
    PUSH (fildes)             !60
    PUSH _LSEEK               !61
    SYS                       !62
    SUB  CX,AX                !63
    PUSH CX                   !64
    PUSH buf                  !65
    PUSH (fildes)             !66
    PUSH _READ                !67
    SYS                       !68
    ADD  SP,4                 !69
    PUSH 1                    !70
    PUSH _WRITE               !71
    SYS                       !72
    ADD  SP,14                !73
    JMP  3b                   !74
8:  PUSH scanerr              !75
    PUSH _PRINTF              !76
    SYS                       !77
    ADD  SP,4                 !78
    JMP  3b                   !79
7:  PUSH 0                    !80
    PUSH _EXIT                !81
    SYS                       !82
fillbuf:                      !83
    PUSH bufsiz               !84
    PUSH buf                  !85
    PUSH (fildes)             !86
    PUSH _READ                !87
    SYS                       !88
    ADD  SP,8                 !89
    MOV  CX,AX                !90
    ADD  BX,CX                !91
    MOV  DI,buf               !92
    RET                       !93

getnum:                       !94
    MOV  DI, linein           !95
    PUSH _GETCHAR             !96
1:  SYS                       !97
    CMPB AL,'\n'              !98
    JL   9b                   !99
    JE   1f                   !100
    STOSB                     !101
    JMP  1b                   !102
1:  MOVB (DI),'\0'            !103
    PUSH curlin               !104
    PUSH numfmt               !105
    PUSH linein               !106
    PUSH _SSCANF              !107
    SYS                       !108
    ADD  SP,10                !109
    RET                       !110

.SECT .DATA                   !111
errmess:                      !112
.ASCIZ "Open %s failed\n"     !113
numfmt: .ASCIZ "%d"           !114
scanerr:                      !115
.ASCIZ "Type a number.\n"     !116
.ALIGN 2                      !117
.SECT .BSS                    !118
linein: .SPACE 80             !119
fildes: .SPACE 2              !120
linh: .SPACE 8192             !121
curlin: .SPACE 4              !122
buf: .SPACE bufsiz+2          !123
count: .SPACE 2               !124
```

C.19. ábra. *Egy program pufferelt olvasással és véletlen fájleléréssel*

A puffer feltöltését a *fillbuf* rutin intézi a 83–93. sorban. Miután a read számára az argumentumokat a veremre tettük, következik a rendszerhívás, amely a ténylegesen beolvasott karakterek számát AX-be teszi. Ezt a számot másoljuk CX-be, és ezután CX a még a pufferben maradt karakterek számát tárolja. Az első be nem olvasott karakter fájlpozícióját BX tárolja, tehát CX-et BX-hez kell adni a 91. sorban.

A 92. sorban a puffer aljának címét DI-be tesszük, ezzel előkészítve a következő újsor karakter megkeresését a pufferben.

Miután a *fillbuf* rutinból visszatérünk, a 24. sor ellenőrzi, hogy egyáltalán sikerült-e valamit beolvasni. Ha nem, a 25. sorban kiugrunk a pufferelt olvasási ciklusból a program második részére.

Készen állunk a puffer bejárására. A 26. sorban a \n szimbólumot AL-be töltjük, majd a 27. sorban ezt az értéket keressük a REP SCASB ciklussal a pufferbeli szimbólumok között. Két módon léphetünk ki a ciklusból: vagy amikor CX eléri a nullát, vagy pedig ha a vizsgált szimbólum az újsor karakter. Ha a zéró flag be van állítva, akkor a legutóbb vizsgált szimbólum egy \n volt, és az aktuális szimbólum fájlpozíciója (eggyel az újsor után) a *linh* tömbben tárolandó. A 29–31. sorban a számlálót növeljük, a fájlpozíciót BX-ből és CX-ből számítjuk ki (CX-ben van a még rendelkezésre álló karakterek száma). A 32–34. sor végzi a tényleges tárolást, de mivel a STOS feltételezi, hogy a céloperandus a DI, nem pedig az SI, így ezeket a regisztereket a STOS előtt és után fel kell cserélni. A 35–37. sor megvizsgálja, hogy van-e még adat a pufferben, és CX értékének megfelelően elugrik.

Amikor elérjük a fájl végét, elkészül a sorok kezdetének fájlpozíciójából álló teljes lista. Mivel a *linh* tömböt a 0 szóval kezdtük, tudjuk, hogy az első sor a 0 címen kezdődik, a következő sor a *linh* + 2 által megadott címen stb. Az *n*. sor mérete az *n* + 1. és az *n*. sorok kezdő pozícióiból számítható ki.

A program további részének célja a sor számának beolvasása, aztán az adott sornak a pufferbe töltése, majd annak kiírása egy write hívás segítségével. Minden szükséges információ megtalálható a *linh* tömbben, amelynek *n*. bejegyzése tartalmazza az *n*. sor kezdetét a fájlban. Ha a kért sor száma 0, vagy tartományon kívülre esik, a program a 7-es címkére ugrik, és kilép.

A programnak ez a része a *getnum* szubrutin hívásával kezdődik a 46. sorban. Ez a rutin beolvas egy sort a szabványos bemenetről, és eltárolja azt a *linein* pufferben (95–103. sor). Utána előkészítjük a SSCANF hívást. Figyelembe véve az argumentumok fordított sorrendjét, először a *curlin* címét vermeljük, amely majd egy egész számot fog tárolni, utána jön a *numfmt* egész formátum-string címe, majd végül a *linein* puffer címe, amely a számot tartalmazza decimális alakban. A SSCANF rendszerszubrutin, amennyiben lehetséges, a bináris értéket a *curlin* változóba teszi. Hiba esetén AX-ben 0-val tér vissza. Ezt a visszatérési értéket ellenőrizzük a 48. sorban, és hiba esetén a program a *8*-as címkénél hibaüzenetet generál.

Ha a *getnum* szubrutin érvényes egésszel tér vissza *curlin*-be, akkor először azt BX-be másoljuk. Utána összehasonlítjuk az értéket a tartománnyal a 49–53. sorban, és amennyiben a szám a tartományon kívülre esik, EXIT-et generálunk.

Ezután meg kell keresni a kiválasztott sor végét a fájlban, és a beolvasandó bájtok számát, ezért BX-et megszorozzuk 2-vel egy SHL balra léptető utasítással. A keresett sor fájlpozícióját az 55. sorban AX-be másoljuk. A következő sor fájlpozícióját CX-be tesszük, és később majd az aktuális sor bájtjai számának kiszámítására használjuk.

A fájlból való véletlen olvasáshoz szükség van egy lseek hívásra, amellyel a fájlpozíciót (offset) a következőnek olvasni kívánt bájtra állíthatjuk. Az lseek a fájl kezdetéhez viszonyítva történik, ezért ennek jelölésére első argumentumként

0 kerül a verembe az 57. sorban. A következő argumentum a fájleltolás. Definíció szerint ez az argumentum egy hosszú (azaz 32 bites) egész, tehát először egy 0 szót majd AX értékét tesszük a verembe az 58. és 59. sorokban, így alkotva egy 32 bites egészet. Ezután a fájlleíró és az LSEEK kódja másolódik a verembe, majd a 62. sorban megtörténik a hívás. Az LSEEK visszatérési értéke, a fájl aktuális pozíciója, a DX : AX regiszter-összetételben található. Ha a szám elfér egy gépi szóban (ami igaz 65 536 bájtnál rövidebb fájlok esetén), akkor AX tartalmazza a címet, tehát ezt CX-ből kivonva (63. sor) megkapjuk a sornak a pufferbe töltéséhez szükséges beolvasandó bájtok számát.

A program további része egyszerű. A sort beolvassuk a fájlból a 64–68. sorban, azután kiírjuk az *1*-es fájlleírón keresztül a szabványos kimenetre a 70–72. sorban. Megjegyezzük, hogy a 69. sorban történő részleges verem takarítás után, a számláló és a puffer értéke még mindig a veremben marad. Végül, a 73. sorban teljesen helyreállítjuk a veremmutatót, és készen állunk a következő lépésre, tehát visszaugrunk a *3*-as címkére, majd kezdjük elölről egy újabb *getnum* hívással.

Köszönetnyilvánítás

Az ebben a függelékben használt assembler az „Amsterdam Compiler Kit" része. A teljes eszközkészlet elérhető a *www.cs.vu.nl/ack* címen. Köszönjük az eredeti tervezésben részt vevő Johan Stevenson, Hans Schaminee és Hans de Vries munkáját. Különösen hálásak vagyunk a programcsomagot karbantartó Ceriel Jacobsnak, aki többször is segített a programot az oktatási követelményeinknek megfelelő formára alakítani, valamint Elth Ogstonnak a kézirat átolvasásáért, valamint a példák és gyakorlatok teszteléséért.

Szintén köszönjük Robbert van Renesse és Jan-Mark Wams munkáját, akik a PDP–11 és a Motorola 68000 számára terveztek nyomkövetőket. Számos ötletüket használtuk fel ennek a nyomkövetőnek a megtervezésében. Köszönjük továbbá a tanársegédek és rendszerüzemeltetők nagy csapatának, hogy segítették munkánkat a hosszú évek alatt megtartott számos assembly nyelvű programozás kurzus idején.

C.9. Feladatok

1. A MOV AX,702 utasítás végrehajtása után AH és AL értéke mennyi decimálisan?
2. A CS regiszter értéke 4. Mi a kódszegmens abszolút memóriacímeinek tartománya?
3. Melyik a 8088-as által elérhető legmagasabb memóriacím?
4. Tegyük fel, hogy CS = 40, DS = 8000 és IP = 20.
   a) Mi a következő utasítás abszolút címe?
   b) A MOV AX,(2) utasítás végrehajtásakor melyik memóriaszó töltődik AX-be?
5. Egy szubrutin három egész argumentummal, a szövegben leírt hívási szekvenciával kerül meghívásra, azaz a hívó fordított sorrendben helyezi a verembe

az argumentumokat, és utána végrehajtja a CALL utasítást. A hívott ezután elmenti a régi BP-t és beállítja az új BP-t, hogy az az elmentett régire mutasson. Ezeket a szabályokat figyelembe véve adja meg az első argumentumnak AX-be töltéséhez szükséges utasítást.

6. A C.1. ábrán operandusként használtuk a *de – hw* kifejezést. Ez az érték két címke különbsége. Elképzelhető-e olyan körülmény, amikor a *de + hw* érvényes operandusként használható? Indokolja válaszát.
7. Adja meg az assembly kódot az alábbi kifejezés kiszámítására:

```
x = a + b + 2
```

7. Egy C függvényt az alábbi módon hívunk:

```
foobar(x, y);
```

Adja meg az assembly kódot erre a hívásra.

8. Írjon assembly nyelvű programot, amely bemenetként elfogad egy egész számból, egy műveleti jelből és egy másik egész számból álló kifejezést, majd kiírja a kifejezés értékét. Engedje meg a +, –, * és a / műveleti jeleket.

Angol–magyar tárgymutató

8051 (see Intel 8051) – 8051 (lásd Intel 8051)
8088 (see Intel 8088) – 8088 (lásd Intel 8088)

A

absolute path – abszolút útvonal, 502
Accelerated Graphics Port (AGP) – gyorsított grafikus port (AGP), 125, 226
Access Control List (ACL) – elérést vezérlő lista (ACL), 508
access token – hozzáférést vezérlő jel, 508
accumulator – akkumulátor, 34, 343, 368
accumulator register – akkumulátorregiszter, 722
ACK (Amsterdam Compiler Kit) – ACK (Amsterdam Compiler Kit, Amsterdam fordítókészlet), 742
acknowledgement packet – nyugtázó csomag, 235
ACL (Access Control List) – ACL (elérést vezérlő lista), 508
active matrix display – aktív mátrixmegjelenítő, 124
actual parameter – aktuális paraméter, 536
adder – összeadó, 171–173
  carry select – átvitelkiválasztó, 173
  full – teljes, 171–172
  half – fél, 171–172
  ripple carry – átvitelt tovább terjesztő, 173
additive inverse – additív inverz, 398
address – cím, 84, 720
  memory – memória, 84
address decoding – címdekódolás, 242
address space – címtartomány, 38, 446
  physical – fizikai, 447
  virtual – virtuális, 447
addressing – címzés, 371, 380–394
  based-indexed – bázis-index, 384
  branch instruction – elágazó utasítások, 382–384
  direct – direkt, 381, 728
  immediate – közvetlen, 380
  implied – implicit, 729
  indexed – indexelt, 382
  Intel 8051 – Intel 8051, 392–393
  Intel 8088 – Intel 8088, 726–730
  Pentium 4 – Pentium 4, 390–392
  register – regiszter, 381
  register displacement – indexelt, 729
  register indirect – regiszter-indirekt, 381–382, 728
  register with index – bázis relatív, 729
  register with index and displacement – bázis relatív eltolással, 729
  stack – verem, 384–388
  UltraSPARC III – UltraSPARC III, 392
addressing mode – címzési mód, 380
  discussion – összefoglalás, 393
ADSL (Asymmetric DSL) – ADSL (aszimmetrikus DSL), 135
Advanced Microcontroller Bus Architecture (AMBA) – fejlett mikrokontroller sínarchitektúra (AMBA), 583
Advanced Programmable Interrupt Controller (APIC) – fejlett programozható megszakításvezérlő (APIC), 212
aggregate bandwidth – összesített sávszélesség, 662
AGP (Accelerated Graphics Port) – AGP (gyorsított grafikus port), 125, 226
AGP bus – AGP sín, 125, 226
Aiken, Howard – Aiken, Howard, 31
algorithm – algoritmus, 24
allocation/renaming unit – lefoglaló/átnevező egység, 336
Alpha – Alpha, 40
alpha channel mask – áttetszőségi csatorna maszk, 595
ALU (Arithmetic Logic Unit) – ALU (aritmetikai-logikai egység), 21, 67, 173–175

AMBA (Advanced Microcontroller Bus Architecture) – AMBA (fejlett mikrokontroller sínarchitektúra), 583
Amdahl's law – Amdahl törvénye, 664
American Standard Code for Information Interchange (ASCII) – az információcsere amerikai szabványos kódrendszere (ASCII), 143
amplitude modulation – amplitúdómoduláció, 133
Amsterdam Compiler Kit (ACK) – Amsterdam Compiler Kit (Amsterdam fordítókészlet), 742
analytical engine – analitikus gép, 30
APIC (Advanced Programmable Interrupt Controller) – APIC (fejlett programozható megszakításvezérlő), 212
Apple II – Apple II, 39
Apple Macintosh – Apple Macintosh, 39
application layer – alkalmazási réteg, 669
Application Programming Interface (API) – alkalmazásfejlesztést támogató interfész (API), 494
Application-Specific Integrated Circuit (ASIC) – alkalmazáshoz kifejlesztett egyedi integrált áramkör (ASIC), 587
arbitration, PCI bus – sínütemezés, PCI sín, 227
architecture – architektúra, 23, 71
argument – argumentum, 737
arithmetic circuit – aritmetikai áramkörök, 170–175
Arithmetic Logic Unit (ALU) – aritmetikai-logikai egység (ALU), 21, 67, 173–175
arithmetic, binary – aritmetika, bináris, 705
array processor – tömbprocesszor, 80
ASCII (American Standard Code for Information Interchange) – ASCII (az információcsere amerikai szabványos kódrendszere, 143
ASIC (Application-Specific Integrated Circuit) – ASIC (alkalmazáshoz kifejlesztett egyedi integrált áramkör), 587
assembler – assembler, 23, 525, 539, 718, 741
  directive – direktíva, 531
  Intel 8088 – Intel 8088, 741
  pass one – első menet, 540–544
  pass two – második menet, 544–545
  symbol table – szimbólumtábla, 545
  two-pass – kétmenetes, 539
  8088 – 8088, 741
assembly language – assembly nyelv, 524, 717
  characteristics – jellemzői, 525
  level – szint, 524–562
  pseudoinstuction – pszeudoutasítás, 531–534
  statement – utasítás, 528
  why use – miért használatos, 526–528
assembly language program – assembly nyelvű program, 525, 717, 741
assembly language programming – assembly nyelvű programozás, 717
  overview – áttekintés, 718
assembly process – assembler menetei, 539–547
  pass one – első menet, 540–544
  pass two – második menet, 544–545
  two-pass – kétmenetes, 539
asserted signal – beállított jel, 187
associative memory – asszociatív memória, 461, 545
Asymmetric DSL (ADSL) – aszimmetrikus DSL (ADSL), 135
asynchronous bus – aszinkron sín, 201–202
AT Attachment – AT kiegészítő, 102
ATA Packet Interface (ATAPI) – ATA-csomaginterfész (ATAPI), 102
ATA-3 – ATA-3, 102
Atanasoff, John – Atanasoff, John, 31, 33
ATAPI-4 – ATAPI-4, 102
attraction memory – vonzásmemória, 628
auto decrement – automatikus csökkentés, 730
auto increment – automatikus növelés, 730
auxiliary carry flag – kiegészítő átvitel-flag, 734

B

Babbage, Charles – Babbage, Charles, 30
baby feeding algorithm – gyeregetetési algoritmus, 452
backward compatibility – visszafelé kompatibilis, 353
Bardeen, John – Bardeen, John, 34
base – bázis, 153
base pointer – bázismutató, 724
base register – bázisregiszter, 722
based-indexed addressing – bázis-index címzés, 384
basic block – alapblokk, 329
Basic Input Output System (BIOS) – alapvető be-kimeneti rendszer (BIOS), 101
batch system – kötegelt rendszer, 27
baud – baud, 134
Bayer filter – Bayer-szűrő, 142
BCD (Binary Coded Decimai) – BCD (binárisan kódolt decimális), 84, 406, 727
Bechtolsheim, Andy – Bechtolsheim, Andy, 57
benchmark – tesztágy, 526
Berkeley UNIX – Berkeley UNIX, 486, 487, 489
best fit algorithm – legjobb illesztés algoritmus, 460
big endian machine – nagy endian gép, 86
binary arithmetic – bináris aritmetika, 705
Binary Coded Decimal (BCD) – Binárisan kódolt decimális (BCD), 84, 406, 727
binary number – bináris szám, 696
  addition of – összeadás, 705
  negative – negatív, 703
binary program – bináris program, 718

binary search – bináris keresés, 546
binding time – hozzárendelési idő, 553
BIOS (Basic Input Output System) – BIOS (alapvető be-kimeneti rendszer), 101
bipolar transistor – bipoláris tranzisztor, 155
bisection bandwidth – kettévágott sávszélesség, 633
bit – bit, 84, 699
BIT Block Transfer (bitblt) – bitblokk-átvitel (bitblt), 596
bit map – bittérkép, 370
bit slice – bitszelet, 174
bitblt (BIT Block Transfer) – bitblt (bitblokk-átvitel), 596
block cache – blokkgyorsító, 489
Block Started by Symbol (BSS) – szimbólummal kezdődő blokk (BSS), 742
block transfer, bus – blokkátvitel, sín, 205
blocking network – blokkoló hálózat, 619
BlueGene – BlueGene, 635–640
BlueGene/L – BlueGene/L, 635–640
Blu-Ray – Blu-Ray, 117
Boole, George – Boole, George, 155
Boolean algebra – Boole-algebra, 155–157
boundle – köteg, 435
branch – elágazás, 414–415
branch history shift register – elágazási előzmények blokkos regisztere, 323
branch prediction – elágazásjóvendolés, 319–324
  dynamic – dinamikus, 321–323
  static – statikus, 324
branch table – ugrótábla, 339
Branch Target Buffer (BTB) – elágazási célpuffer (BTB), 335
Brattain, Walter – Brattain, Walter, 34
breakpoint – töréspont, 749
broadband – szélessávú, 135
BSS (Block Started by Symbol) – BSS (szimbólummal kezdődő blokk), 742
bubblejet – festékbuborékos nyomtató, 129
bucket – rés, 547
buffer, circular – körkörös puffer, 478
buffered message passing – pufferelt üzenetátadás, 650
Burroughs B5000 – Burroughs B5000, 36
bus – sín, 35, 66, 117–120, 194, 207
  AGP – AGP, 226
  arbiter – ütemező, 119
  arbitration – sínütemezés, 119, 202–205
  arbitration, PCI – sínütemezés, PCI sín, 227
  asynchronous – aszinkron, 201
  block transfer – blokkátvitel, 205
  clocking – időzítés, 198
  cycle – ciklus, 198
  driver – vezérlő, 195
  EISA – EISA, 223
  grant – használati engedély, 202, 227
  handshaking – kézfogás, 202
  IBM PC – IBM PC, 222
  ISA – ISA, 222
  master – mester, 195, 201, 213, 218, 230
  multiplxed – multiplxelt, 197
  operation – műveletek, 205–207
  PCI – PCI, 223
  PCI Express – PCI Express, 232
  Pentium 4 – Pentium 4, 212
  protocol – protokoll, 194
  receiver – vevő, 195
  skew – aszimmetria, 197
  slave – szolga, 195
  synchronous – szinkron, 198
  timing – időzítés, 198–202
  tranceiver – adóvevő, 195
  USB – univerzális soros sín (USB), 236
  width – szélesség, 196–197
busy waiting – tevékeny várakozás, 403
Byron, Lord – Byron, Lord, 30
byte – bájt, 86, 727
byte instruction – bájtos utasítás, 727
byte ordering – bájtsorrend, 86–88
byte register – bájtos regiszter, 727

C

cable Internet – kábeles internet, 137
cache coherence protocol – gyorsítótár-koherencia protokoll, 612
  MESI – MESI, 614
cache coherency – gyorsítótár-koherencia, 611
Cache Coherent NUMA (CC-NUMA) – gyorsítótár-koherens NUMA (CC-NUMA), 619–628
cache consistency – gyorsítótár-konzisztencia, 611
cache hit – gyorsítótár-találat, 316
cache line – gyorsítósor, 94, 315, 612
cache memory – gyorsítótár, 54, 92, 313, 611
  direct-mapped – direkt leképezésű, 315
  invalidate strategy – érvénytelenítő stratégia, 613
  level 2 – 2. szintű, 314
  MESI protocol – MESI protokoll, 614
  multiple levels – többszintű, 313
  set-associative – halmazkezelésű, 317
  snooping – szimatoló, 612
  split – szétválasztott, osztott, 94, 313
  unified – egyesített, 94
  update strategy – frissítő stratégia, 613
  write-allocate – írásallokálás, 319, 613
  write-back – visszaírás, 319
  write-deferred – késleltetett írás, 319
  write-through – írásáteresztés, 318
cache miss – gyorsítótárhiány, 316
Cache Only Memory Access (COMA) – gyorsítótáras memóriaelérés (COMA), 605, 628, 687–688

call gate – híváskapu, 466
camera, digital – kamera, digitális, 141
carrier – vivőhullám, 133
carry select adder – átvitelkiválasztó összeadó, 173
catamount – catamount, 642
Cathode Ray Tube (CRT) – katódsugárcső, 121
CCD (Charge Coupled Device) – CCD (töltéscsatolt eszköz), 142
CC-NUMA (Cache Coherent NUMA) – CC-NUMA (gyorsítótár-kohcrens NUMA), 619–628
CDC 6600 – CDC 6600, 36, 79, 325
CDC Cyber – CDC Cyber, 71
CD-R (CD-recordable – CD-R (írható CD), 112
CD-recordable (CD-R) – írható CD (CD-R), 112
CD-rewritable (CD-RW) – újraírható CD (CD-RW), 114
CD-ROM (Compact Disc Read Only Memory) – CD-ROM, 108
  rewritable (CD-RW) – újraírható (CD-RW), 114
  sector – szektor, 110
  track – sáv, 113
  XA – XA, 113
CD-RW (CD-rewritable) – CD-RW (újraírható CD), 114
Celeron – Celeron, 55
cell, memory – rekesz (cella), memória, 84
Central Processing Unit (CPU) – központi feldolgozóegység (CPU), 34, 66
character code – karakterkód, 143
Charge Coupled Device (CCD) – töltéscsatolt eszköz (CCD), 142
checkerboarding – lyukacsosodás, 460
child process – gyerek processzus, 510
chip – lapka, 163
  CPU – CPU, 192
circuit – áramkör
  arithmetic – aritmetikai, 170–175
  combinational – kombinációs, 165
circuit equivalence – áramköri ekvivalencia, 159–162
CISC (Complex Instruction Set Computer) – CISC (összetett utasításkészletű számítógép), 73
clock – óra, 175
clock cycle time – ciklusidő, 175
clocked D latch – időzített D-tároló, 179
clocked SR latch – időzített SR-tároló, 178
clone – klón, 39
closure – zártság, 697
cluster – klaszter, 50, 606
  google – google, 645–649
  NTFS – NTFS, 508
  workstation– munkaállomás, 50–51, 606, 644
cluster computer – klaszterszámítógép, 644
Cluster Of Workstations (COW) – munkaállomások klasztere (COW), 50–51, 606, 644
CLUT (Color Look Up Table) – CLUT (színkereső tábla), 595
CMYK printer – CMYK-nyomtató, 131
coarse-grained multithreading – durva szemcsézettségű többszálúság, 573
COBOL program – COBOL program, 51
code generation – kódgenerálás, 742
code page – kódlap, 145
code point – kódpozíció, 146
code segment – kódszegmens, 720
codesign – együttes tervezés, 41
codeword – kódszó, 88
collective layer – kollektív szolgáltató réteg, 669
collector – kollektor, 153
color gamut – szín gamut, 131
Color Look Up Table (CLUT) – színkereső tábla (CLUT), 595
color palette – színpaletta, 125
COLOSSUS – COLOSSUS, 32
COMA (Cache Only Memory Access) – COMA (gyorsítótáras memóriaelérés), 605, 628, 687–688
combinational circuit – kombinációs áramkör, 165–170
  comparator – összehasonlító, 168
  decoder – dekódoló, 167
  multiplexer – multiplexer, 165–167
committed page – egyeztetett lap, 497
Commodity Off The Shelf (COTS) – készen kapható termék, 51
communicator – kommunikátor, 651
Compact Disc Read Only Memory (CD-ROM) – Compact Disc Read Only Memory (CD-ROM), 108
comparator – összehasonlító, 168
comparision and branch instructions – összehasonlító és elágazó utasítások, 398
comparison of architectures – architektúrák összehasonlítása, 346
compiler – fordítóprogram, 23, 525
Complex Instruction Set Computer (CISC) – összetett utasításkészletű számítógép (CISC), 73
computer architecture – számítógép-architektúra, 23
  milestones – mérföldkövek, 28–41
computer center – számítóközpont, 38
condition code – feltételkód, 359
condition code register – feltételkód-regiszter, 724
condition variable – feltételváltozó, 513
conditional execution – feltételes végrehajtás, 436
consistency model (see Memory semantics) – memóriaszemantika

constant pool – konstans mező, 268
Constant Pool Pointer (CPP) – konstans mező mutatója (CPP), 253, 262, 269, 280
consumer – fogyasztó, 478
context – környezet, 467
Control Data Corporation (CDC) – Control Data Corporation (CDC), 36
control signal – vezérlőjel, 257
control store – vezérlőtár, 72, 261
controller – vezérlő, 118
conversion between radices – konverzió számrendszerek között, 700
coprocessor – társprocesszor, 583–592
copy on write – íráskori másolás, 496
core – mag, 579
core dump – memóriamásolat, 25
CoreConnect – CoreConnect, 582
coroutine – korutin/társrutin, 414, 421, 422
counter register – számlálóregiszter, 722
COW (Cluster Of Workstations) – COW (munkaállomások klasztere), 50–51, 606, 644
CP/M – CP/M, 39
CPP (Constant Pool Pointer) – CPP (konstans mező mutatója), 253, 262, 269, 280
CPU chip – CPU lapka, 192
Cray, Seymour – Cray, Seymour, 36
Cray-1 – Cray-1, 36, 71
CRC (Cyclic Redundancy Check/Code) – CRC (ciklikus redundanciakód), 235, 239, 586
critical section – kritikus szakasz, 516
crossbar switch – crossbar (keresztrudas) kapcsoló, 617
crosspoint – találkozópont, 617
CRT (Cathode Ray Tube) – CRT (katódsugárcső), 121
Cryptoprocessor – kriptoprocesszor, 597
cube – kocka, 634
cycle stealing – ciklusiopás, 119, 405
Cyclic Redundancy Check (CRC) – ciklikus redundanciakód (CRC), 235, 239, 586
Cyclic Redundancy Code (CRC)– ciklikus redundanciakód (CRC), 235, 239, 586
cylinder – cilinder, 98

D

D latch – D-tároló, 179
daisy chaining – láncolás, 203
data cache – adatgyorsítótár, 339
data movement instruction – adatmozgató utasítás, 394
data path – adatút, 21, 67, 252–258
  Mic-1 – Mic–1, 262
  Mic-2 – Mic–2, 301
  Mic-3 – Mic–3, 304
  Mic-4 – Mic–4, 309
  timing – időzítés, 255
data path cycle – adatútciklus, 68
DATA section – DATA szegmens, 742
data segment – adatszegmens, 734
data type – adattípus, 368–371
  nonnumeric – nem numerikus, 369
  numeric – numerikus, 368
DDR (Double Data Rate memory) – DDR (kétszeres sebességű memória), 190
DEC (Digital Equipment Corporation) – DEC (Digital Equipment Corporation), 35, 38, 71
  Alpha – Alpha, 40
  PDP-1 – PDP–1, 35
  PDP-8 – PDP–8, 35
  VAX – VAX, 71, 73
decimal number – decimális szám, 743
decoder – dekódoló, 167
decoding unit – dekódoló egység, 310
degree – fokszám, 631
deinterlacing – sorduplázó technika, 596
delay slot – eltolási rés, 320
demand paging – kérésre lapozás, 452
DeMorgan's law – De Morgan-szabály, 160–161
demultiplexer – demultiplexer, 167
design principles, RISC – tervezési elvek, RISC, 74–75
destination index – célindex, 724
destination operand – céloperandus, 726
device driver – eszközmeghajtó, 492
device level – eszközszint, 20, 153
device register bus – eszközregisztersín, 583
diameter, network – átmérő, hálózat, 632
dibit modulation – kétbites moduláció, 134
difference engine – differenciagép, 30
digital camera – digitális kamera, 141
Digital Equipment Corporation (DEC) – Digital Equipment Corporation (DEC), 35, 38, 71
digital logic level – digitális logika szintje, 20, 152–250
  bus – sín, 194
  circuit – áramkör, 163
  CPU chip – CPU lapka, 192
  gate – kapu, 153
  I/O interface – B/K interfész, 240
  memory – memória, 176
Digital Subscriber Line (DSL) – digitális előfizetői vonal (DSL), 135
Digital Subscriber Line Access Multiplexer (DSLAM) – digitális előfizetői vonal hozzáférési multiplexer (DSLAM), 137
Digital Versatile Disc (DVD) – sokoldalú digitális lemez (DVD), 115
Digital Video Disc – digitális videolemez, 115
dimensionality – dimenziószám, 633
DIMM (Dual Inline Memory Module) – DIMM (két érintkezősoros memóriamodul), 95
DIP (Dual Inline Package) – DIP (kétlábsoros tokozás), 163
direct addressing – direkt címzés, 381

Direct Memory Access (DMA) – közvetlen memóriaelérés (DMA), 119, 404
direction flag – irány-flag, 730
direct-mapped cache – direkt leképezésű gyorsítótár, 315
directory – könyvtár, 475–476
directory based multiprocessor – katalógusalapú multiprocesszor, 621
disc – lemez, 96
  CD-ROM – CD-ROM, 108
  controller – vezérlő, 100
  DVD – DVD, 115
  floppy – hajlékony, 100
  IDE – IDE, 101
  magnetic – mágnes, 97–108
  optical – optikai, 108–117
  RAID – RAID, 104–108
  SCSI – SCSI, 103–104
  Winchester – Winchester, merev, 98
diskette – hajlékonylemez, 100
dispatch table – ugrótábla, 768
distributed memory system – osztott memóriájú rendszer, 600
Distributed Shared Memory (DSM) – elosztott közös memória (DSM), 602, 654
DLL (Dynamic Link Library) – DLL (dinamikus szerkesztő könyvtár), 557
DMA (Direct Memory Access) – DMA (közvetlen memóriaelérés), 119, 404
dot – pont, 742
Dots Per Inch (dpi) – pont per inch (dpi), 129
double – dupla pontosságú, 722
Double Data Rate memory (DDR) – kétszeres sebességű memória (DDR), 190
double indirect block – kétszeresen indirekt blokk, 504
double pointer – dupla szélességű mutató, 345
double precision – dupla pontosság, 368
double torus – kettős tórusz, 634
dpi (Dots Per Inch) – dpi (pont per inch), 129
DRAM (Dynamic RAM) – DRAM (dinamikus RAM), 189
DSL (Digital Subscriber Line) – DSL (digitális előfizetői vonal), 135
DSLAM (Digital Subscriber Line Access Multiplexer) – DSLAM (digitális előfizetői vonal hozzáférési multiplexer), 137
DSM (Distributed Shared Memory) – DSM (elosztott közös memória), 602, 654
Dual Inline Memory Module (DIMM) – két érintkezősoros memóriamodul (DIMM), 95
Dual Inline Package (DIP) – kétlábsoros tokozás (DIP), 163
DVD (Digital Versatile Disc) – DVD (sokoldalú digitális lemez), 115
dyadic instructions – diadikus műveletek, 395
dye based ink – festékalapú tinta, 132
dye sublimation printer – festékszublimációs nyomtató, 132
Dynamic Link Library (DLL) – dinamikus szerkesztő könyvtár (DLL), 557
Dynamic Linking – dinamikus szerkesztés, 555
Dynamic RAM (DRAM) – dinamikus RAM (DRAM), 189
Dynamic Random Access Memory (DRAM) – dinamikus véletlen elérésű memória, 31
dynamic relocation – dinamikus áthelyezés, 552

E

ECC (Error Correcting Code) – ECC (hibajavító kód), 88
Eckert, J. Presper – Eckert, J. Presper, 32
ECL (Emitter-Coupled Logic) – ECL (emitter csatolású logika), 155
edge triggered flip-flop – élvezérelt flip-flop, 179
EDO (Extended Data Output) memory – EDO (kiterjesztett adatkimenetű memória), 189
EDVAC (Electronic Discrete Variable Automatic Computer EDVAC) – EDVAC, 32
EEPROM (Electrically Eresable PROM) – EEPROM (elektromosan törölhető PROM), 190
effective address – effektív cím, 729
egress processing – kimenő csomagokon végzett feldolgozás, 589
EHCI (Enhanced Host Controller Interface) – EHCI (kibővített kiszolgálóvezérlő interfész), 240
EIDE (Extended IDE) – EIDE (kiterjesztett IDE), 101
EISA (Extended ISA) – EISA (bővített ISA), 120
Electrically Eresable PROM (EEPROM) – elektromosan törölhető (EEPROM), 190
Electronic Discrete Variable Automatic Computer (EDVAC) – EDVAC, 32
Electronic Numerical Integrator And Computer (ENIAC) – ENIAC, 32
embedded computer – beágyazott számítógép, 41
emitter – emitter, 153
Emitter-Coupled Logic (ECL) – emitter csatolású logika (ECL), 155
emulation – emuláció, 37
enable – érvényesítés/engedélyezés, 178
endian – endian, 86–88
  big – nagy, 86
  little – kis, 86, 727
Enhanced Host Controller Interface (EHCI) – kibővített kiszolgálóvezérlő interfész (EHCI), 240
ENIAC – ENIAC, 41
ENIGMA – ENIGMA, 32
enormalized number – normalizálatlan számok, 714

entry point – belépési pont, 551
environmental subsystem, Windows XP – környezeti alrendszer, Windows XP, 493
EPIC (Explicitly Parallel Instruction Computing) – EPIC (explicit utasításszintű párhuzamosság), 433–439
EPROM (Erasable PROM) – EPROM (törölhető PROM), 190
Erasable Programmable ROM (EPROM) – törölhető, programozható ROM (EPROM), 60, 190
Erasable PROM (EPROM) – törölhető PROM (EPROM), 60, 190
Error Correcting Code (ECC) – hibajavító kód (ECC), 88
escape code – kiterjesztő kód, 377
Estridge, Philip – Estridge, Philip, 39
Ethernet – Ethernet, 584
event – esemény, 516
evolution of multilevel computers – többszintű számítógépek fejlődése, 23
example programs – példaprogramok
  Intel 8088 – Intel 8088, 752
excess notation – többletes jelölés, 704
executable binary file – végrehajtható bináris fájl, 741
executable binary program – végrehajtható bináris program, 524, 548
executive, Windows XP – végrehajtó, Windows XP, 492
expanding opcode – műveletikód-kiterjesztés, 374–376
explicit linking – explicit szerkesztés, 558
Explicitly Parallel Instruction Computing (EPIC) – explicit utasításszintű párhuzamosság (EPIC), 433–439
exponent – exponens, 709
Extended Data Output (EDO) memory – kiterjesztett adatkimenetű memória (EDO), 189
Extended IDE (EIDE) – kiterjesztett IDE (EIDE), 101
Extended ISA (EISA) – bővített ISA (EISA), 120
external fragmentation – külső elaprózódás, 460
external reference – külső hivatkozás, 551
external symbol – külső szimbólum, 551
extra segment – extra szegmens, 734

F

fabric layer – grid szerkezeti szint, 668
false sharing – téves megosztás, 655
fanout – terhelhetőségi szám, 631
far call – távoli hívás, 737
far jump – távoli ugrás, 735
Fast Page Mode (FPM) memory – gyors lapkezelésű (FPM) memória, 189
FAT (File Allocation Table) – FAT (fájlallokációs tábla), 505
fat tree – kövér fa, 634
fetch-decode-execute cycle – betöltő-dekódoló-végrehajtó ciklus, 69
fetch-execute cycle – betöltő-végrehajtó ciklus, 252
fiber – fonál, 513, 514
Field Programmable Gate Array (FPGA) – helyszínen programozható kapumátrix (FPGA), 587
FIFO (First In First Out algorithm) – FIFO (először be először ki algoritmus), 454
fifth generation project, Japanese – ötödik generációs számítógépek, Japán, 40
file – fájl, 470–472
File Allocation Table (FAT) – fájlallokációs tábla (FAT), 505
file cache manager, Windows XP – fájlgyorsító-kezelő, Windows XP, 493
file descriptor – fájlleíró, 500, 501, 739
file index – fájlindex, 473
file system – fájlrendszer
  UNIX – UNIX, 488, 499–505
  Windows XP – Windows XP, 473, 492, 505–510
filter – szűrő, 502
fine-grained multithreading – finom szemcsézettségű többszálúság, 572
Finite State Machine (FSM) – véges állapotú gép (FSM), 298
  branch prediction – elágazásjövendolés, 323
  instruction fetch unit (IFU) – utasításbetöltő egység (IFU), 299
finite-precision number – véges pontosságú szám, 696–698
firewall – tűzfal, 585
first fit algorithm – első illesztés algoritmus, 460
First In First Out (FIFO) algorithm – először be először ki (FIFO) algoritmus, 454
first pass – első menet, 742
first-generation computers – első generációs számítógépek, 31
flag register – flag-regiszter, 724
flags register – jelzők regisztere, 358
flash memory – flash memória, 191
flat panel display – lapos megjelenítő, 122
flip-flop – billenőkör/flip-flop, 179
floating-point number – lebegőpontos szám, 708–716
floppy disk – hajlékonylemez, floppy, 100
flow control – folyamatvezérlés, 235
flow of control – vezérlési folyamat, 414–426
  branche – elágazás, 414–415
  coroutine – korutin (társrutin), 421–422
  interrupt – megszakítás, 423–426
  procedure – eljárás, 415–421

trap – csapda, 422–423
Flynn's taxonomy – Flynn-féle osztályozás, 604
fork – fork, 660
formal parameter – formális paraméter, 536
Forrester, Jay – Forrester, Jay, 34
FORTRAN – FORTRAN, 25
FORTRAN Monitor System (FMS) – FORTRAN felügyelő rendszer (FMS), 26
forward reference problem – előre hivatkozási probléma, 539
fourth-generation computers – negyedik generációs számítógépek, 38
FPM (Fast Page Mode) memory – FPM (gyors lapkezelésű) memória, 189
fraction – törtrész, 709
fragmentation – elaprózódás
  external – külső, 460
  internal – belső, 455
frame – keret, 110
frame pointer – keretmutató, 362
free list – szabad lista, 474
free page – szabad lap, 497
frequency modulation – frekvenciamoduláció, 133
frequency shift keying – frekvenciaeltolásos kódolás, 133
FSM (see Finite State Machine) – FSM (lásd véges állapotú gép)
full adder – teljes összeadó, 171–172
full duplex line – full-duplex vonal, 135
full handshake – teljes kézfogás, 202
full interconnect – teljes összekötés, 633
full resource sharing – teljes erőforrás-megosztás, 577
functional unit – funkcionális egység, 79

G

game computer – játékgép, 48
gamut – gamut, 131
gate – kapu, 20, 153
gate delay – kapukésleltetés, 164
GDI (Graphics Device Interface), Windows XP – GDI (grafikus eszköz interfész), Windows XP, 493
GDT (Global Descriptor Table) – GDT (globális leírótábla), 462
general regiszter – általános regiszter, 722
Global Descriptor Table (GDT) – globális leírótábla (GDT), 462
global label – globális címke, 743
Globe – Globe, 661
Goldstine, Herman – Goldstine, Herman, 33
google cluster – google-klaszter, 645–649
Graphical User Interface (GUI) – grafikus felhasználói felület (GUI), 39, 489
Graphics Device Interface (GDI), Windows XP – grafikus eszköz interfész (GDI), Windows XP, 493
Green Book – Green Book (Zöld Könyv), 111
grid – grid, háló, 667, 668
  rács, 634
GUI (Graphical User Interface) – GUI (grafikus felhasználói felület), 489

H

H register – H regiszter, 253, 262
half adder – félösszeadó, 171–172
half-duplex line – fél-duplex vonal, 135
halftone screen frequency – halftone képernyő-frekvencia, 131
halftoning – halftoning, 130
Hamming code – Hamming-kód, 89
Hamming distance – Hamming-távolság, 88
Hamming, Richard – Hamming, Richard, 44
handle – kezelő, 494
hardware – hardver, 23
  equivalence with software – ekvivalenciája a szoftverrel, 24
hardware abstraction layer – hardverabsztrakciós réteg, 491
hardware DSM – hardver DSM, 620
Harvard architecture – Harvard-architektúra, 94
hash coding – tördelő kódolás, 547
hazard – akadály, 307
HDTV (High Definition TeleVision) – HDTV (nagy felbontású televízió), 597
headend – fejállomás, 137
header – fejléc, 233
headless workstation – fej nélküli munkaállomás, 645
hexadecimal number – hexadecimális szám, 698, 743
High Definition TeleVision (HDTV) – nagy felbontású televízió (HDTV), 597
high level language – magas szintű nyelv, 23
  compared to assembly language – összehasonlítása az assembly nyelvvel, 526
High Sierra – High Sierra, 111
history – történeti áttekintés
  1642-1945 – 1642–1945, 30
  1945-1955 – 1945–1955, 31
  1955-1965 – 1955–1965, 34
  1965-1980 – 1965–1980, 36
  1980-present – 1980–napjainkig, 38
  computer systems – számítógéprendszerek
  Intel – Intel, 52
  Intel 8051 – Intel 8051, 59
  microprogramming – mikroprogramozás, 24, 27
  operating system – operációs rendszer, 25
  Sun Microsystems – Sun Microsystems, 57
hit ratio – találati arány, 93
Hoagland, Al – Hoagland, Al, 43
Hoff, Ted – Hoff, Ted, 52

hoisting, code – emelés, kód, 330
host library – gazdakönyvtár, 559
http (HyperText Transfer Protocol) – http, 585
hypercube – hiperkocka, 634
HyperText Transfer Protocol (http) – http, 585
hyperthreading – hyperthreading, 576

I

I/O (Input/Output) – B/K (Bemenet/Kimenet), 117–147
I/O instructions – B/K utasítások, 402–405
I/O manager, Windows XP – B/K kezelő, Windows XP, 492
IA-32 – IA–32, 359
IA-64 – IA–64, 431–439
  bundle – köteg, 435
  EPIC model – EPIC modell, 433
  instruction scheduling – utasításütemezés, 434–436
  predication – predikáció, 436–438
IAS machine – IAS gép, 33
IBM 1401 – IBM 1401, 35
IBM 360 – IBM 360, 37, 41, 583
IBM 701 – IBM 701, 34
IBM 704 – IBM 704, 34
IBM 7094 – IBM 7094, 35, 41
IBM 801 – IBM 801, 73
IBM Corporation – IBM Corporation, 34, 35, 37, 223
IBM PC – IBM PC, 39, 43
  bus – sín, 223
  origin – eredet, 39
IBM PS/2 – IBM PS/2, 223
IC (Integrated Circuit) – IC (integrált áramkör), 36, 163
IDE (Integrated Drive Electronics) – IDE (beépített eszközelektronika), 101
IEEE floating-point standard – IEEE lebegőpontos szabvány, 711
IFU (Instruction Fetch Unit) – IFU (utasításbetöltő egység), 296–300
IJVM (Integer Java Virtual Machine) – IJVM (egész aritmetikájú java virtuális gép), 251–260, 266–275
  constant pool – konstans mező, 268
  data path – adatút, 252
  instruction set – utasításkészlet, 270
  java code – java kód, 273
  local variable frame – lokális változók mezője, 268
  memory model – memóriamodell, 268
  memory operation – memóriaművelet, 257
  method area – metódus mező, 269
  Mic-1 implementation – Mic–1 megvalósítás, 280
  Mic-2 implementation – Mic–2 megvalósítás, 291
  Mic-3 implementation – Mic–3 megvalósítás, 303
  Mic-4 implementation – Mic–4 megvalósítás, 309
  operand stack – operandusverem, 268
  stack – verem, 266
  timing – időzítés, 255
ILC (Instuction Location Counter) – ILC (utasítás-helyszámláló), 540
ILLIAC – ILLIAC, 32
ILLIAC IV – ILLIAC IV, 80, 604
immediate addressing – közvetlen címzés, 380
immediate file – közvetlen fájl, 509
immediate operand – közvetlen operandus, 380
implicit linking – implicit szerkesztés, 558
implied addressing – implicit címzés, 729
import library – import könyvtár, 558
index register – indexregiszter, 723
indexed addressing – indexelt címzés, 383
indexed color – indexelt színelőállítás, 125, 595
indirect block – indirekt blokk, 504
Industry Standard Architecture (ISA) – ipari szabványos felépítés (ISA), 119, 223
infix notation – infix jelölés, 385
informative information, in standard – informatív információ, szabványban, 355
informative – informatív, 355
ingress processing – bejövő csomagokon végzett feldolgozás, 589
initiator, PCI bus – kezdeményező, PCI sín, 227
inkjet printer – tintasugaras nyomtató, 128
i-node – i-csomópont (node), 503
Input/Output (I/O) – Bemenet/Kimenet (B/K), 117–147
input/output device – bemeneti kimeneti eszköz
  CRT monitor – CRT Monitor, 121
  digital camera – digitális kamera, 141
  flat panel display – lapos megjelenítő, 122
  keyboard – billentyűzet, 121
  magnetic disk – mágneslemez, 97
  mice – egér, 126
  modem – modem, 133–141
  optical disk – optikai lemez, 108–117
  printer – nyomtató, 127–133
  telecommunications equipment – telekommunikációs berendezés, 133–141
  terminal – terminál, 121–125
input/output instructions – bemeneti/kimeneti utasítások, 402–405
instruction – utasítás, 359–368, 394–414
  8051 – 8051, 411
  8088 – 8088, 730
  branch – clágazó, 398, 414
  comparison – összehasonlító, 398–399
  data movement – adatmozgató, 394–395
  dyadic – diadikus, 395–396
  I/O – B/K, 402–405

loop – ciklus, 400–401
monadic – monadikus, 396–398
Pentium 4 – Pentium 4, 405–408
procedure call – eljáráshívó, 400
UltraSPARC – UltraSPARC, 408–410
instruction count, relation to RISC – utasításszám, RISC-hez viszonyítva, 72
instruction execution – utasítás-végrehajtás, 68
Instruction Fetch Unit (IFU) – utasításbetöltő egység (IFU), 296–300
instruction formats – utasításformátumok, 371–380
8051 – 8051, 379–380
design criteria – tervezési követelmények, 372
Pentium 4 – Pentium 4, 376–379
UltraSPARC – UltraSPARC, 378
instruction group – utasításcsoport, 435
instruction issue unit – utasításkiosztó egység, 339
instruction level parallelism – utasításszintű párhuzamosság, 75–80, 565–572
Instruction Location Counter (ILC) – utasításhelyszámláló (ILC), 540
Instruction Pointer (IP) – utasításmutató (IP), 720, 724
Instruction Register (IR) – utasításregiszter (IR), 67, 344
instruction scheduling – utasításütemezés, 434–436
Instruction Set Architecture (ISA) – utasításrendszer-architektúra (ISA), 22, 352–443, 717
8051 – 8051, 411
Pentium 4 – Pentium 4, 405–408
UltraSPARC – UltraSPARC, 408–410
Instruction Set Architecture (ISA) level – utasításrendszer-architektúra (ISA) szintje, 22, 352–443, 717
instruction set, Intel 8088 – utasításkészlet, Intel 8088, 730
instruction sets, comparison – utasításrendszer, 411
instruction types – utasítástípusok, 394–414
Integer Java Virtual Machine (IJVM) – egész aritmetikájú java virtuális gép (IJVM), 251–260, 266–275
Integer Unit (IU) – Integer Unit (IU), 58
Integrated Circuit (IC) – integrált áramkör (IC), 36, 163
Integrated Drive Electronics (IDE) – beépített eszközelektronika (IDE), 101
Intel 4004 – Intel 4004, 52
Intel 8008 – Intel 8008, 52
Intel 80286 – Intel 80286, 53
Intel 80386 – Intel 80386, 54
Intel 80486 – Intel 80486, 54
Intel 8051 – Intel 8051, 219–221, 365
addressing – címzés, 392–393
data types – adattípusok, 371
history – történeti áttekintés, 59
instruction formats – utasításformátumok, 379
instructions – utasítások, 411
microarchitecture – mikroarchitektúra, 343–345
overview of the ISA – ISA-szint áttekintése, 365–368
Intel 8080 – Intel 8080, 53
Intel 8086 – Intel 8086, 53
Intel 8088 – Intel 8088, 39, 40, 53, 359, 720
addressing – címzés, 726–730
assembler – assembler, 741
instruction set – utasításkészlet, 730
simulator – szimulátor, 747
tracer – nyomkövető, 747
Intel 8255A – Intel 8255A, 240
Intel Celeron – Intel Celeron, 55
Intel Corporation – Intel Corporation, 52
Intel Pentium (see also Pentium 4) – Intel Pentium (lásd még Pentium 4), 52, 54
Intel Xeon – Intel Xeon, 55
Inter Process Communication (IPC) – processzusok közti kommunikáció (IPC), 489
interconnection networks – összekötő hálózatok, 631–634
bisection bandwidth – kettévágott sávszélesség, 633
topology – topológia, 632
interlaced – váltottsoros, 596
interleaved memory – tagolt memória, 619
internal fragmentation – belső elaprózódás, 455
Internet over cable – kábeles internet, 137
Internet Protocol (IP) – internetprotokoll (IP), 586
Internet Service Provider (ISP) – internetszolgáltató (ISP), 585
interpretation – értelmezés, 18
interpreter – értelmező, 18, 70, 718
interrupt – megszakítás, 119, 423–426
imprecise – pontatlan, 327
precise – pontos, 327
transperent – átlátszó, 424
interrupt handler – megszakításkezelő, 119
Interrupt Service Routine (ISR) – megszakításkezelő (ISR), 423, 426
interrupt vector – megszakításvektor, 207
intersector gap – szektorrés, 98
invalidate strategy – érvénytelenítő stratégia, 613
inversion bubble – inverziós gömb, 154
inverter – inverter, nem kapu, 154
inverting buffer – invertáló puffer, 185
invisible computer – láthatatlan számítógép, 41

IP (Instruction Pointer) – IP (utasításmutató), 720
IP (Internet Protocol) – IP (internetprotokoll), 586
IPC (Inter Process Communication) – IPC (processzusok közti kommunikáció), 489
Iron Oxide Valley – Vas-oxid Völgy, 43
ISA (Industry Standard Architecture) – ISA (ipari szabványos felépítés), 119, 223
ISA (Instruction Set Architecture) – ISA (utasításrendszer-architektúra), 22, 352–443, 717
ISA bus – ISA sín, 222
ISA level – ISA-szint, 22, 352–443
addressing – címzés, 380–394
data types – adattípusok, 368–371
flow of control – vezérlési folyamat, 414–426
IA-64 – IA-64, 431–439
instruction formats – utasításformátumok, 371–380
instructions types – utasítástípusok, 394–414
ISP (Internet Service Provider) – ISP (internetszolgáltató), 585
ISR (Interrupt Service Routine) – ISR (megszakításkezelő), 423
Itanium 2 – Itanium 2, 431–439
IU (Integer Unit) – IU (Integer Unit), 58

J

Java – Java, 23
Java Virtual Machine (JVM) – java virtuális gép (JVM), 251
Jobs, Steve – Jobs, Steve, 39
JOHNIAC – JOHNIAC, 32
Joint Photographic Experts Group (JPEG) – JPEG, 142
Joint Test Action Group (JTAG) – JTAG, 596
Joy, Bill – Joy, Bill, 57
JPEG (Joint Photographic Experts Group) – JPEG, 142
JTAG (Joint Test Action Group) – JTAG, 596
jump (see branch) – ugrás (lásd elágazás)
JVM (Java Virtual Machine) – JVM (java virtuális gép), 251
JVM (see also IJVM) – JVM (lásd még IJVM)

K

kernel – kernel (rendszermag), 491
kernel mode – kernel mód, 355
key – kulcs, 473
keyboard – billentyűzet, 121
Khosla, Vinod – Khosla, Vinod, 57
Kildall, Gary – Kildall, Gary, 39

L

L1 BTB – L1 BTB, 335
label – címke, 718
LAN (Local Area Network) – LAN (helyi hálózat), 584
land – szint, 108
lane – sáv, 234
language – nyelv, 17
laser printer – lézernyomtató, 129
latch – tároló, 77, 177–179
latency – késleltetés, 77
rotational – forgási, 99
latency hiding – késleltetési idő elrejtés, 666
Latin-1 – Latin-1, 145
layer – réteg, 19
LBA (Logical Block Addressing) – LBA (logikai blokk címzés), 102
LCD (Liquid Crystal Display) – LCD (folyadékkristályos kijelző), 123
LDT (Local Descriptor Table) – LDT (lokális leírótábla), 462
Least Recently Used (LRU) algorithm – legrégebben használt (LRU) algoritmus, 318, 453
LED (Light Emitting Diode) – LED (fénykibocsátó dióda), 127
left value – balérték, 727
legacy – hagyományos, 209
Leibniz, Gottfried von – Leibniz, Gottfried von, 30
level – szint, 19
level 2 cache – 2. szintű gyorsítótár, 314
level-triggered latch – szintvezérelt tároló, 179
Light Emitting Diode (LED) – fénykibocsátó dióda (LED), 127
Linda – Linda, 656
linear address – lineáris cím, 463
Lines Per Inch (lpi) – vonal per inch (lpi), 131
link – link (kötés, kapocs), 502
link layer – kapcsolati réteg, 235
linkage editor – szerkesztő-editor, 547
linkage segment – csatoló szegmens, 555
linker – szerkesztő, 547–551, 741
linking – szerkesztés, 547–551, 741
binding time – hozzárendelési idő, 552
dynamic – dinamikus, 555
MULTICS – MULTICS, 555
object modul – tárgymodul, 551
task performed – feladatai, 548
UNIX – UNIX, 559
Windows – Windows, 557
linking loader – szerkesztő-betöltő, 547
Liquid Crystal Display (LCD) – folyadékkristályos kijelző (LCD), 123
literal – literál, 542
little endian – kis endián, 86, 727
load/store architecture – betöltő/tároló architektúra, 365
loader – betöltés, 547
Local Area Network (LAN) – helyi hálózat (LAN), 584

Local Descriptor Table (LDT) – lokális leírótábla (LDT), 462
local label – lokális címke, 743
local loop – lokális ciklus/hurok, 135
Local Variable (LV) pointer – lokális változó (LV) mutató, 253, 262, 266, 280
local variable frame – lokális változók mezője, 266, 268
locality principle – lokalitási elv, 93, 452
location counter – helyszámláló, 742
Logical Block Addressing (LBA) – logikai blokk címzés (LBA), 102
logical record – logikai rekord, 471
long – hosszú, 722
long word – hosszú szó, 727
loop control – ismétléses vezérlés, 400–401
loosely coupled – lazán kapcsolt, 564
Lovelace, Ada Augusta – Lovelace, Ada Augusta, 30
lpi (Lines Per Inch) – lpi (vonal per inch), 131
LRU (Least Recently Used) algorithm – LRU (legrégebben használt) algoritmus, 453
LV (Local Variable) pointer – LV (lokális változó) mutató, 253, 262, 266, 280

M

machine language – gépi nyelv, 17, 717
Macintosh, Apple – Macintosh, Apple, 39
macro – makró, 534
  call – hívás, 535
  definition – definíció, 534
  expansion – kifejtés, 535
  formal parameter – formális paraméter, 536
  implementation – megvalósítás, 537
  parameter – paraméter, 536
macroarchitecture – makroarchitektúra, 266
magnetic disk – mágneslemez, 97
mailslot – levélbedobó nyílás, 515
mainframe – nagyszámítógép, 51
MAL (Micro Assembly Language) – MAL (mikro assembly nyelv), 276
MANIAC – MANIAC, 32
mantissa – mantissza, 709
MAR (Memory Address Register) – MAR (memóriacím-regiszter), 257
Mark I – Mark I, 31
mask – maszk, 395
MASM – MASM, 528
Massively Parallel Processors (MPP) – erősen párhuzamos processzorok (MPP), 606, 635–644
Master File Table (MFT) – mesterfájltábla (MFT), 509
master, bus – mester, sín, 195
matrix printer – mátrixnyomtató, 127
Mauchley, John – Mauchley, John, 32
MBR (Memory Buffer Register) – MBR (memóriapuffer-regiszter), 253, 259, 262
McNealy, Scott – McNealy, Scott, 57
MCS-51 family – MCS–51 család, 59
MDR (Memory Data Register) – MDR (memóriaadat-regiszter), 253, 257, 259, 262
media processor – médiaprocesszor, 592–597
memory – memória, 83, 176
  8088 – 8088, 725–730
  address – memóriacím, 84
  associative – asszociatív, 461
  attraction – vonzásmemória, 628
  cache – gyorsítótár, 92, 313, 611–614
  chip – lapka, 186
  DDR – DDR, 190
  DRAM – DRAM, 189
  dynamic RAM – dinamikus RAM, 189
  EDO – EDO, 189
  EEPROM – EEPROM, 190
  EPROM – EPROM, 190
  flash – flash, 191
  FPM – FPM, 189
  hierarchy – hierarchia, 96
  map – térkép, 447
  model – modell, 356–358
  organization – szervezés, 183
  organization, Intel 8088 – szervezés, Intel 8088, 725
  packaging – tokozás, 95
  primary – központi, 83
  PROM – PROM, 190
  Random Access (RAM) – véletlen elérésű (RAM), 60, 188
  refresh – frissítés, 189
  ROM – ROM, 190
  SDRAM – SDRAM, 189
  secondary – háttérmemória, 96
  SRAM – SRAM, 189
  static RAM – statikus RAM, 189
  virtual – virtuális, 445–469, 493, 495–498
Memory Address Register (MAR) – memóriacím-regiszter (MAR), 257
Memory Buffer Register (MBR) – memóriapuffer-regiszter (MBR), 253, 259, 262
Memory Data Register (MDR) – memóriaadat-regiszter (MDR), 257
Memory Management Unit (MMU) – memóriakezelő egység (MMU), 449
memory mapped I/O – memóriára leképezett B/K, 242
memory semantics – memóriaszemantika, 357, 606–610
  processor consistency – processzorkonzisztencia, 608
  release consistency – elengedési konzisztencia, 610
  sequential consistency – soros konzisztencia, 607

  strict consistency – szigorú konzisztencia, 607
  weak consistency – gyenge konzisztencia, 609
mesh – háló, 634
MESI cache coherence protocol – MESI gyorsítótár-koherencia protokoll, 614
message passing – üzenetátadás, 650–652
message passing multicomputers – üzenetátadásos multiszámítógép, 629–667
message queue – üzenetsor, 511
Message-Passing Interface (MPI) – üzenetátadás interfész (MPI), 651
Metal Oxide Semiconductor (MOS) – fém-oxid félvezető (MOS), 155
method – metódus, 400
method area – metódus mező, 269
metric units – mértékegységek, 61
MFT (Master File Table) – MFT (mesterfájltábla), 509
Mic-1 – Mic–1, 261, 280
  data path – adatút, 262
  implementation – megvalósítás, 280
Mic-2 – Mic–2, 300
  data path – adatút, 301
  implementation – megvalósítás, 300
Mic-3 – Mic–3, 305
  data path – adatút, 304
  implementation – megvalósítás, 303–309
  pipeline – csővezeték, 303–309
Mic-4 – Mic–4, 309
  Data Path – Adatút, 309
  implementation – megvalósítás, 309
mickey – mickey, 127
Micro Assembly Language (MAL) – mikro assembly nyelv (MAL), 276
microarchitecture – mikroarchitektúra, 251
  8051 – 8051, 343
  Pentium 4 – Pentium 4, 332
  UltraSPARC – UltraSPARC, 338
microarchitecture level – mikroarchitektúra szintje, 21, 251
  branch prediction – elágazásjövendölés, 319–324
  cache memory – gyorsítótár, 313–319
  design – tervezés, 291–312
  example – példa, 332–347
  IJVM example – IJVM-példa, 251–291
microcontroller – mikrovezérlő, 46
microdrive – mikromeghajtó, 143
microinstruction – mikroutasítás, 72, 258
  notation – jelölés, 275
microinstruction control – mikroutasítás-vezérlés, 261
MicroInstruction Register (MIR) – mikroutasítás-regiszter (MIR), 261
micro-operation – mikroművelet, 310
microprogram – mikroprogram, 21, 24, 721
MicroProgram Counter (MPC) – mikroutasítás-számláló (MPC), 261
microprogramming – mikroprogramozás, 24
  history – történeti áttekintés, 24, 27
Microsoft Corporation – Microsoft Corporation, 40, 490, 473
microstep – mikrolépés, 306
milestones in computer architecture – mérföldkövek a számítógépek fejlődésében, 28–41
MIMD (Multiple Instruction stream Multiple Data stream) – MIMD (többszörös utasításáram többszörös adatáram), 604–605
MIPS (acronym, Millions of Instructions Per Second) – MIPS (betűszó, millió utasítás másodpercenként), 75
MIPS (chip) – MIPS (lapka), 73
MIR (MicroInstruction Register) – MIR (mikroutasítás-regiszter), 261
MISD (Multiple Instruction stream Single Data stream) – MISD (többszörös utasításáram egyszeres adatáram), 604–605
miss ratio – hibaarány, 93
MMU (Memory Management Unit) – MMU (memóriakezelő egység), 449
mnemonic – mnemonik, 718, 741
modem – modem, 120, 133
modulation – moduláció, 133
  amplitude – amplitúdó, 133
  frequency – frekvencia, 133
  phase – fázis, 133
monadic instructions – monadikus műveletek, 396
Moore, Gordon – Moore, Gordon, 42
Moore's law – Moore-szabály, 42
MOS (Metal Oxide Semiconductor) – MOS (fém-oxid félvezető), 155
motherboard – alaplap, 117
Motif – Motif, 489
Motion Picture Expert Group (MPEG) – MPEG, 581
Motorola 68000 – Motorola 68000, 72
mouse – egér, 126
MPC (MicroProgram Counter) – MPC (mikroutasítás-számláló), 261
MPEG (Motion Picture Expert Group) – MPEG, 581
MPI (Message-Passing Interface) – MPI (üzenetátadás interfész), 651
MPP (Massively Parallel Processor) – MPP (erősen párhuzamos processzor), 635, 606–644
MS-DOS – MS-DOS, 40
multicomputer – multiszámítógép, 83, 600, 681
  BlueGene/L – BlueGene/L, 618, 627, 635–640
  google cluster – google-klaszter, 645–649
  MPP – MPP, 606, 635

Red Storm – Red Storm, 640–643
performance – teljesítménye, 652–653
scheduling – ütemezése, 652–653
software – szoftver, 650
MULTICS (MULTiplexed Information and Computing Service (MULTICS) – MULTICS, 461, 466, 556
multilevel machine – többszintű gép, 20
MultiMedia eXtension (MMX) – multimédiás kiegészítések (MMX), 54
Multiple Instruction stream Multiple Data stream (MIMD) – többszörös utasításáram többszörös adatáram (MIMD), 604–605
Multiple Instruction stream Single Data stream (MISD) – többszörös utasításáram egyszeres adatáram (MISD), 604–605
multiplexed bus – multiplexelt sín, 197
MULTiplexed Information and Computing Service (MULTICS) – MULTICS, 461, 466, 556
multiplexer – multiplexer, 165
multiprocessor – multiprocesszor, 82, 340, 578, 583, 598–629
bus-based – sínalapú, 610–616
crossbar-based – crossbar-alapú, 616–619
on chip – egylapkás, 578–583
switching network – kapcsolóhálózat, 617–619
multiprogramming – multiprogramozás, 37
multisession CD-ROM – többszekciós CD-ROM, 113
multistage switching network – többszintű kapcsoló hálózat, 618
multithreading – többszálúság, 572–578
mutex – mutex, 513, 515
mutual exclusion – kölcsönös kizárás, 513
Myhrvold, Nathan – Myhrvold, Nathan, 43

N

NaN (Not a Number) – NaN (nem szám), 715
Nathan's first law of software – Nathan első szoftvertörvénye, 43
NC-NUMA (No Cashing NUMA) – NC-NUMA (gyorsítótár nélküli NUMA), 620, 603
near call – közeli hívás, 737
near jump – közeli ugrás, 735
negated signal – negált jel, 187
negative logic – negatív logika, 162
NetBurst microarchitecture – NetBurst mikroarchitektúra, 332
Network Interface Device (NID) – hálózati interfész eszköz (NID), 136
Network of Workstations (NOW) – munkaállomások hálózata (NOW), 606
network processor – hálózati processzor, 587–582
Nexperia – Nexperia, 592
nibble – falat, 406
NID (Network Interface Device) – NID (hálózati interfész eszköz), 136
No Cashing NUMA (NC-NUMA) – gyorsítótár nélküli NUMA (NC-NUMA), 620
NO Remote Memory Access (NORMA) – távoli memória elérése nélküli (NORMA), 606
nonblocking message passing – nem blokkoló üzenetküldés, 650
nonblocking network – nem blokkoló hálózat, 617
noninverting buffer – nem invertáló puffer, 185
NonUniform Memory Access (NUMA) – nem egységes memóriaelérés (NUMA), 605, 619–628
nonvolatile memory – nem felejtő memória, 190
NORMA (NO Remote Memory Access) – NORMA (távoli memória elérése nélküli), 606
normative information, in standard – normatív információ, szabványban, 355
Not a Number (NaN) – nem szám (NaN), 715
NOW (Network of Workstations) – NOW (munkaállomások hálózata), 606
NT File System (NTFS) – NT-fájlrendszer (NTFS), 505
NTFS (NT File System) – NTFS (NT-fájlrendszer), 505
NUMA (NonUniform Memory Access) – NUMA (nem egységes memóriaelérésű), 605, 619–628
n-way set-associative cache – n-utas halmazkezelésű gyorsítótár, 317

O

object file – tárgymodul, 741
object manager Windows XP – objektumkezelő Windows XP, 492
object module – tárgymodul, 551
object program – tárgyprogram, 524
OCP-IP (Open Core Protocol-International Partnership) – OCP-IP (nyílt magprotokoll nemzetközi társaság), 583
octal number – oktális szám, 698, 743
off-line – kapcsolat nélküli, 475
OGSA (Open Grid Services Architecture) – OGSA (nyílt grid szolgáltatások architektúrája), 669
OHCI (Open Host Controller Interface) – OHCI (nyílt kiszolgálóvezérlő interfész), 240
Old Program Counter (OPC) – régi utasításszámláló (OPC), 253, 262, 280
omega network – omega hálózat, 618
omnibus, PDP-8 – omnibus, PDP-8
on-chip multithreading – lapkaszintű többszálúság, 572–578
on-chip parallelism – lapkaszintű párhuzamosság, 564–583
one's complement – egyes komplemens, 703
on-line – on-line, 475

OPC (Old Program Counter) – OPC (régi utasításszámláló), 253, 262, 280
opcode – művkód, 252
open collector – nyílt gyűjtő/kollektor, 195
Open Core Protocol-International Partnership (OCP-IP) – nyílt magprotokoll nemzetközi társaság (OCP-IP), 583
Open Grid Services Architecture (OGSA) – nyílt grid szolgáltatások architektúrája (OGSA), 669
Open Host Controller Interface (OHCI) – nyílt kiszolgálóvezérlő interfész (OHCI), 240
operand stack – operandusverem, 267, 268
operating system – operációs rendszer, 25, 444, 461
CP/M – CP/M, 39
history – történeti áttekintés, 25
MS-DOS – MS-DOS, 40
OS/2 – OS/2, 40
UNIX – UNIX, 486–489, 495–496, 499–505, 510–513
Windows – Windows, 40
Windows XP – Windows XP, 489–495, 496–498, 505–510, 513–516
Operating System Machine (OSM) – operációs rendszer gép (OSM), 444–523
operating system machine (OSM) level – operációs rendszer gép (OSM) szintje, 22, 444–523
operating system macro – operációs rendszeri makro(utasítás), 27
operation – művelet, 567, 659
operation code – műveleti kód, 252
optical disk – optikai lemez, 108
Orange Book – Orange Book (Narancssárga Könyv), 113
Orca – Orca, 658
ormalized number – normalizált szám, 710
OS/2 – OS/2, 40
Osborne 1 – Osborne 1, 40
OSM (Operating System Machine) – OSM (operációs rendszer gép), 444
ounding – kerekítés, 710
out-of-order execution – sorrendtől eltérő végrehajtás, 324–329
overflow error – túlcsordulási hiba, 709
overlay – átfedés, 445

P

packet – csomag, 584, 631
packet processing engine – csomagfeldolgozó egység, 588
packet switching – csomagkapcsolás, 585
page – lap, 447
committed – egyeztetett, 497
reserved – foglalt, 497
page directory – lapkönyvtár, 464
page fault – laphiány, 452
page frame – lapkeret, 448
page map – laptábla, 447
page replacement algorithm – lapcserélő algoritmus, 453, 454
FIFO – FIFO, 454
LRU – LRU, 453
page replacement policy – lapcserélő eljárás, 453
page scanner – lapfelügyelő, 620
paging – lapozás, 447
demand – kérésre, 452
implementation – megvalósítása, 448
transparent – transzparens, 448
parallel computer architecture – párhuzamos számítógép-architektúra, 563–673
coprocessor – társprocesszor (koprocesszor), 583–598
instruction level parallelism – utasításszintű párhuzamosság, 565–572
media processor – médiaprocesszor, 592–597
multithreading – többszálúság, 572–578
multicomputer – multiszámítógép, 600–603
multiprocessor – multiprocesszor, 598–600
network processor – hálózati processzor, 568–575
on-chip parallelism – lapkaszintű, 564–583
taxonomy – osztályozás, 603–606
Parallel Input/Output (PIO) – párhuzamos be/kimenet (PIO), 241
Parallel Virtual Machine (PVM) – virtuális párhuzamos számítógép (PVM), 651
parallelism – párhuzamosság
instruction-level – utasításszintű, 75
processor-level – processzorszintű, 80
parent process – szülő processzus, 510
parity bit – paritásbit, 88
parity flag – paritás-flag, 734
partial address decoding – részleges címdekódolás, 244
partitioned resource sharing – erőforrás-felosztás, 576
Pascal, Blaise – Pascal, Blaise, 30
pass, assembler – menet, assembler, 539
passive matrix display – passzív mátrixmegjelenítő, 124
path – útvonal(név), 502
path length – úthossz, 292
reducing – csökkentés, 294
payload – hasznos adat, 233
PC (Program Counter) – PC (programszámláló, utasításszámláló), 252–262, 720
PCI (Peripheral Component Interconnect) – PCI (Peripheral Component Interconnect), PCI sín, 120, 223–231
signal – jel, 228
transaction – tranzakció, 230
PCI Express bus – PCI Express sín, 232

PDP-1 – PDP–1, 35
PDP-11 – PDP–11, 38
PDP-8 – PDP–8, 35
Pentium 4 – Pentium 4, 208, 359, 486
  addressing – címzés, 390–392
  bus – sín, 212
  data types – adattípusok, 370
  history – történeti áttekintés, 52
  instruction formats – utasításformátumok, 376–378
  instructions – utasítások, 405–408
  introduction – bevezetés, 37–42
  microarchitecture – mikroarchitektúra, 332
  overview of ISA – ISA-szintjének áttekintése, 359–362
  photograph – fénykép, 55
  pinout – lábkiosztás, 210
  problems – problémái, 431–432
performance of parallel computers – párhuzamos számítógépek teljesítménye
  achieving – elérése, 665–667
  Amdahl's law – Amdahl törvénye, 664
  hardware metrics – hardvermértékek, 661–663
  improving – növelés, 312–332, 661–667
  software metrics – szoftvermértékek, 663–665
Peripheral Component Interconnect (PCI) – PCI sín, Peripheral Component Interconnect (PCI), 120, 223–231
peripherial bus – perifériális sín, 582
perpendicular recording – merőleges rögzítés, 98
personal computer (PC) – személyi számítógép (PC), 49
Personal Digital Assisntant (PDA) – digitális személyi asszisztens (PDA), 41
pervasive computing – mindenütt jelenlévő számítástechnika, 41
phase modulation – fázismoduláció, 133
physical address space – fizikai címtartomány, 447
physical layer – fizikai réteg, 234
pigment based ink – pigmentalapú tinta, 132
pinout – lábkiosztás, 191
PIO (Parallel Input/Output) – PIO (párhuzamos be/kimenet), 241
pipe – cső/csővezeték, 511
pipeline – csővonal, csővezeték, szállítószalag, 76
  Mic-3 – Mic–3, 307
  Mic-4 – Mic–4, 311
  Pentium 4 – Pentium 4, 334
  seven-stage – hétszakaszú, 309
  UltraSPARC – UltraSPARC, 340
pipeline stage – csővezetékfázis, 77
pipeline stall – csővezeték bedugulás, 320
pipelined data path – csővonalas adatút, 305
pipelining – csővezeték alkalmazása, 76
pit – üreg, 108
pixel – képpont, pixel, 125
PLA (Programmable Logic Array) – PLA (programozható logikai tömb), 169
Plain Old Telephone Service (POTS) – egyszerű régi telefonszolgáltatás (POTS), 135
Playstation 2 – Playstation 2, 48
pointer – mutató, 381
pointer register – mutatóregiszter, 723
poison bit – mérgezésbit, 332
Polish notion – lengyel jelölés, 385–388
Portable Operating System-IX (POSIX) – POSIX, 487
position independent code – helyfüggetlen kód, 555
positive logic – pozitív logika, 162
POSIX (Portable Operating System-IX) – POSIX, 487
posxtfix notation – postfix jelölés, 385
POTS (Plain Old Telephone Service) – POTS (egyszerű régi telefonszolgáltatás), 135
poxtfix notation – postfix jelölés, 385
PPE (Protocol/Programmed/Packet Processing Engine) – PPE (protokoll-/programozott/csomag- feldolgozó egység), 588
preabmle – fejléc (szektorban), 98
precise interrupt – pontos megszakítás, 327
predication – predikáció/megalapozás, 436
prefetch buffer – előolvasási puffer, 76
prefetch cache – előrebetöltő gyorsítótár, 339
prefetching – előolvasás, 666
prefix byte – prefix bájt, 287, 377, 406
present/absent bit – jelenlét/hiány bit, 450
print engine – nyomtatómű, 130
printer – nyomtató, 127–133
  color – színes, 131
  dye sublimation – festékszublimációs, 132
  laser – lézer, 129
  monochrom – monokróm, 127
  solid ink – szilárd tintás, 132
  wax – viasz, 132
procedure – eljárás, 415–421
procedure call instruction – eljáráshívó utasítás, 400
procedure epilog – eljárásepilógus, 419
procedure prolog – eljárásprológus, 419
process – processzus, 445, 477
process and thread manager Windows XP – processzus- és szálkezelő Windows XP, 493
process management – processzuskezelés
  UNIX – UNIX, 510
  Windows XP – Windows XP, 513
process synchronization – processzus-szinkronizálás, 482, 489
processor – processzor, 66

processor bandwidth – processzor áteresztőképessége, 77
processor bus – processzorsín, 582
processor consistency – processzorkonzisztencia, 608
processor level parallelism – processzorszintű párhuzamosság, 80–83
producer-consumer problem – termelő-fogyasztó probléma, 478, 511
program – program, 17
Program Counter (PC) – programszámláló, utasításszámláló (PC), 252–262, 720
Program Status Word (PSW) – programállapotszó (PSW), 344, 358, 466
Programmable Logic Array (PLA) – programozható logikai tömb (PLA), 169
Programmable ROM (PROM) – programozható ROM (PROM), 190
programmed I/O – programozott B/K, 402
progressive scan – progresszív pásztázás, 596
PROM (Programmable ROM) – PROM (programozható ROM), 190
protected mode – védett üzemmód, 360
protocol – protokoll, 194, 234, 585
Protocol/Programmed/Packet Processing Engine (PPE) – protokoll-/programozott/csomag-feldolgozó egység (PPE), 588
pseudoinstruction – pszeudoutasítás, 531, 718, 744
PSW (Program Status Word) – PSW (programállapotszó), 344, 358, 466
pthread – pthread, 512
public key cryptography – nyilvános kulcsú titkosítás, 598
PVM (Parallel Virtual Machine) – PVM (virtuális párhuzamos számítógép), 651

Q

queueing unit – sorba állító egység, 310

R

race condition – versenyhelyzet, 482, 489
Radio Frequency IDentification (RFID) – rádiófrekvenciás azonosító (RFID), 44
radix – alapszám (számrendszeré), 698
  conversion between – konverzió számrendszerek között, 698, 700
radix number system – számrendszerek, 698
RAID (Redundant Array of Inexpensive Disks) – RAID (olcsó lemezek redundáns tömbje), 104–108
RAM (Random Access Memory) – RAM (véletlen elérésű memória), 60, 188
  dynamic (DRAM) – dinamikus (DRAM), 189
  static (SRAM) – statikus (SRAM), 189
  synchronous DRAM (SDRAM) – szinkron DRAM (SDRAM), 189
Random Access Memory (RAM) – véletlen elérésű memória, 60, 188
ranging – távolságbehatárolás, 139
raster scan – raszteres pásztázó, 122
RAW dependence – RAW függőség, 307
Read Only Memory (ROM) – csak olvasható memória (ROM), 60, 190
read/write pointer – olvasás/írás mutató, 739
real mode – valós üzemmód, 360
recursion – rekurzió, 400
recursive procedure – rekurzív eljárás, 415
Red Book – Red Book (Vörös Könyv), 108
Red Storm – Red Storm, 640
Reduced Instruction Set Computer (RISC) – csökkentett utasításkészletű számítógép (RISC), 73
  RISC versus CISC – RISC és CISC, 72
Redundant Array of Inexpensive Disks (RAID) – olcsó lemezek redundáns tömbje (RAID), 104–108
Reed-Solomon code – Reed–Solomon-kód, 98
register – regiszter, 21, 720
  accumulator – akkumulátor, 722
  addressing – címzés, 381
  displacement – indexelt, 728–727
  indirect addressing – indirekt címzés, 381, 382
  mode – mód, 381
  base – bázis, 722
  base pointer – bázismutató, 724
  condition code – feltételkód, 724
  counter – számláló, 722
  data – adat, 722
  destination index – célindex, 724
  flag – flag, 724
  index – index, 723
  instruction pointer (IP) – utasításmutató (IP), 720, 724
  pointer – mutató, 723
  program counter (PC) – programszámláló, utasításszámláló (PC), 252–262, 720
  renaming – regiszterátnevezés, 329
  source index – forrásindex, 724
  window – regiszterablak, 363
relative error – relatív hiba, 710
relative path – relatív útvonal, 502
release consistency – elengedési konzisztencia, 610
relocation constant – áthelyezési konstans, 551
relocation problem – áthelyezési probléma, 549
relocation, dynamic – áthelyezés, dinamikus, 552–555
ReOrder Buffer (ROB) – átrendező puffer (ROB), 336
replicated worker model – többszörözött munkás modell, 658
reserved page – foglalt lap, 497
resource layer – erőforrásréteg, 668

retirement unit – befejező egység, 337
reverse Polish notation – fordított lengyel jelölés, 385
RFID chip – RFID lapka, 44
right justified data – jobbra igazított adat, 395
ring – gyűrű, 634
ripple carry adder – átvitelt tovább terjesztő, 173
RISC (Reduced Instruction Set Computer) – RISC (csökkentett utasításkészletű számítógép), 72
RISC design principles – RISC-tervezési elvek
RISC versus CISC – RISC és CISC, 72
ROM (Read Only Memory) – ROM (Read Only Memory), 190
root directory – gyökérkönyvtár, 502
root hub, USB – központi csomópont, USB, 237
rotational latency – forgási késleltetés, 99
rounding – kerekítés, 710
router – útvonalválasztó, 585

S

SATA (Serial ATA) – SATA (soros ATA), 102
saturated arithmetic – telített módú aritmetika, 569
scalable – skálázható, 601, 665
Scalable Processor ARChitecture (SPARC) – skálázható processzorarchitektúra (SPARC), 57–59
Scale, Index, Base (SIB) – skála, index, bázis (SIB), 377, 391
scheduler – ütemező, 336
scheduling multicomputers – ütemezés, multiszámítógép, 652–653
scoreboard – eredményjelző, 325
SCSI (Small Computer System Interface) – SCSI (kis számítógéprendszerek interfésze), 103–104
SDRAM (Synchronous DRAM) – SDRAM (szinkron DRAM), 189
SRAM (Static RAM) – SRAM (statikus RAM), 189
Seastar – Seastar, 641
second pass – második menet, 742
secondary memory – háttérmemória, 96
second-generation computers – második generációs számítógépek, 34
sector, disk – szektor, lemez, 98, 110
security descriptor – biztonságleíró, 495, 508
Security ID (SID) – biztonsági azonosító (SID), 508
security reference monitor, Windows XP – biztonságiutalás-kezelő, Windows XP, 493
seek – keresés, 98
segment – szegmens, 725–726
segment override – szegmensregiszter-választó, 740
prefix – prefix, 740
segment register group – szegmensregiszter-csoport, 724
segmentation – szegmentálás, 456–462
best fit algorithm – legjobb illesztés algoritmus, 460
first fit algorithm – első illesztés algoritmus, 460
implementation – megvalósítása, 459
self-modifying program – önmódosító program, 382
semaphore – szemafor, 482, 511
sequencer – sorba állító, 261
sequential consistency – soros konzisztencia, 607
Serial ATA (SATA) – soros ATA (SATA), 102
server – kiszolgáló, szerver, 50
server farm – kiszolgáló farm, 51
session, CD-ROM – szekció, CD-ROM, 113
set-associative cache – csoportasszociatív gyorsítótár, halmazkezelésű gyorsítótár, 317
shard – szelet, 646
shared library – megosztott könyvtár, 559
shared memory (see multiprocessor) – közös memóriás (lásd multiprocesszor)
shared memory application-level – alkalmazásszintű közös memória, 654–661
shared memory multiprocessor – közös memóriás multiprocesszor, 598–629
shell – héj (parancsértelmező, shell), 489
shifter – léptető, 171
Shockley, William – Shockley, William, 34
SID (Security ID – SID (biztonsági azonosító), 508
sign extension – előjel-kiterjesztés, 258
signed magnitude – előjeles abszolút érték, 703
significand – meghatározó rész, szignifikáns, 713
SIMD (Single Instruction-stream Multiple Data stream) computer – SIMD számítógép, 81
SIMM (Single Inline Memory Module) – SIMM (egy érintkezősoros memóriamodul), 95
simple COMA – egyszerű COMA, 629
simplex line – szimplex vonal, 135
simultaneous multithreading – egyidejű többszálúság, 574
Single-chip multiprocessor – egylapkás multiprocesszorok, 578–583, 610–619
Single Inline Memory Module (SIMM) – egy érintkezősoros memóriamodul (SIMM), 95
Single Instruction-stream Multiple Data stream (SIMD) computer – SIMD számítógép, 81
Single Large Expensive Disk (SLED) – egyetlen nagy, drága lemez (SLED), 104
skew, bus – aszimmetria, sín, 197
slave, bus – szolga, sín, 195
SLED (Single Large Expensive Disk) – SLED (egyetlen nagy, drága lemez), 104

Small Computer System Interface (SCSI) – kis számítógéprendszerek interfésze (SCSI), 103–104
small model, 8088 – small modell, 8088, 746
Small Outline DIMM (SO-DIMM) – kisméretű kétsoros érintkezős memóriamodul (SO-DIMM), 95
SMP (Symmetric MultiProcesszor) – SMP (szimmetrikus multiprocesszor), 600, 610–619
snooping cache – szimatoló gyorsítótár, 609, 611–614
snoopy cache – szimatoló gyorsítótár, 609, 611–614
socket – socket (csatlakozási pont), 487, 515
software – szoftver, 24
equivalence with hardware – ekvivalenciája a hardverrel, 24
software layer – szoftverréteg, 236
Solaris – Solaris, 487, 511
solid ink printer – szilárd tintás nyomtató, 132
source index – forrásindex, 724
source language – forrásnyelv, 524
source operand – forrásoperandus, 726
SP (Stack Pointer) – SP (veremmutató), 253, 262, 266, 280, 723
SPARC (Scalable Processor ARChitecture) – skálázható processzorarchitektúra (SPARC), 57–59
spatial locality – térbeli lokalitás, 314
speculative execution – feltételezett végrehajtás, 330
speculative load – spekulatív betöltés, 438
split cache – szétválasztott gyorsítótár, 94, 313
splitter – szétválasztó, 136
SR latch – SR-tároló, 177
SRAM (Static RAM) – SRAM (statikus RAM), 189
SSE – SSE, 54
stack – verem, 266
stack addressing – veremcímzés, 384–388
stack frame – veremkeret, 723
Stack Pointer (SP) – veremmutató (SP), 253, 262, 266, 280, 723
stage, pipeline – fázis, csővezeték, 77
stale data – elavult adat, 611
stalled, pipeline – bedugult, csővezeték, 320
stalling – elakadás, 308
standard error – szabványos hibacsatorna, 502
standard input – szabványos bemeneti csatorna, 501
standard output – szabványos kimeneti csatorna, 502
star – csillag, 633
8051 – 8051, 345
state – állapot
finite state machine – véges állapotú gép, 298
Static RAM (SRAM) – statikus RAM (SRAM), 189
Stibbitz, George – Stibbitz, George, 31
storage – tároló, 83
store (see memory) – tár (lásd memória)
store-and-forward packet switching – tárold-és-továbbítsd csomagkapcsolás, 585
store-to-load – tárolás utáni betöltés, 337
stream – stream (folyam), 488
Streaming SIMD Extensions (SSE) – Streaming SIMD-kiegészítések (SSE), 54
strict consistency – szigorú konzisztencia, 607
striping – csíkozás, 105
strobe – kapuzójel, 178
structured computer organization – struktúrált számítógép-felépítés, 17
subroutine – szubrutin, 400, 737
Sun Fire E25K – Sun Fire E25K, 617, 624–628
Sun Microsystem – Sun Microsystem, 57–59
supercomputer – szuperszámítógép, 36, 51
superscalar architecture – szuperskaláris architektúra, 78–80
superuser – rendszeradminisztrátor, 503
supervisor call – rendszerhívás, 27
switching algebra – kapcsolóalgebra, 155
switching network – kapcsolóhálózat, 617–619
symbol table – szimbólumtábla, 545–547, 741
symbolic name – szimbolikus név, 718
symmetric key cryptography – szimmetrikus kulcsú titkosítás, 598
Symmetric MultiProcesszor (SMP) – szimmetrikus multiprocesszor (SMP), 600, 610–619
synchronous bus – szinkron sín, 198
Synchronous DRAM (SDRAM) – szinkron DRAM (SDRAM), 189
synchronous message passing – szinkron üzenetátadás, 650
system bus – rendszersín, 194
system call – rendszerhívás, 27, 444, 487, 494
system interface – rendszerinterfész, 494
system on a chip – egylapkás rendszer, 578–583
system programmer – rendszerprogramozó, 22
system services, Windows XP – rendszerszolgáltatások, Windows XP493

T

target language – célnyelv, 524
target library – célkönyvtár, 559
target, PCI bus – céleszköz, PCI sín, 227
task bag – feladatzsák, 658
TAT-12/13 – TAT-12/13, 43
TCP (Transmission Control Protocol) – TCP, 586
TCP header – TCP-fejléc, 586
telco (telephone company) – telco (telefontársaság), 135

telecommunications equipment telekommunikációs berendezés, 133
telephone company (telco) – telefontársaság (telco), 135
template – minta, 657
temporal locality – időbeli lokalitás, 314
TEXT section – TEXT szegmens, 742
TFT (Thin Film Transistor) – TFT (vékonyfilm-tranzisztor), 124
TFT display – TFT megjelenítő, 124
Thin Film Transistor (TFT) – vékonyfilm-tranzisztor (TFT), 124
third-generation computers – harmadik generációs számítógépek, 36
thrashing – vergődés, 454
thread – szál, 479, 511
three-bus architecture – háromsínes architektúra, 295
threshold sharing – küszöbölt (erőforrás-) megosztás, 577
tightly coupled – szorosan kapcsolt, 564
time sharing system – időosztásos rendszer, 27
tiny model, 8088 – tiny modell, 8088, 746
TLB (Translation Lookaside Buffer) – TLB (lapkezelő segédpuffer), 467
TLB miss – TLB-hiány, 468
TN (Twisted Nematic) display – TN (elforgatott molekulájú) kijelző, 123
token – token, vezérjel, 582
Top Of Stack (TOS) – veremtető (TOS), 280
TOS (Top Of Stack) – TOS (veremtető), 253, 262, 280
towers of Hanoi – Hanoi tornyai, 415–421, 427–431
  Pentium – Pentium, 427–429
  UltraSPARC – UltraSPARC, 429–431
trace BTB – nyomkövető BTB, 336
trace cache – nyomkövető gyorsítótár, 334
tracer – nyomkövető, 718
  Intel 8088 – Intel 8088, 747
track – sáv, 98
transaction layer – tranzakciós réteg, 235
transistor, invention – tranzisztor, feltalálása, 34
Transistor-Transistor Logic (TTL) – tranzisztor-tranzisztor logika (TTL), 155
transition – átmenet, 298
transiation – fordítás, 18
Translation Lookaside Buffer (TLB) – lapkezelő segédpuffer (TLB), 340, 467
Translation Storage Buffer (TSB) – lapkezelő tárolópuffer (TSB), 468
transiation table – fordítótábla, 468
translator – fordító, 524
Transmission Control Protocol (TCP) – TCP, 586
transparency – átlátszóság (megszakítás), 424
transparent (paging) – transzparens/átlátszó (lapozási mechanizmus), 448
trap – csapda, 422, 733
trap handler – csapdakezelő, 422
tree – fa, 633
TriMedia – TriMedia, 567–572
triple indirect block – háromszorosan indirekt blokk, 504
tri-state device – *háromállapotú eszköz*, 185
true dependence – valódi függőség, 307
truth table – igazságtábla, 156
TSB (Translation Storage Buffer) – TSB (lapkezelő tárolópuffer), 468
TTL (Transistor-Transistor Logic) – TTL (tranzisztor-tranzisztor logika), 155
tuning, program – hangolás, program, 527
tuple – adategység, 656
tuple space – adategységtér, 656
twin – iker, 656
Twisted Nematic (TN) display – elforgatott molekulájú (TN) kijelző, 123
two's complement arithmetic – kettes komplemens aritmetika, 703
two-pass translator – kétmenetes fordító, 539
TX-0 – TX-0, 34
TX-2 – TX-2, 34

U

U pipeline – fő csővezeték, szállítószalag, 78
UART (Universal Asynchronous Receiver/Transmitter) – UART (univerzális aszinkron adó/vevő), 240
ubiquitous computing – mindenütt jelenlévő számítástechnika, 41
UDB II (UltraSPARC Data Buffer II) – UDB II, 217
UHCI Universal Host Controller Interface) – UHCI (univerzális kiszolgálóvezérlő interfész), 240
Ultra Port Architecture (UPA) – UPA, 217
UltraSPARC Data Buffer II (UDB II) – UDB II, 217
UltraSPARC I – UltraSPARC I, 58
UltraSPARC III – UltraSPARC III, 214–218, 378
  addressing – címzés, 392
  data buffer – adatpuffer, 217
  data types – adattípusok, 370–371
  history – történeti áttekintés, 57
  instructions – utasítások, 408–410
  instruction formts – utasításformátumok, 378–379
  microarchitecture – mikroarchitektúra, 338–343
  overview of ISA – ISA szintjének áttekintése, 362–365
  towers of Hanoi – Hanoi tornyai, 429–431
  virtual memory – virtuális memória, 466–469
UMA (Uniform Memory Access) – UMA (egységes memóriaelérés), 605

underflow error – alulcsordulási hiba, 710
UNICODE – UNICODE, 145
unified cache – egyesített gyorsítótár, 94
Uniform Memory Access (UMA) – egységes memóriaelérés (UMA), 605
Universal Asynchronous Receiver/Transmitter (UART) – univerzális aszinkron adó/vevő (UART), 240
Universal Host Controller Interface (UHCI) – univerzális kiszolgálóvezérlő interfész (UHCI), 240
Universal Serial Bus (USB) – univerzális soros sín (USB), 236–240
Universal Synchronous Asynchronous Receiver/Transmitter (USART) – univerzális szinkron, aszinkron adó-vevő (USART), 241
UNIX – UNIX
  file I/O – fájl B/K, 499–505
  introduction – bevezetés, 486
  linking – szerkesztés, 559
  process management – processzuskezelés, 489, 510
  virtual memory – virtuális memória, 495
UPA (Ultra Port Architecture) – UPA, 217
update strategy – frissítő stratégia, 613
USART (Universal Synchronous Asynchronous Receiver/Transmitter) – USART (univerzális szinkron, aszinkron adó-vevő), 241
USB (Universal Serial Bus) – USB (univerzális soros sín), 236–240
user mode – felhasználói mód, 355

V

V pipeline – második csővezeték, szállítószalag, 78
vampire tap – vámpírcsatlakozó, 584
VAX – VAX, 71
VCI (Virtual Component Interconnect) – VCI (virtuális komponens összeköttetés), 583
vector register – vektorregiszter, 81
vectorprocessor – vektorprocesszor, 81
Very Large Scale Integration (VLSI) – nagyon magas fokú integráltság (VLSI), 38
Very Long Instruction Word (VLIW) – nagyon hosszú utasításszavú (VLIW), 565–567
Video RAM – Video RAM, videomemória, 125
virtual 8086 mode – virtuális 8086 mód, 360
virtual address space – virtuális címtartomány, 447
  UNIX – UNIX, 499
virtual circuit – virtuális áramkör, 235
Virtual Component Interconnect (VCI) – virtuális komponens összeköttetés (VCI), 583
virtual cut through – virtuális vágás, 639
virtual I/O – virtuális B/K, 469
  implementation – megvalósítás, 505
  Windows XP – Windows XP, 505
virtual machine – virtuális gép, 18
virtual memory – virtuális memória, 445–469
  compared to caching – és gyorsítótár, 469 452–453
  manager, Windows XP – kezelő, Windows XP, 493
  Pentium 4 – Pemtium 4, 462–466
  UltraSPARC – UltraSPARC,466–469
  UNIX – UNIX, 495–496
  Windows XP – Windows XP, 496
virtual organization – virtuális szervezet, 668
virtual register – virtuális regiszter, 265
virtual topology – virtuális topológia, 652
virtuous circle – bűvös kör, 43
Visual Instruction Set (VIS) – vizuális utasításkészlet (VIS), 58
VLIW (Very Long Instruction Word) – nagyon hosszú utasításszavú (VLIW), 565–567
VLSI (Very Large Scale Integration) – VLSI (nagyon magas fokú integráltság), 38, 490
Volume Table Of Contents (VTOC) – kötet tartalomjegyzék (VTOC), 113
von Neumann machine – Neumann-gép, 33
von Neumann, John – Neumann János, 33
VTOC (Volume Table Of Contents) – VTOC (kötet tartalomjegyzék), 113

W

wait state – várakozó állapot, 199
WAN (Wide Area Network) – WAN (nagyterületű hálózat), 584
Wattel, Evert – Wattel, Evert, 717
wax printer – viasznyomtató, 132
weak consistency – gyenge konzisztencia, 609
WEIZAC – WEIZAC, 32
Whirlwind I – Whirlwind I, 34
Wide Area Network (WAN) – nagyterületű hálózat (WAN), 584
Wilkes, Maurice – Wilkes, Maurice, 24, 32, 71
Win32 API – Win32 API, 494
Win32 subsystem – Win32 alrendszer, 494
Winchester disk – merevlemez, 98
Windows – Windows, 40, 490
  history – történet, 40
  linking – szerkesztés, 557
Windows 95 – Windows 95, 490
Windows 98 – Windows 98, 490
Windows New Technology (NT) – Windows NT, 490
Windows XP – Windows XP
  file I/O – fájl B/K, 505–510
  introduction – bevezetés, 489–495
  process management – processzuskezelés 513–516
  virtual memory – virtuális memória, 496–498
wired-OR – huzalozott VAGY, 196
word – szó, 86, 727

word instruction – szavas utasítás, 727
word register – szavas regiszter, 727
working directory – munkakönyvtár, 502
working set – munkahalmaz, 452
Wozniak, Steve – Wozniak, Steve, 39
Write After Read (WAR) dependence – olvasás utáni írás (WAR) függőség, 327
Write After Write (WAW) dependence – írás utáni írás (WAW) függőség, 327
write once protocol – egyszer író protokoll, 614
write through – írásáteresztő, 612
write-allocate cache – íráskori feltöltés módszere, 319, 613
write-back protocol – késleltetett írású protokoll, 319, 614
write-deferred cache – késleltetett írás, 298
write-through cache – írásáteresztő gyorsítótár, 318

X, Y, Z

X Window System – X Window System, 489
XP file system – XP-fájlrendszer, 505
Y2K bug – 2000. év probléma, 51
Yellow Book – Yellow Book (Sárga Könyv), 110
zeroth-generation computers – nulladik generációs számítógépek, 30
Zilog Z8000 – Zilog Z8000, 72
Zuse, Konrad – Zuse, Konrad, 31

Magyar–angol tárgymutató

2. szintű gyorsítótár – level 2 cache, 314
2000. év probléma – Y2K bug, 51
8051 (lásd Intel 8051) – 8051 (see Intel 8051)
8088 (lásd Intel 8088) – 8088 (see Intel 8088)

A, Á

abszolút útvonal – absolute path, 502
ACK (Amsterdam Compiler Kit, Amsterdam fordítókészlet) – ACK (Amsterdam Compiler Kit), 742
ACL (elérést vezérlő lista) – ACL (Access Control List), 508
adategység – tuple, 656
adategységtér – tuple space, 656
adatgyorsítótár – data cache, 339
adatmozgató utasítás – data movement instruction, 394
adatszegmens – data segment, 734
adattípus – data type, 368
  nem numerikus – nonnumeric, 369
  numerikus – numeric, 368
adatút – data path, 21, 67, 252–258
  időzítés – timing, 255
  Mic–1 – Mic-1, 262
  Mic–2 – Mic-2, 301
  Mic–3 – Mic-3, 304
  Mic–4 – Mic-4, 309
adatútciklus – data path cycle, 68
additív inverz – additive inverse, 398
ADSL (aszimmetrikus DSL) – ADSL (Asymmetric DSL), 135
AGP (gyorsított grafikus port) – AGP (Accelerated Graphics Port), 125, 226
AGP sín – AGP bus, 125, 226
Aiken, Howard – Aiken, Howard, 31
akadály – hazard, 307
akkumulátor – accumulator, 34, 343, 368
akkumulátorregiszter – accumulator register, 722
aktív mátrixmegjelenítő – active matrix display, 124
aktuális paraméter – actual parameter, 536
alapblokk – basic block, 329
alaplap – motherboard, 117
alapszám (számrendszeré) – radix, 698
  konverzió számrendszerek között – conversion between, 698, 700
alapvető be-kimeneti rendszer (BIOS) – Basic Input Output System (BIOS), 101
algoritmus – algorithm, 24
alkalmazásfejlesztést támogató interfész (API) – Application Programming Interface (API), 494
alkalmazáshoz kifejlesztett egyedi integrált áramkör (ASIC) – Application-Specific Integrated Circuit (ASIC), 587
alkalmazási réteg – application layer, 669
  8051 – 8051, 345
alkalmazásszintű közös memória – shared memory application-level, 654–661
állapot – state
  véges állapotú gép – finite state machine, 298
Alpha – Alpha, 40
általános regiszter – general regiszter, 722
ALU (aritmetikai-logikai egység) – ALU (Arithmetic Logic Unit), 21, 67, 173–175
alulcsordulási hiba – underflow error, 710
AMBA (fejlett mikrokontroller sínarchitektúra) – AMBA (Advanced Microcontroller Bus Architecture), 583
Amdahl törvénye – Amdahl's law, 664
amplitúdómoduláció – amplitude modulation, 133
Amsterdam Compiler Kit (Amsterdam fordítókészlet) – Amsterdam Compiler Kit (ACK), 742
analitikus gép – analytical engine, 30
APIC (fejlett programozható megszakításvezérlő) – APIC (Advanced Programmable Interrupt Controller), 212
Apple II – Apple II, 39

Apple Macintosh – Apple Macintosh, 39
áramkör – circuit
  aritmetikai – arithmetic, 170–175
  kombinációs – combinational, 165
áramköri ekvivalencia – circuit equivalence, 159–162
architektúra – architecture, 23, 71
architektúrák összehasonlítása – comparison of architectures, 346
argumentum – argument, 737
aritmetika, bináris – arithmetic, binary, 705
aritmetikai áramkörök – arithmetic circuit, 170–175
aritmetikai-logikai egység (ALU) – Arithmetic Logic Unit (ALU), 21, 67, 173–175
ASCII (az információcsere amerikai szabványos kódrendszere – ASCII (American Standard Code for Information Interchange), 143
ASIC (alkalmazáshoz kifejlesztett egyedi integrált áramkör) – ASIC (Application-Specific Integrated Circuit), 587
assembler – assembler, 23, 525, 539, 718, 741
  direktíva – directive, 531
  első menet – pass one, 540–544
  Intel 8088 – Intel 8088, 741
  kétmenetes – two-pass, 539
  második menet – pass two, 544–545
  szimbólumtábla – symbol table, 545
assembler menetei – assembly process, 539–547
  első menet – pass one, 540–544
  kétmenetes – two-pass, 539
  második menet – pass two, 544–545
  8088 – 8088, 741
assembly nyelv – assembly language, 524, 717
  jellemzői – characteristics, 525
  miért használatos – why use, 526–528
  pszeudoutasítás – pseudoinstuction, 531–534
  szint – level, 524–562
  utasítás – statement, 528
assembly nyelvű program – assembly language program, 525, 717, 741
assembly nyelvű programozás – assembly language programming, 717
  áttekintés – overview, 718
aszimmetria, sín – skew, bus, 197
aszimmetrikus DSL (ADSL) – Asymmetric DSL (ADSL), 135
aszinkron sín – asynchronous bus, 201–202
asszociatív memória – associative memory, 461, 545
AT kiegészítő – AT Attachment, 102
ATA-3 – ATA-3, 102
ATA-csomaginterfész (ATAPI) – ATA Packet Interface (ATAPI), 102
Atanasoff, John – Atanasoff, John, 31, 33
ATAPI-4 – ATAPI-4, 102
átfedés – overlay, 445
áthelyezés, dinamikus – relocation, dynamic, 552
áthelyezési konstans – relocation constant, 551
áthelyezési probléma – relocation problem, 549
átlátszóság (megszakítás) – transparency, 424
átmenet – transition, 298
átmérő, hálózat – diameter, network, 632
átrendező puffer (ROB) – ReOrder Buffer (ROB), 336
áttetszőségi csatorna maszk – alpha channel mask, 595
átvitelkiválasztó összeadó – carry select adder, 173
átvitelt tovább terjesztő – ripple carry adder, 173
automatikus csökkentés – auto decrement, 730
automatikus növelés – auto increment, 730

B

B/K (Bemenet/Kimenet) – I/O (Input/Output), 117–147
B/K kezelő, Windows XP – I/O manager, Windows XP, 492
B/K utasítások – I/O instructions, 402–405
Babbage, Charles – Babbage, Charles, 30
bájt – byte, 86, 727
bájtos regiszter – byte register, 727
bájtos utasítás – byte instruction, 727
bájtsorrend – byte ordering, 86–88
balérték – left value, 727
Bardeen, John – Bardeen, John, 34
baud – baud, 134
Bayer-szűrő – Bayer filter, 142
bázis – base, 153
bázis-index címzés – based-indexed addressing, 384
bázismutató – base pointer, 724
bázisregiszter – base register, 722
BCD (binárisan kódolt decimális) – BCD (Binary Coded Decimal), 84, 406, 727
beágyazott számítógép – embedded computer, 41
beállított jel – asserted signal, 187
Bechtolsheim, Andy – Bechtolsheim, Andy, 57
bedugult, csővezeték – stalled, pipeline, 320
beépített eszközelektronika (IDE) – Integrated Drive Electronics (IDE), 101
befejező egység – retirement unit, 337
bejövő csomagokon végzett feldolgozás – ingress processing, 589
belépési pont – entry point, 551
belső elaprózódás – internal fragmentation, 455
Bemenet/Kimenet (B/K) – Input/Output (I/O), 117–147
bemeneti/kimeneti eszköz – input/output device
  billentyűzet – keyboard, 121
  CRT Monitor – CRT monitor, 121
  digitális kamera – digital camera, 141
  egér – mice, 126
  lapos megjelenítő – flat panel display, 122

  mágneslemez – magnetic disk, 97
  modem – modem, 133–141
  nyomtató – printer, 127–133
  optikai lemez – optical disk, 108–117
  telekommunikációs berendezés – telecommunications equipment, 133–141
  terminál – terminal, 121–125
bemeneti/kimeneti utasítások – input/output instructions, 402–405
Berkeley UNIX – Berkeley UNIX, 486, 487, 489
betöltés – loader, 547
betöltő/tároló architektúra – load/store architecture, 365
betöltő-dekódoló-végrehajtó ciklus – fetch-decode-execute cycle, 69
betöltő-végrehajtó ciklus – fetch-execute cycle, 252
billenőkör/flip-flop – flip-flop, 179
billentyűzet – keyboard, 121
bináris aritmetika – binary arithmetic, 705
bináris keresés – binary search, 546
bináris program – binary program, 718
bináris szám – binary number, 696
  negatív – negative, 703
  összeadás – addition of, 705
Binárisan kódolt decimális (BCD) – Binary Coded Decimal (BCD), 84, 406, 727
BIOS (alapvető be-kimeneti rendszer) – BIOS (Basic Input Output System), 101
bipoláris tranzisztor – bipolar transistor, 155
bit – bit, 84, 699
bitblokk-átvitel (bitblt) – BIT Block Transfer (bitblt), 596
bitblt (bitblokk-átvitel) – bitblt (BIT Block Transfer), 596
bitszelet – bit slice, 174
bittérkép – bit map, 370
biztonsági azonosító (SID) – Security ID (SID), 508
biztonságiutalás-kezelő, Windows XP – security reference monitor, Windows XP, 493
biztonságleíró – security descriptor, 495, 508
blokkátvitel, sín – block transfer, bus, 205
blokkgyorsító – block cache, 489
blokkoló hálózat – blocking network, 619
BlueGene – BlueGene, 635–640
BlueGene/L – BlueGene/L, 635–640
Blu-Ray – Blu-Ray, 117
Boole, George – Boole, George, 155
Boole-algebra – Boolean algebra, 155–157
bővített ISA (EISA) – Extended ISA (EISA), 120
Brattain, Walter – Brattain, Walter, 34
BSS (szimbólummal kezdődő blokk) – BSS (Block Started by Symbol), 742
Burroughs B5000 – Burroughs B5000, 36
bűvös kör – virtuous circle, 43
Byron, Lord – Byron, Lord, 30

C, CS

catamount – catamount, 642
CCD (töltéscsatolt eszköz) – CCD (Charge Coupled Device), 142
CC-NUMA (gyorsítótár-koherens NUMA) – CC-NUMA (Cache Coherent NUMA), 619–628
CDC (töltéscsatolt eszköz) – CDC (Charge Coupled Device), 142
CDC 6600 – CDC 6600, 36, 79, 325
CDC Cyber – CDC Cyber, 71
CD-R (írható CD) – CD-R (CD-recordable, 112
CD-ROM – CD-ROM (Compact Disc Read Only Memory), 108
  sáv – track, 113
  szektor – sector, 110
  újraírható (CD-RW) – rewritable (CD-RW), 114
  XA – XA, 113
CD-RW (újraírható CD) – CD-RW (CD-rewritable), 114
Celeron – Celeron, 55
céleszköz, PCI sín – target, PCI bus, 227
célindex – destination index, 724
célkönyvtár – target library, 559
cella, memória – cell, memory, 84
célnyelv – target language, 524
céloperandus – destination operand, 726
ciklikus redundanciakód (CRC) – Cyclic Redundancy Check (CRC), 235, 239, 586
ciklusidő – clock cycle time, 175
cikluslopás – cycle stealing, 119, 405
cilinder – cylinder, 98
cím – address, 84, 720
  memória – memory, 84
címdekódolás – address decoding, 242
címke – label, 718
címtartomány – address space, 38, 446
  fizikai – physical, 447
  virtuális – virtual, 447
címzés – addressing, 371, 380–394
  bázis relatív – register with index, 729
  bázis relatív eltolással – register with index and displacement, 729
  bázis-index – based-indexed, 384
  direkt – direct, 381, 728
  elágazó utasítások – branch instruction, 388–384
  implicit – implied, 729
  indexelt – indexed, 382
  indexelt – register displacement, 729
  Intel 8051 – Intel 8051, 392–393
  Intel 8088 – Intel 8088, 726–730
  közvetlen – immediate, 380
  Pentium 4 – Pentium 4, 390–392
  regiszter – register, 381

regiszter-indirekt – register indirect, 381–382, 728
UltraSPARC III – UltraSPARC III, 392
verem – stack, 384–388
címzési mód – addressing mode, 380
összefoglalás – discussion, 393
CISC (összetett utasításkészletű számítógép) – CISC (Complex Instruction Set Computer), 73
CLUT (színkereső tábla) – CLUT (Color Look Up Table), 595
CMYK-nyomtató – CMYK printer, 131
COBOL program – COBOL program, 51
COLOSSUS – COLOSSUS, 32
COMA (gyorsítótáras memóriaelérés) – COMA (Cache Only Memory Access), 628, 687–688
Compact Disc Read Only Memory (CD-ROM) – Compact Disc Read Only Memory (CD-ROM), 108
Control Data Corporation (CDC) – Control Data Corporation (CDC), 36
CoreConnect – CoreConnect, 582
COW (munkaállomások klasztere) – COW (Cluster of Workstations), 50–51, 606, 644
CP/M – CP/M, 39
CPP (konstans mező mutatója) – CPP (Constant Pool Pointer), 253, 262, 269, 280
CPU lapka – CPU chip, 192
Cray, Seymour – Cray, Seymour, 36
Cray-1 – Cray-1, 36, 71
CRC (ciklikus redundanciakód) – CRC (Cyclic Redundancy Check/Code), 235, 239, 586
crossbar (keresztrudas) kapcsoló – crossbar switch, 617
CRT (katódsugárcső) – CRT (Cathode Ray Tube), 121
csak olvasható memória (ROM) – Read Only Memory (ROM), 60, 190
csapda – trap, 422, 733
csapdakezelő – trap handler, 422
csatoló szegmens – linkage segment, 555
csíkozás – striping, 105
csillag – star, 633
csomag – packet, 584, 631
csomagfeldolgozó egység – packet processing engine, 588
csomagkapcsolás – packet switching, 585
csoportasszociatív gyorsítótár – set-associative cache, 317
cső/csővezeték – pipe, 511
csökkentett utasításkészletű számítógép (RISC) – Reduced Instruction Set Computer (RISC), 73
RISC és CISC – RISC versus CISC, 72
csővezeték alkalmazása – pipelining, 76
csővezeték bedugulás – pipeline stall, 320
csővezeték fázis – pipeline stage, 77
csővonal, csővezeték – pipeline, 305
hétszakaszú – seven-stage, 309
Mic-3 – Mic-3, 307
Mic-4 – Mic-4, 311
Pentium 4 – Pentium 4, 334
UltraSPARC – UltraSPARC, 340
csővonal, csővezeték, szállítószalag –, 76
csővonalas adatút – pipelined data path, 305

D

D-tároló – D latch, 179
DATA szegmens – DATA section, 742
DDR (kétszeres sebességű memória) – DDR (Double Data Rate memory), 190
De Morgan-szabály – DeMorgan's law, 160–161
DEC (Digital Equipment Corporation) – DEC (Digital Equipment Corporation), 35, 38, 71
Alpha – Alpha, 40
PDP-1 – PDP-1, 35
PDP-8 – PDP-8, 35
VAX – VAX, 71, 73
decimális szám – decimal number, 743
dekódoló – decoder, 167
dekódoló egység – decoding unit, 310
demultiplexer – demultiplexer, 167
diadikus műveletek – dyadic instructions, 395
differenciagép – difference engine, 30
Digital Equipment Corporation (DEC) – Digital Equipment Corporation (DEC), 35, 38, 71
digitális előfizetői vonal (DSL) – Digital Subscriber Line (DSL), 135
digitális előfizetői vonal hozzáférési multiplexer (DSLAM) – Digital Subscriber Line Access Multiplexer (DSLAM), 137
digitális kamera – digital camera, 141
digitális logika szintje – digital logic level, 20, 152–250
áramkör – circuit, 163
B/K interfész – I/O interface, 240
CPU lapka – CPU chip, 192
kapu – gate, 153
memória – memory, 176
sín – bus, 194
digitális személyi asszisztens (PDA) – Personal Digital Assisntant (PDA), 41
digitális videolemez – Digital Video Disc, 115
dimenziószám – dimensonality, 633
DIMM (két érintkezősoros memóriamodul) – DIMM (Dual Inline Memory Module), 95
dinamikus áthelyezés – dynamic relocation, 552–555
dinamikus RAM (DRAM) – Dynamic RAM (DRAM), 189
dinamikus szerkesztés – Dynamic Linking, 555
dinamikus szerkesztő könyvtár (DLL) – Dynamic Link Library (DLL), 557
dinamikus véletlen elérésű memória – Dynamic Random Access Memory (DRAM), 31

DIP (kétlábsoros tokozás) – DIP (Dual Inline Package), 163
direkt címzés – direct addressing, 381
direkt leképezésű gyorsítótár – direct-mapped cache, 315
DLL (dinamikus szerkesztő könyvtár) – DLL (Dynamic Link Library), 557
DMA (közvetlen memóriaelérés) – DMA (Direct Memory Access), 119, 404
dpi (pont per inch) – dpi (Dots Per Inch), 129
DRAM (dinamikus RAM) – DRAM (Dynamic RAM), 189
DSL (digitális előfizetői vonal) – DSL (Digital Subscriber Line), 135
DSLAM (digitális előfizetői vonal hozzáférési multiplexer) – DSLAM (Digital Subscriber Line Access Multiplexer), 137
DSM (elosztott közös memória) – DSM (Distributed Shared Memory), 602, 654
dupla pontosság – double precision, 368
dupla pontosságú – double, 722
dupla szélességű mutató – double pointer, 345
durva szemcsézettségű többszálúság – coarse-grained multithreading, 573
DVD (sokoldalú digitális lemez) – DVD (Digital Versatile Disc), 115

E, É

ECC (hibajavító kód) – ECC (Error Correcting Code), 88
Eckert, J. Presper – Eckert, J. Presper, 32
ECL (emitter csatolású logika) – ECL (Emitter-Coupled Logic), 155
EDO (kiterjesztett adatkimenetű memória) – EDO (Extended Data Output) memory, 189
EDVAC – EDVAC (Electronic Discrete Variable Automatic Computer EDVAC), 32
EEPROM (elektromosan törölhető PROM) – EEPROM (Electrically Eresable PROM), 190
effektív cím – effective address, 729
egér – mouse, 126
egész aritmetikájú java virtuális gép (IJVM) – Integer Java Virtual Machine (IJVM), 251–260, 266–275
egy érintkezősoros memóriamodul (SIMM) – Single Inline Memory Module (SIMM), 95
egyes komplemens – one's complement, 703
egyesített gyorsítótár – unified cache, 94
egyetlen nagy, drága lemez (SLED) – Single Large Expensive Disk (SLED), 104
egyeztetett lap – committed page, 497
egyidejű többszálúság – simultaneous multithreading, 574
egylapkás multiprocesszorok – Single-chip multiprocessor, 578–583, 610–619
egylapkás rendszer – system on a chip, 578–583
egységes memóriaelérés (UMA) – Uniform Memory Access (UMA), 605
egyszer író protokoll – write once protocol, 614
egyszerű COMA – simple COMA, 629
egyszerű régi telefonszolgáltatás (POTS) – Plain Old Telephone Service (POTS), 135
együttes tervezés – codesign, 41
EHCI (kibővített kiszolgálóvezérlő interfész) – EHCI (Enhanced Host Controller Interface), 240
EIDE (kiterjesztett IDE) – EIDE (Extended IDE), 101
EISA (bővített ISA) – EISA (Extended ISA), 120
elágazás – branch, 414–415
elágazási célpuffer (BTB) – Branch Target Buffer (BTB), 335
elágazási előzmények blokkos regisztere – branch history shift register, 323
elágazásjövendölés – branch prediction, 319–324
dinamikus – dynamic, 321–323
statikus – static, 324
elakadás – stalling, 308
elaprózódás – fragmentation
belső – internal, 455
külső – external, 460
elavult adat – stale data, 611
elektromosan törölhető (EEPROM) – Electrically Eresable PROM (EEPROM), 190
elengedési konzisztencia – release consistency, 610
elérést vezérlő lista (ACL) – Access Control List (ACL), 508
elforgatott molekulájú (TN) kijelző – Twisted Nematic (TN) display, 123
eljárás – procedure, 415–421
eljárásepilógus – procedure epilog, 419
eljáráshívó utasítás – procedure call instruction, 400
eljárásprológus – procedure prolog, 419
elosztott közös memória (DSM) – Distributed Shared Memory (DSM), 602, 654
előjeles abszolút érték – signed magnitude, 703
előjel-kiterjesztés – sign extension, 258
előolvasás – prefetching, 666
előolvasási puffer – prefetch buffer, 76
előre hivatkozási probléma – forward reference problem, 539
előrebetöltő gyorsítótár – prefetch cache, 339
először be először ki (FIFO) algoritmus – First In First Out (FIFO) algorithm, 454
első generációs számítógépek – first-generation computers, 31
első illesztés algoritmus – first fit algorithm, 460
első menet – first pass, 742
eltolási rés – delay slot, 320
élvezérelt flip-flop – edge triggered flip-flop, 179
emelés, kód – hoisting, code, 330

emitter – emitter, 153
emitter csatolású logika (ECL) – Emitter-Coupled Logic (ECL), 155
emuláció – emulation, 37
endian – endian, 86–88
  kis – little, 86, 727
  nagy – big, 86
ENIAC – Electronic Numerical Integrator And Computer (ENIAC), 32, 41
ENIGMA – ENIGMA, 32
EPIC (explicit utasításszintű párhuzamosság) – EPIC (Explicitly Parallel Instruction Computing), 433–439
EPROM (törölhető PROM) – EPROM (Erasable PROM), 190
eredményjelző – scoreboard, 325
erőforrás-felosztás – partitioned resource sharing, 576
erőforrásréteg – resource layer, 668
erősen párhuzamos processzorok (MPP) – Massively Parallel Processors (MPP), 606, 635
értelmezés – interpretation, 18
értelmező – interpreter, 18, 70, 718
érvényesítés/engedélyezés – enable, 178
érvénytelenítő stratégia – invalidate strategy, 613
esemény – event, 516
Estridge, Philip – Estridge, Philip, 39
eszközmeghajtó – device driver, 492
eszközregisztersín – device register bus, 583
eszközszint – device level, 20, 153
Ethernet – Ethernet, 584
Explicit párhuzamos utasítású számítás (EPIC) – Explicitly Parallel Instruction Computing (EPIC), 433–439
explicit szerkesztés – explicit linking, 558
explicit utasításszintű párhuzamosság (EPIC) – Explicitly Parallel Instruction Computing (EPIC), 433–439
exponens – exponent, 709
extra szegmens – extra segment, 734

F

fa – tree, 633
fájl – file, 470–472
fájlallokációs tábla (FAT) – File Allocation Table (FAT), 505
fájlgyorsító-kezelő, Windows XP – file cache manager, Windows XP, 493
fájlindex – file index, 473
fájlleíró – file descriptor, 500, 501, 739
fájlrendszer – file system
  UNIX – UNIX, 488, 499–505
  Windows XP – Windows XP, 473, 492, 505–510
falat – nibble, 406
FAT (fájlallokációs tábla) – FAT (File Allocation Table), 505
fázis, csővezeték – stage, pipeline, 77
fázismoduláció – phase modulation, 133
fej nélküli munkaállomás – headless workstation, 645
fejállomás – headend, 137
fejléc – header, 233
fejléc (szektorban) – preabmle, 98
fejlett mikrokontroller sínarchitektúra (AMBA) – Advanced Microcontroller Bus Architecture (AMBA), 583
fejlett programozható megszakításvezérlő (APIC) – Advanced Programmable Interrupt Controller (APIC), 212
feladatzsák – task bag, 658
fél-duplex vonal – half-duplex line, 135
felhasználói mód – user mode, 355
félösszeadó – half adder, 171–172
feltételes végrehajtás – conditional execution, 436
feltételezett végrehajtás – speculative execution, 330
feltételkód – condition code, 359
feltételkód-regiszter – condition code register, 724
feltételváltozó – condition variable, 513
fém-oxid félvezető (MOS) – Metal Oxide Semiconductor (MOS), 155
fénykibocsátó dióda (LED) – Light Emitting Diode (LED), 127
festékalapú tinta – dye based ink, 132
festékbuborékos nyomtató – bubblejet, 129
festékszublimációs nyomtató – dye sublimation printer, 132
FIFO (először be először ki algoritmus) – FIFO (First In First Out algorithm), 454
finom szemcsézettségű többszálúság – fine-grained multithreading, 572
fizikai címtartomány – physical address space, 447
fizikai réteg – physical layer, 234
flag-regiszter – flag register, 724
flash memória – flash memory, 191
flip-flop/billenőkör – flip-flop, 179
Flynn-féle osztályozás – Flynn's taxonomy, 604
foglalt lap – reserved page, 497
fogyasztó – consumer, 478
fokszám – degree, 631
folyadékkristályos kijelző (LCD) – Liquid Crystal Display (LCD), 123
folyamatvezérlés – flow control, 235
fonál – fiber, 513, 514
fordítás – translation, 18
fordító – translator, 524
fordítóprogram – compiler, 23, 525
fordítótábla – translation table, 468
fordított lengyel jelölés – reverse Polish notation, 385
forgási késleltetés – rotational latency, 99

fork – fork, 660
formális paraméter – formal parameter, 536
forrásindex – source index, 724
forrásnyelv – source language, 524
forrásoperandus – source operand, 726
Forrester, Jay – Forrester, Jay, 34
FORTRAN – FORTRAN, 25
FORTRAN felügyelő rendszer (FMS) – FORTRAN Monitor System (FMS), 26
fő csővezeték, szállítószalag – U pipeline, 78
FPM (gyors lapkezelésű) memória – FPM (Fast Page Mode) memory, 189
frekvenciaeltolásos kódolás – frequency shift keying, 133
frekvenciamoduláció – frequency modulation, 133
frissítő stratégia – update strategy, 613
FSM (lásd véges állapotú gép) – FSM (see Finite State Machine)
full-duplex vonal – full duplex line, 135
funkcionális egység – functional unit, 79

G, GY

gamut – gamut, 131
gazdakönyvtár – host library, 559
GDI (grafikus eszköz interfész), Windows XP – GDI (Graphics Device Interface), Windows XP, 493
GDT (globális leírótábla) – GDT (Global Descriptor Table), 462
gépi nyelv – machine language, 17, 717
globális címke – global label, 743
globális leírótábla (GDT) – Global Descriptor Table (GDT), 462
Globe – Globe, 661
Goldstine, Herman – Goldstine, Herman, 33
google-klaszter – google cluster, 645–649
grafikus eszköz interfész (GDI), Windows XP – Graphics Device Interface (GDI), Windows XP, 493
grafikus felhasználói felület (GUI) – Graphical User Interface (GUI), 39, 489
Green Book (Zöld Könyv) – Green Book, 111
grid szerkezeti szint – fabric layer, 668
grid, háló – grid, 667, 668
GUI (grafikus felhasználói felület) – GUI (Graphical User Interface), 489
gyenge konzisztencia – weak consistency, 609
gyerek processzus – child process, 510
gyereketetési algoritmus – baby feeding algorithm, 452
gyors lapkezelésű (FPM) memória – Fast Page Mode (FPM) memory, 189
gyorsítósor – cache line, 94, 315, 612
gyorsítótár – cache memory, 54, 92, 313, 611
  2. szintű – level 2, 314
  direkt leképezésű – direct-mapped, 315
  egyesített – unified, 94
  érvénytelenítő stratégia – invalidate strategy, 613
  frissítő stratégia – update strategy, 613
  halmazkezelésű – set-associative, 317
  írásallokálás – write-allocate, 319, 613
  írásáteresztés – write-through, 318
  késleltetett írás – write-deferred, 319
  MESI – MESI, 614
  osztott, szétválasztott – split, 94, 313
  szétválasztott, osztott – split, 94, 313
  szimatoló – snooping, 612
  többszintű – multiple levels, 313
  visszaírás – write-back, 319
gyorsítótáras memóriaelérés (COMA) – Cache Only Memory Access (COMA), 605, 628, 687–688
gyorsítótárhiány – cache miss, 316
gyorsítótár-koherencia – cache coherency, 611
gyorsítótár-koherencia protokoll – cache coherence protocol, 612
  MESI – MESI, 614
gyorsítótár-koherens NUMA (CC-NUMA) – Cache Coherent NUMA (CC-NUMA), 619–628
gyorsítótár-konzisztencia – cache consistency, 611
gyorsítótár nélküli NUMA (NC-NUMA) – No Cashing NUMA (NC-NUMA), 620
gyorsítótár-találat – cache hit, 316
gyorsított grafikus port (AGP) – Accelerated Graphics Port (AGP), 125, 226
gyökérkönyvtár – root directory, 502
gyűrű – ring, 634

H

H regiszter – H register, 253, 262
hagyományos – legacy, 209
hajlékonylemez – diskette, 100
hajlékonylemez, floppy – floppy disk, 100
halftone képernyő-frekvencia – halftone screen frequency, 131
halftoning – halftoning, 130
halmazkezelésű gyorsítótár – set-associative cache, 317
háló – mesh, 634
hálózati interfész eszköz (NID) – Network Interface Device (NID), 136
hálózati processzor – network processor, 587–592
Hamming, Richard – Hamming, Richard, 44
Hamming-kód – Hamming code, 89
Hamming-távolság – Hamming distance, 88
hangolás, program – tuning, program, 527
Hanoi tornyai – towers of Hanoi, 415–421, 427–431
  Pentium – Pentium, 427–429

UltraSPARC – UltraSPARC, 429–431
hardver – hardware, 23
ekvivalenciája a szoftverrel – equivalence with software, 24
hardverabsztrakciós réteg – hardware abstraction layer, 491
hardver DSM – hardware DSM, 620
harmadik generációs számítógépek – third-generation computers, 36
háromsínes architektúra – three-bus architecture, 295
háromállapotú eszköz – tri-state device, 185
háromszorosan indirekt blokk – triple indirect block, 504
Harvard-architektúra – Harvard architecture, 94
hasznos adat – payload, 233
háttérmemória – secondary memory, 96
HDTV (nagy felbontású televízió) – HDTV (High Definition TeleVision), 597
héj (parancsértelmező, shell) – shell, 489
helyfüggetlen kód – position independent code, 555
helyi hálózat (LAN) – Local Area Network (LAN), 584
helyszámláló – location counter, 742
helyszínen programozható kapumátrix (FPGA) – Field Programmable Gate Array (FPGA), 587
hexadecimális szám – hexadecimal number, 698, 743
hibaarány – miss ratio, 93
hibajavító kód (ECC) – Error Correcting Code (ECC), 88
High Sierra – High Sierra, 111
hiperkocka – hypercube, 634
híváskapu – call gate, 466
Hoagland, Al – Hoagland, Al, 43
Hoff, Ted – Hoff, Ted, 52
hosszú – long, 722
hosszú szó – long word, 727
hozzáférést vezérlő jel – access token, 508
hozzárendelési idő – binding time, 553
http – http (HyperText Transfer Protocol), HyperText Transfer Protocol (http), 585
huzalozott VAGY – wired-OR, 196
hyperthreading – hyperthreading, 576

I, Í

IA–32 – IA-32, 359
IA–64 – IA-64, 431–439
EPIC modell – EPIC model, 433
köteg – bundle, 435
predikáció – predication, 436–438
utasításütemezés – instruction scheduling, 434–436
IAS gép – IAS machine, 33
IBM 1401 – IBM 1401, 35
IBM 360 – IBM 360, 37, 41, 583
IBM 701 – IBM 701, 34
IBM 704 – IBM 704, 34
IBM 7094 – IBM 7094, 35, 41
IBM 801 – IBM 801, 73
IBM Corporation – IBM Corporation, 34, 35, 37, 223
IBM PC – IBM PC, 39, 43
eredet – origin, 39
sín – bus, 223
IBM PS/2 – IBM PS/2, 223
IC (integrált áramkör) – IC (Integrated Circuit), 36, 163
i-csomópont (node) – i-node, 503
IDE (beépített eszközelektronika) – IDE (Integrated Drive Electronics), 101
időbeli lokalitás – temporal locality, 314
időosztásos rendszer – time sharing system, 27
időzített D-tároló – clocked D latch, 179
időzített SR-tároló – clocked SR latch, 178
IEEE lebegőpontos szabvány – IEEE floating-point standard, 711
IFU (utasításbetöltő egység) – IFU (Instruction Fetch Unit), 296–300
igazságtábla – truth table, 156
IJVM (egész aritmetikájú java virtuális gép) – IJVM (Integer Java Virtual Machine), 251–260, 266–275
adatút – data path, 252
időzítés – timing, 255
java kód – java code, 273
konstans mező – constant pool, 268
lokális változók mezője – local variable frame, 268
memóriamodell – memory model, 268
memóriaművelet – memory operation, 257
metódus mező – method area, 269
Mic–1 megvalósítás – Mic-1 implementation, 280
Mic–2 megvalósítás – Mic-2 implementation, 291
Mic–3 megvalósítás – Mic-3 implementation, 303
Mic–4 megvalósítás – Mic-4 implementation, 309
operandusverem – operand stack, 268
utasításkészlet – instruction set, 270
verem – stack, 266
iker – twin, 656
ILC (utasítás-helyszámláló) – ILC (Instuction Location Counter), 540
ILLIAC – ILLIAC, 32
ILLIAC IV – ILLIAC IV, 80, 604
implicit címzés – implied addressing, 729
implicit szerkesztés – implicit linking, 558
import könyvtár – import library, 558

indexelt címzés – indexed addressing, 383
indexelt színelőállítás – indexed color, 125, 595
indexregiszter – index register, 723
indirekt blokk – indirect block, 504
infix jelölés – infix notation, 385
információcsere amerikai szabványos kódrendszere, az (ASCII) – American Standard Code for Information Interchange (ASCII), 143
informatív – informative, 355
informatív információ, szabványban – informatíve information, in standard, 355
Integer Unit (IU) – Integer Unit (IU), 58
integrált áramkör (IC) – Integrated Circuit (IC), 36, 163
Intel 4004 – Intel 4004, 52
Intel 8008 – Intel 8008, 52
Intel 80286 – Intel 80286, 53
Intel 80386 – Intel 80386, 54
Intel 80486 – Intel 80486, 54
Intel 8051 – Intel 8051, 219–221, 365
adattípusok – data types, 371
címzés – addressing, 392–393
ISA-szint áttekintése – overview of the ISA, 365–368
mikroarchitektúra – microarchitecture, 343–345
történeti áttekintés – history, 59
utasításformátumok – instruction formats, 379
utasítások – instructions, 411
Intel 8080 – Intel 8080, 53
Intel 8086 – Intel 8086, 53
Intel 8088 – Intel 8088, 39, 40, 53, 359, 720
assembler – assembler, 741
címzés – addressing, 726–730
nyomkövető – tracer, 747
szimulátor – simulator, 747
utasításkészlet – instruction set, 730
Intel 8255A – Intel 8255A, 240
Intel Celeron – Intel Celeron, 55
Intel Corporation – Intel Corporation, 52
Intel Pentium (lásd még Pentium 4) – Intel Pentium (see also Pentium 4), 52, 54
Intel Xeon – Intel Xeon, 55
internetprotokoll (IP) – Internet Protocol (IP), 586
internetszolgáltató (ISP) – Internet Service Provider (ISP), 585
invertáló puffer – inverting buffer, 185
inverter, nem kapu – inverter, 154
inverziós gömb – inversion bubble, 154
IP (internetprotokoll) – IP (Internet Protocol), 586
IP (utasításmutató) – IP (Instruction Pointer), 720
ipari szabványos felépítés (ISA) – Industry Standard Architecture (ISA), 119, 223
IPC (processzusok közti kommunikáció) – IPC (Inter Process Communication), 489
irány-flag – direction flag, 730
írás utáni írás (WAW) függőség – Write After Write (WAW) dependence, 327
írásáteresztő – write through, 612
írásáteresztő gyorsítótár – write-through cache, 318
íráskori feltöltés módszere – write-allocate cache, 319, 613
íráskori másolás – copy on write, 496
írható CD (CD-R) – CD-recordable (CD-R), 112
ISA (ipari szabványos felépítés) – ISA (Industry Standard Architecture), 119, 223
ISA (utasításrendszer-architektúra) – ISA (Instruction Set Architecture), 22, 352–443, 717
ISA sín – ISA bus, 222
ISA-szint – ISA level, 22, 352–443
adattípusok – data types, 368–371
címzés – addressing, 380–394
IA–64 – IA-64, 431–439
utasítástípusok – instructions types, 394–414
utasításformátumok – instruction formats, 371–380
vezérlési folyamat – flow of control, 414–426
ismétléses vezérlés – loop control, 400–401
ISP (internetszolgáltató) – ISP (Internet Service Provider), 585
ISR (megszakításkezelő) – ISR (Interrupt Service Routine), 423
Itanium 2 – Itanium 2, 431–439
IU (Integer Unit) – IU (Integer Unit), 58

J

játékgép – game computer, 48
Java – Java, 23
java virtuális gép (JVM) – Java Virtual Machine (JVM), 251
jelenlét/hiány bit – present/absent bit, 450
jelzők regisztere – flags register, 358
jobbra igazított adat – right justified data, 395
Jobs, Steve – Jobs, Steve, 39
JOHNIAC – JOHNIAC, 32
Joy, Bill – Joy, Bill, 57
JPEG – Joint Photographic Experts Group (JPEG), JPEG (Joint Photographic Experts Group), 142
JTAG – Joint Test Action Group (JTAG), JTAG (Joint Test Action Group), 596
JVM (java virtuális gép) – JVM (Java Virtual Machine), 251
JVM (lásd még IJVM) – JVM (see also IJVM)

K

kábeles internet – cable Internet, Internet over cable, 137
kamera, digitális – camera, digital, 141
kapcsolat nélküli – off-line, 475
kapcsolati réteg – link layer, 235

kapcsolóalgebra – switching algebra, 155
kapcsolóhálózat – switching network, 617–619
kapu – gate, 20, 153
kapukésleltetés – gate delay, 164
kapuzójel – strobe, 178
karakterkód – character code, 143
katalógusalapú multiprocesszor – directory based multiprocessor, 621
katódsugárcső – Cathode Ray Tube (CRT), 121
képpont, pixel – pixel, 125
kerekítés – rounding, 710
keresés – seek, 98
kérésre lapozás – demand paging, 452
keret – frame, 110
keretmutató – frame pointer, 362
kernel (rendszermag) – kernel, 491
kernelmód – kernel mode, 355
késleltetés – latency, 77
  forgási – rotational, 99
késleltetési idő elrejtés – latency hiding, 666
késleltetett írás – write-deferred cache, 298
késleltetett írású protokoll – write-back protocol, 319, 614
készen kapható termék – Commodity Off The Shelf (COTS), 51
két érintkezősoros memóriamodul (DIMM) – Dual Inline Memory Module (DIMM), 95
kétbites moduláció – dibit modulation, 134
kétlábsoros tokozás (DIP) – Dual Inline Package (DIP), 163
kétmenetes fordító – two-pass translator, 539
kétszeres sebességű memória (DDR) – Double Data Rate memory (DDR), 190
kétszeresen indirekt blokk – double indirect block, 504
kettes komplemens aritmetika – two's complement arithmetic, 703
kettévágott sávszélesség – bisection bandwidth, 633
kettős tórusz – double torus, 634
kezdeményező, PCI sín – initiator, PCI bus, 227
kezelő – handle, 494
Khosla, Vinod – Khosla, Vinod, 57
kibővített kiszolgálóvezérlő interfész (EHCI) – Enhanced Host Controller Interface (EHCI), 240
kiegészítő átvitel-flag – auxiliary carry flag, 734
Kildall, Gary – Kildall, Gary, 39
kimenő csomagokon végzett feldolgozás – egress processing, 589
kis endián – little endian, 86, 727
kis számítógéprendszerek interfésze (SCSI) – Small Computer System Interface (SCSI), 103–104
kisméretű kétsoros érintkezős memóriamodul (SO-DIMM) – Small Outline DIMM (SO-DIMM), 95
kiszolgáló farm – server farm, 51
kiszolgáló – server, 50
kiterjesztett adatkimenetű memória (EDO) – Extended Data Output (EDO) memory, 189
kiterjesztett IDE (EIDE) – Extended IDE (EIDE), 101
kiterjesztő kód – escape code, 377
klaszter – cluster, 50, 606
  google – google, 645–649
  munkaállomás – workstation, 50–51, 589, 644
  NTFS – NTFS, 508
klaszterszámítógép – cluster computer, 644
klón – clone, 39
kocka – cube, 634
kódgenerálás – code generation, 742
kódlap – code page, 145
kódpozíció – code point, 146
kódszegmens – code segment, 720
kódszó – codeword, 88
kollektív szolgáltató réteg – collective layer, 669
kollektor – collector, 153
kombinációs áramkör – combinational circuit, 165–170
  dekódoló – decoder, 167
  multiplexer – multiplexer, 165–167
  összehasonlító – comparator, 168
kommunikátor – communicator, 651
konstans mező – constant pool, 268
konstans mező mutatója (CPP) – Constant Pool Pointer (CPP), 253, 262, 269, 280
konverzió számrendszerek között – conversion between radices, 700
korutin/társrutin – coroutine, 414, 421, 422
kölcsönös kizárás – mutual exclusion, 513
könyvtár – directory, 475–476
körkörös puffer – buffer, circular, 478
környezet – context, 467
környezeti alrendszer, Windows XP – environmental subsystem, Windows XP, 493
köteg – bundle, 435
kötegelt rendszer – batch system, 27
kötet tartalomjegyzék (VTOC) – Volume Table Of Contents (VTOC), 113
kövér fa – fat tree, 634
közeli hívás – near call, 737
közeli ugrás – near jump, 735
közös memóriás (lásd multiprocesszor) – shared memory (see multiprocessor)
közös memóriás multiprocesszor – shared memory multiprocessor, 598–629
központi csomópont, USB – root hub, USB, 237
központi feldolgozóegység (CPU) – Central Processing Unit (CPU), 34, 66
közvetlen címzés – immediate addressing, 380
közvetlen fájl – immediate file, 509
közvetlen memóriaelérés (DMA) – Direct Memory Access (DMA), 119, 404

közvetlen operandus – immediate operand, 380
kriptoprocesszor – Cryptoprocessor, 597
kritikus szakasz – critical section, 516
kulcs – key, 473
külső elaprózódás – external fragmentation, 460
külső hivatkozás – external reference, 551
külső szimbólum – external symbol, 551
küszöbölt (erőforrás-) megosztás – threshold sharing, 577

L, LY

L1 BTB – L1 BTB, 335
lábkiosztás – pinout, 191
LAN (helyi hálózat) – LAN (Local Area Network), 584
láncolás – daisy chaining, 203
lap – page, 447
  egyeztetett – committed, 497
  foglalt – reserved, 497
lapcserélő algoritmus – page replacement algorithm, 453, 454
  FIFO – FIFO, 454
  LRU – LRU, 453
lapcserélő eljárás – page replacement policy, 453
lapfelügyelő – page scanner, 620
laphiány – page fault, 452
lapka – chip, 163
  CPU – CPU, 192
lapkaszintű többszálúság – on-chip multithreading, 572–578
lapkaszintű párhuzamosság – on-chip parallelism, 564–583
lapkeret – page frame, 448
lapkezelő segédpuffer (TLB) – Translation Lookaside Buffer (TLB), 340, 467
lapkezelő tárolópuffer (TSB) – Translation Storage Buffer (TSB), 468
lapkönyvtár – page directory, 464
lapos megjelenítő – flat panel display, 122
lapozás – paging, 447
  kérésre – demand, 452
  megvalósítása – implementation, 448
  transzparens – transparent, 448
laptábla – page map, 447
láthatatlan számítógép – invisible computer, 41
Latin-1 – Latin-1, 145
lazán kapcsolt – loosely coupled, 564
LBA (logikai blokk címzés) – LBA (Logical Block Addressing), 102
LCD (folyadékkristályos kijelző) – LCD (Liquid Crystal Display), 123
LDT (lokális leírótábla) – LDT (Local Descriptor Table), 462
lebegőpontos szám – floating-point number, 708–716
LED (fénykibocsátó dióda) – LED (Light Emitting Diode), 127
lefoglaló/átnevező egység – allocation/renaming unit, 336
legjobb illesztés algoritmus – best fit algorithm, 460
legrégebben használt (LRU) algoritmus – Least Recently Used (LRU) algorithm, 318, 453
Leibniz, Gottfried von – Leibniz, Gottfried von, 30
lemez – disc, 96
  CD-ROM – CD-ROM, 108
  DVD – DVD, 115
  hajlékony – floppy, 100
  IDE – IDE, 101
  mágnes – magnetic, 97–108
  optikai – optical, 108–117
  RAID – RAID, 104–108
  SCSI – SCSI, 103–104
  vezérlő – controller, 100
  Winchester – Winchester, 98
lengyel jelölés – Polish notion, 385–388
léptető – shifter, 171
levélbedobó nyílás – mailslot, 515
lézernyomtató – laser printer, 129
Linda – Linda, 656
lineáris cím – linear address, 463
link (kötés, kapocs) – link, 502
literál – literal, 542
logikai blokk címzés (LBA) – Logical Block Addressing (LBA), 102
logikai rekord – logical record, 471
lokális ciklus/hurok – local loop, 135
lokális címke – local label, 743
lokális leírótábla (LDT) – Local Descriptor Table (LDT), 462
lokális változó (LV) mutató – Local Variable (LV) pointer, 253, 262, 266, 280
lokális változók mezője – local variable frame, 266, 268
lokalitási elv – locality principle, 93, 452
Lovelace, Ada Augusta – Lovelace, Ada Augusta, 30
lpi (vonal per inch) – lpi (Lines Per Inch), 131
LRU (legrégebben használt) algoritmus – LRU (Least Recently Used) algorithm, 453
LV (lokális változó) mutató – LV (Local Variable) pointer, 253, 262, 266, 280
lyukacsosodás – checkerboarding, 460

M

Macintosh, Apple – Macintosh, Apple, 39
mag – core, 579
magas szintű nyelv – high level language, 23
  összehasonlítása az assembly nyelvvel – compared to assembly language, 526
mágneslemez – magnetic disk, 97–108
makró – macro, 534
  definíció – definition, 534

formális paraméter – formal parameter, 536
hívás – call, 535
kifejtés – expansion, 535
megvalósítás – implementation, 537
paraméter – parameter, 536
makroarchitektúra – macroarchitecture, 266
MAL (mikro assembly nyelv) – MAL (Micro Assembly Language), 276
MANIAC – MANIAC, 32
mantissza – mantissa, 709
MAR (memóriacím-regiszter) – MAR (Memory Address Register), 257
Mark I – Mark I, 31
MASM – MASM, 528
második csővezeték, szállítószalag – V pipeline, 78
második generációs számítógépek – second-generation computers, 34
második menet – second pass, 742
maszk – mask, 395
mátrixnyomtató – matrix printer, 127
Mauchley, John – Mauchley, John, 32
MBR (memóriapuffer-regiszter) – MBR (Memory Buffer Register), 253, 259, 262
McNealy, Scott – McNealy, Scott, 57
MCS–51 család – MCS-51 family, 59
MDR (memóriaadat-regiszter) – MDR (Memory Data Register), 253, 257, 259, 262
médiaprocesszor – media proeessor, 592–597
meghatározó rész – significand, 713
megosztott könyvtár – shared library, 559
megszakítás – interrupt, 119, 423–426
átlátszó – transperent, 424
pontatlan – impreeise, 327
pontos – precise, 327
megszakításkezelő – interrupt handler, 119
megszakításkezelő (ISR) – Interrupt Service Routine (ISR), 423, 426
megszakításvektor – interrupt vector, 207
8088 – 8088 709
memória – memory, 83, 176
8088 – 8088, 725–730
asszociatív – associative, 461
DDR – DDR, 190
dinamikus RAM – dynamic RAM, 189
DRAM – DRAM, 189
EDO – EDO, 189
EEPROM – EEPROM, 190
EPROM – EPROM, 190
flash – flash, 191
FPM – FPM, 189
frissítés – refresh, 189
gyorsítótár – cache, 92, 313, 611–614
háttérmemória – secondary, 96
hierarchia – hierarchy, 96
központi – primary, 83
lapka – chip, 186
memóriacím – address, 84
modell – model, 356–358
PROM – PROM, 190
ROM – ROM, 190
SDRAM – SDRAM, 189
SRAM – SRAM, 189
statikus RAM – static RAM, 189
szervezés – organization, 183
szervezés, Intel 8088 – organization, Intel 8088, 725
térkép – map, 447
tokozás – packaging, 95
véletlen elérésű (RAM) – Random Access (RAM), 60, 188
virtuális – virtual, 445–469, 493, 495–498
vonzásmemória – attraction, 628
memóriapuffer-regiszter (MBR) – Memory Buffer Register (MBR), 253, 259, 262
memóriaadat-regiszter (MDR) – Memory Data Register (MDR), 253, 257, 259, 262
memóriacím-regiszter (MAR) – Memory Address Register (MAR), 257
memóriakezelő egység (MMU) – Memory Management Unit (MMU), 449
memóriamásolat – core dump, 25
memóriára leképezett B/K – memory mapped I/O, 242
memóriaszemantika – consistency model (see Memory semantics), memory semantics, 357, 606–610
elengedési konzisztencia – release consistency, 610
gyenge konzisztencia – weak consistency, 609
processzorkonzisztencia – processor consistency, 608
soros konzisztencia – sequential consistency, 607
szigorú konzisztencia – strick consistency, 607
menet, assembler – pass, assembler, 539
merevlemez – Winchester disk, 98
mérföldkövek a számítógépek fejlődésében – milestones in computer architecture, 28–41
mérgezésbit – poison bit, 332
merőleges rögzítés – perpendicular recording, 98
mértékegységek – metric units, 61
MESI gyorsítótár-koherencia protokoll – MESI cache coherence protocol, 614
mester, sín – master, bus, 195
mesterfájltábla (MFT) – Master File Table (MFT), 509
metódus – method, 400
metódus mező – method area, 269
MFT (mesterfájltábla) – MFT (Master File Table), 509
Mic–1 – Mic-1, 261, 280
adatút – data path, 262
megvalósítás – implementation, 280

Mic–2 – Mic-2, 300
adatút – data path, 301
megvalósítás – implementation, 300
Mic–3 – Mic-3, 305
adatút – data path, 304
csővezeték – pipeline, 303–309
megvalósítás – implementation, 303–309
Mic–4 – Mic-4, 309
Adatút – Data Path, 309
megvalósítás – implementation, 309
mickey – mickey, 127
Microsoft Corporation – Microsoft Corporation, 40, 490, 473
mikro assembly nyelv (MAL) – Micro Assembly Language (MAL), 276
mikroarchitektúra – microarchitecture, 251
8051 – 8051, 343
Pentium 4 – Pentium 4, 332
UltraSPARC – UltraSPARC, 338
mikroarchitektúra szintje – microarchitecture level, 21, 251
elágazásjövendölés – branch prediction, 319–324
gyorsítótár – cache memory, 313–319
IJVM-példa – IJVM example, 251–292
példa – example, 332–347
tervezés – design, 291–312
mikrolépés – microstep, 306
mikromeghajtó – microdrive, 143
mikroművelet – micro-operation, 310
mikroprogram – microprogram, 21, 24, 721
mikroprogramozás – microprogramming, 24
történeti áttekintés – history, 24, 27
mikroutasítás – microinstruction, 72, 258
jelölés – notation, 275
mikroutasítás-regiszter (MIR) – MicroInstruction Register (MIR), 261
mikroutasítás-számláló (MPC) – MicroProgram Counter (MPC), 261
mikroutasítás-vezérlés – microinstruction control, 261
mikrovezérlő – microcontroller, 46
MIMD (többszörös utasításáram többszörös adatáram) – MIMD (Multiple Instruction stream Multiple Data stream), 604–605
mindenütt jelenlévő számítástechnika – pervasive computing, ubiquitous computing, 41
minta – template, 657
MIPS (betűszó, millió utasítás másodpercenként) – MIPS (acronym, Millions of Instructions Per Second), 75
MIPS (lapka) – MIPS (chip), 73
MIR (mikroutasítás-regiszter) – MIR (MicroInstruction Register), 261
MISD (többszörös utasításáram egyszeres adatáram) – MISD (Multiple Instruction stream Single Data stream), 604–605
MMU (memóriakezelő egység) – MMU (Memory Management Unit), 449
mnemonik – mnemonic, 718, 741
modem – modem, 120, 133
moduláció – modulation, 133
amplitúdó – amplitude, 133
fázis – phase, 133
frekvencia – frequency, 133
monadikus műveletek – monadic instructions, 396
Moore, Gordon – Moore, Gordon, 42
Moore-szabály – Moore's law, 42
MOS (fém-oxid félvezető) – MOS (Metal Oxide Semiconductor), 155
Motif – Motif, 489
Motorola 68000 – Motorola 68000, 72
MPC (mikroutasítás-számláló) – MPC (MicroProgram Counter), 261
MPEG – Motion Picture Expert Group (MPEG), 581
MPI (üzenetátadás interfész) – MPI (Message-Passing Interface), 651
MPP (erősen párhuzamos processzor) – MPP (Massively Parallel Processor), 635, 606
MS-DOS – MS-DOS, 40
MULTICS – MULTICS (MULTiplexed Information and Computing Service (MULTICS), 461, 466, 556
multimédiás kiegészítések (MMX) – MultiMedia eXtension (MMX), 54
multiplexeit sín – multiplexed bus, 197, 214
multiplexer – multiplexer, 165
multiprocesszor – multiprocessor, 82, 340, 578, 583, 598–629
crossbar-alapú – crossbar-based, 616–619
egylapkás – on chip, 578–583
kapcsolóhálózat – switching network, 617–619
sínalapú – bus-based, 610–616
multiprogramozás – multiprogramming, 37
multiszámítógép – multicomputer, 83, 600, 681
BlueGene/L – BlueGene/L, 618, 627, 635–640
google-klaszter – google cluster, 645–649
MPP – MPP, 606, 635
Red Storm – Red Storm, 622–643
szoftver – software, 650
teljesítménye – performance, 660–667
ütemezése – scheduling, 652–653
munkaállomások hálózata (NOW) – Network of Workstations (NOW), 606
munkaállomások klasztere (COW) – Cluster Of Workstations (COW), 50–51, 606, 644
munkahalmaz – working set, 452
munkakönyvtár – working directory, 502
mutató – pointer, 381
mutatóregiszter – pointer register, 723
mutex – mutex, 513, 515
művelet – operation, 567, 659

műveleti kód – operation code, 252
műveletikód-kiterjesztés – expanding opcode, 374–376
művkód – opcode, 252
Myhrvold, Nathan – Myhrvold, Nathan, 43

N, NY
nagy endian gép – big endian machine, 86
nagy felbontású televízió (HDTV) – High Definition TeleVision (HDTV), 597
nagyon hosszú utasításszavú (VLIW) – Very Long Instruction Word (VLIW), 565–567
nagyon magas fokú integráltság (VLSI) – Very Large Scale Integration (VLSI), 38, 490
nagyon hosszú utasításszavú (VLIW) – VLIW (Very Long Instruction Word), 565–567
nagyszámítógép – mainframe, 51
nagyterületű hálózat (WAN) – Wide Area Network (WAN), 584
NaN (nem szám) – NaN (Not a Number), 715
Nathan első szoftvertörvénye – Nathan's first law of software, 43
NC-NUMA (gyorsítótár nélküli NUMA) – NC-NUMA (No Cashing NUMA), 620, 603
negált jel – negated signal, 187
negatív logika – negative logic, 162
negyedik generációs számítógépek – fourth-generation computers, 38
nem blokkoló hálózat – nonblocking network, 617
nem blokkoló üzenetküldés – nonblocking message passing, 650
nem egységes memóriaelérés (NUMA) – NonUniform Memory Access (NUMA), 605, 619–628
nem felejtő memória – nonvolatile memory, 190
nem invertáló puffer – noninverting buffer, 185
nem szám (NaN) – Not a Number (NaN), 715
NetBurst mikroarchitektúra – NetBurst microarchitecture, 332
Neumann János – von Neumann, John, 33
Neumann-gép – von Neumann machine, 33
Nexperia – Nexperia, 592
NID (hálózati interfész eszköz) – NID (Network Interface Device), 136
NORMA (távoli memória elérése nélküli) – NORMA (NO Remote Memory Access), 606
normalizálatlan számok – enormalized number, 714
normalizált szám – ormalized number, 710
normatív információ, szabványban – normative information, in standard, 355
NOW (munkaállomások hálózata) – NOW (Network of Workstations), 606
NT-fájlrendszer (NTFS) – NT File System (NTFS), 505
NTFS (NT-fájlrendszer) – NTFS (NT File System), 505
nulladik generációs számítógépek – zeroth-generation computers, 30
NUMA (nem egységes memóriaelérés) – NUMA (NonUniform Memory Access), 605, 619–628
n-utas halmazkezelésű gyorsítótár – n-way set-associative cache, 317
nyelv – language, 17
nyílt grid szolgáltatások architektúrája (OGSA) – Open Grid Services Architecture (OGSA), 669
nyílt gyűjtő/kollektor – open collector, 195
nyílt kiszolgálóvezérlő interfész (OHCI) – Open Host Controller Interface (OHCI), 240
nyílt magprotokoll nemzetközi társaság (OCP-IP) – Open Core Protocol-International Partnership (OCP-IP), 583
nyilvános kulcsú titkosítás – public key cryptography, 598
nyomkövető – tracer, 718
  Intel 8088 – Intel 8088, 747
nyomkövető BTB – trace BTB, 336
nyomkövető gyorsítótár – trace cache, 334
nyomtató – printer, 127–133
  festékszublimációs – dye sublimation, 132
  lézer – laser, 129
  monokróm – monochrom, 127
  szilárd tintás – solid ink, 132
  színes – color, 131
  viasz – wax, 132
nyomtatómű – print engine, 130
nyugtázó csomag – acknowledgement packet, 235

O
objektumkezelő Windows XP – object manager Windows XP, 492
OCP-IP (nyílt magprotokoll nemzetközi társaság) – OCP-IP (Open Core Protocol-International Partnership), 583
OGSA (nyílt grid szolgáltatások architektúrája) – OGSA (Open Grid Services Architecture), 669
OHCI (nyílt kiszolgálóvezérlő interfész) – OHCI (Open Host Controller Interface), 240
oktális szám – octal number, 698, 743
olcsó lemezek redundáns tömbje (RAID) – Redundant Array of Inexpensive Disks (RAID), 104
olvasás utáni írás (WAR) függőség – Write After Read (WAR) dependence, 327
olvasás/írás mutató – read/write pointer, 739
omega hálózat – omega network, 618
omnibus, PDP-8 – omnibus, PDP-8
on-line – on-line, 475
OPC (régi utasításszámláló) – OPC (Old Program Counter), 253, 262, 280
operációs rendszer – operating system, 25, 444, 461

CP/M – CP/M, 39
MS-DOS – MS-DOS, 40
OS/2 – OS/2, 40
történeti áttekintés – history, 25
UNIX – UNIX, 486–489, 495–496, 499–505, 510–513
Windows – Windows, 40
Windows XP – Windows XP, 489–495, 496–498, 505–510, 513–516
operációs rendszer gép (OSM) – Operating System Machine (OSM), 444–523
operációs rendszer gép (OSM) szintje – operating system machine (OSM) level, 22, 444–523
operációs rendszeri makro(utasítás) – operating system macro, 27
operandusverem – operand stack, 267, 268
optikai lemez – optical disk, 108–117
óra – clock, 175
Orange Book (Narancssárga Könyv) – Orange Book, 113
Orca – Orca, 658
OS/2 – OS/2, 40
Osborne I – Osborne 1, 40
OSM (operációs rendszer gép) – OSM (Operating System Machine), 444
osztott memóriájú rendszer – distributed memory system, 600

Ö
önmódosító program – self-modifying program, 382
összeadó – adder, 171–173
  átvitelkiválasztó – carry select, 173
  átvitelt tovább terjesztő – ripple carry, 173
  fél – half, 171–172
  teljes – full, 171–172
összehasonlító – comparator, 168
összehasonlító és elágazó utasítások – comparision and branch instructions, 398
összekötő hálózatok – interconnection networks, 631–634
  kettévágott sávszélesség – bisection bandwidth, 633
  topológia – topology, 632
összesített sávszélesség – aggregate bandwidth, 662
összetett utasításkészletű számítógép (CISC) – Complex Instruction Set Computer (CISC), 73
ötödik generációs számítógépek, Japán – fifth generation project, Japanese, 40

P
parancsértelmező (shell, héj) – shell, 489
párhuzamos be/kimenet (PIO) – Parallel Input/Output (PIO), 241
párhuzamos számítógépek teljesítménye – performance of parallel computers
  Amdahl törvénye – Amdahl's law, 664
  elérése – achieving, 665–667
  hardvermértékek – hardware metrics, 661–663
  növelés – improving, 312–332, 661–667
  szoftvermértékek – software metrics, 663–665
párhuzamos számítógép-architektúra – parallel computer architecture, 563–673
  hálózati processzor – network processor, 568–575
  lapkaszintű – on-chip parallelism, 564–583
  médiaprocesszor – media processor, 592–597
  multiprocesszor – multiprocessor, 598–600
  multiszámítógép – multicomputer, 600–603
  osztályozás – taxonomy, 603–606
  társprocesszor (koprocesszor) – coprocessor, 583–598
  többszálúság – multithreading, 572–578
  utasításszintű párhuzamosság – instruction level parallelism, 565–572
párhuzamosság – parallelism
  processzorszintű – processor-level, 80
  utasításszintű – instruction-level, 75
paritásbit – parity bit, 88
paritás-flag – parity flag, 734
Pascal, Blaise – Pascal, Blaise, 30
passzív mátrixmegjelenítő – passive matrix display, 124
PC (programszámláló, utasításszámláló) – PC (Program Counter), 252–262, 720
PCI (Peripheral Component Interconnect) – PCI (Peripheral Component Interconnect), 120, 223–231
PCI Express sín – PCI Express bus, 232
PCI sín – PCI (Peripheral Component Interconnect), 120, 223–231
  jel – signal, 228
  tranzakció – transaction, 230
PDP–1 – PDP-1, 35
PDP–11 – PDP-11, 38
PDP–8 – PDP-8, 35
példaprogramok – example programs
  Intel 8088 – Intel 8088, 752
Pentium 4 – Pentium 4, 208, 359, 486
  adattípusok – data types, 370
  bevezetés – introduction, 37–42
  címzés – addressing, 390–392
  fénykép – photograph, 55
  ISA-szintjének áttekintése – overview of ISA, 359–362
  lábkiosztás – pinout, 210
  mikroarchitektúra – microarchitecture, 332
  problémái – problems, 431–432
  sín – bus, 212
  történeti áttekintés – history, 52

utasításformátumok – instruction formats, 376–378
utasítások – instructions, 405–408
virtuális memória – virtual memory, 462
perifériális sín – peripherial bus, 582
Peripheral Component Interconnect (PCI) – Peripheral Component Interconnect (PCI), 120
pigmentalapú tinta – pigment based ink, 132
PIO (párhuzamos be/kimenet) – PIO (Parallel Input/Output), 241
pixel, képpont – pixel, 125
PLA (programozható logikai tömb) – PLA (Programmable Logic Array), 169
Playstation 2 – Playstation 2, 48
pont – dot, 742
pont per inch (dpi) – Dots Per Inch (dpi), 129
pontos megszakítás – precise interrupt, 327
POSIX – Portable Operating System-IX (POSIX), 487
postfix jelölés – posxtfix notation, 385
POTS (egyszerű régi telefonszolgáltatás) – POTS (Plain Old Telephone Service), 135
pozitív logika – positive logic, 162
PPE (protokoll-/programozott/csomag-feldolgozó egység) – PPE (Protocol/Programmed/Packet Processing Engine), 588
predikáció/megalapozás – predication, 436
prefix bájt – prefix byte, 287, 377, 406
processzor – processor, 66
processzor áteresztőképessége – processor bandwidth, 77
processzorkonzisztencia – processor consistency, 608
processzorsín – processor bus, 582
processzorszintű párhuzamosság – processor level parallelism, 80–83
processzus – process, 445, 477
processzus- és szálkezelő – process and thread manager
Windows XP – Windows XP, 493
processzusszinkronizálás – process synchronization, 482, 489
processzuskezelés – process management
UNIX – UNIX, 510
Windows XP – Windows XP, 513
processzusok közti kommunikáció (IPC) – Inter Process Communication (IPC), 489
program – program, 17
programállapotszó (PSW) – Program Status Word (PSW), 344, 358, 466
programozható logikai tömb (PLA) – Programmable Logic Array (PLA), 169
programozható ROM (PROM) – Programmable ROM (PROM), 190
programozott B/K – programmed I/O, 402
programszámláló (PC) – Program Counter (PC), 252–262, 720
progresszív pásztázás – progressive scan, 596
PROM (programozható ROM) – PROM (Programmable ROM), 190
protokoll – protocol, 194, 234, 585
protokoll-/programozott/csomag- feldolgozó egység (PPE) – Protocol/Programmed/Packet Processing Engine (PPE), 588
PSW (programállapotszó) – PSW (Program Status Word), 358, 466
pszeudoutasítás – pseudoinstruction, 531, 718, 744
pthread – pthread, 512
pufferelt üzenetátadás – buffered message passing, 650
PVM (virtuális párhuzamos számítógép) – PVM (Parallel Virtual Machine), 651

R

rács – grid, 634
rádiófrekvenciás azonosító (RFID) – Radio Frequency IDentification (RFID), 44
RAID (olcsó lemezek redundáns tömbje) – RAID (Redundant Array of Inexpensive Disks), 104
RAM (véletlen elérésű memória) – RAM (Random Access Memory), 60, 188
dinamikus (DRAM) – dynamic (DRAM), 189
statikus (SRAM) – static (SRAM), 189
szinkron DRAM (SDRAM) – synchronous DRAM (SDRAM), 189
raszteres pásztázó – raster scan, 122
RAW függőség – RAW dependence, 307
Red Book (Vörös Könyv) – Red Book, 108
Red Storm – Red Storm, 640
Reed–Solomon-kód – Reed-Solomon code, 98
régi utasításszámláló (OPC) – Old Program Counter (OPC), 253, 262, 280
regiszter – register, 21, 720
ablak – register window, 363
adat – data, 722
átnevezés – register renaming, 329
akkumulátor – accumulator, 722
bázis – base, 722
bázismutató – base pointer, 724
célindex – destination index, 724
címzés – addressing, 381
feltételkód – condition code, 724
flag – flag, 724
forrásindex – source index, 724
index – index, 723
indexelt – displacement, 728–727
indirekt címzés – indirect addressing, 381–382
mód – mode, 381
mutató – pointer, 723
programszámláló (PC) – program counter (PC), 252–262, 720
számláló – counter, 722

utasításmutató (IP) – instruction pointer (IP), 720, 724
utasításszámláló (PC), 66, 720
rekesz, memória – cell, memory, 84
rekurzió – recursion, 400
rekurzív eljárás – recursive procedure, 415
relatív hiba – relative error, 710
relatív útvonal – relative path, 502
rendszeradminisztrátor – superuser, 503
rendszerhívás – supervisor call, 27
rendszerinterfész – system interface, 494
rendszerprogramozó – system programmer, 22
rendszersín – system bus, 194
rendszerszolgáltatások, Windows XP – system services, Windows XP493
rés – bucket, 547
részleges címdekódolás – partial address decoding, 244
réteg – layer, 19
RFID lapka – RFID chip, 44
RISC (csökkentett utasításkészletű számítógép) – RISC (Reduced Instruction Set Computer), 72
RISC és CISC – RISC versus CISC, 72
RISC-tervezési elvek – RISC design principles
ROM (Read Only Memory) – ROM (Read Only Memory), 190

S, SZ

SATA (soros ATA) – SATA (Serial ATA), 102
sáv – track, 98
sáv/pálya – lane, 234
SCSI (kis számítógéprendszerek interfésze) – SCSI (Small Computer System Interface), 103–104
SDRAM (szinkron DRAM) – SDRAM (Synchronous DRAM), 189
SRAM (statikus RAM) – SRAM (Static RAM), 189
Seastar – Seastar, 641
shell (parancsértelmező, héj) – shell, 489
Shockley, William – Shockley, William, 34
SID (biztonsági azonosító) – SID (Security ID, 508
SIMD számítógép – SIMD (Single Instruction-stream Multiple Data stream) computer, 81
SIMM (egy érintkezősoros memóriamodul) – SIMM (Single Inline Memory Module), 95
sín – bus, 35, 66, 117–120, 194, 207
adóvevő – tranceiver, 195
AGP – AGP, 226
aszimmetria – skew, 197
aszinkron – asynchronous, 201–202
blokkátvitel – block transfer, 205
ciklus – cycle, 198
EISA – EISA, 223
használati engedély – grant, 202, 227
IBM PC – IBM PC, 222
időzítés – clocking, 198
időzítés – timing, 198–202
ISA – ISA, 222
kézfogás – handshaking, 202
mester – master, 195, 201, 213, 218, 230
multipixelt – multiplxed, 197
műveletek – operation, 205–207
PCI – PCI, 223–231
PCI Express – PCI Express, 232
Pentium 4 – Pentium 4, 212
protokoll – protocol, 194
sínütemezés – arbitration, 119, 202–205
sínütemezés, PCI sín – arbitration, PCI, 227
szélesség – width, 196–197
szinkron – synchronous, 198
szolga – slave, 195
univerzális soros sín (USB) – USB, 236–240
ütemező – arbiter, 119
vevő – receiver, 195
vezérlő – driver, 195
sínütemezés, PCI sín – arbitration, PCI bus, 227
skála, index, bázis (SIB) – Scale, Index, Base (SIB), 377, 391
skálázható – scalable, 601, 665
skálázható processzorarchitektúra (SPARC) – Scalable Processor ARChitecture (SPARC), 57–59
SLED (egyetlen nagy, drága lemez) – SLED (Single Large Expensive Disk), 104
small modell, 8088 – small model, 8088, 746
SMP (szimmetrikus multiprocesszor) – SMP (Symmetric MultiProcesszor), 600, 610–619
socket (csatlakozási pont) – socket, 487, 515
sokoldalú digitális lemez (DVD) – Digital Versatile Disc (DVD), 115
Solaris – Solaris, 487, 511
sorba állító – sequencer, 261
sorba állító egység – queueing unit, 310
sorduplázó technika – deinterlacing, 596
soros ATA (SATA) – Serial ATA (SATA), 102
soros konzisztencia – sequential consistency, 607
sorrendtől eltérő végrehajtás – out-of-order execution, 324–329
SP (veremmutató) – SP (Stack Pointer), 253, 262, 266, 280, 723
SPARC, skálázható processzorarchitektúra – Scalable Processor ARChitecture (SPARC), 57–59
spekulatív betöltés – speculative load, 438
SR-tároló – SR latch, 177
SRAM (statikus RAM) – SRAM (Static RAM), 189
SSE – SSE, 54
statikus RAM (SRAM) – Static RAM (SRAM), 189
Stibbitz, George – Stibbitz, George, 31
stream (folyam) – stream, 488

Streaming SIMD-kiegészítések (SSE) – Streaming SIMD Extensions (SSE), 54
struktúrált számítógép-felépítés – structured computer organization, 17
Sun Fire E25K – Sun Fire E25K, 617, 624–628
Sun Microsystem – Sun Microsystem, 57–59
szabad lap – free page, 497
szabad lista – free list, 474
szabványos bemeneti csatorna – standard input, 501
szabványos hibacsatorna – standard error, 502
szabványos kimeneti csatorna – standard output, 502
szál – thread, 479, 511
számítógép-architektúra – computer architecture, 23
  mérföldkövek – milestones, 28–41
számítóközpont – computer center, 38
számlálóregiszter – counter register, 722
számrendszerek – radix number system, 698
szavas regiszter – word register, 727
szavas utasítás – word instruction, 727
szegmens – segment, 725–726
szegmensregiszter-csoport – segment register group, 724
szegmensregiszter-választó – segment override, 740
  prefix – prefix, 740
szegmentálás – segmentation, 456 462
  legjobb illesztés algoritmus – best fit algorithm , 460
  első illesztés algoritmus – first fit algorithm, 460
  megvalósítása – implementation, 459
szekció, CD-ROM – session, CD-ROM, 113
szektor, lemez – sector, disk, 98, 110
szektorrés – intersector gap, 98
szélessávú – broadband, 135
szelet – shard, 646
szemafor – semaphore, 482, 511
személyi számítógép (PC) – personal computer (PC), 49
szerkesztés – linking, 547–551, 741
  dinamikus – dynamic, 555
  feladatai – task performed, 548
  hozzárendelési idő – binding time, 552
  MULTICS – MULTICS, 555
  tárgymodul – object modul, 551
  UNIX – UNIX, 559
  Windows – Windows, 557
szerkesztő – linker, 547–551, 741
szerkesztő-betöltő – linking loader, 547
szerkesztő-editor – linkage editor, 547
szerver – server, 50
szétválasztó – splitter, 136
szétválasztott gyorsítótár – split cache, 94, 313
szignifikáns – significand, 713
szigorú konzisztencia – strict consistency, 607
szilárd tintás nyomtató – solid ink printer, 132
szimatoló gyorsítótár – snooping/snoopy cache, 609, 611–614
szimbolikus név – symbolic name, 718
szimbólummal kezdődő blokk (BSS) – Block Started by Symbol (BSS), 742
szimbólumtábla – symbol table, 545–547, 741
szimmetrikus kulcsú titkosítás – symmetric key cryptography, 598
szimmetrikus multiprocesszor (SMP) – Symmetric MultiProcesszor (SMP), 600, 610–619
szimplex vonal – simplex line, 135
szín gamut – color gamut, 131
színkereső tábla (CLUT) – Color Look Up Table (CLUT), 595
szinkron DRAM (SDRAM) – Synchronous DRAM (SDRAM), 189
szinkron sín – synchronous bus, 198
szinkron üzenetátadás – synchronous message passing, 650
színpaletta – color palette, 125
szint – land, 108
  level, 19
szintvezérelt tároló – level-triggered latch, 179
szó – word, 86, 727
szoftver – software, 24
  ekvivalenciája a hardverrel – equivalence with hardware, 24
szoftverréteg – software layer, 236
szolga, sín – slave, bus, 195
szorosan kapcsolt – tightly coupled, 564
szubrutin – subroutine, 400, 737
szuperskaláris architektúra – superscalar architecture, 78–80
szuperszámítógép – supercomputer, 36, 51
szülő processzus – parent process, 510
szűrő – filter, 502

T

tagolt memória – interleaved memory, 619
találati arány – hit ratio, 93
találkozópont – crosspoint, 617
tár (lásd memória) – store (see memory)
tárgymodul – object file, object module, 551, 741
tárgyprogram – object program, 524
tárolás utáni betöltés – store-to-load, 337
tárold-és-továbbítsd csomagkapcsolás – store-and-forward packet switching, 585
tároló – latch, storage, 77, 177–179
társprocesszor – coprocessor, 583–592
társrutin/korutin – coroutine, 414, 421, 422
TAT–12/13 – TAT-12/13, 43
távoli hívás – far call, 737
távoli memória elérése nélküli (NORMA) – NO Remote Memory Access (NORMA), 606
távoli ugrás – far jump, 735

távolságbehatárolás – ranging, 139
TCP – TCP (Transmission Control Protocol), 586
TCP-fejléc – TCP header, 586
telco (telefontársaság) – telco (telephone company), 135
telefontársaság (telco) – telephone company (telco), 135
telekommunikációs berendezés – telecommunications equipment, 133
telített módú aritmetika – saturated arithmetic, 569
teljes erőforrás-megosztás – full resource sharing, 577
teljes kézfogás – full handshake, 202
teljes összeadó – full adder, 171–172
teljes összekötés – full interconnect, 633
térbeli lokalitás – spatial locality, 314
terhelhetőségi szám – fanout, 631
termelő-fogyasztó probléma – producer-consumer problem, 478, 511
tervezési elvek, RISC – design principles, RISC, 74
tesztágy – benchmark, 526
tevékeny várakozás – busy waiting, 403
téves megosztás – false sharing, 655
TEXT szegmens – TEXT section, 742
TFT (vékonyfilm-tranzisztor) – TFT (Thin Film Transistor), 124
TFT megjelenítő – TFT display, 124
tintasugaras nyomtató – inkjet printer, 128
tiny modell, 8088 – tiny model, 8088, 746
TLB (lapkezelő segédpuffer) – TLB (Translation Lookaside Buffer), 467
TLB-hiány – TLB miss, 468
TN (elforgatott molekulájú) kijelző – TN (Twisted Nematic) display, 123
token, vezérjel – token, 582
TOS (veremtető) – TOS (Top Of Stack), 253, 262, 280
többletes jelölés – excess notation, 704
többszálúság – multithreading, 572–578
többszekciós CD-ROM – multisession CD-ROM, 113
többszintű gép – multilevel machine, 20
többszintű kapcsoló hálózat – multistage switching network, 618
többszintű számítógépek fejlődése – evolution of multilevel computers, 23
többszörös utasításáram egyszeres adatáram (MISD) – Multiple Instruction stream Single Data stream (MISD), 604–605
többszörös utasításáram többszörös adatáram (MIMD) – Multiple Instruction stream Multiple Data stream (MIMD), 604–605
többszörözött munkás modell – replicated worker model, 658
töltéscsatolt eszköz (CCD) – Charge Coupled Device (CCD), 142
tömbprocesszor – array processor, 80
tördelő kódolás – hash coding, 547
töréspont – breakpoint, 749
törölhető PROM (EPROM) – Erasable PROM (EPROM), 60, 190
törölhető, programozható ROM (EPROM) – Erasable Programmable ROM (EPROM), 60, 190
történeti áttekintés – history
  1642–1945 – 1642-1945, 30
  1945–1955 – 1945-1955, 31
  1955–1965 – 1955-1965, 34
  1965–1980 – 1965-1980, 36
  1980–napjainkig – 1980-present, 38
  Intel – Intel, 52
  Intel 8051 – Intel 8051, 59
  mikroprogramozás – microprogramming, 24, 27
  operációs rendszer – operating system, 25
  Sun Microsystems – Sun Microsystems, 57
  számítógéprendszerek – computer systems
törtrész – fraction, 709
transzparens/átlátszó (lapozási mechanizmus) – transparent (paging), 448
tranzakciós réteg – transaction layer, 235
tranzisztor, feltalálása – transistor, invention, 34
tranzisztor-tranzisztor logika (TTL) – Transistor-Transistor Logic (TTL), 155
TriMedia – TriMedia, 567–572
TSB (lapkezelő tárolópuffer) – TSB (Translation Storage Buffer), 468
TTL (tranzisztor-tranzisztor logika) – TTL (Transistor-Transistor Logic), 155
túlcsordulási hiba – overflow error, 709
tűzfal – firewall, 585
TX–0 – TX-0, 34
TX–2 – TX-2, 34

U, Ú

UART (univerzális aszinkron adó/vevő) – UART (Universal Asynchronous Receiver/Transmitter), 240
UDB II – UDB II (UltraSPARC Data Buffer II), 217
ugrás (lásd elágazás) – jump (see branch)
ugrótábla – branch table, 339
UHCI (univerzális kiszolgálóvezérlő interfész) – UHCI Universal Host Controller Interface), 240
újraírható CD (CD-RW) – CD-rewritable (CD-RW), 114
UltraSPARC I – UltraSPARC I, 58
UltraSPARC III – UltraSPARC III, 214–218, 378
  adatpuffer – data buffer, 217
  adattípusok – data types, 370–371
  címzés – addressing, 392
  Hanoi tornyai – towers of Hanoi, 429–431

ISA-szintjének áttekintése – overview of ISA, 362–365
mikroarchitektúra – microarchitecture, 338–343
történeti áttekintés – history, 57
utasítások – instructions, 408–410
utasításformátumok – instruction formats, 378–379
virtuális memória – virtual memory, 466–469
UMA (egységes memóriaelérés) – UMA (Uniform Memory Access), 605
UNICODE – UNICODE, 145
univerzális aszinkron adó/vevő (UART) – Universal Asynchronous Receiver/Transmitter (UART), 240
univerzális kiszolgálóvezérlő interfész (UHCI) – Universal Host Controller Interface (UHCI), 240
univerzális soros sín (USB) – Universal Serial Bus (USB), 236–240
univerzális szinkron, aszinkron adó-vevő (USART) – Universal Synchronous Asynchronous Receiver/Transmitter (USART), 241
UNIX – UNIX
bevezetés – introduction, 486
fájl B/K – file I/O, 499–505
processzuskezelés – process management, 489, 510
szerkesztés – linking, 559
virtuális memória – virtual memory, 495
UPA – Ultra Port Architecture (UPA), 217
USART (univerzális szinkron, aszinkron adó-vevő) – USART (Universal Synchronous Asynchronous Receiver/Transmitter), 241
USB (univerzális soros sín) – USB (Universal Serial Bus), 236–240
utasítás – instruction, 359–368, 394–414
8051 – 8051, 411
8088 – 8088, 730
adatmozgató – data movement, 394–395
B/K – I/O, 402–405
ciklus – loop, 400–401
diadikus – dyadic, 395–396
elágazó – branch, 398, 414
eljáráshívó – procedure call, 400
monadikus – monadic, 396–398
összehasonlító – comparison, 398–399
Pentium 4 – Pentium 4, 405–408
UltraSPARC – UltraSPARC, 408–410
utasításbetöltő egység (IFU) – Instruction Fetch Unit (IFU), 296–300
utasításcsoport – instruction group, 435
8051 – 8051, 379
utasításformátumok – instruction formats, 371–380
8051 – 8051, 379–380
Pentium 4 – Pentium 4, 376–379
tervezési követelmények – design criteria, 372
UltraSPARC – UltraSPARC, 378
utasítás-helyszámláló (ILC) – Instruction Location Counter (ILC), 540
utasításkészlet, Intel 8088 – instruction set, Intel 8088, 730
utasításkiosztó egység – instruction issue unit, 339
utasításmutató (IP) – Instruction Pointer (IP), 720, 724
utasításregiszter (IR) – Instruction Register (IR), 67, 344
utasításrendszer – instruction sets, comparison, 411
utasításrendszer-architektúra (ISA) – Instruction Set Architecture (ISA), 22, 352–443, 717
Pentium 4 – Pentium 4, 405–408
UltraSPARC – UltraSPARC, 408–410
utasításrendszer-architektúra (ISA) szintje – Instruction Set Architecture (ISA) level, 22, 352–443, 717
8051 – 8051, 411
utasításszám, RISC-hez viszonyítva – instruction count, relation to RISC, 72
utasításszámláló (PC) – program counter, PC, 252–262, 720
utasítástípusok – instruction types, 394–414
utasításütemezés – instruction scheduling, 434–436
utasítás-végrehajtás – instruction execution, 68
utasításszintű párhuzamosság – instruction level parallelism, 75–80, 565–572
úthossz – path length, 292
csökkentés – reducing, 294
útvonal(név) – path, 502
útvonalválasztó – router, 585

Ü

üreg – pit, 108
ütemezés, multiszámítógép – scheduling multicompnters, 652–653
ütemező – scheduler, 336
üzenetátadás – message passing, 650–652
üzenetátadás interfész (MPI) – Message-Passing Interface (MPI), 651
üzenetátadásos multiszámítógép – message passing multicomputers, 629–667
üzenetsor – message queue, 511

V

valódi függőség – true dependence, 307
valós üzemmód – real mode, 360
váltottsoros – interlaced, 596
vámpírcsatlakozó – vampire tap, 584
várakozó állapot – wait state, 199

Vas-oxid Völgy – Iron Oxide Valley, 43
VAX – VAX, 71
VCI (virtuális komponens összeköttetés) – VCI (Virtual Component Interconnect), 583
védett üzemmód – protected mode, 360
véges állapotú gép (FSM) – Finite State Machine (FSM), 298
elágazásjövendölés – branch prediction, 323
utasításbetöltő egység (IFU) – instruction fetch unit (IFU), 299
véges pontosságú szám – finite-precision number, 696–698
végrehajtható bináris fájl – executable binary file, 741
végrehajtható bináris program – executable binary program, 524, 548
végrehajtó, Windows XP – executive, Windows XP, 492
vékonyfilm-tranzisztor (TFT) – Thin Film Transistor (TFT), 124
vektorprocesszor – vectorprocessor, 81
vektorregiszter – vector register, 81
véletlen elérésű memória – Random Access Memory (RAM), 60, 188
verem – stack, 266
veremcímzés – stack addressing, 384–388
veremkeret – stack frame, 723
veremmutató (SP) – Stack Pointer (SP), 253, 262, 266, 280, 723
veremtető (TOS) – Top Of Stack (TOS), 280
vergődés – thrashing, 454
versenyhelyzet – race condition, 482, 489
vezérjel, token – token, 582
vezérlési folyamat – flow of control, 414–426
csapda – trap, 422–423
elágazás – branche, 414–415
eljárás – procedure, 415–421
korutin (társrutin) – coroutine, 421–422
megszakítás – interrupt, 423–426
vezérlő – controller, 118
vezérlőjel – control signal, 257
vezérlőtár – control store, 72, 261
viasznyomtató – wax printer, 132
Video RAM, videomemória – Video RAM, 125
virtuális 8086 mód – virtual 8086 mode, 360
virtuális áramkör – virtual circuit, 235
virtuális B/K – virtual I/O, 469
megvalósítás – implementation, 505
Windows XP – Windows XP, 505
virtuális címtartomány – virtual address space, 447
UNIX – UNIX, 499
virtuális gép – virtual machine, 18
virtuális komponens összeköttetés (VCI) – Virtual Component Interconnect (VCI), 583
virtuális memória – virtual memory, 445–469
és gyorsítótár – compared to caching, 469 452–453
kezelő, Windows XP – manager, Windows XP, 493
Pentium 4 – Pemtium 4, 462–466
UltraSPARC – UltraSPARC,466–469
UNIX – UNIX, 495–496
Windows XP – Windows XP, 496
virtuális párhuzamos számítógép (PVM) – Parallel Virtual Machine (PVM), 651
virtuális regiszter – virtual register, 265
virtuális szervezet – virtual organization, 668
virtuális topológia – virtual topology, 652
virtuális vágás – virtual cut through, 639
visszafelé kompatibilis – backward compatibility, 353
visszaírás gyorsítótár – write-back cache, 319
vivőhullám – carrier, 133
vizuális utasításkészlet (VIS) – Visual Instruction Set (VIS), 58
VLSI (nagyon magas fokú integráltság) – VLSI (Very Large Scale Integration), 38, 490
vonal per inch (lpi) – Lines Per Inch (lpi), 131
vonzásmemória – attraction memory, 628
VTOC (kötet tartalomjegyzék) – VTOC (Volume Table Of Contents), 113

W, X Y, Z

WAN (nagyterületű hálózat) – WAN (Wide Area Network), 584
Wattel, Evert – Wattel, Evert, 717
WEIZAC – WEIZAC, 32
Whirlwind I – Whirlwind I, 34
Wilkes, Maurice – Wilkes, Maurice, 24, 32, 71
Win32 alrendszer – Win32 subsystem, 494
Win32 API – Win32 API, 494
Windows – Windows, 40, 490
szerkesztés – linking, 557
történet – history, 40
Windows 95 – Windows 95, 490
Windows 98 – Windows 98, 490
Windows NT – Windows New Technology (NT), 490
Windows XP – Windows XP
bevezetés – introduction, 489–495
fájl I/O – file B/K, 505–510
processzuskezelés – process management, 513–516
virtuális memória – virtual memory, 496–498
Wozniak, Steve – Wozniak, Steve, 39
X Window System – X Window System, 489
XP-fájlrendszer – XP file system, 505
Yellow Book (Sárga Könyv) – Yellow Book, 110
zártság – closure, 697
Zilog Z8000 – Zilog Z8000, 72
Zuse, Konrad – Zuse, Konrad, 31