ASSEMBLY PROGRAMOZÁS

AJÁNLOTT IRODALOM

1) Pethő Ádám: Assembly alapismeretek 1. kötet, Számalk, Budapest, 1992
2) Peter Norton-John Socha: Az IBM PC Assembly nyelvű programozása, Novotrade, Budapest, 1991
3) Peter Norton: Az IBM PC programozása, Műszaki Könyvkiadó, Budapest, 1992
4) László József: A VGA-kártya programozása Pascal és Assembly nyelven, ComputerBooks, Budapest, 1994
5) Abonyi Zsolt: PC hardver kézikönyv
6) Dr. Kovács Magda: 32 bites mikroprocesszorok 80386/80486 I. és II. kötet, LSI, Budapest

TARTALOM

1 AZ ASSEMBLY NYELV TULAJDONSÁGAI, JELENTŐSÉGE

A számítógépes problémamegoldás során a kitűzött célt megvalósító algoritmust mindig valamilyen *programozási nyelven* (programming language) írjuk, kódoljuk le. A nyelvet sokszor az adott feladat alapján választjuk meg, míg máskor aszerint döntünk egy adott nyelv mellett, hogy az hozzánk, az emberi gondolkodáshoz mennyire áll közel. Ez utóbbi tulajdonság alapján csoportosíthatók a számítógépes programozási nyelvek: megkülönböztetünk *alacsony szintű* (low-level) és *magas szintű programozási nyelveket* (high-level programming language). Az előbbire jó példák az Assembly és részben a C, az utóbbira pedig a Pascal ill. a BASIC nyelvek. Ha a nyelvek szolgáltatásait tekintjük, akkor szembeötlő, hogy ahogy egyre fentebb haladunk az alacsony szintű nyelvektől a magas szintűek felé, úgy egyre nagyobb szabadsággal, egyre általánosabb megoldásokkal találkozunk.

Az Assembly tehát egy alacsony szintű programozási nyelv, méghozzá nagyon alacsony szintű, ebből következően pedig sokkal közelebb áll a hardverhez, mint bármely más nyelv. Főbb jellemzői:

- **nagyon egyszerű, elemi műveletek**
- **típustalanság**
- **rögzített utasításkészlet**
- **világos, egyszerű szintaxis**
- **kevés vezérlési szerkezet**
- **nagyon kevés adattípus; ha több is van, akkor általában egymásból származtathatók valahogyan**

De miért is van szükség az Assemblyre, ha egyszer ott van a többi nyelv, amikben jóval kényelmesebben programozhatunk? Erre egyik indok, hogy a magas szintű nyelvek eljárásai, függvényei sokszor általánosra lettek megírva, így teljesen feleslegesen foglalkoznak olyan dolgokkal, amikre esetleg soha sem lesz szükségünk. Erre jó példák lehetnek a Borland Pascal/C grafikus eljárásai, valamint ki-/bemeneti (I/O) szolgáltatásai. Kört rajzolhatunk a Circle eljárással is, de ennél gyorsabb megoldást kapunk, ha vesszük a fáradságot, és mi magunk írunk egy olyan körrajzolót, ami semmi mást nem csinál, csak ami a feladata: helyesen kirajzolja a kört a képernyőre, de nem foglalkozik pl. hibaellenőrzéssel, a képernyő szélén kívülre kerülő pontok kiszűrésével stb. Hasonló a helyzet a fájlkezeléssel is. Ha nem akarunk speciális típusokat (mondjuk objektumokat, rekordokat) állományba írni, mindössze valahány bájtot szeretnénk beolvasni vagy kiírni a lemezre, akkor felesleges a fenti nyelvek rutinjait használni. Mindkét feladat megoldható Assemblyben is, méghozzá hatékonyabban, mint a másik két nyelvben.

Akkor miért használják mégis többen a C-t, mint az Assemblyt? A választ nem nehéz megadni: magasabb szintű nyelvekben a legtöbb probléma gyorsabban leírható, a forrás rövidebb, strukturáltabb, s ezáltal áttekinthetőbb lesz, könnyebb lesz a későbbiekben a program karbantartása, és ez nem csak a program szerzőjére vonatkozik. Mégsem mellőzhetjük az Assemblyt, sok dolgot ugyanis vagy nagyon nehéz, vagy egyszerűen képtelenség megcsinálni más nyelvekben, míg Assemblyben némi energia befektetése árán ezek is megoldhatók. Aki Assemblyben akar programozni, annak nagyon elszántnak, türelmesnek, kitartónak kell lennie. A hibalehetőségek ugyanis sokkal gyakoribbak itt, és egy-egy ilyen baki megkeresése sokszor van olyan nehéz, mint egy másik program megírása.

Régen, mikor még nem voltak modern programozási nyelvek, fordítóprogramok, akkor is kellett valahogy dolgozni az embereknek. Ez egy *gépi kódnak* (machine code) nevezett nyelven történt. A gépi kód a processzor saját nyelve, csak és kizárólag ezt érti meg. Ez volt ám a fárasztó dolog! A gépi kód ugyanis nem más, mint egy rakás szám egymás után írva. Akinek ebben kellett programozni, annak fejből tudnia kellett az összes utasítás összes lehetséges változatát, ismernie kellett a rendszert teljes mértékben. Ha egy kívülálló rátekintett egy gépi kódú programra, akkor annak működéséből, jelentéséből jobbára semmit sem értett meg. Nézzünk egy példát: 0B8h 34h 12h // 0F7h 26h 78h 56h // 0A3h 78h 56h, ahol a dupla törtvonal az utasításhatárt jelzi. Ez a tíz hexadecimális (tizenhatos szám-rendszerbeli) szám nem mond túl sokat első ránézésre. Éppen ezért kidolgoztak egy olyan jelölésrendszert, nyelvet, amiben emberileg emészthető formában leírható bármely gépi kódú program. Ebben a nyelvben a hasonló folyamatot végrehajtó gépi kódú utasítások csoportját egyetlen szóval, az ú.n. *mnemonikkal* (mnemonic) azonosítják. Természetesen van olyan mnemonik is, ami egyetlen egy utasításra vonatkozik. Ez a nyelv lett az Assembly. Ebben az új jelölésrendszerben a fenti programrészlet a következő formát ölti:

```
MOV     AX,1234h                ;0B8h 34h 12h
MUL     WORD PTR [5678h]        ;0F7h 26h 78h 56h
MOV     [5678h],AX              ;0A3h 78h 56h
```

Némi magyarázat a programhoz:

- **az első sor egy számot (1234h) rak be az AX regiszterbe, amit most tekinthetünk mondjuk egy speciális változónak**
- **a második sor a fenti értéket megszorozza a memória egy adott címén (5678h) található értékkel**
- **a harmadik sor az eredményt berakja az előbbi memóriarekeszbe**

- **a pontosvessző utáni rész csak megjegyzés**
- **az első oszlop tartalmazza a mnemonikot**
- **a memóriahivatkozásokat szögletes zárójelek ([és]) közé írjuk**

Tehát a fenti három sor egy változó tartalmát megszorozza az 1234h számmal, és az eredményt visszarakja az előbbi változóba.

Mik az Assembly előnyei?

- **korlátlan hozzáférésünk van a teljes hardverhez, beleértve az összes perifériát (billentyűzet, nyomtató stb.)**
- **pontosan ellenőrizhetjük, hogy a gép tényleg azt teszi-e, amit elvárunk tőle**
- **ha szükséges, akkor minimalizálhatjuk a program méretét és/vagy sebességét is (ez az ú.n. optimalizálás)**

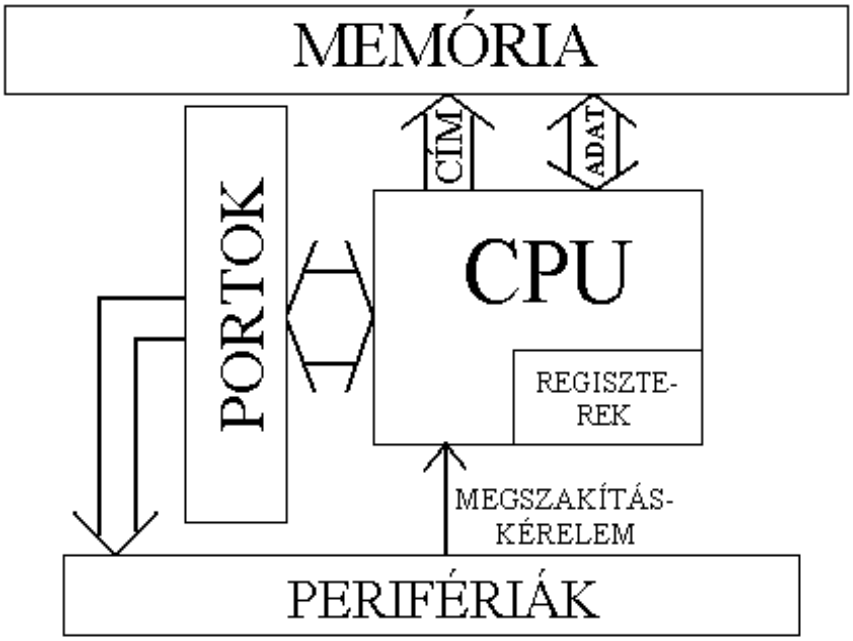
Most lássuk a hátrányait:

- **a forrás sokszor áttekinthetetlen még a szerzőnek is**
- **a kódolás nagy figyelmet, türelmet, és főleg időt igényel**
- **sok a hibalehetőség**
- **a hardver alapos ismerete elengedhetetlen**
- **a forrás nem hordozható (nem portolható), azaz más alapokra épülő számítógépre átírás nélkül nem vihető át (ez persze igaz a gépi kódra is)**

Bár úgy tűnhet, több hátránya van mint előnye, mégis érdemes alkalmazni az Assemblyt ott, ahol más eszköz nem segít. A befektetett erőfeszítések pedig meg fognak térülni.

2 A PC-K HARDVERÉNEK FELÉPÍTÉSE

Az IBM PC-k felépítését szemlélteti a következő ábra eléggé leegyszerűsítve:



Az első IBM PC az Intel 8086-os *mikroprocesszorával* jelent meg, és hamarosan követte az IBM PC XT, ami már Intel 8088-os maggal ketyegett. Később beköszöntött az AT-k időszaka, s vele jöttek újabb processzorok is: Intel 80286, 80386 (SX és DX), 80486 (SX, DX, DX2 és DX4), majd eljött az 586-os és 686-os gépek világa (Pentium, Pentium Pro, Pentium II stb.) Nem csak az Intel gyárt processzorokat PC-kbe, de az összes többi gyártó termékére jellemző, hogy (elvileg) 100%-osan kompatibilis az Intel gyártmányokkal, azaz ami fut Intel-en, az ugyanúgy elfut a másik procin is, és viszont. Az összes későbbi processzor alapja tulajdonképpen a 8086-os volt, éppen ezért mondják azt, hogy a processzorok ezen családja az Intel 80x86-os (röviden x86-os) architektúrájára épül.

A továbbiakban kizárólag az Intel 8086/8088-os processzorok által biztosított programozási környezetet vizsgáljuk. (A két proci majdnem teljesen megegyezik, a különbség mindössze az adatbuszuk szélessége: a 8086 16 bites, míg a 8088-as 8 bites külső adatbusszal rendelkezik.)

A fenti ábrán a *CPU* jelöli a processzort (Central Processing Unit=központi feldolgozó egység). Mint neve is mutatja, ő a gép agya, de persze gondolkodni nem tud, csak végrehajtja, amit parancsba adtak neki. A CPU-n többek között található néhány különleges, közvetlenül elérhető tárolóhely. Ezeket *regiszterekneк* (register) nevezzük. A processzor működés közbeni újraindítását *reset*-nek hívjuk. Reset esetén a processzor egy jól meghatározott állapotba kerül.

A *memória* adja a számítógép "emlékezetét". Hasonlóan az emberi memóriához, van neki felejtő (az információt csak bizonyos ideig megőrző) és emlékező változata. Az előbbi neve *RAM* (Random Access Memory=véletlen hozzáférésű memória), míg az utóbbié *ROM* (Read Only Memory=csak olvasható memória). A ROM fontos részét képezi az ú.n. *BIOS* (Basic Input/Output System=alapvető ki-/bemeneti rendszer). A gép bekapcsolása (és reset) után a BIOS-ban levő egyik fontos program indul el először (pontosabban minden processzort úgy terveznek, hogy a BIOS-t kezdje el végrehajtani ilyenkor). Ez leellenőrzi a hardverelemeket, teszteli a memóriát, megkeresi a jelenlevő perifériákat, majd elindítja az operációs rendszert, ha lehet. A BIOS ezenkívül sok hasznos rutint tartalmaz, amikkel vezérelhetjük például a billentyűzetet, videokártyát, merev-lemezt stb.

***Busznak* (bus) nevezzük "vezetékek" egy csoportját, amik bizonyos speciális célt szolgálnak, és a CPU-t kötik össze a számítógép többi fontos részével. Megkülönböztetünk adat-, cím- ill. vezérlőbuszt (data bus, address bus, control bus). A címbusz szélessége (amit bitekben ill. a "vezetékek" számában**

mérünk) határozza meg a megcímezhető memória maximális nagyságát.

A CPU a *perifériákkal* (pl. hangkártya, videovezérlő, DMA-vezérlő, nyomtató stb.) az ú.n. *portokon* keresztül kommunikál. Ezeket egy szám azonosítja, de ne úgy képzeljük el, hogy annyi vezeték van bekötve, ahány port van. Egyszerűen, ha egy eszközt el akar érni a CPU, akkor kiírja a címbuszára a port számát, és ha ott "van" eszköz (tehát egy eszköz arra van beállítva, hogy erre a portszámra reagáljon), akkor az válaszol neki, és a kommunikáció megkezdődik.

Azonban nem csak a CPU szólhat valamelyik eszközhöz, de azok is jelezhetik, hogy valami mondanivalójuk van. Erre szolgál a *megszakítás-rendszer* (interrupt system). Ha a CPU érzékel egy *megszakítás-kérelmet* (IRQ–Interrupt ReQuest), akkor abbahagyja az éppen aktuális munkáját, és kiszolgálja az adott eszközt. A megszakítások először a *megszakítás-vezérlőhöz* (interrupt controller) futnak be, s csak onnan mennek tovább a processzorhoz. Az XT-k 8, az AT-k 16 db. független *megszakítás-vonallal* rendelkeznek, azaz ennyi perifériának van lehetősége a megszakítás-kérésre. Ha egy periféria használ egy megszakítás-vonalat, akkor azt kizárólagosan birtokolja, tehát más eszköz nem kérhet megszakítást ugyanazon a vonalon.

A fentebb felsorolt rendszerelemek (CPU, memória, megszakítás-vezérlő, buszok stb.) és még sok minden más egyetlen áramköri egységen, az ú.n. *alaplapon* (motherboard) található.

Van még három fontos eszköz, amikről érdemes szót ejteni. Ezek az órajel-generátor, az időzítő és a DMA-vezérlő.

A CPU működését szabályos időközönként megjelenő elektromos impulzusok vezérlik. Ezeket nevezik *órajelnek* (clock, clocktick), másodpercenkénti darabszámuk mértékegysége a Hertz (Hz). Így egy 4.77 MHz-es órajel másodpercenként 4770000 impulzust jelent. Az órajelet egy

kvarckristályon alapuló *órajel-generátor* (clock generator) állítja elő.

A RAM memóriák minden egyes memóriarekeszét folyamatosan ki kell olvasni és vissza kell írni másodpercenként többször is, különben tényleg "felejtővé" válna. Erre a *frissítésnek* (memory refresh) nevezett műveletre felépítésük miatt van szükség. A műveletet pontos időközönként kell végrehajtani, ezt pedig egy különleges egység, az *időzítő* (timer) intézi el. Ennek egyik dolga az, hogy kb. 15.09 μs-onként elindítsa a frissítést (ez nagyjából 66287 db. frissítést jelent másodpercenként). Ezenkívül a rendszerórát (system clock) is ez a szerkezet szinkronizálja.

A memória elérése a CPU-n keresztül igen lassú tud lenni, ráadásul erre az időre a processzor nem tud mással foglalkozni. E célból bevezették a *közvetlen memória-hozzáférés* (DMA– Direct Memory Access) módszerét. Ez úgy működik, hogy ha egy perifériának szüksége van valamilyen adatra a memóriából, vagy szeretne valamit beírni oda, akkor nem a CPU-nak szól, hanem a *DMA-vezérlőnek* (DMA controller), és az a processzort kikerülve elintézi a kérést.

3 SZÁMRENDSZEREK, GÉPI ADATÁBRÁZOLÁS, ARITMETIKA ÉS LOGIKA

Az emberek általában tízes (*decimális* – decimal) szám-rendszerben számolnak a mindennapjaik során, hiszen ezt tanították nekik, és ez az elfogadott konvenció a világon. A processzort azonban (de a többi hardverösszetevőt, pl. a memóriát is) feleslegesen túlbonyolítaná, ha neki is ezekkel a számokkal kellene dolgoznia. Ennél jóval egyszerűbb és

kézenfekvő megoldás, ha kettes alapú (*bináris* – binary) számrendszerben kezel minden adatot. Az információ alapegysége így a bináris számjegy, a *bit* (BInary digiT) lesz, ezek pedig a 0 és az 1. Bináris számok ábrázolásakor ugyanúgy helyiértékes felírást használunk, mint a decimális számok esetén. Pl. a 10011101 bináris számnak 128+16+8+4+1=157 az értéke. Az egyes helyiértékek jobbról balra 2-nek egymás után következő hatványai, tehát 1, 2, 4, 8, 16, 32 stb. Ha sokszor dolgozunk bináris számokkal, akkor nem árt, ha a hatványokat fejből tudjuk a 0-diktól a 16-odikig.

Egy-egy aránylag kicsi szám bináris leírásához sok 0-t és 1-et kell egymás mellé raknunk, ez pedig néha fárasztó. Ezt kiküszöbölendő a számokat sokszor írjuk tizenhatos alapú (*hexadecimális* – hexadecimal) számrendszerben. Itt a szám-jegyek a megszokott 10 arab számjegy, plusz az angol ábécé első hat betűje (A, B, C, D, E, F), továbbá A=10, B=11 stb. Mivel $16=2^4$, ezért négy bináris számjegy éppen egy hexadecimális (röviden hexa) számjegyet tesz ki. Az előző példa alapján 10011101b=9Dh=157d. Ahhoz, hogy mindig tudjuk, a leírt számot milyen alapú rendszerben kell értelmezni, a szám után írunk egy "b", "h" vagy "d" betűt. Szokás még néha a nyolcas alapú (*oktális* – octal) felírást is alkalmazni. Ekkor a 0..7 számjegyeket használjuk, és 3 bináris jegy tesz ki egy oktális számjegyet. Az oktális számok végére "o" betűt írunk.

Most elevenítsük fel a legegyszerűbb, közismert logikai műveleteket. Ezek ugyanis fontos szerepet játszanak mind a programozás, mind a processzor szempontjából. A két logikai igazságértéket itt most bináris számjegyek fogják jelölni.

A negáció (tagadás – negation) egyváltozós (unáris) művelet, eredménye a bemeneti igazságérték ellentettje. A műveletet jelölje NOT az angol tagadás mintájára.

NOT	
0	1
1	0

A konjunkció ("ÉS") már kétváltozós (bináris) művelet. Jele AND (az "és" angolul), eredményét a következő táblázat szemlélteti:

AND	0	1
0	0	0
1	0	1

A diszjunkció ("VAGY") szintén bináris művelet. Jele OR (a "vagy" angolul), és a két változón MEGENGEDŐ VAGY műveletet hajt végre:

OR	0	1
0	0	1
1	1	1

Utolsó műveletünk az antivalencia ("KIZÁRÓ VAGY"). Jele a XOR (eXclusive OR), hatása az alábbi ábrán követhető:

XOR	0	1
0	0	1
1	1	0

A legtöbb processzor kizárólag egész számokkal tud számolni, esetleg megenged racionális értékeket is. Az ábrázolható számok tartománya mindkét esetben véges, ezt ugyanis a processzor regisztereinek bitszélessége határozza meg. A számítástechnikában a legkisebb ábrázolható információt a bit képviseli, de mindenhol az ennél nagyobb, egészen pontosan 8 db. bitből álló *bájtot* (byte) használják az adatok alapegységeként, és pl. a regiszterek és az adatbusz szélessége is ennek többszöröse. Szokás még más, a bájt fogalmára épülő

mértékegységet is használni, ilyen a *szó* (word; általában 2 vagy 4 bájt), a *duplaszó* (doubleword; a szó méretének kétszerese) és a *kvadraszó* (quadword; két duplaszó méretű). A bájtban a biteket a leírás szerint jobbról balra 0-tól kezdve számozzák, és egy bájtot két hexadecimális számjeggyel lehet leírni.

A számítástechnikában a kilo- (k, K) és mega- (M) előtét-szavak a megszokott 1000 és 1000000 helyett 1024-et (=2^{10}) ill. 1048576-ot (=2^{20}) jelentenek. A giga- (G) hasonlóan 2^{30}-t jelent.

Fontos szólni egy fogalomról, az ú.n. *endianizmusról* (endianism). Ez azt a problémát jelenti, hogy nincs egyértelműen rögzítve a több bájt hosszú adatok ábrázolása során az egyes bájtok memóriabeli sorrendje. Két logikus verzió létezik: a legkevésbé értékes bájttal kezdünk, és sorban haladunk a legértékesebb bájt felé (*little-endian* tárolás), ill. ennek a fordítottja, tehát a legértékesebbtől haladunk a legkevésbé értékes bájt felé (*big-endian* tárolás). Mindkét megoldásra találhatunk példákat a különböző hardvereken.

Nézzük meg, hogy ábrázoljuk az egész számokat. Először tételezzük fel, hogy csak nemnegatív (*előjeltelen* – unsigned) számaink vannak, és 1 bájtot használunk a felíráshoz. Nyolc biten 0 és 255 (=2^8-1) között bármilyen szám felírható, ezért az ilyen számokkal nincs gond.

Ha negatív számokat is szeretnénk használni, akkor két lehetőség adódik:

- **csak negatív számokat ábrázolunk**
- **az ábrázolási tartományt kiterjesztjük a nemnegatív számokra is**

Ez utóbbi módszert szokták választani, és a tartományt praktikus módon úgy határozzák meg, hogy a pozitív és negatív számok nagyjából azonos mennyiségben legyenek. Ez esetünkben azt jelenti, hogy -128-tól +127-ig tudunk számokat felírni (beleértve a 0-t is). De hogy különböztessük meg a negatív számokat a pozitívaktól? Természetesen az *előjel* (sign) által. Mivel ennek két értéke lehet (ha a nullát pozitívnak

tekintjük), tárolására elég egyetlen bit. Ez a kitüntetett bit az *előjelbit* (sign bit), és megegyezés szerint az adat legértékesebb (most significant), azaz legfelső bitjén helyezkedik el. Ha értéke 0, akkor a tekintett *előjeles* (signed) szám pozitív (vagy nulla), 1 esetén pedig negatív. A fennmaradó biteken (esetünkben az alsó 7 bit) pedig tároljuk magát a számot előjele nélkül. Megtehetnénk, hogy azonos módon kezeljük a pozitív és negatív számokat is, de kényelmi szempontok és a matematikai műveleti tulajdonságok fenntartása végett a negatív számok más alakban kerülnek leírásra. Ez az alak az ú.n. *kettes komplemens* (2's complement). Ennek kiszámítása úgy történik, hogy az ábrázolható legnagyobb előjeltelen számnál eggyel nagyobb számhoz (ez most 256) hozzáadjuk a kérdéses számot, és az eredményt modulo 256 vesszük. Még egyszerűbb megérteni, ha bevezetjük az *egyes komplemens* (1's complement) fogalmát is. Ezt úgy képezzük bármilyen értékű bájt (szó stb.) esetén, hogy annak minden egyes bitjét negáljuk. Ha vesszük egy tetszőleges szám egyes komplemensét, majd ahhoz hozzáadunk 1-et (az összeadást az alábbiak szerint elvégezve), akkor pont a szám kettes komplemensét kapjuk meg. Egy szám kettes komplemensének kettes komplemense a kiinduló számot adja vissza.

Egy példán illusztrálva: legyenek az ábrázolandó számok 0, 1, 2, 127, 128, 255, -1, -2, -127 és -128. A számok leírása ekkor így történik:

- **0 (előjeltelen vagy előjeles pozitív) = 00000000b**
- **1 (előjeltelen vagy előjeles pozitív) = 00000001b**
- **2 (előjeltelen vagy előjeles pozitív) = 00000010b**
- **127 (előjeltelen vagy előjeles pozitív) = 01111111b**
- **128 (előjeltelen) = 10000000b**
- **255 (előjeltelen) = 11111111b**
- **-1 (előjeles negatív) = 11111111b**
- **-2 (előjeles negatív) = 11111110b**
- **-127 (előjeles negatív) = 10000001b**
- **-128 (előjeles negatív) = 10000000b**

Láthatjuk, hogy az előjeltelen és előjeles ábrázolás tartományai között átfedés van (0..127), míg más értékek esetén ütközés áll fenn (128..255 ill. -1..-128). Azaz a 11111111b számot olvashatjuk 255-nek de akár -1-nek is.

Ha nem bájton, hanem mondjuk 2 bájtos szóban akarjuk tárolni a fenti számokat, akkor ezt így tehetjük meg:

- **0 = 00000000 00000000b**
- **1 = 00000000 00000001b**
- **2 = 00000000 00000010b**
- **127 = 00000000 01111111b**
- **128 = 00000000 10000000b**
- **255 = 00000000 11111111b**
- **-1 = 11111111 11111111b**
- **-2 = 11111111 11111110b**
- **-127 = 11111111 10000001b**
- **-128 = 11111111 10000000b**

Ebben az esetben meg tudjuk különböztetni egymástól a -1-et és a 255-öt, de átfedő rész itt is van (32768..65535 ill. -1..-32768).

Végül megnézzük, hogy végezhetők el a legegyszerűbb matematikai műveletek. A műveletek közös tulajdonsága, hogy az eredmény mindig hosszabb 1 bittel, mint a két szám közös hossza (úgy tekintjük, hogy mindkettő tag azonos bitszélességű). Így két 1 bites szám összege és különbsége egyaránt 2 bites, míg két bájté 9 bites lesz. A +1 bit tulajdonképpen az esetleges átvitelt tárolja.

Két bináris számot ugyanúgy adunk össze, mint két decimális értéket:

1)kiindulási pozíció a legalacsonyabb helyiértékű jegy, innen haladunk balra
2)az átvitel kezdetben 0

3)az aktuális pozícióban levő két számjegyet összeadjuk, majd ehhez hozzáadjuk az átvitelt (az eredmény két jegyű bináris szám lesz)
4)az eredmény alsó jegyét leírjuk az összeghez, az átvitel pedig felveszi a felső számjegy értékét
5)ha még nem értünk végig a két összeadandón, akkor menjünk ismét a 3)-ra
6)az átvitelt mint számjegyet írjuk hozzá az összeghez

Az összeadási szabályok pedig a következőek:

- 0+0 = 00, átvitel=0
- 0+1 = 01, átvitel=0
- 1+0 = 01, átvitel=0
- 1+1 = 10, átvitel=1
- 10+01 = 11, átvitel=1

Ezek alapján ellenőrizhetjük, hogy a fent definiált kettes komplemens alak valóban teljesíti azt az alapvető algebrai tulajdonságot, hogy egy számnak és additív inverzének (tehát -1-szeresének) összege 0 kell legyen. És valóban:

1d+(-1d)=00000001b+11111111b=1 00000000b

Az eredmény szintén egy bájt lesz (ami tényleg nulla), valamint keletkezik egy átvitel is. Az egyes komplemens is rendelkezik egy érdekes tulajdonsággal. Nevezetesen, ha összeadunk egy számot és annak egyes komplemensét, akkor egy csupa egyesekből álló számot, azaz -1-et (avagy a legnagyobb előjeltelen számot) fogunk kapni!

Kivonáskor hasonlóan járunk el, csak más szabályokat alkalmazunk:

- 0-0 = 00, átvitel=0
- 0-1 = 11, átvitel=1
- 1-0 = 01, átvitel=0
- 1-1 = 00, átvitel=0
- 11-01 = 10, átvitel=1

továbbá a fenti algoritmusban a 3) lépésben az átvitelt le kell vonni a két számjegy különbségéből. Ezek alapján ugyancsak teljesül, hogy

1d-1d=00000001b-00000001b=0 00000000b

azaz egy számot önmagából kivonva 0-t kapunk. Viszont lényeges eltérés az előző esettől (amikor a kettes komplemens alakot adtuk hozzá a számhoz), hogy itt sose keletkezik átvitel a művelet elvégzése után.

A szorzás és az osztás már macerásabbak. Mindkét művelet elvégzése előtt meghatározzuk az eredmény (szorzat ill. hányados) előjelét, majd az előjeleket leválasztjuk a tagokról. Mindkét műveletet ezután ugyanúgy végezzük, mint ahogy papíron is csinálnánk. Osztás alatt itt maradékos egész osztást értünk. A szorzat hossza a kiinduló tagok hosszainak összege, a hányadosé az osztandó és osztó hosszainak különbsége, míg a maradék az osztandó hosszát örökli. A maradék előjele az osztandóéval egyezik meg, továbbá teljesül, hogy a maradék abszolút értékben kisebb mint az osztó.

A műveletek egy speciális csoportját alkotják a 2 hatványaival való szorzás és osztás. Ezeket közös néven *shiftelésnek* (eltolásnak, léptetésnek) hívjuk.

2-vel úgy szorozhatunk meg egy számot a legegyszerűbben, ha a bináris alakban utána írunk egy 0-t. De ez megegyezik azzal az esettel, hogy minden számjegy eggyel magasabb helyiértékre csúszik át, az alsó, üresen maradó helyet pedig egy 0-val töltjük ki. Erre azt mondjuk, hogy a számot egyszer balra shifteltük. Ahányszor balra shiftelünk egy számot, mindannyiszor megszorozzuk 2-vel, végeredményben tehát 2-nek valamely hatványával szorozzuk meg. Ez a módszer minden számra alkalmazható, legyen az akár negatív, akár pozitív.

Hasonlóan definiálható a jobbra shiftelés is, csak itt a legfelső megüresedő bitpozíciót töltjük fel 0-val. Itt azonban felmerül egy bökkenő, nevezetesen negatív számokra nem fogunk helyes eredményt kapni. A probléma az előjelbitben

keresendő, mivel az éppen a legfelső bit, amit pedig az előbb 0-val helyettesítettünk. A gond megszüntethető, ha bevezetünk egy előjeles és egy előjel nélküli jobbra shiftelést. Az előjeltelen változat hasonlóan működik a balra shifteléshez, az előjelesnél pedig annyi a változás, hogy a legfelső bitet változatlanul hagyjuk (vagy ami ugyanaz, az eredeti előjelbittel töltjük fel).

Megjegyezzük, hogy az angol terminológia megkülönbözteti az összeadáskor keletkező *átvitelt* a kivonásnál keletkezőtől. Az előbbit carry-nek, míg az utóbbit borrow-nak nevezik.

***Túlcsordulásról* (overflow) beszélünk, ha a művelet eredménye már nem tárolható a kijelölt helyen. Ez az aritmetikai műveleteknél, pl. shifteléskor, osztáskor fordulhat elő.**

Egy előjeles bájt *előjeles kiterjesztésén* (sign extension) azt a műveletet értjük, mikor a bájtot szó méretű számmá alakítjuk át úgy, hogy mindkettő ugyanazt az értéket képviselje. Ezt úgy végezzük el, hogy a cél szó felső bájtjának minden bitjét a kiinduló bájt előjelbitjével töltjük fel. Tehát pl. a -3d=11111101b szám előjeles kiterjesztése az 11111111 11111101b szám lesz, míg a +4d=00000100b számból 00000000 00000100b lesz. Ezt a fogalmat általánosíthatjuk is: bájt kiterjesztése duplaszóvá, szó kiterjesztése duplaszóvá, kvadraszóvá stb.

4 A 8086-OS PROCESSZOR JELLEMZŐI, SZOLGÁLTATÁSAI

Ismerkedjünk most meg az Intel 8086-os mikroprocesszorral közelebbről.

4.1 Memóriakezelés

A számítógép memóriáját úgy tudjuk használni, hogy minden egyes memóriarekeszt megszámozunk. Azt a módszert, ami meghatározza, hogy hogyan és mekkora területhez férhetünk hozzá egyszerre, *memória-szervezésnek* vagy *memória-modellnek* nevezzük. Több elterjedt modell létezik, közülük a legfontosabbak a lineáris és a szegmentált modellek.

***Lineáris modellen* (linear model) azt értjük, ha a memória teljes területének valamely bájtja egyetlen számmal megcímezhető (kiválasztható). Ezt az értéket ekkor *lineáris memóriacímnek* (linear address) nevezzük.**

***Szegmensen* (segment) a memória egy összefüggő, rögzített nagyságú darabját értjük most (létezik egy kicsit másféle szegmens-fogalom is), ennek kezdőcíme (tehát a szegmens legelső bájtjának memóriacíme) a *szegmens báziscím* avagy *szegmenscím* (segment base address). A szegmensen belüli bájtok elérésére szükség van azok szegmenscímhez képesti relatív távolságára, ez az *offszetcím* (offset address). *Szegmentált modell* (segmented model) esetén a *logikai memóriacím* (logical address) tehát két részből tevődik össze: a szegmenscímből és az offszetcímből. E kettő érték már elegendő a memória lefedéséhez. Jelölés:**

SZEGMENSCÍM:OFFSZETCÍM

tehát először leírjuk a szegmenscímet, azután egy kettőspontot, majd az offszet jön.

A 8086-os processzor 20 bites címbusszal rendelkezik, tehát a memóriacímek 20 bitesek lehetnek. Ez 1 Mbájt (=1024*1024 bájt) méretű memória megcímzéséhez elegendő. A processzor csak a szegmentált címzést ismeri. A szegmensek méretét 64 Kbájtban szabták meg, mivel így az offszet 16 bites lesz, ez pedig belefér bármelyik regiszterbe (ezt majd később

látni fogjuk). A szegmensek kezdőcímére is tettek azonban kikötést, mégpedig azt, hogy minden szegmens csak 16-tal osztható memóriacímen kezdődhet (ezek a címek az ú.n. *paragrafus-határok*). Ez annyit tesz, hogy minden szegmenscím alsó 4 bitje 0 lesz, ezeket tehát felesleges lenne eltárolni. Így marad pont 16 bit a szegmenscímből, azaz mind a szegmens-, mind az offszetcím 16 bites lett. Nézzünk egy példát:

1234h:5678h

A szegmenscím felső 16 bitje 1234h, ezért ezt balra shifteljük 4-szer, így kapjuk az 12340h-t. Ehhez hozzá kell adni az 5678h számot. A végeredmény 1 79B8h. Észrevehetjük, hogy ugyanazt a memóriarekeszt többféleképpen is megcímezhetjük, így pl. a 179Bh:0008h, 15AFh: 1EC8h, 130Dh:48E8h címek mind az előző helyre hivatkoznak. Ezek közül van egy kitüntetett, a 179Bh:0008h. Ezt a címet úgy alkottuk meg, hogy a lineáris cím alsó négy bitje (1000b=0008h) lesz az offszetcím, a felső 16 bit pedig a szegmenscímet alkotja. Ezért az ilyen címeket nevezhetjük bizonyos szempontból "normáltnak" (normalized address), hiszen az offszet itt mindig 0000h és 000Fh között lesz.

4.2 Regiszterek

A 8086-os proci összesen 14 db. 16 bites regiszterrel gazdálkodhat:

- **4 általános célú adatregiszter (AX, BX, CX, DX)**
- **2 indexregiszter (SI és DI)**
- **3 mutatóregiszter (SP, BP, IP)**
- **1 státuszregiszter vagy Flags regiszter (SR vagy Flags)**
- **4 szegmensregiszter (CS, DS, ES, SS)**

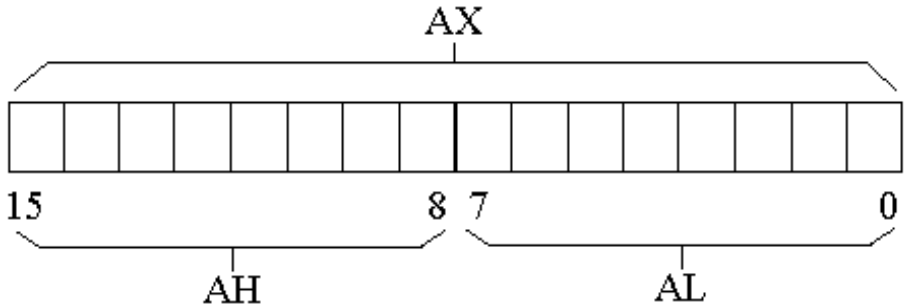
Most nézzük részletesebben:

- **AX (Accumulator) – sok aritmetikai utasítás használja a forrás és/vagy cél tárolására**
- **BX (Base) – memóriacímzésnél bázisként szolgálhat**
- **CX (Counter) – sok ismétléses utasítás használja számlálóként**
- **DX (Data) – I/O utasítások használják a portszám tárolására, ill. egyes aritmetikai utasítások számára is különös jelentőséggel bír**
- **SI (Source Index) – sztringkezelő utasítások használják a forrás sztring címének tárolására**
- **DI (Destination Index) – a cél sztring címét tartalmazza**
- **SP (Stack Pointer) – a verem tetejére mutat (azaz a verembe legutóbb berakott érték címét tartalmazza)**
- **BP (Base Pointer) – általános pointer, de alapesetben a verem egy elemét jelöli ki**
- **IP (Instruction Pointer) – a következő végrehajtandó utasítás memóriabeli címét tartalmazza; közvetlenül nem elérhető, de tartalma írható és olvasható is a vezérlésátadó utasításokkal**
- **SR avagy Flags (Status Register) – a processzor aktuális állapotát, az előző művelet eredményét mutató, ill. a proci működését befolyásoló biteket, ú.n. *flag-eket* (jelzőket) tartalmaz; szintén nem érhető el közvetlenül, de manipulálható különféle utasításokkal**
- **CS (Code Segment) – a végrehajtandó program kódját tartalmazó szegmens címe; nem állítható be közvetlenül, csak a vezérlésátadó utasítások módosíthatják**
- **DS (Data Segment) – az alapértelmezett, elsődleges adatterület szegmensének címe**
- **ES (Extra Segment) – másodlagos adatszegmens címe**
- **SS (Stack Segment) – a verem szegmensének címe**

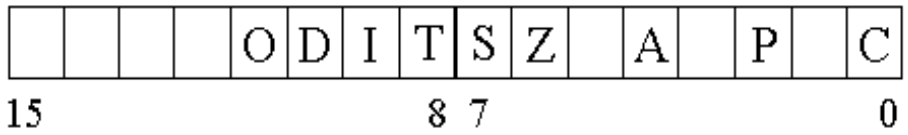
A négy általános célú regiszter (AX, BC, CX és DX) alsó és felső nyolc bitje (azaz alsó és felső bájtja) külön neveken érhető el: az alsó bájtokat az AL, BL, CL, DL, míg a felső bájtokat az AH, BH, CH, DH regiszterek jelölik (a rövidítésekben L=Low, H=High). Éppen ezért a következő (Pascal-szerű) műveletek:

```
AX:=1234h
AL:=78h
AH:=56h
```

végrehajtása után AX tartalma 5678h lesz, AH 56h-t, AL pedig 78h-t fog tartalmazni.



A Flags regiszter a következő mezőket tartalmazza:



Az egyes bitek a következő információt szolgáltatják:

- **C (Carry) – 1, ha volt aritmetikai átvitel az eredmény legfelső bitjénél (előjeltelen aritmetikai túlcsordulás), 0, ha nem**
- **P (Parity even) – 1, ha az eredmény legalsó bájtja páros számú 1-es bitet tartalmaz, különben 0**
- **A (Auxilliary carry/Adjust) – 1 jelzi, ha volt átvitel az eredmény 3. és 4. bitje között**
- **Z (Zero) – értéke 1, ha az eredmény zérus lett**
- **S (Sign) – értéke az eredmény legfelső bitjének, az előjelbitnek a tükörképe**

- **T (Trap) – ha 1, akkor a lépésenkénti végrehajtás (single-step execution) engedélyezve van**
- **I (Interrupt enable) – ha 1, akkor a maszkolható hardver-megszakítások engedélyezettek**
- **D (Direction) – ha 0, akkor a sztringutasítások növelik SI-t és/vagy DI-t, különben csökkentés történik**
- **O (Overflow) – ha 1, akkor előjeles aritmetikai túlcsordulás történt**

Ha hivatkozunk valamelyik flag-re, akkor a fenti betűjelekhez még hozzáírjuk az "F" betűt is. A CF, IF és DF flag-ek értékét közvetlenül is befolyásolhatjuk. Ezért pl. a CF nem csak akkor lehet 1, ha volt túlcsordulás (átvitel), hanem saját célból egyéb dolgot is jelezhet.

A Flags regiszter 1, 3, 5, 12, 13, 14 és 15 számú bitjei fenntartottak. Ezek értéke gyárilag rögzített, és nem is módosíthatjuk őket.

***Aritmetikai flag*-ek alatt, ha külön nem mondjuk, a CF, PF, AF, ZF, SF és OF flag-eket értjük.**

Ha azt akarjuk kifejezni, hogy két vagy több regiszter tartalmát egymás után fűzve akarunk megadni egy értéket, akkor az egyes regisztereket egymástól kettősponttal elválasztva soroljuk fel. Tehát pl. a DX:AX jelölés azt jelenti, hogy a 32 bites duplaszónak az alsó szava AX-ben, felső szava DX-ben van (a leírási sorrend megfelel a bitek sorrendjének a bájtokban). Röviden: a DX:AX *regiszterpár* (register pair) egy 32 bites duplaszót tartalmaz. Ez a jelölés azonban mást jelent, mint a memóriacímek ábrázolására használt SZEGMENS:OFFSZET felírások!

Ha *általános (célú) regiszterekről* beszélünk, akkor általában nemcsak az AX, BX, CX és DX regiszterekre gondolunk, hanem ezek 8 bites párjaira, továbbá az SI, DI, SP, BP regiszterekre is.

A következő végrehajtásra kerülő utasítás címét a CS:IP, míg a verem tetejét az SS:SP regiszterpárok által meghatározott értékek jelölik.

A CS és IP regisztereken kívül mindegyik regiszter tartalmazhat bármilyen értéket, tehát nem csak az eredeti funkciójának megfelelő tartalommal tölthetjük fel őket. Ennek ellenére az SS, SP és Flags regiszterek más célú használata nem javasolt.

4.3 Adattípusok

A szó méretét 2 bájtban állapították meg az Intel-nél, ezért pl. a BX és DI regiszterek szavasak, BL és DH bájtosak, míg az 1234 5678h szám egy duplaszó.

A processzor csak egészekkel tud dolgozni, azon belül ismeri az előjeles és előjeltelen aritmetikát is. A több bájt hosszú adatokat little-endian módon tárolja, ezért pl. a fenti 1234 5678h szám a következő módon lesz eltárolva: a legalacsonyabb memóriacímre kerül a 78h bájt, ezt követi az 56h, majd a 34h, végül pedig a 12h. Logikai, lebegőpontos racionális típusokat nem támogat a proci!

Az egészek részhalmazát alkotják a *binárisan kódolt decimális egész számok* (Binary Coded Decimal numbers–BCD numbers). Ezek két csoportba sorolhatók: vannak *pakolt* (packed) és *pakolatlan* (unpacked) *BCD* számok. (Használják még a "csomagolt" és "kicsomagolt" elnevezéseket is.) Pakolt esetben egy bájtban két decimális számjegyet tárolnak úgy, hogy a 4..7 bitek a szám magasabb helyiértékű jegyét, míg a 0..3 bitek az alacsonyabb helyiértékű jegyet tartalmazzák. A számjegyeket a 0h..9h értékek valamelyike jelöli. Pakolatlan esetben a bájtnak a felső fele kihasználatlan, így csak 1 jegyet tudunk 1 bájtban tárolni.

A *sztring* (string) adattípus bájtok vagy szavak véges hosszú folytonos sorát jelenti. Ezzel a típussal bővebben egy későbbi fejezetben foglalkozunk.

Speciális egész típus a *mutató* (pointer). Mutatónak hívunk egy értéket, ha az egy memóriacímet tartalmaz, vagy azt a memória elérésénél felhasználjuk. Két típusa van: a *közeli* vagy *rövid mutató* (near/short pointer) egy offszetcímet jelent, míg *távoli*, *hosszú* vagy *teljes mutatón* (far/long/full pointer) egy teljes logikai memóriacímet, tehát szegmens:offszet alakú címet értünk. A közeli mutató hossza 16 bit, a távolié pedig 32 bit. Fontos, hogy mutatóként bármilyen értéket felhasználhatunk.

4.4 Memóriahivatkozások, címzési módok

Láttuk, hogy a memória szervezése szegmentált módon történik. Most lássuk, ténylegesen hogy adhatunk meg memóriahivatkozásokat.

Azokat a, regiszterek és konstans számok (kifejezések) kombinációjából álló jelöléseket, amelyek az összes lehetséges szabályos memóriacímzési esetet reprezentálják, *címzési módoknak* (addressing mode) nevezzük. Több típusuk van, és mindenhol használható mindegyik, ahol valamilyen memória-operandust meg lehet adni. A memóriahivatkozás jelzésére a szögletes zárójeleket ([és]) használjuk.

4.4.1 Közvetlen címzés

A címzés alakja [offs16], ahol offs16 egy 16 bites abszolút, szegmensen belüli offszetet (rövid mutatót) jelöl.

4.4.2 Báziscímzés

A címzés egy bázisregisztert használ az offszet megadására. A lehetséges alakok: [BX], [BP]. A cél bájt offszetcímét a használt bázisregiszterből fogja venni a processzor.

4.4.3 Indexcímzés

Hasonló a báziscímzéshez, működését tekintve is annak tökéletes párja. Alakjai: [SI], [DI].

4.4.4 Bázis+relatív címzés

Ide a [BX+rel8], [BX+rel16], [BP+rel8], [BP+rel16] formájú címmegadások tartoznak. A rel16 egy előjeles szó, rel8 pedig egy előjeles bájt, amit a processzor előjelesen kiterjeszt szóvá. Ezek az ú.n. *eltolások* (displacement). Bármelyik változatnál a megcímzett bájt offszetjét a bázisregiszter tartalmának és az előjeles eltolásnak az összege adja meg.

4.4.5 Index+relatív címzés

Hasonlóan a báziscímzés/indexcímzés pároshoz, ez a bázis+relatív címzési mód párja. Alakjai: [SI+rel8], [SI+rel16], [DI+rel8], [DI+rel16].

4.4.6 Bázis+index címzés

A két nevezett címzési mód keveréke. A következő formákat öltheti: [BX+SI], [BX+DI], [BP+SI] és [BP+DI]. A regiszterek sorrendje természetesen közömbös, így [BP+SI] és

[SI+BP] ugyanazt jelenti. A cél bájt offszetjét a két regiszter értékének összegeként kapjuk meg.

4.4.7 Bázis+index+relatív címzés

Ez adja a legkombináltabb címzési lehetőségeket. Általános alakjuk [bázis+index+rel8] és [bázis+index+rel16] lehet, ahol bázis BX vagy BP, index SI vagy DI, rel8 és rel16 pedig előjeles bájt ill. szó lehet.

A bázis+relatív és index+relatív címzési módok másik elnevezése a *bázisrelatív* ill. *indexrelatív címzés*.

Minden címzési módhoz tartozik egy alapértelmezett (default) szegmensregiszter-előírás. Ez a következőket jelenti:

- **bármely, BP-t NEM tartalmazó címzési mód a DS-t fogja használni**
- **a BP-t tartalmazó címzési módok SS-t használják**

Ha az alapértelmezett beállítás nem tetszik, akkor mi is előírhatjuk, hogy melyik szegmensregisztert használja a cím meghatározásához a proci. Ezt a *szegmensfelülbíráló prefixekkel* tehetjük meg. A prefixek alakja a következő: CS:, DS:, ES:, SS:, tehát a szegmensregiszter neve plusz egy kettőspont. Ezeket a prefixeket legjobb közvetlenül a címzési mód jelölése elé írni.

Most lássunk néhány példát szabályos címzésekre:

- **[12h*34h+56h]**
- **ES:[09D3h]**
- **CS:[SI-500d]**
- **SS:[BX+DI+1999d]**
- **DS:[BP+SI]**

A következő jelölések viszont hibásak:

- **[AX]**
- **[CX+1234h]**
- **[123456789ABCh]**
- **[SP+BP]**
- **[IP]**

A használt címzésmód által hivatkozott memóriacímet *tényleges címnek* (effective address) nevezzük.

4.5 Veremkezelés

***Vermen* (stack) a következő tulajdonságokkal rendelkező adatszerkezetet értjük:**

- **értelmezve van rajta egy Push művelet, ami egy értéket rak be a verem tetejére**
- **egy másik művelet, a Pop a verem tetején levő adatot olvassa ki és törli a veremből**
- **működése a *LIFO* (Last In First Out) elvet követi, azaz a legutoljára betett adatot vehetjük ki először a veremből**
- **verem méretén azt a számot értjük, ami meghatározza, hogy maximálisan mennyi adatot képes eltárolni**
- **verem magassága a benne levő adatok számát jelenti**
- **ha a verem magassága 0, akkor a verem üres**
- **ha a verem magassága eléri a verem méretét, akkor a verem tele van**
- **üres veremből nem vehetünk ki adatot**
- **teli verembe nem rakhatunk be több adatot**

Az Intel 8086-os processzor esetén a verem mérete maximálisan 64 Kbájt lehet. Ez annak a következménye, hogy a verem csak egy szegmensnyi területet foglalhat el, a szegmensek mérete pedig szintén 64 Kbájt. A verem szegmensét az SS

regiszter tárolja, míg a verem tetejére az SP mutat. Ez a verem *lefelé bővülő* (expand-down) verem, ami azt jelenti, hogy az újonnan betett adatok egyre alacsonyabb memóriacímen helyezkednek el, és így SP értéke is folyamatosan csökken a Push műveletek hatására. SP-nek mindig páros értéket kell tartalmaznia, és a verembe betenni ill. onnan kiolvasni szintén csak páros számú bájtot lehet.

A Push és Pop műveleteknek megfelelő utasítások mnemonikja szintén PUSH ill. POP, így megjegyzésük nem túl nehéz. A PUSH először csökkenti SP-t a betenni kívánt adat méretének megfelelő számú bájttal, majd az SS:SP által mutatott memóriacímre berakja a kérdéses értéket. A POP ezzel ellentétes működésű, azaz először kiolvas az SS:SP címről valahány számú bájtot, majd SP-hez hozzáadja a kiolvasott bájtok számát.

Nézzünk egy példát a verem működésének szemléltetésére! Legyen AX tartalma 1212h, BX tartalma 3434h, CX pedig legyen egyenlő 5656h-val. Tekintsük a következő Assembly programrészletet:

```
PUSH    AX
PUSH    BX
POP     AX
PUSH    CX
POP     BX
POP     CX
```

SP legyen 0100h, SS értéke most közömbös. A verem és a regiszterek állapotát mutatják a következő ábrák az egyes utasítások végrehajtása után:

Kezdetben:

	SS:0102h : ??
SP=>	**SS:0100h : ??**

	SS:00FEh : ?? SS:00FCh : ?? SS:00FAh : ??

AX=1212h, BX=3434h, CX=5656h

***PUSH AX** után:*

	SS:0102h : ?? SS:0100h : ??
SP=>	SS:00FEh : 1212h SS:00FCh : ?? SS:00FAh : ??

AX=1212h, BX=3434h, CX=5656h

***PUSH BX** után:*

	SS:0102h : ?? SS:0100h : ?? SS:00FEh : 1212h
SP=>	SS:00FCh : 3434h SS:00FAh : ??

AX=1212h, BX=3434h, CX=5656h

***POP AX** után:*

	SS:0102h : ?? SS:0100h : ??
SP=>	SS:00FEh : 1212h SS:00FCh : 3434h SS:00FAh : ??

AX=3434h, BX=3434h, CX=5656h

***PUSH CX** után:*

	SS:0102h : ?? SS:0100h : ?? SS:00FEh : 1212h
SP=>	SS:00FCh : 5656h

	SS:00FAh : ??

AX=3434h, BX=3434h, CX=5656h

***POP BX** után:*

	SS:0102h : ??
	SS:0100h : ??
SP=>	SS:00FEh : 1212h
	SS:00FCh : 5656h
	SS:00FAh : ??

AX=3434h, BX=5656h, CX=5656h

***POP CX** után:*

	SS:0102h : ??
SP=>	SS:0100h : ??
	SS:00FEh : 1212h
	SS:00FCh : 5656h
	SS:00FAh : ??

AX=3434h, BX=5656h, CX=1212h

Az utasítások végrehajtása tehát azt eredményezi, hogy AX felveszi BX értékét, BX a CX értékét, CX pedig az AX eredeti értékét (olyan, mintha "körbeforgattuk" volna a regiszterek tartalmát egymás között). Mivel ugyanannyi Push volt, mint amennyi Pop, SP értéke a végrehajtás után visszaállt a kezdetire.

A verem sok dologra használható a programozás során:

- **regiszterek, változók értékének megcserélésére**
- **regiszterek ideiglenes tárolására**
- **lokális változók elhelyezésére**
- **eljáráshíváskor az argumentumok átadására, ill. a visszatérési érték tárolására**

A veremregisztereknek (SS és SP) mindig érvényes értékeket kell tartalmazniuk a program futásakor, mivel azt a processzor és más programok (pl. a hardver-megszakításokat lekezelő rutinok) is használják.

4.6 I/O, megszakítás-rendszer

Ez a processzor 16 biten ábrázolja a portszámokat, így összesen 65536 db. portot érhetünk el a különféle I/O (Input/Output=ki-/bemeneti) utasításokkal. Ezek közül a 0000h és 00FFh közötti tartományt az alaplapi eszközök használják, és mivel a legfontosabb perifériák címét rögzítették, a különböző PC-s rendszerekben ezeket ugyanazon a porton érhetjük el. Bármelyik porton kezdeményezhetünk olvasást és írást egyaránt, de ezzel azért nem árt vigyázni, egy félresikeredett írással ugyanis könnyen "megadásra" késztethetjük a számítógépet.

Egy megszakítást kiválthat egy hardvereszköz (ez a *hardver-megszakítás*), de a felhasználó és a programok is elindíthatnak ilyen eseményt (ez a *szoftver-megszakítás*). A megszakítások két típusa nem csak a számukban, de tulajdonságaikban is eltér egymástól.

A hardver-megszakításokat a megszakítás-vezérlőn keresztül érzékeli a processzor. A vezérlő 8 különböző megszakítás vonalat tart fent (AT-k esetén két vezérlőt kötöttek sorba, így lett a 8-ból 16 vonal). Ezen megszakításokat az IRQ0, IRQ1, …, IRQ7 (továbbá IRQ8, …, IRQ15) szimbólumokkal jelöljük. Fontos jellemzőjük, hogy a program futásával párhuzamosan aszinkron keletkeznek, és hogy *maszkolhatók*. Ez utóbbi fogalom azt takarja, hogy az egyes IRQ vonalakat egyenként engedélyezhetjük ill. tilthatjuk le. Ha egy vonal tiltva (másképpen maszkolva) van, akkor az arra vonatkozó kérések

nem jutnak el a processzorhoz. Ezen kívül a processzor Flags regiszterének IF bitjével az összes hardver-megszakítás érzékelését letilthatjuk ill. újra engedélyezhetjük.

Ezzel szemben szoftver-megszakításból 256 darab van, jelölésükre az INT 00h, …, INT 0FFh kifejezéseket használjuk. A program futása során szinkron keletkeznek (hiszen egy gépi kódú utasítás váltja ki őket), és nem maszkolhatók, nem tilthatók le. Az IF flag állásától függetlenül mindig érzékeli őket a CPU.

A két megszakítás típus lekezelésében közös, hogy a CPU egy speciális rutint (programrészt) hajt végre a megszakítás detektálása után. Ez a rutin a *megszakítás-kezelő* (interrupt handler). Az operációs rendszer felállása után sok hardver- és szoftver-megszakításnak már van kezelője (pl. a BIOS is jópár ilyen megszakítást üzemeltet), de lehetőség van arra is, hogy bármelyik ilyen rutint lecseréljük egy saját programunkra.

A hardver-megszakítások a szoftveresek egy adott tartományára képződnek le, ez alapállapotban az INT 08h, …, INT 0Fh (IRQ8-tól IRQ15-ig pedig az INT 70h, …, INT 77h) megszakításokat jelenti, tehát ezek a megszakítások másra nem használhatók. Így pl. IRQ4 keletkezésekor a processzor az INT 0Ch kezelőjét hívja meg, ha IF=1 és az IRQ4 nincs maszkolva.

A megszakítások egy speciális csoportját alkotják azok, amiket valamilyen vészhelyzet, súlyos programhiba vált ki. Ezeket összefoglalóan *kivételnek* (exception) nevezzük. Kivételt okoz pl. ha nullával próbálunk osztani, vagy ha a verem túl- vagy alulcsordul.

5 AZ ASSEMBLY NYELV SZERKEZETE, SZINTAXISA

Az Assembly nyelven megírt programunkat egy szöveges forrásfájlban tároljuk, amit aztán egy erre specializált fordítóprogrammal átalakítunk *tárgykóddá* (object code). A tárgykód még nem futtatható közvetlenül, ezért a *szerkesztő* (linker) elvégzi rajta az átalakításokat. Végül az egy vagy több tárgykód összefűzése után kialakul a futtatható program. Az említett fordítóprogramot *assemblernek* nevezzük. A forrásfájlok állománynév-kiterjesztése általában .ASM vagy .INC, a tárgykódé .OBJ, a futtatható fájloké pedig .COM vagy .EXE. Ezek az elnevezések a PC-s DOS rendszerekre vonatkoznak, más operációs rendszerek alatt (pl. UNIX, OS/2, Linux) ezek eltérhetnek. Az Assembly forrásfájloknak tiszta szöveges (text) állományoknak kell lennie, mert az assemblerek nem szeretik a mindenféle "bináris szemetet" tartalmazó dokumentumokat.

Az Assembly nem érzékeny a kis- és nagybetűkre, így tehát a PUSH, Push, push, PusH szavak ugyanazt a jelentést képviselik. Ez elég nagy szabadságot ad a különböző azonosítók elnevezésére, és a forrást is áttekinthetőbbé teszi.

Az Assembly forrás sorai a következő elemeket tartalmazhatják:

- **assembler utasítások (főleg ezek alkotják a tényleges program kódját)**
- **pszeudo utasítások (pl. makrók, helyettesítő szimbólumok)**
- **assembler direktívák**

A *direktívák* (directive) olyan, az assemblernek szóló utasítások, melyek közvetlenül gépi kódot nem feltétlenül generálnak, viszont a fordítás menetét, az elkészült kódot befolyásolják.

Az érvényes assembler utasítások szintaxisa a következő:

{Címke:} <Prefix> {Utasítás {Operandus<,Operandus>}} {;Megjegyzés}

A kapcsos zárójelek ({ és }) az opcionális (nem kötelező) részeket jelzik, míg a csúcsos zárójelek (< és >) 0 vagy több előfordulásra utalnak.

A *címke* (label) deklarációja az azonosítójából és az azt követő kettőspontból áll. Értéke az aktuális sor memóriacíme (helyesebben offszetje). Szerepe azért fontos, mert segítségével könnyen írhatjuk le a különféle vezérlési szerkezeteket. A gépi kódú utasítások ugyanis relatív vagy abszolút memóriacímekkel dolgoznak, nekünk viszont kényelmesebb, ha egy szimbólummal (a címke nevével) hivatkozunk a kérdéses sorra. A tényleges cím megállapítása a szerkesztő feladata lesz.

A *prefix* olyan utasításelem, amely csak az őt követő gépi kódú utasításra (avagy Assembly mnemonikra) hat, annak működését változtatja meg ill. egészíti ki.

Ezeket követi az *utasítás* (instruction), ami egy assembler direktíva, egy pszeudo utasítás vagy egy Assembly mnemonik lehet. Emlékeztetőül, a *mnemonik* (mnemonic) azonos műveletet végző különböző gépi kódú utasítások csoportját jelölő szimbólum (szó). Az Intel mnemonikok legalább kétbetűsek, és némelyik még számot is tartalmaz. A mnemonikok gépi kódú alakját *műveleti kódnak* (operation code, opcode) nevezzük.

Az mnemonik által meghatározott utasítás valamilyen "dolgokon" hajtja végre a feladatát. Ezeket a dolgokat mutatják az *operandusok* (operand), amelyeket (ha több van) vesszővel elválasztva sorolunk fel. Operandusok építőkövei a következők lehetnek:

- **regiszterek**
- **numerikus (szám) konstansok**
- **karakteres (sztring) konstansok**
- **szimbólumok**
- **operátorok (műveletek)**

Regiszterekre a következő szavakkal utalhatunk: AL, AH, AX, BL, BH, BX, CL, CH, CX, DL, DH, DX, SI, DI, SP, BP, CS, DS, ES és SS.

Numerikus konstansnak **számjeggyel kell kezdődnie, azután tartalmazhatja a tíz számjegyet ill. az A..F betűket, a végén pedig a számrendszerre utaló jelölés állhat. Alapesetben minden számot decimálisnak értelmez az assembler, kivéve ha:**

- **átállítottuk az alapértelmezést a RADIX direktívával**
- **a szám végén a "d" (decimális), "b" (bináris), "h" (hexadecimális) vagy "o" (oktális) betű áll**

Fontos, hogy a betűvel kezdődő hexadecimális számok elé tegyünk legalább egy 0-t, mert különben szimbólumnak próbálná értelmezni a számunkat az assembler.

Karakteres konstansokat **aposztrófok vagy idézőjelek között adhatunk meg. Ha numerikus konstans helyett állnak, akkor lehetséges hosszuk az adott direktívától (pl. a DW esetén 2 bájt) ill. a másik operandustól függ.**

Szimbólumok **például a konstans azonosítók, változók, címkék, szegmensek, eljárások nevei. Szimbólum azonosítójának mindig betűvel kell kezdődnie, azután tartalmazhat számjegyet, betűket (csak az angol ábécé betűit), aláhúzásjelet (_), dollárjelet ($) és "kukacot" (@).**

Speciális szimbólum az egymagában álló dollárjel, ennek neve pozíció-számláló (location counter). Értéke az aktuális sor szegmensbeli (avagy szegmenscsoportbeli) offszetje.

Operátorból **rengeteg van, a legfontosabbak talán a következők:**

- **kerek zárójelek**
- **szokásos aritmetikai műveleti jelek (+, -, *, /, MOD)**
- **mezőkiválasztó operátor (.)**
- **szegmens-előírás/-felülbírálás (:)**
- **memóriahivatkozás ([…])**
- **bitenkénti logikai műveletek (AND, OR, XOR, NOT)**
- **típusfelülbírálás (PTR)**

- **adattípus megadás (BYTE, WORD, DWORD)**
- **duplikáló operátor (DUP)**
- **relációs operátorok (EQ, NE, GT, GE, LT, LE)**
- **pointer típusok (NEAR, FAR)**
- **szegmens/offszet lekérdezés (SEG, OFFSET)**
- **léptető operátorok (SHL, SHR)**
- **méretlekérdező operátorok (LENGTH, SIZE, TYPE, WIDTH)**

Az operandusok ezek alapján három csoportba sorolhatók: regiszter-, memória- és konstans (közvetlen értékű) operandusok. A különböző opcode-ok lehetnek nulla-, egy-, kettő-, három- vagy négyoperandusúak.

Fontos megjegyezni, hogy a kétoperandusú mnemonikok első operandusa a CÉL, a második pedig a FORRÁS. Ezt a konvenciót egyébként csak az Intel Assembly használja, az összes többi (pl. Motorola) Assemblyben az első a forrás, a második pedig a cél operandus. Az elnevezések arra utalnak, hogy a művelet eredménye a cél–operandusban tárolódik el. Ha szükséges, a céloperandus második forrásként is szolgálhat. Egy operandust használó utasításoknál az az egy operandus sokszor egyben forrás és cél is.

Végül az assembler utasítás sorának végén *megjegyzést* is írhatunk egy pontosvessző után. Ezeket a karaktereket az assembler egyszerűen átugorja, így tehát bármilyen szöveget tartalmazhat (még ékezetes betűket is).

Most tisztáznunk kell még néhány fogalmat. Korábban már említettük, hogy a processzor szegmentált memória-modellt használ. Hogy teljes legyen a kavarodás, az assemblernek is elő kell írnunk a memória-modellt, és szegmenseket is létre kell hoznunk. A két szegmens- és memória-modell fogalmak azonban egy picit különböznek.

Az assemblerben létrehozott szegmenseknek van nevük (azonosítójuk, ami tehát egy szimbólum), méretük legfeljebb 64 Kbájt lehet, és nem feltétlenül kell összefüggőnek lennie egy szegmens definíciójának. Ez utóbbi tulajdonság több szabadságot enged a programozónak, mivel megteheti, hogy elkezd egy szegmenst, majd definiál egy másikat, azután megint visszatér az első szegmensbe. Az azonos néven megkezdett szegmensdefiníciókat az assembler és a szerkesztő szépen összefűzi egyetlen darabbá, tehát ez a jelölésrendszer a processzor számára átlátszó lesz. Másrészt a méretre vonatkozó kijelentés mindössze annyit tesz, hogy ha nem töltünk ki teljesen (kóddal vagy adatokkal) egy memóriaszegmenst (ami viszont 64 Kbájt nagyságú), akkor a fennmaradó bájtokkal az assembler nem foglalkozik, a program futásakor ezek valami "szemetet" fognak tartalmazni.

A memória-modell előírása akkor fontos, ha nem akarunk foglalkozni különböző szegmensek (adat-, kód-, verem- stb.) létrehozásával, és ezt a folyamatot az assemblerre akarjuk hagyni. Ehhez viszont jeleznünk kell az assemblernek, hogy előreláthatólag mekkora lesz a programunk memóriaigénye, hány és milyen típusú szegmensekre van szükségünk. Ezt a típusú szegmensmegadást *egyszerűsített szegmensdefiníciónak* (simplified segment definition) nevezzük.

A két módszert keverve is használhatjuk kedvünk és igényeink szerint, nem fogják egymást zavarni (mi is ezt fogjuk tenni). Ha magas szintű nyelvhez írunk külső Assembly rutinokat, akkor az egyszerűsített szegmensmegadás sokszor egyszerűbb megoldást szolgáltat.

Most pedig lássunk néhány fontos direktívát:

- **modul/forrásfájl lezárása (END)**
- **szegmens definiálása (SEGMENT…ENDS)**
- **szegmenscsoport definiálása (GROUP)**

- **szegmens hozzárendelése egy szegmensregiszterhez (ASSUME)**
- **értékadás a $ szimbólumnak (ORG)**
- **memória-modell megadása (MODEL)**
- **egyszerűsített szegmensdefiníciók (CODESEG, CONST, DATASEG, FARDATA, STACK, UDATASEG, UFARDATA, .CODE, .DATA, .STACK)**
- **helyfoglalás (változó létrehozása) (DB, DW, DD, DF, DQ, DT)**
- **konstans/helyettesítő szimbólum létrehozása (=, EQU)**
- **eljárás definiálása (PROC…ENDP)**
- **külső szimbólum definiálása (EXTRN)**
- **szimbólum láthatóvá tétele a külvilág számára (PUBLIC)**
- **feltételes fordítás előírása (IF, IFccc, ELSE, ELSEIF, ENDIF)**
- **külső forrásfájl beszúrása az aktuális pozícióba (INCLUDE)**
- **felhasználói típus definiálása (TYPEDEF)**
- **struktúra, unió, rekord definiálása (STRUC…ENDS, UNION…ENDS, RECORD)**
- **a számrendszer alapjának átállítása (RADIX)**
- **makródefiníció (MACRO…ENDM)**
- **makróműveletek (EXITM, IRP, IRPC, PURGE, REPT, WHILE)**
- **utasításkészlet meghatározása (P8086, P186, P286, P386 stb.; .8086, .186, .286, .386 stb.)**

A direktívák használatára később még visszatérünk.

Assemblerből több is létezik PC-re, ilyenek pl. a Microsoft Macro Assembler (MASM), Turbo Assembler (TASM), Netwide Assembler (NASM), Optimizing Assembler

(OPTASM). Ebben a jegyzetben a programokat a TASM jelölésmódban írjuk le, de több-kevesebb átírással másik assembler alá is átvihetők a források. A TASM specialitásait nem fogjuk kihasználni, ahol nem muszáj. A TASM saját linkert használ, ennek neve Turbo Linker (TLINK).

6 A 8086-OS PROCESSZOR UTASÍTÁSKÉSZLETE

Itt az ideje, hogy megismerkedjünk a kiválasztott processzorunk által ismert utasításokkal. Az utasítások közös

tulajdonsága, hogy az operandusoknak teljesíteniük kell a következő feltételeket:

- **Ha egyetlen operandusunk van, akkor az általában csak az általános regiszterek közül kerülhet ki, vagy pedig memóriahivatkozás lehet (néhány esetben azonban lehet konstans vagy szegmensregiszter is).**
- **Két operandus esetén bonyolultabb a helyzet: mindkét operandus egyszerre nem lehet szegmensregiszter, memóriahivatkozás és közvetlen (konstans) érték, továbbá szegmensregiszter mellett vagy általános regiszternek vagy memóriahivatkozásnak kell szerepelnie. (A sztring-utasítások kivételek, ott mindkét operandus memóriahivatkozás lesz, de erről majd később.) A két operandus méretének majdnem mindig meg kell egyeznie, ez alól csak néhány speciális eset kivétel.**

6.1 Prefixek

Először lássuk a prefixeket, a gépi kódjukkal együtt:

6.1.1 Szegmensfelülbíráló prefixek

Ezek a CS: (2Eh), DS: (3Eh), ES: (26h) és SS: (36h). A TASM ezeken kívül ismeri a SEGCS, SEGDS, SEGES és SEGSS mnemonikokat is. A SCAS és a STOS utasítások kivételével bármely más utasítás előtt megadva őket az alapértelmezés szerinti szegmensregiszter helyett használandó regisztert adják meg. Természetesen ez csak a valamilyen memóriaoperandust használó utasításokra vonatkozik.

6.1.2 Buszlezáró prefix

Mnemonikja LOCK (0F0h). Hatására az adott utasítás által módosított memóriarekeszt az utasítás végrehajtásának idejére a processzor zárolja úgy, hogy a buszokhoz más hardverelem nem férhet hozzá a művelet lefolyásáig. Csak a következő utasítások esetén használható: ADD, ADC, SUB, SBB, AND, OR, XOR, NOT, NEG, INC, DEC, XCHG, más esetekben hatása definiálatlan.

6.1.3 Sztringutasítást ismétlő prefixek

Lehetséges alakjaik: REP/REPE/REPZ (0F3h), REPNE/REPNZ (0F2h). Csak a sztringkezelő utasítások előtt használhatók, más esetben következményük kiszámíthatatlan. Működésükről és használatukról később szólunk.

6.2 Utasítások

Most jöjjenek az utasítások, típus szerint csoportosítva:

6.2.1 Adatmozgató utasítások

- **MOV – adatok mozgatása**
- **XCHG – adatok cseréje**
- **PUSH – adat betétele a verembe**
- **PUSHF – Flags regiszter betétele a verembe**
- **POP – adat kivétele a veremből**
- **POPF – Flags regiszter kivétele a veremből**
- **IN – adat olvasása portról**
- **OUT – adat kiírása portra**
- **LEA – tényleges memóriacím betöltése**
- **LDS, LES – teljes pointer betöltése szegmensregiszter: általános regiszter regiszterpárba**
- **CBW – AL előjeles kiterjesztése AX-be**
- **CWD – AX előjeles kiterjesztése DX:AX-be**
- **XLAT/XLATB – AL lefordítása a DS:BX című fordító táblázattal**
- **LAHF – Flags alsó bájtjának betöltése AH-ba**
- **SAHF – AH betöltése Flags alsó bájtjába**
- **CLC – CF flag törlése**
- **CMC – CF flag komplementálása (invertálása)**
- **STC – CF flag beállítása**
- **CLD – DF flag törlése**
- **STD – DF flag beállítása**
- **CLI – IF flag törlése**
- **STI – IF flag beállítása**

6.2.2 Egész számos aritmetika

- **ADD – összeadás**
- **ADC – összeadás átvitellel (CF) együtt**
- **SUB – kivonás**
- **SBB – kivonás átvitellel (CF) együtt**
- **CMP – összehasonlítás (flag-ek beállítása a cél és a forrás különbségének megfelelően)**
- **INC – inkrementálás (növelés 1-gyel), CF nem változik**
- **DEC – dekrementálás (csökkentés 1-gyel), CF nem változik**
- **NEG – kettes komplemens képzés (szorzás -1-gyel)**
- **MUL – előjeltelen szorzás**
- **IMUL – előjeles szorzás**
- **DIV – előjeltelen maradékos osztás**
- **IDIV – előjeles maradékos osztás**

6.2.3 Bitenkénti logikai utasítások (Boole-műveletek)

- **AND – logikai AND**
- **OR – logikai OR**
- **XOR – logikai XOR**
- **NOT – logikai NOT (egyes komplemens képzés)**
- **TEST – logikai bit tesztelés (flag-ek beállítása a két op. logikai AND-jének megfelelően)**

6.2.4 Bitléptető utasítások

- **SHL – előjeltelen léptetés (shiftelés) balra**
- **SAL – előjeles léptetés balra (ugyanaz, mint SHL)**
- **SHR – előjeltelen léptetés jobbra**
- **SAR – előjeles (aritmetikai) léptetés jobbra**
- **ROL – balra forgatás (rotálás)**
- **RCL – balra forgatás CF-en át**

- **ROR – jobbra forgatás**
- **RCR – jobbra forgatás CF-en át**

6.2.5 Sztringkezelő utasítások

- **MOVS, MOVSB, MOVSW – sztring mozgatása**
- **CMPS, CMPSB, CMPSW – sztringek összehasonlítása**
- **SCAS, SCASB, SCASW – keresés sztringben**
- **LODS, LODSB, LODSW – betöltés sztringből**
- **STOS, STOSB, STOSW – tárolás sztringbe**

6.2.6 Binárisan kódolt decimális (BCD) aritmetika

- **AAA – ASCII igazítás összeadás után**
- **AAS – ASCII igazítás kivonás után**
- **AAM – ASCII igazítás szorzás után**
- **AAD – ASCII igazítás osztás előtt**
- **DAA – BCD rendezés összeadás után**
- **DAS – BCD rendezés kivonás után**

6.2.7 Vezérlésátadó utasítások

- **JMP – feltétel nélküli ugrás**
- **JCXZ – ugrás, ha CX=0000h**
- **Jccc – feltételes ugrás ("ccc" egy feltételt ír le)**
- **LOOP – CX dekrementálása és ugrás, ha CX≠0000h**
- **LOOPE, LOOPZ – CX dekrementálása és ugrás, ha ZF=1 és CX≠0000h**
- **LOOPNE, LOOPNZ – CX dekrementálása és ugrás, ha ZF=0 és CX≠0000h**
- **CALL – eljárás (szubrutin) hívása**
- **RET, RETF – visszatérés szubrutinból**
- **INT – szoftver-megszakítás kérése**

- **INTO – INT 04h hívása, ha OF=1, különben NOP-nak felel meg**
- **IRET – visszatérés megszakításból**

6.2.8 Rendszervezérlő utasítások

- **HLT – processzor leállítása amíg megszakítás (vagy reset) nem érkezik**

6.2.9 Koprocesszor-vezérlő utasítások

- **WAIT – adatszinkronizálás a koprocesszorral**
- **ESC – utasítás küldése a koprocesszornak**

6.2.10 Speciális utasítások

- **NOP – üres utasítás (igazából XCHG AX, AX), kódja 90h**

Az adatmozgató utasítások a leggyakrabban használt utasítások kategóriáját alkotják. Ez természetes, hiszen más (magasabb szintű) programozási nyelvekben is sokszor előfordul, hogy valamilyen adatot olvasunk ki vagy töltünk be egy változóba, avagy valamelyik portra. Kicsit kilógnak ebből a sorból a CBW, CWD, XLAT utasítások, de mondjuk a flag-eket állító CLC, CMC stb. utasítások is kerülhettek volna másik kategóriába.

Az egész aritmetikás utasításokat szintén nagyon sokszor használjuk programozáskor. A CMP utasításnak pl. kulcsszerepe lesz a különféle elágazások (feltételes vezérlési szerkezetek) lekódolásában.

A Boole-műveletekkel mindenféle bitműveletet (beállítás, maszkolás, tesztelés) el tudunk végezni, a munkák során éppen ezért nélkülözhetetlenek.

Shiftelő műveleteket főleg a már említett, 2 hatványával való szorzás gyors megvalósítására alkalmazunk. Ezenkívül nagy precizitású, saját aritmetikai műveletek fejlesztésekor is fontos szerepet kapnak a rotáló utasításokkal együtt.

***Sztringen* (string) itt bájtok vagy szavak véges hosszú folytonos sorát értjük. A sztringeken operáló utasítások pl. a nagy mennyiségű adatok másolásánál, vizsgálatánál segítenek.**

BCD aritmetikát csak az emberi kényelem miatt támogat a processzor. Ezek az utasítások két csoportra oszthatók: pakolatlan (AAA, AAD, AAM, AAS) és pakolt (DAA,DAS) BCD aritmetikára. Kifejezetten ritkán használjuk őket, bár ez elég szubjektív kijelentés.

A vezérlésátadó utasítások a programok egyik alappillérét alkotják, helyes használatuk a jól működő algoritmus alapfeltétele.

Rendszer- és koprocesszor-vezérlő utasításokat csak speciális esetekben, külső hardverrel való kapcsolatfenntartásra és kommunikációra szokás használni.

Különleges "csemege" a NOP (No OPeration), ami eredete szerint adatmozgató utasítás lenne (mivel az XCHG AX,AX utasítás álneve), viszont működését tekintve gyakorlatilag nem csinál semmit. Hasznos lehet, ha a kódba olyan üres bájtokat akarunk beszúrni, amit később esetleg valami egyéb célra használunk majd, de nem akarjuk, hogy a processzor azt valamilyen utasításnak értelmezve nemkívánatos mellékhatás lépjen fel. A NOP kódja egyetlen bájt (90h).

Szegmensregiszter-operandust (ami nem egyenlő a szegmensfelülbíráló prefixszel) a fenti utasítások közül csak a MOV, PUSH és POP kaphat, de ezzel egyelőre ne foglalkozzunk, később még visszatérünk rájuk.

7 VEZÉRLÉSI SZERKEZETEK MEGVALÓSÍTÁSA

Most megnézzük, hogy kódolható le néhány gyakran használt vezérlési szerkezet Assemblyben. Előtte azonban ejtünk néhány szót a programírás-fordítás-szerkesztés menetéről.

A programok forrásállományának a könnyebb azonosítás végett (és a hagyományok betartása érdekében is) célszerű egy .ASM kiterjesztésű nevet adni. A forrást bármilyen szövegszerkesztővel (pl. a DOS-os EDIT, valamilyen commander program szerkesztője, Windows NotePad-je stb.) elkészíthetjük, csak az a fontos, hogy a kimentett állomány ne tartalmazzon mindenféle formázási információt, képeket stb., pusztán a forrás szövegét.

Ha ez megvan, akkor jöhet a fordítás. A TASM program szintaxisa a következő:

TASM {opciók} forrás{,tárgykód}{,lista}{,xref}

Az opciók különféle kapcsolók és paraméterek, amik a fordítás menetét és a generált fájlok tartalmát befolyásolják. A kimeneti állomány tárgykód (object, .OBJ kiterjesztéssel) formátumú, és alapesetben a fájl neve megegyezik a forrás nevével. Ha akarjuk, ezt megváltoztathatjuk. Kérhetjük az assemblert, hogy a fordítás végeztével készítsen listát és/vagy keresztreferencia-táblázatot (cross reference table, xref). A listában a forrás minden egyes sorához generált gépi kód fel van tüntetve, ezenkívül tartalmazza a definiált szegmensek adatai, valamint a használt szimbólumokat összegyűjtő szimbólumtáblát. A keresztreferencia-táblázat tartalmazza, hogy a különböző

szimbólumokat hol definiálták, ill. mely helyeken hivatkoztak rájuk. Ez a fájl bináris (tehát nem szöveges), értelmes információt belőle a TCREF programmal nyerhetünk.

Legtöbbször csak a lefordítandó forrás nevét adjuk meg a TASM-nak, hacsak nincs valamilyen különleges igényünk. Néhány fontos kapcsolót azért felsorolunk:

- **/a, /s – A szegmensek szerkesztési sorrendjét befolyásolják. Az első ábécé rend szerint, míg a második a definiálás sorrendje szerinti szegmens-sorrendet ír elő. Alapértelmezés a /s.**
- **/l, /la – A listázás formátumát befolyásolják: normál (/l) ill. kibővített (/la) lista fog készülni. Alapértelmezés a /l.**
- **/m# – A fordítás # menetes lesz. Alapérték a 2.**
- **/z – Hiba esetén a forrás megfelelő sorát is kiírja.**
- **/zi, /zd, /zn – A tárgykódba bekerülő nyomkövetési információt befolyásolják: teljes (/zi), csak a programsorok címei (/zd) vagy semmi (/zn). Alapértelmezés a /zn.**

Miután mindegyik forrást lefordítottuk tárgykódra, jöhet a szerkesztés. A TLINK is számos argumentummal indítható:

TLINK {opciók} tárgykód fájlok{, futtatható fájl}{, térkép}{, könyvtárak}

Az opciók esetleges megadása után kell felsorolni azoknak a tárgykódú állományoknak a neveit, amiket egy futtatható fájlba akarunk összeszerkeszteni. A főprogram tárgykódját kell legelsőnek megadnunk. A futtatható fájl nevét kívánságunk szerint megadhatjuk, különben a legelső tárgykódú fájl (a főprogram) nevét fogja kapni. A térképfájl (map file) az egyes modulokban definiált szegmensek, szimbólumok adatait tartalmazza, valamint külön kérésre azoknak a programsoroknak a sorszámait, amikhez kód került lefordításra. A könyvtárakban (library) több tárgykódú állomány lehet összegyűjtve, a

program írásakor tehát ezekből is használhatunk rutinokat, ha akarunk.

Most lássunk néhány megadható opciót:

- **/x, /m, /l, /s – A térkép (map) tartalmát befolyásolják: nincs térkép (/x), publikus szimbólumok listája lesz (/m), kódhoz tartozó sorszámok lesznek (/l), szegmensinformáció lesz (/s).**
- **/c – Érzékeny lesz a kisbetűkre és nagybetűkre.**
- **/v – Engedélyezi a nyomkövetési információk beépítését a futtatható fájlba.**
- **/t – .EXE helyett .COM típusú futtatható fájlt készít.**

7.1 Szekvenciális vezérlési szerkezet

A processzor mindig szekvenciális utasítás-végrehajtást alkalmaz, hacsak nem használunk valamilyen vezérlésátadó utasítást. Lássuk hát első Assembly programunkat, ami nem sok hasznos dolgot csinál, de kezdésnek megteszi.

Pelda1.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Kar         DB      'a'
Szo         DW      "Ha"
Szam        DB      12h
Ered        DW      ?
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT
```

```
@Start:
            MOV     CX,ADAT
            MOV     DS,CX
            MOV     BL,[Kar]
            MOV     AX,[Szo]
            XOR     BH,BH
            ADD     AX,BX
            SUB     AL,[Szam]
            MOV     [Ered],AX
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

A programban három szegmenst hozunk létre: egy ADAT nevűt az adatoknak, egy KOD nevűt az utasításoknak, illetve egy vermet. Utóbbit a .STACK egyszerűsített szegmensdefiníciós direktíva készíti el, a szegmens neve STACK, mérete 0400h (1 Kbájt) lesz. Ezt a direktívát egészíti ki a MODEL, itt a memória-modellnek SMALL-t írtunk elő, ami annyit tesz, hogy a kód- és az adatterület mérete külön-külön max. 64 Kbájt lehet.

Szegmensdefiníciót a szegmens nevéből és a SEGMENT direktívából álló sorral kezdünk, ezt követi a szegmens tartalmának leírása. Lezárásként a szegmens nevét és az azt követő ENDS foglalt szót tartalmazó sort írjuk. Azaz:

```
Név     SEGMENT {attribútumok}
definíciók
Név     ENDS
```

A SEGMENT kulcsszó után a szegmens jellemzőit meghatározó különféle dolgokat írhatunk még, ezek a szegmens illeszkedését (alignment), kombinálását (combine type), osztályát (class), operandusméretét (operand size) és hozzáférési módját (access mode) határozzák meg. Ezekkel egyelőre még ne törődjünk.

Az adatszegmensben négy változót hozunk létre. Ebből háromnak (Kar, Szo és Szam) kezdeti értéket is adunk, ezek tehát inkább hasonlítanak a Pascal típusos konstansaihoz. Változónak így foglalhatunk helyet:

```
Név         Dx      KezdetiÉrték<,KezdetiÉrték>
```

ahol Dx helyén DB, DW, DD, DF, DQ és DT állhat (jelentésük Define Byte, Word, Doubleword, Farword, Quadword és Ten byte unit). Ezek sorban 1, 2, 4, 6, 8 ill. 10 bájtnyi területet foglalnak le a Név nevű változónak. A kezdeti érték megadásánál írhatunk közönséges számokat (12h), karakter-konstansokat ('a'), sztring-konstansokat ("Ha"), ezek valamilyen kifejezését, illetve ha nem akarunk értéket adni neki, akkor egy ?-et is. A kérdőjellel definiált változókat a legtöbb assembler 00h-val tölti fel, de az is megoldható, hogy ezeket a területeket ne tárolja el a futtatható állományban, s indításkor ezek helyén valami véletlenszerű érték legyen. Vesszővel elválasztva több értéket is felsorolhatunk, ekkor mindegyik értékhez egy újabb terület lesz lefoglalva. Így pl. a következő példa 10 bájtot foglal le, a terület elejére pedig a Valami címke fog mutatni:

```
Valami      DB      1,2,'#',"Szöveg",6+4
```

A kódszegmens megnyitása után jeleznünk kell az assemblernek, hogy ez valóban "kódszegmens", illetve meg kell adnunk, hogy melyik szegmens szerint számolja a memóriahivatkozások offszetcímét. Mindezeket az ASSUME direktívával intézzük el. Az ASSUME direktíva szintaxisa:

```
ASSUME SzegReg:SzegNév<,SzegReg:SzegNév>
ASSUME NOTHING
```

A SzegReg valamilyen szegmensregisztert jelent, a SzegNev helyére pedig egy definiált szegmens vagy szegmenscsoport nevét, vagy a NOTHING kulcsszót írhatjuk. Utóbbi esetben az adott szegmensregisztert egyetlen szegmenshez sem köti hozzá, és nem tételez fel semmit sem az adott regiszterrel kapcsolatban. Az ASSUME NOTHING minden előző hozzárendelést megszüntet.

Rögtön ezután egy címke következik, aminek a @Start nevet adtuk (a "kukac" most csak arra utal, hogy ez nem változó-, hanem címkeazonosító). Rendeltetése az, hogy a program indulásának helyét jelezze, erre az információra ugyanis az assemblernek, linkernek és az operációs rendszernek is szüksége van.

Az első utasítást, ami végrehajtható gépi kódot generál, a MOV (MOVe data) képviseli. Ez két operandussal rendelkezik: az első a cél, a második a forrás (ez persze az összes kétoperandusú utasítást jellemzi). Működése: egyszerűen veszi a forrás tartalmát, és azt bemásolja (eltárolja) a cél op. területére. Eközben a forrás tartalmát, ill. a flag-eket békén hagyja, csak a cél tartalma módosul(hat).

Az első két sor az adatszegmens-regiszter (DS) tartalmát állítja be, hogy az általunk létrehozott ADAT-ra mutasson. Ezt nem teszi meg helyettünk az assembler annak ellenére sem, hogy az ASSUME direktívában már szerepel a DS:ADAT előírás. Látható, hogy az ADAT szegmens címét először betöltjük egy általános regiszterbe (CX), majd ezt bemozgatjuk DS-be. Viszont a MOV DS,ADAT utasítás érvénytelen lenne, mert az Intel 8086-os proci csak memória vagy általános regiszter tartalmát tudja szegmensregiszterbe átrakni, és ez a fordított irányra is vonatkozik (mellesleg pl. a MOV 1234h,DS utasításnak nem lenne semmi értelme).

Most jegyezzünk meg egy fontos dolgot: CS tartalmát ilyen módon nem tölthetjük fel (azaz a MOV CS,… forma ér-

vénytelen), csak a vezérlésátadó utasítások változtathatják meg CS-t. Olvasni viszont lehet belőle, így pl. a MOV SI,CS helyes utasítás lesz.

A következő sor a Kar változó tartalmát BL-be rakja. Látható, hogy a memóriahivatkozást a szögletes zárójelpár jelzi. Ez egyébként jó példa a közvetlen címzésre, a sort ugyanis az assembler a MOV BL,[0000h] gépi utasításra fordítja le. Ha jobban megnézzük, 0000h pont a Kar változó ADAT szegmensbeli offszetje. Ezért van tehát szükség az ASSUME direktíva megadására, mert különben az assembler nem tudta volna, hogy a hivatkozás mire is vonatkozik.

AX-be hasonló módon a Szo változó értéke kerül.

A XOR BH,BH kicsit trükkös dolgot művel. Ha visszaemlékszünk a KIZÁRÓ VAGY értéktáblázatára, akkor könnyen rájöhetünk, hogy ha egy számot önmagával hozunk KIZÁRÓ VAGY kapcsolatba, akkor minden esetben nullát kapunk. Így tehát ez a sor megfelel a MOV BH,00h utasításnak, de ezt írhattuk volna még SUB BH,BH-nak is. Tehát már háromféleképpen tudunk kinullázni egy regisztert!

Hogy végre számoljunk is valamit, az ADD AX,BX összeadja a BX tartalmát az AX tartalmával, majd az eredményt az AX-be teszi. Röviden: hozzáadja BX-et az AX regiszterhez. A művelet elvégzése során beállítja az OF, SF, ZF, AF , PF és CF flag-eket.

Kitalálható, hogy a következő sorban a Szam változó tartalmát vonjuk le az AL regiszterből, a SUB (SUBtract) ugyanis a forrás tartalmát vonja ki a célból.

Számolásunk eredményét el is kell tenni, ezt végzi el a MOV [Ered],AX sor.

Ha befejeztük dolgunkat, és "ki akarunk lépni a programból", akkor erről az aktuális operációs rendszert (ez most a DOS) kell értesíteni, s az majd cselekedni fog. Ezt végzi el programunk utolsó két sora. Magyarázatképpen csak annyit, hogy a 21h szoftver-megszakítás alatt érhetjük el a DOS

funkcióit, ennek 4Ch számú szolgáltatása jelenti a programból való kilépést (vagy másképpen a program befejezését, terminálását). A szolgáltatás számát AH-ban kell megadnunk, AL-be pedig egy visszatérési értéket, egyfajta hibakódot kell írni (ez lesz az ERRORLEVEL nevű változó értéke a DOS-ban). A megszakítást az egyoperandusú INT (INTerrupt request) utasítás váltja ki, s ezzel a program futása befejeződik. Az INT operandusa csak egy bájt méretű közvetlen érték lehet. A megszakítások használatát később tárgyaljuk, most egyelőre fogadjuk el, hogy így kell befejezni a programot. (Persze másképpen is lehetne, de most így csináljuk.)

Az aktuális forrásfájlt az END direktívával kell lezárni, különben az assembler morcos lesz. Ha ez a fő forrásfájl, azaz ez tartalmazza azt a programrészt, aminek el kell indulnia a program betöltése után, akkor az END után egy címke nevének kell szerepelnie. Az itt megadott címre fog adódni a vezérlés a futtatható programfájl betöltése után.

Vajon mi lett a számolás eredménye? BL-be az 'a' karaktert raktuk, ennek ASCII kódja 61h. A Szo értékének megadásához tudni kell, hogy a sztring-konstansokat is helyiértékes rendszerben értelmezi az assembler, 256-os alapú számként tekintve őket. Ezért a Szo tartalma 4861h. Az összeadás után így 4861h+61h=48C2h lesz AX-ben. Ebből még levonjuk a Szam tartalmát (12h), s ezzel kialakul a végeredmény: 48B0h.

7.2 Számlálásos ismétléses vezérlés

Magas szintű nyelvekben igen gyakran alkalmaznak olyan ciklusokat, ahol a ciklusváltozó egy bizonyos tartományt fut be,

és a ciklus magja a tartomány minden értékére lefut egyszer. Ezt valósítják meg a különféle FOR utasítások.

Assemblyben is mód nyílik számlálásos ismétléses vezérlésre, viszont van néhány megkötés a többi nyelvvel szemben: ciklusváltozónak csak CX használható, és nem adható meg az ismétlés tartománya, csak a lefutások számát írhatjuk elő.

A következő program két vektor (egydimenziós tömb) tartalmát összegzi egy harmadik vektorba. Az egyik vektor előjeles bájtokból, míg a másik előjeles szavakból fog állni.

Pelda2.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
ElemSzam    EQU     5
Vekt1       DB      6,-1,17,100,-8
Vekt2       DW      1000,1999,-32768,4,32767
Eredm       DW      ElemSzam DUP (?)
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            MOV     CX,ElemSzam
            XOR     BX,BX
            MOV     SI,BX
@Ciklus:
            MOV     AL,[Vekt1+BX]
            CBW
            ADD     AX,[Vekt2+SI]
            MOV     [Eredm+SI],AX
```

```
            INC     BX
            LEA     SI,[SI+2]
            LOOP    @Ciklus
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Az első újdonságot az adatszegmensben szereplő EQU (define numeric EQUate) kulcsszó képviseli. Ez a fordítási direktíva numerikus konstanst (pontosabban helyettesítő makrót) tud létrehozni, ezért leginkább a C nyelv #define utasításához hasonlítható. Hatására az ElemSzam szimbólum minden előfordulását az EQU jobb oldalán álló kifejezés értékével (5) fogja helyettesíteni az assembler.

Az eredmény vektor nyilván ugyanolyan hosszú lesz, mint a két kiinduló vektor, típusa szerint szavas lesz. Számára a helyet a következőképpen is lefoglalhattuk volna:

```
Eredm       DW      0,0,0,0,0
```

Ez a megoldás azonban nem túl rugalmas. Ha ugyanis megváltoztatjuk az ElemSzam értékét, akkor az Eredm definíciójához is hozzá kell nyúlni, illetve ha nagy az elemek száma, akkor sok felesleges 0-t vagy kérdőjelet kell kiírnunk. Ezeket a kényelmetlenségeket szünteti meg a duplikáló operátor. A DUP (DUPlicate) hatására az operátor bal oldalán álló számú (ElemSzam), adott típusú (DW) elemekből álló terület lesz lefoglalva, amit az assembler a DUP jobb oldalán zárójelben szereplő kifejezéssel (most ?) inicializál (azaz ez lesz a kezdőérték). Ez a módszer tehát jóval leegyszerűsíti a nagy változóterületek létrehozását is.

A ciklusváltozó szerepét a CX regiszter tölti be, ami nem véletlen, hiszen a regiszter neve is innen származik (Counter).

A vektorelemek elérésére most nem használhatunk közvetlen címzést, hiszen azzal csak a vektorok legelső elemét tudnánk kiválasztani. Ehelyett két megoldás kínálkozik: minden adatterülethez hozzárendelünk egy-egy pointert (mutatót), vagy pedig valamilyen relatív címzést alkalmazunk.

A pointeres megoldásnak akkor van értelme, ha mondjuk egy olyan eljárást írunk, ami a célterületnek csak a címét és elemszámának méretét kapja meg. Pointer alatt itt most rövid, szegmensen belüli mutatót értünk, ami egy offszet érték lesz. Pointereket csak bázis- vagy indexregiszterekben tárolhatunk, hiszen csak ezekkel végezhető memóriahivatkozás.

A másik módszer lehetővé teszi, hogy a változóinkat ténylegesen tömbnek tekinthessük, és ennek megfelelően az egyes elemeket indexeléssel (tömbelem-hivatkozással) érhessük el. Ehhez a megoldáshoz szükség van a tömbök bázisára (kezdőcímére), illetve az elemkiválasztást megvalósító tömbindexre. A tömbök kezdőcímét most konkrétan tudjuk, így ez konstans numerikus érték lesz. A tömbindexet tetszésünk szerint tárolhatjuk bármely bázis- vagy indexregiszterben. Mivel minden vektorban elemről-elemre sorban haladunk, valamint a Vekt2 és az Eredm vektorok is szavasak, ezért most csak kettő tömbindexre van szükségünk. BX a Vekt1-et indexeli, és egyesével kell növelni. SI-t ezzel szemben Vekt2 és Eredm indexelésére is használjuk, növekménye 2 bájt lesz.

Assemblyben a ciklus magja a ciklus kezdetét jelző címkétől (most @Cimke) a ciklusképző utasításig (ez lesz a LOOP) tart.

A Vekt1 következő elemét olvassuk ki először AL-be, bázisrelatív címzést használva. Azután ehhez kéne hozzáadnunk a Vekt2 megfelelő elemét. A két vektor elemmérete azonban eltérő, egy bájtot pedig nem adhatunk hozzá egy szóhoz. Ez megoldható, ha a bájtot átkonvertáljuk szóvá. A CBW (Convert Byte to Word) utasítás pont ezt teszi, az AL-ben levő értéket

előjelesen kiterjeszti AX-be. Így már elvégezhető az összeadás, ami után az eredményt is a helyére tesszük.

Az INC (INCrement) utasítás az operandusát növeli meg 1-gyel, de természetesen nem lehet közvetlen érték ez az operandus. Működése az ADD ??,1 utasítástól csak annyiban tér el, hogy nem változtatja meg a CF értékét, és ez néha fontos tud lenni. Az INC párja a DEC (DECrement), ami eggyel csökkenti operandusát.

SI-t többféleképpen is megnövelhetnénk 2-vel:

1)

```
INC     SI
INC     SI
```

2)

```
ADD     SI,2
```

Mindkét megoldásnak van egy pici szépséghibája: az első feleslegesen ismétel meg egy utasítást, a második viszont túl nagy kódot generál (4 bájtot), és a flag-eket is elállítja. Ha kicsi a hozzáadandó érték (mondjuk előjelesen 1 bájtos), akkor érdemesebb a LEA (Load Effective Address) utasítást használni. Ez a forrás memóriaoperandus tényleges címét (effective address) tölti be a céloperandusba, ami csak egy 16 bites általános regiszter lehet. Az összes flag-et békén hagyja. Tényleges cím alatt a használt címzési mód által jelölt, meghatározott memóriacímet (offszetet) értjük. Így a LEA SI, [SI+2] utasítás a DS:(SI+2) című memóriarekesz offszetcímét tölti be SI-be, azaz 2-vel megnöveli SI-t. Ez így csak 3 bájtos utasítást eredményez. Az utasítás jelentősége jobban látszik, ha bonyolultabb címzésmódot használunk:

```
LEA     AX,[BX+DI-100]
```

Itt AX-be egy háromtagú összeg értéke kerül, s mindezt úgy hajthatjuk végre, hogy más regiszterre nem volt szükségünk, továbbá ez az utasítás is csak három bájtot foglal el.

A tényleges cikluskép zést a LOOP utasítás végzi. Működése: csökkenti CX-et, ha az zéró, akkor a LOOP utáni utasításra kerül a vezérlés, különben elugrik az operandus által mutatott memóriacímre. A csökkentés során egyetlen flag értéke sem változik meg. A LOOP ú.n. relatív ugrást használ, ami azt jelenti, hogy a cél címet a LOOP után következő utasítás elejéhez képest relatívan kell megadni (ez csak a gépi kód szinten számít, Assemblyben nyugodtan használhatunk címkéket). Ezt a relatív címet 1 bájton tárolja el, így a -128.. +127 bájtnyi távolságra levő címekre tudunk csak ciklust szervezni. Ha az operandus által mutatott cím túl messze lenne, akkor az assembler hibát fog jelezni. Ilyenkor csak azt tehetjük, hogy más módon (pl. feltételes és feltétel nélküli ugrások kombinációjával) oldjuk meg a problémát. A szabályos ciklusokban egyébként csak visszafelé mutató címet szoktunk megadni a LOOP után (tehát a relatív cím negatív lesz). Fontos még tudni, hogy a LOOP először csökkenti CX-et, s csak ezután ellenőrzi, hogy az nullává vált-e. Így ha a ciklusba belépéskor CX zéró volt, akkor nem egyszer, hanem 65536-szor fog lefutni ciklusunk! A LOOP utasítással megírt ciklus tehát legalább 1-szer mindenképpen lefut.

A LOOP utasításnak még két változata létezik. A LOOPE/ LOOPZ (LOOP while Equal/Zero/ZF=1) csökkenti CX-et, majd akkor ugrik a megadott címre, ha CX≠0000h és ZF=1. Ha valamelyik vagy mindkét feltétel nem teljesül, a LOOP utáni utasításra kerül a vezérlés. A két mnemonik ugyanazt a műveleti kódot generálja. Hasonlóan a LOOPNE/LOOPNZ (LOOP while Not Equal/Not Zero/ZF=0) utasítás CX csökkentése után akkor ugrik, ha CX≠0000h és ZF=0 is teljesül.

7.3 Egyszerű és többszörös szelekciós vezérlés

Ez a vezérlési szerkezet egy vagy több feltétel teljesülése vagy nem teljesülése szerinti elágazást tesz lehetővé a program menetében. Megfelel az IF…THEN…ELSE utasítások láncolatának.

A megoldandó probléma: konvertáljunk át egy karaktersorozatot csupa nagybetűsre. Az ékezetes betűkkel most nem törődünk, a szöveget pedig az ASCII 00h kódú karakter fogja lezárni.

Pelda3.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Szoveg      DB      "Ez egy PELDA szoveg."
            DB      "Jo proci a 8086-os!",00h
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            PUSH    DS
            POP     ES
            LEA     SI,[Szoveg]
            MOV     DI,SI
            CLD
@Ciklus:
            LODSB
            OR      AL,AL
            JZ      @Vege
            CMP     AL,'a'
            JB      @Tarol
```

```
            CMP     AL,'z'
            JA      @Tarol
            SUB     AL,'a'-'A'
@Tarol:
            STOSB
            JMP     @Ciklus
@Vege:
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

A program elején két ismerős utasítással is találkozhatunk, ezek a verem tárgyalásánál már említett PUSH és POP. Mindkettő egyoperandusú, s ez az operandus nemcsak általános regiszter vagy memóriahivatkozás, de szegmensregiszter is lehet. 8 bites regisztert viszont nem fogadnak el, mivel a verem szavas szervezésű. A programban szereplő PUSH-POP pár tulajdonképpen DS tartalmát tölti ES-be, amit rövidebben írhattunk volna MOV ES,AX-nek is, mindössze szemléltetni akarjuk, hogy így is lehet értéket adni egy szegmensregiszternek.

Bár nem tartozik szorosan a kitűzött feladathoz, két probléma is felmerül ezzel a két utasítással kapcsolatban. Az egyik, hogy vajon mit csinál a PUSH SP utasítás? Nos, a 8086-os processzor esetén ez SP 2-vel csökkentése után SP új (csökkentett) értékét teszi le a verem SS:SP címére, míg a későbbi processzorok esetén a régi érték kerül eltárolásra. Furcsa egy megoldás, az biztos… Mindenesetre ezzel nem kell foglalkoznunk, csak egy érdekesség.

A másik dolog, amit viszont fontos megjegyezni, az az, hogy a POP CS utasítást a MOV CS,?? -hez hasonlóan nem szereti sem az assembler, sem a processzor. Ha belegondolunk hogy ez mit jelentene, láthatjuk, hogy feltétel nélküli vezérlésátadást valósítana meg, csakhogy úgy, hogy közben csak CS értéke változna, IP-é nem! Ez pedig veszélyes és

kiszámíthatatlan eredményekkel járna. Így a CS nem lehet a POP operandusa. (Igazság szerint a 8086-os processzor értelmezni tudja ezt az utasítást, aminek gépi kódja 0Fh, viszont a későbbi processzorokon ennek a kódnak más a szerepe. Ezért és az előbb említettek miatt NE használjuk!)

A következő három utasítás a sztringkezelő utasításokat (LODSB és STOSB) készíti elő.

A forrássztring címét DS:SI fogja tartalmazni, így a Szoveg offszetjét betöltjük SI-be. Megjegyezzük, hogy a TASM ezt az utasítást MOV SI,OFFSET [Szoveg] -ként fogja lefordítani, talán azért, mert ez kicsit gyorsabb lesz, mint az eredeti változat.

A célsztring az ES:DI címen fog elhelyezkedni. Most helyben fogunk konvertálni, így DI-t egyenlővé tesszük SI-vel.

A CLD (CLear Direction flag) a DF flag-et törli. Ennek szükségessége rögvest kiderül.

A LODSB (LOaD String Byte) a DS:SI címen levő bájtot betölti AL-be, majd SI-t eggyel növeli, ha DF=0, ill. csökkenti, ha DF=1. Másik alakja, a LODSW (LOaD String Word) AX-be tölti be a DS:SI-n levő szót, és SI-t 2-vel növeli vagy csökkenti DF-től függően. Egyetlen flag-et sem változtat meg.

Az OR AL,AL utasítás ismét egy trükköt mutat be. Ha egy számot önmagával hozunk logikai VAGY kapcsolatba, akkor maga a szám nem változik meg. Cserébe viszont CF, AF és OF törlődik, SF, ZF és PF pedig az operandusnak megfelelően módosulnak. Itt most annak tesztelésére használtuk, hogy AL zérus-e, azaz elértük-e a sztringünk végét. Ezt megtehettük volna a később szereplő CMP AL,00h utasítással is, csak így elegánsabb.

Elérkeztünk a megoldás kulcsához, a *feltételes ugrásokhoz* (conditional jumps). Ezek olyan vezérlésátadó utasítások, amik valamilyen flag (vagy flag-ek) állása alapján vagy elugranak az operandus szerinti címre, vagy a következő utasításra térnek. Összesen 17 db van belőlük, de mnemonikból ennél több van,

mivel némelyik egynél több elnevezés alatt is elérhető. Három csoportba oszthatók: előjeles aritmetikai, előjeltelen aritmetikai és egyéb feltételes ugrások.

Először jöjjenek az előjeles változatok:

Mnemo.	*Hatás*
JL, JNGE	**Ugrás, ha kisebb/nem nagyobb vagy egyenlő**
JNL, JGE	**Ugrás, ha nem kisebb/nagyobb vagy egyenlő**
JLE, JNG	**Ugrás, ha kisebb vagy egyenlő/nem nagyobb**
JNLE, JG	**Ugrás, ha nem kisebb vagy egyenlő/nagyobb**

A mnemonikokat tehát úgy képezzük, hogy a J betűt (ami a Jump if … kifejezést rövidíti) egy, kettő vagy három betűs rövidítés követi. Némi angol tudással könnyebben megjegyezhetők az egyes változatok, ha tudjuk, mit jelölnek az egyes rövidítések:

- **L=Less**
- **G=Greater**
- **N=Not**
- **LE=Less or Equal**
- **GE=Greater or Equal**

Most nézzük az előjel nélküli ugrásokat:

Mnemonik	*Hatás*
JB, JNAE, JC	**Ugrás, ha kisebb/nem nagyobb vagy egyenlő/CF=1**
JNB, JAE, JNC	**Ugrás, ha nem kisebb/nagyobb vagy egyenlő/CF=0**
JBE, JNA	**Ugrás, ha kisebb vagy egyenlő/nem nagyobb**
JNBE, JA	**Ugrás, ha nem kisebb vagy egyenlő/nagyobb**

Az alkalmazott rövidítések:

- **B=Below**
- **C=Carry**
- **A=Above**
- **BE=Below or Equal**
- **AE=Above or Equal**

Mint várható volt, a JB és JC ugyanazt jelentik, hiszen mindkettő azt fejezi ki, hogy a legutolsó művelet előjel nélküli aritmetikai túlcsordulást okozott.

Lássuk a megmaradt további ugrásokat:

Mnemo.	*Hatás*
JE, JZ	Ugrás, ha egyenlő/ZF=1
JNE, JNZ	Ugrás, ha nem egyenlő/ZF=0
JO	Ugrás, ha OF=1
JNO	Ugrás, ha OF=0
JS	Ugrás, ha SF=1
JNS	Ugrás, ha SF=0
JP, JPE	Ugrás, ha PF=1/páros paritás
JNP, JPO	Ugrás, ha PF=0/páratlan paritás
JCXZ	Ugrás, ha CX=0000h

A rövidítések magyarázata pedig:

- **E=Equal**
- **Z=Zero**
- **O=Overflow**
- **S=Sign**
- **P=Parity**
- **PE=Parity Even**
- **PO=Parity Odd**
- **CXZ=CX is Zero (vagy CX equals Zero)**

Mint látható, a JCXZ kakukktojás a többi ugrás között, mivel nem egy flag, hanem a CX regiszter állása szerint cselekszik. Mindegyik feltételes ugró utasítás egyetlen operandusa a cél memóriacím, de ez is előjeles relatív címként 1 bájtban tárolódik el, a LOOP utasításhoz hasonlóan. Pontosan ezen megkötés miatt található meg mindegyik feltételnek (a JCXZ-t kivéve) a negált párja is. A következő példát ugyanis nem fogja lefordítani az assembler (a TASM-ra ez nem teljesen igaz, a JUMPS és NOJUMPS direktívákkal megadható, hogy magától kijavítsa-e az ilyen helyzeteket, avagy utasítsa vissza):

```
            XOR     AX,AX
            JZ      @Cimke
KamuAdat    DB      200 DUP (?)
@Cimke:
            …
```

A 200 bájtos területfoglalás biztosítja, hogy a JZ utasítást követő bájt messzebb legyen a @Cimke címkétől, mint a megengedett +127 bájt. Az ugrást ráadásul végre kell hajtani, mivel a XOR AX,AX hatására ZF=1 lesz. Ezt a hibás szituációt feloldhatjuk, ha a JZ … helyett JNZ-JMP párost használunk (a JMP leírása kicsit lentebb lesz):

```
            XOR     AX,AX
            JNZ     @KamuCimke
            JMP     @Cimke
@KamuCimke:
KamuAdat    DB      200 DUP (?)
@Cimke:
            …
```

A példaprogramhoz visszatérve, az OR AL,AL és a JZ … hatására megoldható, hogy az algoritmus leálljon, ha elértük a sztring végét.

Most jön maga a szelekció: el kell dönteni, hogy az aktuális karaktert (ami AL-ben van) kell-e konvertálni. Konverziót kell végezni, ha 61h≤AL≤7Ah ('a' kódja 61h, 'z' kódja 7Ah), különben változatlanul kell hagyni a karaktert. Először azt nézzük meg, hogy AL<61h teljesül-e. Ha igen, akkor nem kell konverzió. Különben ha AL>7Ah, akkor ugyancsak nincs konverzió, különben pedig elvégezzük az átalakítást. A leírtakat CMP-Jccc utasításpárosokkal oldjuk meg. A CMP (CoMPare) utasítás kiszámítja a céloperandus és a forrásoperandus különbségét, beállítja a flag-eket, az operandusokat viszont békén hagyja (és eldobja az eredményt is).

A konverziót az egyetlen SUB AL,'a'-'A' utasítás végzi el.

Akár volt konverzió, akár nem, az algoritmus végrehajtása a @Tarol címkétől folytatódik. Megfigyelhetjük, hogy ez a rész közös rész a szelekció mindkét ágában (t.i. volt-e konverzió avagy nem), így felesleges lenne mindkét esetben külön leírni. Ehelyett azt tesszük, hogy ha nem kell konvertálni, akkor elugrunk a @Tarol címkére, különben elvégezzük a szükséges átalakítást, és "rácsorgunk" erre a címkére. Az ilyen megoldások igen gyakoriak az Assembly programokban, használatuk rövidebbé, gyorsabbá teszi azokat, az algoritmusok pedig áttekinthetőbbek lesznek általuk.

Most jön a kérdéses karakter eltárolása. Ezt egy másik sztringkezelő utasítás, a STOSB (STOre String Byte) intézi el. Működése: AL-t eltárolja az ES:DI címre, majd DI-t megnöveli eggyel, ha DF=0, illetve csökkenti, ha DF=1. A STOSW (STOre String Word) utasítás, hasonlóan a LODSW-hez, AX-szel dolgozik, és DI-t 2-vel növeli vagy csökkenti.

Ezután vissza kell ugranunk a @Ciklus címkére, mert a többi karaktert is fel kell dolgozni a sztringünkben. Erre szolgál a JMP (JuMP) feltétel nélküli ugrás (unconditional jump). Működése: az operandusként megadott címre állítja be IP-t (van egy másik, távoli formája is, ekkor CS-t is átállítja), és

onnan folytatja a végrehajtást. Érdekes, hogy a JMP is relatív címként tárolja el a cél memóriacímét, viszont a feltételes ugrásokkal és a LOOP-pal ellentétben a JMP 16 bites relatív címet használ, így a -32768..+32767 bájt távolságban levő címekre tudunk előre-hátra ugrálni. (Ez csak a közeli, ugyanazon kódszegmensen belüli ugrásra vonatkozik, de erről majd később.)

Ha jobban megnézzük, észrevehetjük, hogy az algoritmus elején szereplő OR AL,AL // JZ @Vege és az utóbbi JMP @Ciklus utasítások egy hurok vezérlési szerkezetet írnak elő, amiben az OR-JZ páros alkotja a kilépési feltételt, a JMP pedig maga a ciklusszervező utasítás (t.i. végtelen ciklust valósít meg). De ha jobban tetszik, ezt tekinthetjük akár előfeltételes vezérlési szerkezetnek is (WHILE…DO ciklus).

A @Vege címke elérésekor a kiinduló sztringünk már nagybetűsen virít a helyén.

7.4 Eljárásvezérlés

A program eljárásokra és függvényekre (egyszóval szubrutinokra) bontása a strukturált programozás alapját képezi. Nézzük meg, hogyan használhatunk eljárásokat Assemblyben.

A feladat: írjunk olyan eljárást, ami az AX-ben található előjel nélküli számot kiírja a képernyőre decimális alakban!

Pelda4.ASM:

```
MODEL SMALL

.STACK

KOD         SEGMENT
```

```
            ASSUME CS:KOD,DS:NOTHING

DecKiir     PROC
            PUSH    AX
            PUSH    BX
            PUSH    CX
            PUSH    DX
            MOV     BX,10
            XOR     CX,CX
@OsztCikl:
            OR      AX,AX
            JZ      @CiklVege
            XOR     DX,DX
            DIV     BX
            PUSH    DX
            INC     CX
            JMP     @OsztCikl
@CiklVege:
            MOV     AH,02h
            JCXZ    @NullaVolt
@KiirCikl:
            POP     DX
            ADD     DL,'0'
            INT     21h
            LOOP    @KiirCikl
            JMP     @KiirVege
@NullaVolt:
            MOV     DL,'0'
            INT     21h
@KiirVege:
            POP     DX
            POP     CX
            POP     BX
            POP     AX
            RET
DecKiir     ENDP

UjSor       PROC
            PUSH    AX
```

```
            PUSH    DX
            MOV     AH,02h
            MOV     DL,0Dh
            INT     21h
            MOV     DL,0Ah
            INT     21h
            POP     DX
            POP     AX
            RET
UjSor       ENDP

@Start:
            XOR     AX,AX
            CALL    DecKiir
            CALL    UjSor
            MOV     AX,8086
            CALL    DecKiir
            CALL    UjSor
            MOV     AX,8000h
            CALL    DecKiir
            CALL    UjSor
            MOV     AX,0FFFFh
            CALL    DecKiir
            CALL    UjSor
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Ez egy kicsit hosszúra sikeredett, de hát a probléma sem annyira egyszerű. Nézzük először a főprogramot (azaz a @Start címke utáni részt)! Itt minden ismerősnek tűnik, kivéve a CALL utasítást. Ez az egyoperandusú utasítás szolgál az eljárások végrehajtására, más szóval az *eljáráshívásra* (procedure call). Működése: a verembe eltárolja az IP aktuális értékét (tehát szimbolikusan végrehajt egy PUSH IP-t), majd IP-t beállítja az operandus által mutatott címre, és onnan folytatja az utasítások végrehajtását. A célcímet a JMP-hez hasonlóan 16 bites előjeles

relatív címként tárolja. (Létezik távoli formája is, ekkor CS-t is lerakja a verembe az IP előtt, illetve beállítja a cél kódszegmenst is.) Az operandus helyén most egy eljárás neve áll (ami végül is egy olyan címke, ami az eljárás kezdetére mutat). Két eljárást írtunk, a DecKiir végzi el AX tartalmának kiírását, az UjSor pedig a képernyő következő sorára állítja a kurzort. Tesztelés céljából a 0, 8086, 32768 és 65535 értékeket íratjuk ki.

Most térjünk vissza a program elejére, ahol az eljárásokat helyeztük el (egyébként bárhol lehetnének, ez csak megszokás kérdése). Eljárást a következő módon definiálhatunk:

```
Név    PROC {opciók}
eljárástörzs
Név    ENDP
```

Az eljárás törzsét (azaz annak működését leíró utasításokat) a PROC…ENDP direktívák (nem pedig mnemonikok!) fogják közre. Van néhány olyan rész, ami a legtöbb eljárás törzsében közös. Az eljárás elején szokásos az eljárásban később módosított regiszterek tartalmát a verembe elmenteni, hogy azokat később visszaállíthassuk, és a hívó program működését ne zavarjuk be azzal, hogy a regiszterekbe zagyvaságokat teszünk. Természetesen ha egy regiszterben akarunk visszaadni valamilyen értéket, akkor azt nem fogjuk elmenteni. Az eljárásból a hívóhoz visszatérés előtt a verembe berakott regiszterek tartalmát szépen helyreállítjuk, méghozzá pontosan a berakás fordított sorrendjében.

A DecKiir eljárásban a 4 általános adatregisztert fogjuk megváltoztatni, ezért ezeket szépen PUSH-sal berakjuk a verembe az elején, majd az eljárás végén fordított sorrendben POP-pal kivesszük.

Bináris számot úgy konvertálunk decimálisra, hogy a kiinduló számot elosztjuk maradékosan 10-zel, és a maradékot szépen eltároljuk. Ha a hányados nulla, akkor készen vagyunk, különben a hányadost tekintve az új számnak, azon folytatjuk a

10-zel való osztogatást. Ha ez megvolt, akkor nincs más dolgunk, mint hogy az osztások során kapott maradékokat a megkapás fordított sorrendjében egymás mellé írjuk mint számjegyeket. Így tehát az első maradék lesz a decimális szám utolsó jegye.

Az osztót most BX-ben tároljuk, CX pedig az eltárolt maradékokat (számjegyeket) fogja számolni, ezért kezdetben kinullázzuk.

Az osztásokat egy előfeltételes ciklusba szervezve végezzük el. A ciklusnak akkor kell megállnia, ha AX (ami kezdetben a kiírandó szám, utána pedig a hányados) nullává válik. Ha ez teljesül, akkor kiugrunk a ciklusból, és a @CiklVege címkére ugrunk.

Ha AX≠0000h, akkor osztanunk kell. A DIV (unsigned DIVision) utasítás szolgál az előjeltelen osztásra. Egyetlen operandusa van, ami 8 vagy 16 bites regiszter vagy memóriahivatkozás lehet. Ha az operandus bájt méretű, akkor AX-et osztja el az adott operandussal, a hányadost AL-be, a maradékot pedig AH-ba teszi. Ha szavas volt az operandus, akkor DX:AX-et osztja el az operandussal, a hányados AX-be, a maradék DX-be kerül. A 6 aritmetikai flag értékét elrontja. Nekünk most a második esetet kell választanunk, mert előfordulhat, hogy az első osztás hányadosa nem fog elférni egy bájtban. Így tehát BX-szel fogjuk elosztani DX:AX-et. Mivel a kiinduló számunk csak 16 bites volt, a DIV viszont 32 bites számot követel meg, DX-et az osztás elvégzése előtt törölnünk kell.

A maradékokat a veremben fogjuk tárolni az egyszerűbb kiíratás kedvéért, ezért DX-et berakjuk oda, majd a számlálót megnöveljük. Végül visszaugrunk az osztó ciklus elejére.

A számjegyek kiírására a legegyszerűbb módszert választjuk. Már említettük, hogy az INT 21h-n keresztül a DOS szolgáltatásait érhetjük el. A 02h számú szolgáltatás (azaz ha AH=02h) a DL-ben levő karaktert kiírja a képernyőre az

aktuális kurzorpozícióba. AH-ba ezért most berakjuk a szolgáltatás számát.

Ha az eljárás hívásakor AX nulla volt, akkor az osztó ciklus egyszer sem futott le, hanem rögtön az elején kiugrott. Ekkor viszont CX=0, hiszen így állítottuk be. Ezek hatására a JCXZ utasítás segítségével a @NullaVolt címkén folytatjuk az eljárás végrehajtását, ahol a 21h-s szoftver-megszakítás segítségével kiírjuk azt az egy számjegyet, majd "rácsorgunk" a @KiirVege címkére.

Ha nemzéró értékkel hívtuk meg eljárásunkat, akkor itt az ideje, hogy kiírjuk a CX db jegyet a képernyőre. Ezt egy egyszerű számlálásos ismétléses vezérléssel (LOOP ciklussal) elintézhetjük. Mivel a verem LIFO működésű, így a legutoljára betett maradékot vehetjük ki legelőször, és ezt is kell első számjegyként kiírnunk. Az érvényes decimális jeggyé konvertálást az ADD DL,'0' utasítás végzi el, majd a karaktert megszakítás-hívással kiküldjük a monitorra. A kiírás végeztével szintén a @KiirVege címkére megyünk.

Miután visszaállítottuk a módosított eredeti regiszterek tartalmát, vissza kell térnünk arra a helyre, ahonnan meghívták az eljárást. Erre szolgál a RET (RETurn) utasítás. Két változata van: ha nem írunk mellé operandust, akkor a veremből kiszedi IP-t (amit előzőleg a CALL rakott oda), és onnan folytatja a végrehajtást. De írhatunk egy 16-bites közvetlen értéket (numerikus konstanst) is operandusként, ekkor az IP kiszedése után ezt a számot hozzáadja SP-hez (olyan, mintha Szám/2 db. POP-ot hajtana végre), majd az új CS:IP címre adja a vezérlést. Mi most nem akarunk változtatni a veremmutatón, így az egyszerű RET-tel visszatérünk a főprogramba.

Az UjSor eljárás nagyon egyszerű és rövid. A regiszterek elmentésén-visszaállításán kívül csak két megszakítás-hívást tartalmaz, amik a 0Dh és 0Ah ASCII kódú karaktereket írják ki a képernyőre. Az első karakter az ú.n. kocsivissza (CR–

Carriage Return), a másik pedig a soremelés (LF–Line Feed). Ezek alkotják az *új sor* (new line) karakterpárt.

Ha sok regisztert kell elmentenünk és/vagy visszaállítanunk, akkor fárasztó és felesleges minden egyes regiszterhez külön PUSH vagy POP utasítást írni. Ezt megkönnyítendő, a TASM lehetővé teszi, hogy egynél több operandust írjunk ezen utasítások után, az egyes operandusokat egymástól szóközzel elválasztva. Így a PUSH AX BX CX DX // POP SI DI ES utasítások ekvivalensek a következő sorozattal:

```
PUSH   AX
PUSH   BX
PUSH   CX
PUSH   DX
POP    SI
POP    DI
POP    ES
```

8 A TURBO DEBUGGER HASZNÁLATA

Az eddig megírt programjaink működését nem tudtuk ellenőrizni, úgy pedig elég kényelmetlen programozni, hogy nem látjuk, tényleg azt csinálja-e a program, amit elvárunk tőle. Ha valami rosszul működik, netán a számítógép semmire sem reagál (ezt úgy mondjuk, hogy "kiakadt" vagy "lefagyott"), a hiba megkeresése egyszerűen reménytelen feladat segítség nélkül. Ezt a természetes igényt elégítik ki a különböző debugger (nyomkövető, de szó szerint "bogártalanító") szoftverek ill. hardverek. (Az elnevezés még a számítástechnika hőskorszakából származik, amikor is az egyik akkori

számítógép működését valamilyen rovar zavarta meg. Azóta hívják a hibavadászatot "bogárirtásnak".)

A Turbo Assemblert és Linkert készítő Borland cég is kínál egy szoftveres nyomkövető eszközt, Turbo Debugger (TD) néven. Most ennek használatával fogunk megismerkedni.

A Turbo Debugger fő tulajdonsága, hogy képes egy futtatható állományt (.EXE vagy .COM kiterjesztéssel) betölteni, majd annak gépi kódú tartalmát Assembly forrásra visszafejteni. Ezt hívjuk *disassemblálásnak* (disassembly). A legszebb a dologban az, hogy a programban szereplő kód- és memóriahivatkozásokat konkrét számok helyett képes az eredeti forrásban szereplő szimbólumokkal megjeleníteni. Ezenkívül minden utasítást egyenként hajthatunk végre, ha akarjuk, többször is, sőt megadhatjuk, hogy a program végrehajtása egy bizonyos feltétel teljesülése esetén szakadjon meg. Figyelhetjük a memória tetszőleges területének tartalmát, a regiszterek és flag-ek értékeit, s mindezeket meg is változtathatjuk. A program futását az eredeti forrásokon is képes követni. Szóval csupa-csupa hasznos szolgáltatással bír, amikkel pont olyan kényelmesen figyelhetjük programunk tevékenységét, mintha mondjuk a Borland C fejlesztő rendszerében (IDE) dolgoznánk.

Ahhoz hogy mindezt a kényelmet élvezhessük, nem kell mást tenni, mint:

- **minden forrást a /zi vagy /zd kapcsolóval lefordítani**
- **a tárgykódokat a /v kapcsolóval összeszerkeszteni**

Ha ezt a két műveletet elvégeztük, akkor a .EXE állományunk (remélhetőleg) tartalmazza az összes szimbolikus információt, amire a nyomkövetés során szükség lehet.

```
≡  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help                READY
CPU 80486                                                                          2
#pelda#@start:  MOV CX,ADAT
  cs:0000 B9C084          mov     cx,84C0          ax 0000   c=0
#pelda#16:  MOV DS,CX                              bx 0061   z=0
  cs:0003 8ED9            mov     ds,cx            cx 84C0   s=0
#pelda#17:  MOV BL,[Kar]                           dx 0000   o=0
  cs:0005 8A1E0000        mov     bl,[#pelda#kar]  si 0000   p=0
#pelda#18:  MOV AX,[Szo]                           di 0000   a=0
  cs:0009►A10100          mov     ax,[#pelda#szo]  bp 0000   i=1
#pelda#19:  XOR BH,BH                              sp 0400   d=0
  cs:000C 32FF            xor     bh,bh            ds 84C0
#pelda#20:  ADD AX,BX                              es 8470
  cs:000E 03C3            add     ax,bx            ss 8480
#pelda#21:  SUB AL,[Szam]                          cs 84C1
                                                   ip 0009
  es:0000 CD 20 00 A0 00 9A F0 FE =  á ü≡▪
  es:0008 1D F0 E0 01 62 39 AA 01 ↔≡α♀b9¬♀
  es:0010 62 39 89 02 BD 33 2F 25 b9ë☻╜3/%        ss:0402 1248
  es:0018 01 01 01 00 02 FF FF FF ♀♀♀ ☻           ss:0400►6161
[▪]═Module: pelda File: pelda.asm 18═══════════4═[↑][↓]═
                ASSUME CS:KOD, DS:ADAT
  @Start:
                MOV                 CX,ADAT
                MOV                 DS,CX
                MOV                 BL,[Kar]
►               MOV                 AX,[Szo]
                XOR                 BH,BH
                ADD                 AX,BX
                SUB                 AL,[Szam]
                MOV                 [Ered],AX
                MOV                 AX,4C00h
                INT                 21h
  KOD           ENDS
                END                 @Start
Stack═══3
Watches═══1
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

A dolognak két hátránya van: egyrészt .COM programokba nem kerülhet nyomkövetési info, másrészt ez az adathalmaz az .EXE fájl méretét igencsak megnövelheti (előfordulhat, hogy az eredeti többszöröse lesz a kimenet mérete).

A legelső példaprogramot (Pelda1.ASM) begépeltük és elmentettük PELDA.ASM néven, majd lefordítottuk és linkeltük, az összes debug információt belerakva. A TD PELDA.EXE parancs kiadása utáni állapotot tükrözi a fenti kép.

A képernyő tetején ill. alján a már megszokott menüsor és státuszsor találhatók.

A kép nagy részét elfoglaló munkaasztalon (desktop) helyezkednek el a különféle célt szolgáló ablakok.

A legfelső ablak (CPU ablak) több részre osztható. A bal felső sarok az aktuális kód disassemblált változatát mutatja, és most éppen úgy van beállítva, hogy a forrás eredeti sorait is

megjeleníti. A következő végrehajtandó sor a bal oldalon egy kis jobbra mutató háromszöggel van megjelölve (ez most a CS:0009h című sor).

Emellett az egyes regiszterek és flag-ek tartalma látható. Az előző állapothoz képest megváltozott dolgokat fehér színnel jelöli meg.

A jobb alsó sarok a memória egy adott területének tartalmát mutatja hexadecimális és ASCII alakban (ez az ú.n. memory dump).

Végül a jobb alsó sarokban a verem aktuális állapotát szemlélhetjük meg. A veremmutatót (azaz a verem tetejét) szintén egy jobbra mutató sötét háromszög jelzi.

A kép közepén levő nagyobb ablakban a forrásban követhetjük nyomon a program futását. Itt mindig azt a fájlt látjuk, amihez az éppen végrehajtott kód tartozik. A következő végrehajtandó sort a bal oldalon kis fehér háromszög mutatja.

Az alatta látható, Stack címkéjű ablak a különböző eljárás- és függvényhívások során a verembe rakott argumentumokat mutatja.

A legalsó, keskeny ablak a Watches címkét viseli. Ennek megfelelően az általunk óhajtott változók, memóriaterületek aktuális értékét követhetjük itt figyelemmel.

A felső sorban található menürendszeren kívül minden ablakban előhívható egy helyi menü (local menu) az <ALT>+<F10> billentyűkombinációval, ahol az aktuális ablakban (vagy részablakban) rendelkezésre álló plusz szolgáltatásokat érhetjük el. Ilyenek pl. az utasítás assemblálása, regiszter tartalmának megváltoztatása, flag törlése, megfigyelendő változó felvétele a Watches listába stb.

Az ablakok között az <F6> és <Shift>+<F6> billentyűkkel mozoghatunk, míg az aktuális ablakon belül a <Tab> és a <Shift>+<Tab> kombinációkkal lépkedhetünk a részablakok között.

Most áttekintjük az egyes menük funkcióit. A File menü a szokásos állományműveleteket tartalmazza, de itt kaphatunk információt a betöltött programról is.

Az Edit menü szintén a gyakori másolás-beszúrás típusú funkciókat rejti.

A View menü készlete igen gazdag. Innen nézhetjük meg a változókat (Variables), a CPU ablakot, másik programmodult, tetszőleges állományt és még sok minden mást.

A Run menüben, mint neve is sejteti, a futtatással kapcsolatos tevékenységek vannak összegyűjtve, de itt állíthatók be a program futtatási (parancssoros) argumentumai is. Szinte az összes itteni szolgáltatáshoz tartozik valamilyen gyorsbillentyű (hot key) is. Néhány ezek közül: futtatás <F9>, újraindítás <Ctrl>+<F2>, adott sorig végrehajtás <F4>, lépésenkénti végrehajtás <F7>, CALL és INT utasítások átugrása <F8>.

A Breakpoints menüvel a töréspontokat tarthatjuk karban. A *töréspont* egy olyan hely a kódban, ahol a program végrehajtásának valamilyen feltétel teljesülése esetén (vagy mindenképpen) meg kell szakadnia. Ilyenkor a vezérlést ismét visszakapjuk a TD képernyőjével együtt, és kedvünk szerint beavatkozhatunk a program menetébe.

A Data menü az adatok, változók manipulálását segíti.

Az Options menüben találhatók a TD beállításai. Ilyenek pl. a forrás nyelve (Language), helye (Path for source), képernyővel kapcsolatos dolgok (Display options).

A Window menü az ablakokkal való bűvészkedésre jó, a Help menü pedig a szokásos súgót tartalmazza.

A TD egyébként nemcsak Assembly, de Pascal és C nyelvű programot is képes nyomon követni, és az ezekben a nyelvekben meglevő összes adattípust is képes kezelni.

9 SZÁMOLÁS ELŐJELES SZÁMOKKAL, BITMŰVELETEK

Ebben a fejezetben a gyakorlati problémák megoldása során gyakran előforduló feladatokkal foglalkozunk. Teljes példaprogramokat most nem közlünk, a megoldás módszerét egy-egy rövid programrészleten fogjuk bemutatni.

9.1 Matematikai kifejezések kiértékelése

Az első problémakört a különböző előjelű számokat tartalmazó matematikai kifejezések kiértékelése alkotja. Például tegyük fel, hogy AL-ben van egy előjeles, míg BX-ben egy előjel nélküli érték, és mi ezt a két számot szeretnénk összeadni, majd az eredményt a DX:AX regiszterpárban tárolni. Az ehhez hasonló feladatoknál mindig az a megoldás, hogy a két összeadandót azonos méretűre kell hozni. Ez egész pontosan két dolgot jelent: az előjeles számokat előjelesen, az előjelteleneket zéró-kiterjesztésnek kell alávetni. *Zéró-kiterjesztésen* (zero extension) azt értjük, amikor az adott érték felső, hiányzó bitjeit csupa 0-val töltjük fel, ellentétben az előjeles kiterjesztéssel, ahol az előjelbitet használjuk kitöltő értékként. Azt is vegyük figyelembe, hogy az eredmény mindig hosszabb lesz 1 bittel mint a kiinduló tagok hosszának maximuma. Ha ez megvan, akkor jöhet a tényleges összeadás. Itt figyelnünk kell arra, mit és milyen sorrendben adunk össze. Először az alsó bájtokon/ szavakon végezzük el a műveletet. Itt kapunk egy részeredményt, valamint egy esetleges átvitelt, amit a felső bájtok/szavak összeadásakor is figyelembe kell venni.

Ezek alapján nézzünk egy lehetséges megoldást:

```
CBW
CWD
ADD     AX,BX
ADC     DX,0000h
```

Először tisztázzuk az eredmény méretét. Az előjeles szám 8 bites, az előjeltelen 16 bites, ebből 17 bit jön ki. Az alsó 16 bit meghatározása nem nagy kunszt, a CBW utasítás szépen kiterjeszti előjelesen AL-t AX-be, amihez aztán hozzáadhatjuk BX tartalmát. Az eredmény alsó 16 bitje ezzel már megvan. A legfelső bit azonban, mint gondolhatnánk, nem maga az átvitel lesz. Lássuk mondjuk, mi lesz, ha AL=-1, BX=65535. AX-ben előjeles kiterjesztés után 0FFFFh lesz, de ugyanezt fogja BX is tartalmazni. A két számot összeadva 0FFFEh-t kapunk, és CF=1 lesz. Látható, hogy a helyes eredmény is 0FFFEh lesz, nem pedig 1FFFEh. A megoldás kulcsa, hogy az eredményt 24 vagy 32 bitesnek tekintjük. Most az utóbbit választjuk, hiszen DX:AX-ben várjuk az összeget. Innen már következik a módszer: mindkét kiinduló számot 32 bitesnek képzeljük el, s az összeadást is eszerint végezzük el. Az AX-ben levő előjeles számot az CWD (Convert Word to Doubleword) operandus nélküli utasítás DX:AX-be előjelesen kiterjeszti. A másik, BX-ben levő tagnak zéró-kiterjesztésen kellene átesnie, amit mi most kihagyunk, de az összeadás során figyelembe fogunk venni. Miután AX-ben képeztük az eredmény alsó szavát, itt az ideje, hogy a felső szavakat is összeadjuk az átvitellel együtt. A kétoperandusú ADC (ADd with Carry) utasítás annyiban tér el az ADD-től, hogy a célhoz CF zéró-kiterjesztett értékét is hozzáadja. Az előjeles tag felső szava már DX-ben van, a másik tag felső szava azonban 0000h. Ezért az utolsó utasítással DX-hez hozzáadjuk a közvetlen adatként szereplő nullát és CF-et is. Ha mindenképpen zéró kiterjesztést akarunk, akkor így kell módosítani a programot:

```
CBW
```

```
CWD
XOR     CX,CX
ADD     AX,BX
ADC     DX,CX
```

CX helyett persze használhatunk más regisztert is a nulla tárolására. A példát befejezve, DX 0FFFFh-t fog tartalmazni, amihez a 0000h-t és az átvitelt hozzáadva DX is nullává válik. DX:AX így a helyes eredményt fogja tartalmazni, ami 0000FFFEh.

A feladatban helyettesítsük most az összeadást kivonással, tehát szeretnénk az AL-ben levő előjeles számból kivonni a BX-ben levő előjel nélküli számot, az eredményt ugyancsak DX:AX-ben várjuk. A megoldás a következő lehet:

```
CBW
CWD
SUB     AX,BX
SBB     DX,0000h
```

Teljesen világos, hogy az összeadásokat a programban is kivonásra kicserélve célhoz érünk. Az ADC párja az SBB (SuBtract with Borrow), ami a SUB-tól csak annyiban tér el, hogy a célból CF zéró-kiterjesztett értékét is kivonja.

Az összes additív utasítás az összes aritmetikai flag-et, tehát a CF, PF, AF, ZF, SF és OF flag-eket módosítja.

Térjünk most rá a szorzásra és osztásra. A gépi aritmetika tárgyalásánál már említettük, hogy szorozni és osztani sokkal körülményesebb, mint összeadni és kivonni. A gondot az előjeles és előjeltelen számok csak tovább bonyolítják. Azt is említettük, hogy előjeles esetben a tagok előjelét le kell választani a műveletek elvégzéséhez miután megállapítottuk az eredmény előjelét. Ezen okból mind szorzásból mind osztásból létezik

előjeles és előjel nélküli változat is. Mindegyik utasításban közös, hogy az egyik forrás tag és az eredmény helye is rögzítve van, továbbá mindegyik utasítás egyetlen operandust kap, ami csak egy általános regiszter vagy memóriahivatkozás lehet. Közvetlen értékkel tehát nem szorozhatunk, de nem is oszthatunk.

Nézzük meg először az előjel nélküli változatokat, ezek az egyszerűbbek.

Szorozni a MUL (unsigned MULtiplication) utasítással tudunk. Ennek egyetlen operandusa az egyik szorzó tagot (multiplier) tartalmazza. Az operandus mérete határozza meg a továbbiakat: ha 8 bites az operandus, akkor AL-t szorozza meg azzal, az eredmény pedig AX-be kerül. Ha szó méretű volt az operandus, akkor másik tagként AX-et használja, az eredmény pedig DX:AX-ben keletkezik.

Osztásra a DIV (unsigned DIVision) utasítás szolgál, ezzel már korábban találkoztunk, de azért felelevenítjük használatát. Az egyetlen operandus jelöli ki az osztót (divisor). Ha ez bájt méretű, akkor AX lesz az osztandó (dividend), és a hányados (quotient) AL-be, a maradék (remainder) pedig AH-ba fog kerülni. Ha az operandus szavas volt, akkor DX:AX-et osztja el vele, majd a hányados AX-be, a maradék pedig DX-be kerül.

Az előjeles esetben mindkét utasítás ugyanazokat a regisztereket használja a forrás illetve a cél tárolására, mint az előjel nélküli változatok.

Előjeles szorzást az IMUL (Integer signed MULtiplication) utasítással hajthatunk végre. Az eredmény előjele a szokásos szabály szerint lesz meghatározva, tehát a két forrás előjelbitjét logikai KIZÁRÓ VAGY kapcsolatba hozva kapjuk meg.

Végül előjelesen osztani az IDIV (Integer signed DIVision) utasítás alkalmazásával tudunk. A hányados előjele ugyanúgy lesz meghatározva, mint az IMUL esetén, míg a maradék az osztandó előjelét örökli.

Ezek az utasítások elég "rendetlenül" módosítják a flag-eket: a szorzások a CF és OF flag-et változtatják meg, míg a PF, AF, ZF és SF flag-ek értéke meghatározatlan, az osztó utasítások viszont az összes előbb említett flag-et definiálatlan állapotban hagyják (azaz nem lehet tudni, megváltozik-e egy adott flag értéke, s ha igen, mire és miért).

Az utasítások használatának szemléltetésére nézzünk meg egy kicsit összetett példát! Tegyük fel, hogy a következő kifejezés értékét akarjuk meghatározni:

$$\frac{A*B+C*D}{(E-F)/G}$$

A betűk hét számot jelölnek, ezek közül A és B előjel nélküli bájtok, C és D előjeles bájtok, E és F előjel nélküli szavak, és végül G előjel nélküli bájt. Feltesszük, hogy E nagyobb vagy egyenlő F-nél. Az eredmény egy előjeles szó lesz, amit műveletek elvégzése után AX-ben kapunk meg. Az osztásoknál a maradékkal nem törődünk, így az eredmény csak közelítő pontosságú lesz. Hasonlóan nem foglalkozunk az esetleges túlcsordulásokkal sem. Lássuk hát a megoldást:

```
MOV     AX,[E]
SUB     AX,[F]
DIV     [G]
MOV     CL,AL
XOR     CH,CH
MOV     AL,[A]
MUL     [B]
MOV     BX,AX
MOV     AL,[C]
IMUL    [D]
CWD
ADD     AX,BX
ADC     DX,0000h
IDIV    CX
```

Érdemes átgondolni, hogy a 6 műveletet milyen sorrendben célszerű elvégezni. Először most az E-F kivonást hajtjuk végre, amit rögvest elosztunk G-vel, az eredményt berakjuk CL-be, s ezt rögtön zéró-kiterjesztésnek vetjük alá, azaz CH-t kinullázzuk. Ezután, hogy minél kevesebb regiszter művelet legyen, az előjeltelen szorzást végezzük el előbb, az eredményt BX-ben tároljuk. Most jön a másik, előjeles szorzat kiszámolása, aminek az eredményét rögtön előjelesen kiterjesztjük DX:AX-be, hogy az összeadást végre tudjuk hajtani. Az összeg meghatározása után elvégezzük a nagy előjeles osztást, s ezzel AX-ben kialakul a végleges eredmény.

9.2 Bitforgató utasítások

Következő témánk a bitforgató utasításokat tekinti át. Ezek alapvetően két csoportra oszthatók: shiftelő és rotáló utasításokra. Az összes ilyen utasítás közös jellemzője, hogy céloperandusuk általános regiszter vagy memóriahivatkozás lehet, míg a második operandus a léptetés/forgatás számát adja meg bitekben. Ez az operandus vagy a közvetlen 1-es érték, vagy a CL regiszter lehet. (A TASM megenged 1-nél nagyobb közvetlen értéket is, ekkor a megfelelő utasítás annyiszor lesz lekódolva. Így pl. az SHL AX,3 hatására 3 db. SHL AX,1 utasítást fog az assembler generálni.) CL-nek minden bitjét figyelembe veszi a 8086-os proci, így pl. akár 200-szor is léptethetünk. A legutoljára kicsorgó bit értékét minden esetben CF tartalmazza.

A shiftelés fogalmát már definiáltuk a gépi aritmetika elemzése közben, ezért itt csak annyit említünk meg, hogy shiftelésből megkülönböztetünk előjeles (ez az ú.n. *aritmetikai*)

és előjeltelen (ez a *logikai*) változatot, és mindkettőből van balra és jobbra irányuló utasítás is. Mindegyik shiftelő utasítás módosítja a CF, PF, SF, ZF és OF flag-eket, AF értéke meghatározatlan lesz.

Balra shiftelni az SHL (SHift logical Left) és az SAL (Shift Arithmetical Left) utasításokkal tudunk, közülük a második szolgál az előjeles léptetésre. Balra shiftelésnél azonban mindegy, hogy a céloperandus előjeles vagy előjeltelen érték-e, ezért, bár két mnemonik létezik, ténylegesen csak 1 utasítás van a gépi kód szintjén. Ezért ne csodálkozzunk, ha mondjuk a Turbo Debugger nem ismeri az SAL mnemonikot (de például a JC-t sem ismeri…).

Jobbra shiftelésnél már valóban meg kell különböztetni az előjeles változatot az előjel nélkülitől. Az SHR (SHift logical Right) előjel nélküli, míg az SAR (Shift Arithmetical Right) előjeles léptetést hajt végre a céloperanduson.

Nézzünk most egy egyszerű példát a shiftelés használatára. Tegyük fel, hogy AL-ben egy előjeles bájt van, amit szeretnénk megszorozni 6-tal, az eredményt pedig AX-ben várjuk. Ezt igazán sokféle módszerrel meg lehet oldani (LEA, IMUL, ADD, SUB stb.), de mi most léptetések segítségével tesszük meg. A megoldás majdnem triviális: a 6-tal való szorzás ugyanazt jelenti, mintha az eredeti szám dupláját és négyszeresét adnánk össze.

```
CBW
SHL     AX,1
MOV     BX,AX
SHL     AX,1
ADD     AX,BX
```

Ennél szebb megoldás is lehetséges, de a lényeg ezen is látszik. A program működése remélhetőleg mindenkinek világos.

Hasonló elgondolással megoldható az is, ha mondjuk az előjeles AL tartalmát meg akarjuk szorozni 3/2-del, az eredményt itt is AX-ben várva.

```
CBW
MOV     BX,AX
SAR     AX,1
ADD     AX,BX
```

Rotáláson **(forgatáson) azt a, léptetéshez igen hasonló műveletet értjük, amikor az operandus bitjei szépen egymás után balra vagy jobbra lépnek, miközben a kicsorgó bit a túloldalon visszalép. Ez a fajta rotálás az operandus értékét "megőrzi", legalábbis olyan értelemben, hogy ha mondjuk egy bájtot 8-szor balra vagy jobbra rotálunk, az eredeti változatlan bájtot kapjuk vissza. Rotálni nyilván csak előjeltelen értékeket lehet (tehát az előjelbitnek itt nincs speciális szerepe). Balra a ROL (ROtate Left), jobbra pedig a ROR (ROtate Right) mnemonikokkal lehet forgatni. Bármelyik forgató utasítás csak a CF és OF értékét módosítja.**

A forgatásnak van egy másik változata is, ami egy igen hasznos tulajdonsággal bír. A forgatást ugyanis úgy is el lehet végezni, hogy nem a kilépő bit fog belépni, hanem CF forgatás előtti értéke. Szemléletesen ez annyit jelent, mintha a CF bit az operandus egy plusz bitje lenne: balra forgatás esetén a legfelső utáni, míg jobbra forgatáskor a legalsó bit előtti bit szerepét tölti be. Ilyen módon balra az RCL (Rotate through Carry Left), jobbra az RCR (Rotate through Carry Right) utasítás forgat.

Forgatást például olyankor használunk, ha mondjuk szeretnénk az operandus minden egyes bitjét egyenként végigvizsgálni. A következő program például a BL-ben levő érték bináris alakját írja ki a képernyőre.

```
        MOV     AH,02h
        MOV     CX,8
@Ciklus:
        MOV     DL,'0'
        ROL     BL,1
        ADC     DL,00h
        INT     21h
        LOOP    @Ciklus
```

Az algoritmus működése azon alapul, hogy a ROL a kiforgó bitet CF-be is betölti, amit azután szépen hozzáadunk a "0" karakter kódjához. A forgatást nyolcszor elvégezve BL-ben újra a kiinduló érték lesz, s közben már a kiírás is helyesen megtörtént.

Szintén jó szolgálatot tehet valamelyik normál rotáló utasítás, ha egy regiszter alsó és felső bájtját, vagy alsó és felső bitnégyesét akarjuk felcserélni. Például az

```
ROL     AL,4
ROR     SI,8
```

utasítások hatására AL alsó és felső fele megcserélődik, majd SI alsó és felső bájtjával történik meg ugyanez.

Ha saját magunk akarunk megvalósítani nagy pontosságú aritmetikát, akkor is sokszor nyúlunk a shiftelő és rotáló utasításokhoz. Tegyük fel mondjuk, hogy a shiftelést ki akarjuk terjeszteni duplaszavakra is. A következő két sor a DX:AX-ben levő számot lépteti egyszer balra:

```
SHL     AX,1
RCL     DX,1
```

Az alapmódszer a következő: balra léptetésnél az első utasítás az SHL/SAL legyen, a többi pedig az RCL, és a legalsó bájttól haladunk a legfelső, legértékesebb bájt felé. Jobbra léptetés esetén a helyzet a fordítottjára változik: a legfelső bájttól

haladunk lefelé, az első utasítás a szám típusától függően SHR vagy SAR legyen, a többi pedig RCR. Egyszerre csak 1 bittel tudjuk ilyen módon léptetni a számot, de így is legfeljebb 7 léptetésre lesz szükség, mivel a LepesSzam db. bittel való léptetést megoldhatjuk, ha a számot LepesSzam MOD 8 -szor léptetjük, majd az egész területet LepesSzam/8 bájttal magasabb címre másoljuk.

9.3 Bitmanipuláló utasítások

Utolsó témakörünk a bitmanipulációk (beállítás, törlés, negálás és tesztelés) megvalósításával foglalkozik.

Elsőként a logikai utasítások ilyen célú felhasználását nézzük meg. Ebbe a csoportba 5 olyan utasítás tartozik, amik az operandusok között ill. az operanduson valamilyen logikai műveletet hajtanak végre bitenként. 3 utasítás (AND, OR és XOR) kétoperandusú, a művelet eredménye minden esetben a céloperandusba kerül. A cél általános regiszter vagy memóriahivatkozás lehet, forrásként pedig ezeken kívül közvetlen értéket is írhatunk. A TEST szintén kétoperandusú, viszont az eredményt nem tárolja el. A NOT utasítás egyoperandusú, ez az operandus egyben forrás és cél is, típusa szerint általános regiszter vagy memóriahivatkozás lehet. Az utasítások elnevezése utal az általuk végrehajtott logikai műveletre is, ezért bővebben nem tárgyaljuk őket, a gépi logika leírásában megtalálható a szükséges információ.

Az AND, OR, XOR és TEST utasítások a flag-eket egyformán kezelik: CF-et és OF-et 0-ra állítják, AF meghatározatlan lesz, PF, ZF és SF pedig módosulhatnak. A NOT utasítás nem változtat egyetlen flag-et sem.

A TEST különleges logikai utasítás, mivel a két operandus között bitenkénti logikai ÉS műveletet végez, a flag-eket ennek megfelelően beállítja, viszont a két forrást nem bántja, és a művelet eredményét is eldobja.

Egyszerű logikai azonosságok alkalmazása által ezekkel az utasításokkal képesek vagyunk különféle bitmanipulációk elvégzésére. Az alapprobléma a következő: adott egy bájt, amiben szeretnénk a 2-es bitet 0-ra állítani, a 4-es bitet 1-be billenteni, a 6-os bit értékét negálni, illetve kíváncsiak vagyunk, hogy az előjelbit (7-es bit) milyen állapotú. A bájtot tartalmazza most mondjuk az AL regiszter. Minden kérdés egyetlen utasítással megoldható:

```
AND     AL,0FBh
OR      AL,10h
XOR     AL,40h
TEST    AL,80h
```

A bit törléséhez a logikai ÉS művelet két jól ismert tulajdonságát használjuk fel: Bit AND 1=Bit, illetve Bit AND 0=0. Ezért ha AL-t olyan értékkel (*bitmaszkkal*) hozzuk logikai ÉS kapcsolatba, amiben a törlendő bit(ek) helyén 0, a többi helyen pedig csupa 1-es áll, akkor készen is vagyunk.

Bitek beállításához a logikai VAGY műveletet és annak két tulajdonságát hívjuk segítségül: Bit OR 0=Bit, valamint Bit OR 1=1. AL-t ezért olyan maszkkal kell VAGY kapcsolatba hozni, amiben a beállítandó bit(ek) 1-es, a többiek pedig 0-ás értéket tartalmaznak.

Ha egy bit állapotát kell negálni (más szóval *invertálni* vagy komplementálni), a logikai KIZÁRÓ VAGY művelet jöhet szóba. Ennek két tulajdonságát használjuk most ki: Bit XOR 0= Bit, és Bit XOR 1=NOT Bit. A használt maszk ezek alapján ugyanúgy fog felépülni, mint az előbbi esetben.

Ha az összes bitet negálni akarjuk, azt megtehetjük az XOR operandus,11…1b utasítással, de ennél sokkal egyszerűbb

a NOT operandus utasítás használata, ez ráadásul a flag-eket sem piszkálja meg. (Az 11…1b jelölés egy csupa 1-esekből álló bináris számot jelent.)

Ha valamely bit vagy bitek állapotát kell megvizsgálnunk (más szóval *tesztelnünk*), a célravezető megoldás az lehet, hogy az adott operandust olyan maszkkal hozzuk ÉS kapcsolatba, ami a tesztelendő bitek helyén 1-es, a többi helyen 0-ás értékű. Ha a kapott eredmény nulla (ezt ZF=1 is jelezni fogja), akkor a kérdéses bitek biztosan nem voltak beállítva. Különben két eset lehetséges: ha egy bitet vizsgáltunk, akkor az a bit tutira be volt állítva, ha pedig több bitre voltunk kíváncsiak, akkor a bitek közül valamennyi (de legalább egy) darab 1-es értékű volt az operandusban. Ha ez utóbbi dologra vagyunk csak kíváncsiak (tehát a bitek értéke külön-külön nem érdekel bennünket, csak az a fontos, hogy legalább 1 be legyen állítva), akkor felesleges az AND utasítást használni. A TEST nem rontja el egyik operandusát sem, és ráadásul a flag-eket az ÉS művelet eredményének megfelelően beállítja.

A bitmanipuláció speciális esetét jelenti a Flags regiszter egyes bitjeinek, azaz a flag-eknek a beállítása, törlése stb. Az aritmetikai flag-eket (CF, PF, AF, ZF, SF, OF) elég sok utasítás képes megváltoztatni, ezek eredményét részben mi is irányíthatjuk megfelelő operandusok választásával. A CF, DF és IF flag értékét külön-külön állíthatjuk bizonyos utasításokkal: törlésre a CLC, CLD és CLI (CLear Carry/Direction/Interrupt flag), míg beállításra az STC, STD és STI (SeT Carry/Direction/ Interrupt flag) utasítások szolgálnak. Ezenkívül CF-et negálhatjuk a CMC (CoMplement Carry flag) utasítással. Mindegyik említett utasítás operandus nélküli, és más flag-ek értékét nem befolyásolják.

Ha egy olyan flag értékét akarjuk beállítani, amit egyéb módon nehézkes lenne (pl. AF), vagy egy olyan flag értékére vagyunk kíváncsiak, amit eddig ismert módon nem tudunk írni/

olvasni (pl. TF), akkor más megoldást kell találni. Két lehetőség is van: vagy a PUSHF/POPF, vagy a LAHF/SAHF utasítások valamelyikét használjuk. Mindegyik utasítás operandus nélküli.

A PUSHF (PUSH Flags) utasítás a Flags regiszter teljes tartalmát (mind a 16 bitet) letárolja a verembe, egyébként működése a szokásos PUSH művelettel egyezik meg. A POPF (POP Flags) ezzel szemben a verem tetején levő szót leemeli ugyanúgy, mint a POP, majd a megfelelő biteken szereplő értékeket sorban betölti a flag-ekbe. (Ezt azért fogalmaztuk így, mivel a Flags néhány bitje kihasználatlan, így azokat nem módosíthatjuk.)

A másik két utasítás kicsit másképp dolgozik. A LAHF (Load AH from Flags) a Flags regiszter alsó bájtját az AH regiszterbe másolja, míg a SAHF (Store AH into Flags) AH 0, 2, 4, 6 és 7 számú bitjeit sorban a CF, PF, AF, ZF és SF flag-ekbe tölti be. Más flag-ek ill. a verem/veremmutató értékét nem módosítják.

A szemléltetés kedvéért most AF-et mind a kétféle módon negáljuk:

1)

```
PUSHF
POP     AX
XOR     AL,10h
PUSH    AX
POPF
```

2)

```
LAHF
XOR     AH,10h
SAHF
```

10 AZ ASSEMBLY NYELV KAPCSOLATA MAGAS SZINTŰ NYELVEKKEL, PARAMÉTERÁTADÁS FORMÁI

Ebben a fejezetben azt vizsgáljuk, hogy egy önálló Assembly program esetén egy adott eljárásnak/függvénynek milyen módon adhatunk át paramétereket, függvények esetében hogyan kaphatjuk meg a függvény értékét, illetve hogyan tudunk lokális változókat kezelni, azaz hogyan valósíthatjuk meg mindazt, ami a magas szintű nyelvekben biztosítva van. Ezután megnézzük, hogy a Pascal és a C pontosan hogyan végzi ezt el, de előbb tisztázzunk valamit:

Ha a meghívott szubrutinnak (alprogramnak) nincs visszatérési értéke, akkor *eljárásnak* (procedure), egyébként pedig *függvénynek* (function) nevezzük. Ezt csak emlékeztetőül mondjuk, no meg azért is, mert Assemblyben formálisan nem tudunk különbséget tenni eljárás és függvény között, mivel mindkettőt a PROC/ENDP párral deklaráljuk, viszont ebben a deklarációban nyoma sincs sem paramétereknek, sem visszatérési értéknek, ellentétben pl. a Pascallal (PROCEDURE, FUNCTION), vagy C-vel, ahol végül is minden függvény, de ha visszatérési értéke VOID, akkor eljárásnak minősül, s a fordítók is ekként kezelik őket.

10.1 Paraméterek átadása regisztereken keresztül

Ez azt jelenti, hogy az eljárásokat/függvényeket (a továbbiakban csak eljárásoknak nevezzük őket) úgy írjuk meg,

hogy azok bizonyos regiszterekben kérik a paramétereket. Hogy pontosan mely regiszterekben, azt mi dönthetjük el tetszőlegesen (persze ésszerű keretek között, hiszen pl. a CS-t nem használjuk paraméterátadásra), de nem árt figyelembe vennünk azt sem, hogy hogy a legérdemesebb ezek megválasztása. A visszatérési értékeket (igen, mivel nem egy regiszter van, ezért több dolgot is visszaadhatunk) is ezeken keresztül adhatjuk vissza. Erre rögtön egy példa: van egy egyszerű eljárásunk, amely egy bájtokból álló vektor elemeit összegzi. Bemeneti paraméterei a vektor címe, hossza. Visszatérési értéke a vektor elemeinek előjeles összege. Az eljárás feltételezi, hogy az összeg elfér 16 biten.

Pelda5.ASM:

```
Osszegez        PROC
        PUSH    AX
        PUSHF
        CLD                     ;SI-nek növekednie kell majd
        XOR     BX,BX               ;Kezdeti összeg = 0
@AddKov:
        LODSB                   ;Következő vektorkomponens AL-be
        CBW                     ;előjelesen kiterjesztjük szóvá
        ADD     BX,AX               ;hozzáadjuk a meglevő
összeghez
        LOOP    @AddKov         ;Ismételd, amig CX > 0
        POPF
        POP     AX
        RET
Osszegez        ENDP
.
.
.
;Példa eljárás meghívására
        MOV     CX,13                       ;13 bájt hosszú a
vektor
```

```
        MOV     SI,OFFSET Egyikvektor
                                ;cím offszetrésze
                                ;feltesszük,hogy DS az
                                ;adatszegmensre mutat
        CALL    Osszegez
        MOV     DX,BX           ;Pl. a DX-be töltjük az
                                ;eredményt
.
.
.
;valahol az adatszegmensben deklarálva vannak a változók:
Hossz           DW    ?
Cim             DD    ?
Eredmeny        DW    ?
Probavektor     DB    13 DUP (?)
```

Ezt az eljárást úgy terveztük, hogy a vektor címét a DS:SI regiszterpárban, a vektor hosszát a CX regiszterben kapja meg, ugyanis az eljárás szempontjából ez a legelőnyösebb (a LODSB és LOOP utasításokat majdnem minden előkészület után használhattuk). Az összeget az eljárás a BX regiszterben adja vissza.

Érdemes egy pillantást vetni a PUSHF/POPF párra: az irányjelzőt töröljük, mert nem tudhatjuk, hogy az eljárás hivásakor mire van állítva, de éppen e miatt fontos, hogy meg is őrizzük azt. Ezért mentettük el a flageket.

Az AX regiszter is csak átmeneti „változó”, jobb ha ennek értékét sem rontjuk el.

A példában egy olyan vektor komponenseit összegeztük, amely globális változóként az adatszegmensben helyezkedik el.

10.2 Paraméterek átadása globális változókon keresztül

Ez a módszer analóg az előzővel, annyi a különbség, hogy ilyenkor a paramétereket bizonyos globális változókba teszünk, s az eljárás majd onnan olvassa ki őket. Magas szintű nyelvekben is megtehetjük ezt, azaz nem paraméterezzük fel az eljárást, de az felhasznál bizonyos globális változókat bemenetként (persze ez nem túl szép megoldás). Ennek mintájára a visszatérési értékeket is átadhatjuk globális változókban. Írjuk át az előző példát:

Pelda6.ASM:

```
Osszegez        PROC
        PUSH    AX
        PUSH    BX
        PUSHF
        CLD                     ;SI-nek növekednie kell majd
        XOR     BX,BX               ;Kezdeti összeg = 0
        MOV     CX,Hossz
        LDS     SI,Cim
@AddKov:
        LODSB                   ;Következő vektorkomponens AL-be
        CBW                     ;előjelesen kiterjesztjük szóvá
        ADD     AX,BX               ;hozzáadjuk a meglevő
összeghez

        LOOP    @AddKov         ;Ismételd, amig CX > 0
        MOV     Eredmeny,BX
        POPF
        POP     BX
        POP     AX
        RET
Osszegez        ENDP
.
.
.
;Példa az eljárás meghivására
```

```
        MOV     Hossz,13            ;13 bájt hosszú a vektor
        MOV     WORD PTR Cim[2],DS
                                    ;ami az adatszegmensben
                                    ;van, így a cím szegmens-
                                    ;része az DS
        MOV     WORD PTR Cim,OFFSET Egyikvektor    ;*
                                    ;offszetrésze

                                    ;cím offszetrésze
                                    ;feltesszük,hogy DS az
                                    ;adatszegmensre mutat
        CALL    Osszegez
        MOV     DX,Eredmeny         ;Pl. a DX-be töltjük az
                                    ;eredményt
.
.
.
;valahol az adatszegmensben deklarálva vannak a változók:
Hossz           DW    ?
Cim             DD    ?
Eredmeny        DW    ?
Egyikvektor     DB   13 DUP (?)
Masikvektor     DB   34 DUP (?)
```

A példához nem kívánunk magyarázatot fűzni, talán csak annyit, hogy mivel itt már a BX is egy átmeneti változóvá lett (mert nem rajta keresztül adjuk vissza az összeget), ezért jobb, ha ennek értékét megőrzi az eljárás.

Van egy dolog, amire azonban nagyon vigyázni kell az effajta paraméterátadással. Képzeljük el, hogy már beírtuk a paramétereket a változókba, de még mielőtt meghívnánk az eljárást (a *-gal jelzett helyen), bekövetkezik egy megszakítás. Tegyük fel, hogy a megszakításban egy saját megszakítási rutinunk hajtódik végre, ami szintén meghívja az Osszegez eljárást. Mi történik akkor, ha ő a Masikvektor komponenseit összegezteti? Ő is beírja a paramétereit ugyanezekbe a globális

változókba, meghívja az eljárást, stb., végetér a megszakítási rutin, és a program ugyanott folytatódik tovább, ahol megszakadt. Esetünkben ez azt jelenti, hogy meghívjuk az eljárást, de nem azokkal a paraméterekkel, amiket beállítottunk, mert a megszakítási rutin felülírta azokat a sajátjaival. Ez természetesen a program hibás működését okozza. Az előző esetben, ahol regisztereken keresztül adtuk át a cuccokat, ott ez nem fordulhatott volna elő, hiszen egy megszakításnak kötelessége elmenteni azokat a regisztereket, amelyeket felhasznál. A probléma megoldása tehát az, hogy ha megszakításban hívjuk ezt az eljárást, akkor meg kell el kell mentenünk, majd vissza kell állítanunk a globális változók értékeit, mintha azok regiszterek lennének.

10.3 Paraméterek átadása a vermen keresztül

Ennek lényege, hogy az eljárás meghívása előtt a verembe beletesszük az összes paramétert, az eljárás pedig kiolvassa őket onnan, de nem a POP művelettel. Mivel a paraméterek sorrendje rögzített, és a utánuk a verembe csak a visszatérési cím kerül, így azok az eljáráson belül az SP-hez relatívan elérhetőek. Ezek szerint olyan címzésmódot kellene használni, amelyben SP közvetlenül szerepel, pl. MOV AX,[SP+6], csakhogy ilyen nem létezik. A másik probléma az SP-vel az lenne, hogy valamit ideiglenesen elmentünk a verembe, akkor a paraméterek relatív helye is megváltozik, így nehézkesebb lenne kezelni őket. Épp erre „találták ki" viszont a BP regisztert, emiatt van, hogy a vele használt címzésmódokban az alapértelmezett szegmens nem a DS, hanem az SS (de már a neve is: Base Pointer - Bázismutató, a vermen belül). A módszert ismét az előző példa átírásával mutatjuk be:

Pelda7.ASM:

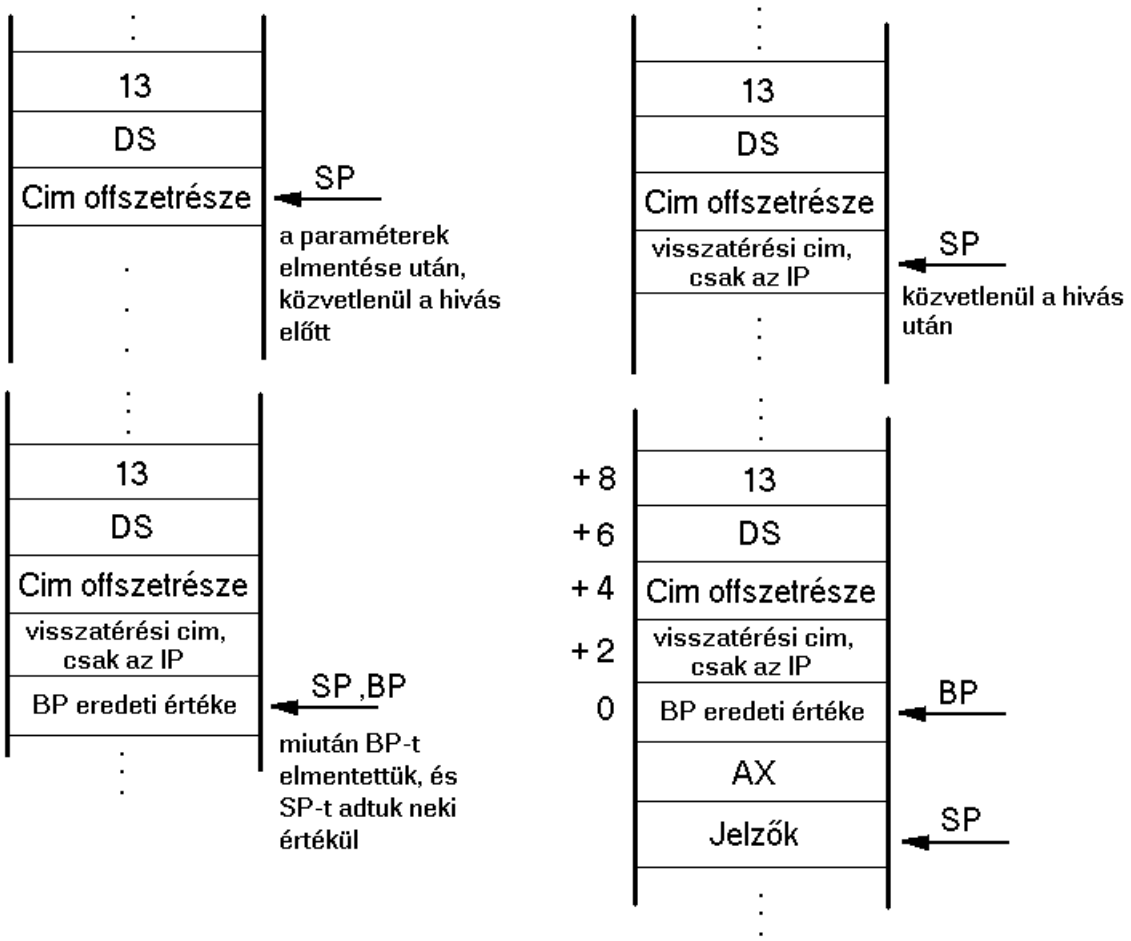
```
Osszegez      PROC

      PUSH    BP
      MOV     BP,SP
      PUSH    AX
      PUSHF
      CLD                ;SI-nek növekednie kell majd
      XOR     BX,BX          ;Kezdeti összeg = 0
      MOV     CX,[BP+8]
      LDS     SI,[BP+4]
@AddKov:
      LODSB              ;Következő vektorkomponens AL-be
      CBW                ;előjelesen kiterjesztjük szóvá
      ADD     BX,AX          ;hozzáadjuk a meglevő
összeghez
      LOOP    @AddKov    ;Ismételd, amig CX > 0
      POPF
      POP     AX
      POP     BP
      RET     6

Osszegez      ENDP
.
.
.
;Példa eljárás meghivására
      MOV     AX,13
      PUSH    AX       ;13 bájt hosszú a vektor
      PUSH    DS
      MOV     AX,OFFSET Egyikvektor
                       ;cím offszetrésze
      PUSH    AX
      CALL    Osszegez
      MOV     DX,BX    ;Pl. a DX-be töltjük az
                       ;eredményt
.
.
.
```

```
;valahol az adatszegmensben deklarálva vannak a változók:
Hossz          DW    ?
Cim            DD    ?
Eredmeny       DW    ?
Probavektor    DB    13 DUP (?)
```

Az eljárásban BP értékét is megőriztük a biztonság kedvéért. A RET utasitásban szereplő operandus azt mondja meg a processzornak, hogy olvassa ki a visszatérési címet, aztán az operandus értékét adja hozzá SP-hez. Ez azért kell, hogy az eljárás a paramétereit kitakaritsa a veremből. Hogy jobban megértsük a folyamatot, ábrákkal illusztráljuk a verem alakulását:



A BP-t tehát ráállítjuk az utolsó paraméterre (pontosabban itt most BP elmentett értékére), ez a bázis, s ehhez képest +4 bájtra található a vektor teljes címe, amit persze úgy helyeztünk a verembe, hogy az alacsonyabb címen szerepeljen

az offszetrész, aztán a szegmens. BP-hez képest pedig +8 bájtra található a vektor hossza. Az eljárásból való visszatérés után persze minden eltűnik a veremből, amit előtte belepakoltunk a hívásához.

A visszatérési értéket most BX-ben adja vissza, erről még lesz szó.

A magas szintű nyelvek is vermen keresztüli paraméterátadást valósítanak meg. Ennek a módszernek kétféle változata is létezik, aszerint, hogy a paraméterek törlése a veremből kinek a feladata:

- Pascal-féle változat: a verem törlése az eljárás dolga, amint azt az előbb is láttuk
- C-féle változat: a verem törlése a hívó fél feladata, erre mindjárt nézünk példát

A Pascal a paramétereket olyan sorrendben teszi be a verembe, amilyen sorrendben megadtuk őket, tehát a legutolsó paraméter kerül bele a verembe utoljára. A C ezzel szemben fordított sorrendben teszi ezt, így a legelső paraméter kerül be utoljára. Ennek megvan az az előnye is, hogy könnyen megvalósítható a változó számú paraméterezés, hiszen ilyenkor az első paraméter relatív helye a bázishoz képest állandó, ugyanígy a másodiké is, stb., csak azt kell tudnia az eljárásnak, hogy hány paraméterrel hívták meg. Most pedig nézzük meg a példát, amikor a hívó törli a vermet:

Pelda8.ASM:

```
Osszegez        PROC

        PUSH    BP
        MOV     BP,SP
        PUSH    AX
        PUSHF
```

```
        CLD                     ;SI-nek növekednie kell majd
        XOR     BX,BX               ;Kezdeti összeg = 0
        MOV     CX,[BP+8]
        LDS     SI,[BP+4]
@AddKov:
        LODSB               ;Következő vektorkomponens AL-be
        CBW                 ;előjelesen kiterjesztjük szóvá
        ADD     BX,AX           ;hozzáadjuk a meglevő
összeghez
        LOOP    @AddKov     ;Ismételd, amig CX > 0
        POPF
        POP     AX
        POP     BP
        RET

Osszegez        ENDP
.
.
.
;Példa eljárás meghivására

        MOV     AX,13
        PUSH    AX              ;13 bájt hosszú a vektor
        PUSH    DS
        MOV     AX,OFFSET Egyikvektor
                                ;cím offszetrésze
        PUSH    AX
        CALL    Osszegez
        ADD     SP,6
        MOV     DX,BX           ;Pl. a DX-be töltjük az
                                ;eredményt
```

Látható, hogy ilyenkor a visszatérési értéket is át lehetne adni a vermen keresztül úgy, hogy pl. az egyik paraméter helyére beírjuk azt, s a hívó fél onnan olvassa ki, mielőtt törölné a vermet (a Pascal-szerű verzió esetén ez persze nem lehetséges). Ezt azonban nem szokás alkalmazni, a C is mindig regiszterekben adja ezt vissza, ez a könnyebben kezelhető megoldás, mi is tegyünk így.

10.4 Lokális változók megvalósítása

A lokális változókat kétféleképpen valósíthatjuk meg: elmentjük valamely regisztert, s felhasználjuk azt változóként, ugyanúgy, ahogy idáig is tettük az Osszegez eljárásban az AX-szel. A másik megoldás, amit a magas szintű nyelvek is alkalmaznak, hogy maga az eljárás kezdetben foglal a veremben a lokális változóknak helyet, ami annyit tesz, hogy SP értékét lecsökkenti valamennyivel. A változókat továbbra is a bázis-technikával (a BP-n keresztül) éri el. Írjuk át ismét az Osszegez-t úgy, hogy ezt a módszert alkalmazza (most a lokális változóban tároljuk ideiglenesen az összeget):

Pelda9.ASM:

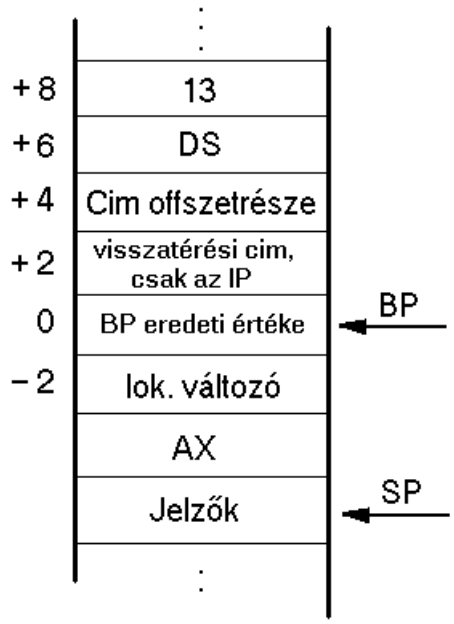
```
Osszegez   PROC

      PUSH    BP
      MOV     BP,SP
      SUB     SP,2            ;Helyet foglalunk a lokális
                              ;változónak
      PUSH    AX
      PUSHF
      CLD                     ;SI-nek növekednie kell majd
      MOV     WORD PTR [BP-2],0h
                              ;Kezdeti összeg = 0
      MOV     CX,[BP+8]
      LDS     SI,[BP+4]
AddKov:
      LODSB               ;Következő vektorkomponens AL-be
      CBW                 ;előjelesen kiterjesztjük szóvá
      ADD     [BP-2],AX;hozzáadjuk a meglevő összeghez
      LOOP    AddKov      ;Ismételd, amig CX > 0
      POPF
      POP     AX
```

```
        MOV     BX,[BP-2]       ;BX = lok. változó
        ADD     SP,2            ;lok. vált. helyének
                                ;felszabadítása
        POP     BP
        RET

Osszegez        ENDP
.
.
.
;Példa eljárás meghívására
        MOV     AX,13
        PUSH    AX              ;13 bájt hosszú a vektor
        PUSH    DS
        MOV     AX,OFFSET Egyikvektor
                                ;cím offszetrésze
        PUSH    AX
        CALL    Osszegez
        ADD     SP,6
        MOV     DX,BX           ;Pl. a DX-be töltjük az
                                ;eredményt
```

Hogyan is néz ki a verem az eljárás futásakor?



11 MŰVELETEK SZTRINGEKKEL

Sztringen bájtok vagy szavak véges hosszú folyamát értjük. A magas szintű nyelvekben is találkozhatunk ezzel az adattípussal, de ott általában van valamilyen megkötés, mint pl. a rögzített maximális hossz, kötött elhelyezkedés a memóriában stb. Assemblyben sztringen nem csak a szegmensekben definiált karakteres sztring-konstansokat értjük, hanem a memória tetszőleges címén kezdődő összefüggő területet. Ez azt jelenti, hogy mondjuk egy 10 bájt méretű változót tekinthetünk 10 elemű tömbnek (vektornak), de kedvünk szerint akár sztringnek is; vagy például a verem tartalmát is kezelhetjük sztringként.

Mivel minden regiszter 16 bites, valamint a memória is szegmentált szervezésű, ezeknek természetes következménye, hogy a sztringek maximális hossza egy szegmensnyi, tehát 64 Kbájt. Persze ha ennél hosszabb folyamokra van szükség, akkor megfelelő feldarabolással ez is megoldható.

A 8086-os mikroprocesszor 5 utasítást kínál a sztringekkel végzett munka megkönnyítésére. Ezek közül kettővel már találkoztunk eddig is. Az utasítások közös jellemzője, hogy a forrássztring címét a DS:SI, míg a cél címét az ES:DI regiszterpárból olvassák ki. Ezek közül a DS szegmenst felülbírálhatjuk, minden más viszont rögzített. Az utasítások csak bájtos vagy szavas elemű sztringekkel tudnak dolgozni. Szintén közös tulajdonság, hogy a megfelelő művelet elvégzése után mindegyik utasítás a sztring(ek) következő elemére állítja SI-t és/vagy DI-t. A következő elemet most kétféleképpen értelmezhetjük: lehet a megszokott, növekvő irányú, illetve lehet fordított, csökkenő irányú is. A használni kívánt irányt a DF flag állása választja ki: DF=0 esetén növekvő, míg DF=1 esetén csökkenő lesz.

Ugyancsak jellemző mindegyik utasításra, hogy az Assembly szintjén több mnemonikkal és különféle operandus-számmal érhetők el. Pontosabban ez azt takarja, hogy mindegyiknek létezik 2, operandus nélküli rövid változata, illetve egy olyan változat, ahol "áloperandusokat" adhatunk meg. *Áloperandus* (pseudo operand) alatt most egy olyan operandust értünk, ami csak szimbolikus, és nem adja meg a tényleges operandus címét/helyét, csak annak méretét, szegmensét jelöli ki. Ezekre a már említett megkötések (DS:SI és ES:DI) miatt van szükség. Az áloperandussal rendelkező mnemonikok a CMPS, LODS, MOVS, SCAS és STOS, míg az egyszerű alakok a CMPSB/CMPSW, LODSB/LODSW, MOVSB/MOVSW, SCASB/SCASW és STOSB/STOSW. Minden rögtön világosabb lesz, ha nézünk rájuk egy példát:

Pelda10.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Szoveg1     DB      "Ez az első szöveg.",00h
Szoveg2     DB      "Ez a második szöveg.",00h
Kozos       DB      10 DUP (?)
Hossz       DW      ?
Vektor      DW      100 DUP (?)
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            MOV     ES,AX
```

```
            LEA     SI,[Szoveg1]
            LEA     DI,[Szoveg2]
            CLD
            XOR     CX,CX
@Ciklus1:
            CMPSB
            JNE     @Vege
            INC     CX
            JMP     @Ciklus1
@Vege:
            LEA     SI,[SI-2]
            LEA     DI,[Kozos]
            ADD     DI,CX
            DEC     DI
            STD
        REP MOVS    ES:[Kozos],DS:[SI]
            LEA     DI,[Szoveg1]
            MOV     CX,100
            XOR     AL,AL
            CLD
     REPNE  SCASB
            STC
            SBB     DI,OFFSET Szoveg1
            MOV     [Hossz],DI
            PUSH    CS
            POP     DS
            XOR     SI,SI
            LEA     DI,[Vektor]
            MOV     CX,100
@Ciklus2:
            LODSB
            CBW
            NOT     AX
            STOSW
            LOOP    @Ciklus2
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

A program nem sok értelmeset csinál, de szemléltetésnek megteszi.

Először meg akarjuk tudni, hogy a Szoveg1 és a Szoveg2, 00h kódú karakterrel terminált sztringek elején hány darab megegyező karakter van. Ezt "hagyományos" módon úgy csinálnánk, hogy egy ciklusban kiolvasnánk az egyik sztring következő karakterét, azt összehasonlítanánk a másik sztring megfelelő karakterével, majd az eredmény függvényében döntenénk a továbbiakról. Számláláshoz CX-et használjuk. A ciklus most is marad csakúgy, mint a JNE-JMP páros. Az összehasonlítást viszont másképp végezzük el.

Első sztringkezelő utasításunk mnemonikja igen hasonlít az egész aritmetikás összehasonlításra. A CMPS (CoMPare String) mnemonik kétoperandusú utasítás. Mindkét operandus memóriahivatkozást kifejező áloperandus, és rendhagyó módon az első operandus a forrás, míg a második a cél. Az utasítás szintaxisa tehát a következő:

CMPS forrás,cél

Az utasítás a forrást összehasonlítja a céllal, beállítja a flag-eket, de az operandusokat nem bántja. A forrás alapértelmezésben a DS:SI, a cél pedig az ES:DI címen található, mint már említettük. Az első operandus szegmensregisztere bármi lehet, a másodiké kötelezően ES. Ha egyik operandusból sem derül ki azok mérete (bájt vagy szó), akkor a méretet nekünk kell megadni a BYTE PTR vagy a WORD PTR kifejezés valamelyik operandus elé írásával. A PTR operátor az ő jobb oldalán álló kifejezés típusát a bal oldalán álló típusra változtatja meg, ezt más nyelvekben hasonló módon típusfelülbírálásnak (type casting/overloading) hívják. Ha nem kívánunk az operandusok megadásával bajlódni, akkor használhatunk két másik rövidítést. A CMPSB és CMPSW (CoMPare String Byte/Word) operandus nélküli mnemonikok

rendre bájt ill. szó elemű sztringeket írnak elő a DS:SI és ES:DI címeken, tehát formálisan a CMPS BYTE/WORD PTR DS:[SI], ES:[DI] utasításnak felelnek meg.

A CMPSB utasítás tehát a sztringek következő karaktereit hasonlítja össze, majd SI-t és DI-t is megnöveli 1-gyel, hiszen DF=0 volt. Ha ZF=0, akkor vége a ciklusnak, különben növeljük CX-et, és visszaugrunk a ciklus elejére.

A @Vege címkére eljutva már mindenképpen megtaláltuk a közös rész hosszát, amit CX tartalmaz. SI és DI a két legutóbb összehasonlított karaktert követő bájtra mutat, így SI-t kettővel csökkentve az a közös rész utolsó karakterére fog mutatni.

Ezt követően a közös karakterláncot a Kozos változó területére fogjuk másolni. Mivel SI most a sztring végére mutat, célszerű, ha a másolást fordított irányban végezzük el. Ennek megfelelően állítjuk be DI értékét is. DF-et 1-re állítva készen állunk a másolásra.

Következő új sztringkezelő utasításunk a MOVS/MOVSB/ MOVSW (MOVe String Byte/Word). Az utasítás teljes alakja a következő:

MOVS cél,forrás

Az utasítás a forrás sztringet átmásolja a cél helyére, flag-et persze nem módosít. A példában jól látszik, hogy a megadott operandusok tényleg nem valódi címre hivatkoznak: célként nem az ES:[Kozos] kifejezés tényleges címe lesz használva, hanem az ES:DI által mutatott érték, de forrásként is adhattunk volna meg bármit SI helyett. Ezek az operandusok csak a program dokumentálását, megértését segítik, belőlük az assembler csak a forrás szegmenst (DS) és a sztring elemméretét (ami most a Kozos elemmérete, tehát bájt) használja fel, a többit figyelmen kívül hagyja. A MOVSB/MOVSW mnemonikok a CMPS-hez hasonlóan formálisan a MOVS BYTE/WORD PTR ES:[DI],DS:[SI] utasítást rövidítik.

Ha egy adott sztring minden elemére akarunk egy konkrét sztringműveletet végrehajtani, akkor nem kell azt feltétlenül ciklusba rejtenünk. A sztringutasítást ismétlő prefixek minden sztringkezelő utasítás mnemonikja előtt megadhatók. Három fajtájuk van: REP, REPE/REPZ és REPNE/REPNZ (REPeat, REPeat while Equal/Zero/ZF=1, REPeat while Not Equal/Not Zero/ZF=0).

A REP prefix működése a következő:

1)ha CX=0000h, akkor kilépés, az utasítás végrehajtása befejeződik
2)a prefixet követő sztringutasítás egyszeri végrehajtása
3)CX csökkentése 1-gyel, a flag-ek nem változnak
4)menjünk 1)-re

Ha tehát kezdetben CX=0000h, akkor nem történik egyetlen művelet sem, hatását tekintve gyakorlatilag egy NOP-nak fog megfelelni az utasítás. Különben a sztringkezelő utasítás 1-szer végrehajtódik, CX csökken, és ez mindaddig folytatódik, amíg CX zérus nem lesz. Ez végül is azt eredményezi, hogy a megadott utasítás pontosan annyiszor kerül végrehajtásra, amennyit CX kezdetben tartalmazott. REP prefixet a LODS, MOVS és STOS utasításokkal használhatunk.

A REPE/REPZ prefix a következő módon működik:

1)ha CX=0000h, akkor kilépés, az utasítás végrehajtása befejeződik
2)a prefixet követő sztringutasítás egyszeri végrehajtása
3)CX csökkentése 1-gyel, a flag-ek nem változnak
4)ha ZF=0, akkor kilépés, az utasítás végrehajtása befejeződik
5)menjünk 1)-re

A REPE/REPZ tehát mindaddig ismétli a megadott utasítást, amíg CX≠0000h és ZF=1 is fennállnak. Ha valamelyik vagy mindkét feltétel hamis, az utasítás végrehajtása befejeződik.

A REPNE/REPNZ prefix hasonló módon akkor fejezi be az utasítás ismételt végrehajtását, ha a CX=0000h, ZF=1 feltételek közül legalább az egyik teljesül (tehát a fenti pszeudokódnak a 4-es lépése változik meg).

A REPE/REPZ és REPNE/REPNZ prefixeket a CMPS és a SCAS utasításoknál illik használni (hiszen ez a két utasítás állítja a flag-eket is), bár a másik három utasítás esetében működésük a REP-nek felel meg.

Még egyszer hangsúlyozzuk, hogy ezek a prefixek kizárólag egyetlen utasítás ismételt végrehajtására használhatóak, s az az utasítás csak egy sztringkezelő mnemonik lehet.

Gépi kódú szinten csak két prefix létezik: REP/REPE/ REPZ és REPNE/REPNZ, az első prefix tehát különböző módon működik az őt követő utasítástól függően.

A programhoz visszatérve, a REP MOVS … utasítás hatására megtörténik a közös karakterlánc átmásolása. Mivel a REP prefix CX-től függ, nem volt véletlen, hogy mi is ebben tároltuk a másolandó rész hosszát. Ezt a sort persze írhattuk volna az egyszerűbb REP MOVSB alakban is, de be akartuk mutatni az áloperandusokat is.

Ha ez megvolt, akkor meg fogjuk mérni a Szoveg1 sztring hosszát, ami ekvivalens a 00h kódú karaktert megelőző bájtok számának meghatározásával. Pontosan ezt fogjuk tenni mi is: megkeressük, hol van a lezáró karakter, annak offszetjéből már meghatározható a sztring hossza.

A SCAS/SCASB/SCASW (SCAn String Byte/Word) utasítás teljes alakja a következő:

SCAS cél

Az utasítás AL illetve AX tartalmát hasonlítja össze a cél ES:[DI] bájttal/szóval, beállítja a flag-eket, de egyik operandust sem bántja. Az összehasonlítást úgy kell érteni, hogy AL-t/AX-et kivonja a cél tartalmából, s az eredmény alapján módosítja a szükséges flag-eket. A SCASB/SCASW mnemonikok a SCAS BYTE/WORD PTR ES:[DI] utasítást rövidítik.

Esetünkben egészen addig akarjuk átvizsgálni a Szoveg1 bájtjait, amíg meg nem találjuk a 00h kódú karaktert. Mivel nem tudjuk, mennyi a sztring hossza, így azt sem tudjuk, hányszor fog lezajlani a ciklus. CX-et tehát olyan értékre kell beállítani, ami biztosan nagyobb vagy egyenlő a sztring hosszánál. Most feltesszük, hogy a sztring rövidebb 100 bájtnál. DF-et törölni kell, mivel a sztring elejétől növekvő sorrendben vizsgálódunk. A REPNE prefix hatására egészen mindaddig ismétlődik a SCAS, amíg ZF=1 lesz, ami pedig azt jelenti, hogy a legutolsó vizsgált karakter a keresett bájt volt.

Ha az utasítás végzett, DI a 00h-s karakter utáni bájtra fog mutatni, ezért DI-t 1-gyel csökkentjük, majd levonjuk belőle a Szoveg1 kezdőoffszetjét. Ezzel DI-ben kialakult a hossz, amit rögvest el is tárolunk. A kivonást kicsit cselesen oldjuk meg. Az STC utasítás CF-et állítja 1-re, amit a célból a forrással együtt kivonunk az SBB-vel. Az OFFSET operátor nevéből adódóan a tőle jobbra álló szimbólum offszetcímét adja vissza.

A maradék programrész az előzőekhez képest semmi hasznosat nem csinál. A 100 db. szóból álló Vektor területet fogjuk feltölteni a következő módon: bármelyik szót úgy kapjuk meg, hogy a kódszegmens egy adott bájtját előjelesen kiterjesztjük szóvá, majd ennek vesszük az egyes komplemensét. Semmi értelme, de azért jó. :)

Mivel a kódszegmens lesz a forrás, CS-t berakjuk DS-be, SI-t a szegmens elejére állítjuk.

A ciklusban majdnem minden ismerős. A LODS/LODSB/LODSW és STOS/STOSB/STOSW utasításokkal már találkoz-

tunk korábban, de azért ismétlésképpen felidézzük működésüket. Teljes alakjuk a következő:

LODS forrás
STOS cél

Forrás a szokott módon DS:SI, ebből DS felülbírálható, a cél pedig ES:DI, ami rögzített. Az utasítások AL-t vagy AX-et használják másik operandusként, a flag-eket nem módosítják. A LODS a forrást tölti be AL-be/AX-be, a STOS pedig AL-t/AX-et írja a cél területére. A LODS-nél a forrás, míg a STOS-nál AL/AX marad változatlan. SI-t ill. DI-t a szokott módon frissítik. A LODSB/LODSW, valamint a STOSB/STOSW utasítások formálisan sorban a LODS BYTE/WORD PTR DS:[SI], ill. a STOS BYTE/WORD PTR ES:[DI] utasításokat rövidítik.

Az AL-be beolvasott bájtot a CBW terjeszti ki előjelesen AX-be, majd ezt az egyoperandusú NOT utasítás bitenként logikailag negálja (azaz képezi az egyes komplemensét), végül a STOS utasítás a helyére teszi a kész szót. A ciklus lefutása után a program futása befejeződik.

Az utasítások teljes alakjának használatát csak a forrás szegmens megváltoztatása indokolhatná, minden más esetben a rövid alakok célravezetőbbek és jobban átláthatóak. Azonban az egyszerűbb változatok esetén is lehetőségünk van a forrás szegmens kiválasztására, mégpedig kétféle módon. Az egyik, hogy az utasítás előtt alkalmazzuk valamelyik SEGxx prefixet, a másik, hogy az utasítás előtti sorban kézzel berakjuk az adott prefix gépi kódját. Az első módszer MASM esetén nem működik (esetleg más assemblernél sem), a másodikban meg annyi a nehézség, hogy tudni kell a kívánt prefix kódját. Ezek alapján a LODS SS:[SI] utasítást az alábbi két módon helyettesíthetjük:

1)

```
SEGSS LODSB
```

2)

```
        DB      36h
        LODSB
```

Az SS: prefix gépi kódja 36h, a LODSB-é 0ACh, így mindkettő megoldás a 36h 0ACh bájtsorozatot generálja.

12 AZ .EXE ÉS A .COM PROGRAMOK KÖZÖTTI KÜLÖNBSÉGEK, A PSP

Az, hogy a futtatható állományoknak milyen a formátumuk, az adott operációs rendszertől függ. A .COM és . EXE fájlok a DOS operációs rendszer állományai, csak ő képes ezeket futtatni.

12.1 A DOS memóriakezelése

Hamarosan megnézzük, hogy hogyan történik általában egy program elindítása, de hogy világosan lássunk, tudnunk kell egyet s mást a DOS-ról, azon belül is annak memóriakezeléséről. Ugyebár 1 Mbájt memóriát tudunk megcímezni (így a DOS is), viszont ennyit mégsem tudunk kihasználni, ugyanis az 1 Mbájt memória felső területén (azaz a magas címtartományban) helyezkedik el pl. a BIOS programkódja (ráadásul az ROM memória) és a képernyőmemória, tehát pl. ezeket nem tudjuk általános tárolásra használni. A memóriából mindenesetre összesen 384 Kbájtot tartanak fenn, így az 1024 Kbájtból csak 640 Kbájtot tud a DOS ténylegesen használni (ez a 640 ismerős szám kell, hogy legyen). Ezen a(z alsó) 640 Kbájton belül a DOS memóriablokkokat (memóriaszeleteket) kezel, amelyeknek négy fő jellemzőjük van:

- **memóriablokk helye (kezdőcíme)**
- **memóriablokk mérete**
- **memóriablokk állapota (foglalt - szabad)**
- **memóriablokk (tulajdonosának) neve**

Ezekről az adatokról a DOS nem vezet külön adminisztrációt egy saját, operációs rendszeri memóriaterületen, ahol pl. különböző listákban tárolná őket (amihez természetesen senkinek semmi köze), hanem egyszerűen az adott memóriablokk elején helyezi el őket. Ezek az adatok szintén egy blokkot alkotnak, amelynek mérete 16 bájt, neve pedig memóriavezérlő blokk (MCB–Memory Control Block). Ezzel megengedi a felhasználói programoknak is azt, hogy elérhessék ezeket az információkat, ami a DOS szemszögéből elég nagy hiba, ugyanis bárki „turkálhat" bennük, ami miatt tönkremehet a memória nyilvántartása. Ezalatt azt értem, hogy adott esetben egy program a memóriablokkokra vonatkozó műveleteket nem a DOS szabványos eljárásain keresztül végzi, hanem saját maga, mégpedig úgy, hogy egyszerűen átírja az MCB-t, és/vagy új(ak) at hoz létre, ha újabb memóriablokk(ok) keletkeztek. Egyébként három műveletet értelmez a DOS a memória-blokkjain, mindegyik eljárásnak vannak hibakódjaik (visszatérési értékeik):

- ***adott méretű memóriablokk lefoglalása (memória foglalása)***
  - **Ha sikerült lefoglalni a kívánt méretű memóriát, akkor visszakapjuk a lefoglalt terület címét. Hibakódok: nincs elég memória, ekkor visszakapjuk a legnagyobb szabad blokk méretét is.**
- ***adott kezdőcimű blokk felszabaditása***
  - **Hibakódok: a kezdőcím nem egy blokk kezdetére mutat.**

- *adott kezdőcimű foglalt blokk méretének megváltoztatása*
  - **Hibakódok: nincs elég memória (ha növelni akarjuk), érvénytelen blokkcím.**

Ha memóriát foglalunk le, akkor a DOS létrehoz egy új blokkot a kívánt mérettel, a blokk legelején az MCB-vel, s a blokk tényleges kezdőcíménél egy 16 bájttal nagyobb címet ad vissza mint a lefoglalt memória kezdetét. A felhasználói program (aki a memóriát kérte) szempontjából az MCB tehát nem tartozik a blokkhoz, gondoljunk csak a magas szintű programozási nyelvekre: ott is használunk ilyen függvényeket, s bennünket egyáltalán nem érdekel, hogy a lefoglalt memóriát „ki" és milyen módon tartja nyilván. A DOS tehát eleve egy 16 bájttal nagyobb blokkot foglal le, hogy az MCB-t is el tudja helyezni benne.

...	MCB 16 byte	Memóriablokk azon része, amely tényleges adatokat tartalmaz (ha foglalt), vagy használaton kivül van (ha szabad)	...

Egy memóriablokk felépítése

Lényeges, hogy soha egyetlen memóriablokk MCB-je se sérüljön meg, mindig helyes adatokat tartalmazzon, különben az említett memóriakezelő eljárások a következő hibakóddal térnek vissza: az MCB-k tönkrementek. Ilyenkor természetesen az adott művelet sem hajtódik végre. Erre példa: ha az adott blokk kezdőcíméhez hozzáadjuk annak méretét, akkor megkapjuk a következő blokk kezdőcímét (pontosabban annak MCB-jének kezdőcímét), és így tovább, azt kell kapnunk, hogy a legutolsó blokk utáni terület kezdőcíme a 640 Kbájt. Ha pl. ez nem teljesül, az azt jelenti, hogy valamelyik (de lehet, hogy több) MCB tönkrement, s attól kezdve talán már olyan területeket vizsgáltunk MCB-ként, amik valójában nem is azok.

Mindesetre ilyenkor használhatatlanná válik a nyilvántartás. A DOS is az említett (saját) eljárásokat használja. Ha az előbbi hibával találkozik, akkor azt úgy kezeli le, hogy a "Memory Allocation Error–memóriakiosztási hiba" üzenettel szépen meghal.

Bootoláskor a DOS lefoglal magának egy blokkot (a memória elején), oda rakja adott esetben a saját programkódját, a többi memóriát pedig bejelöli egyetlen nagy szabad blokknak, amelyet aztán a programok szabadon használhatnak.

Az 1M memória felépítése DOS esetén

12.2 Általában egy programról

Tekintsük át <u>elméleti</u> szempontból, hogy általában hogyan épül fel egy program, mire van szüksége a futáshoz és futás közben, aztán megnézzük, hogy mindezeket hogyan biztosították DOS alatt.

Egy program három fő részből tevődik össze:

- ***Kód*** **– ez alatt magát az alacsony szintű programkódot értjük, a forditóprogramok feladata, hogy a magas szintű nyelven leírt algoritmust a processzor számára feldolgozható kódra alakítsák át.**
- ***Adat*** **– ebben a részben helyezkednek el a program <u>globális</u> változói, azaz a statikusan lefoglalt memória.**

- ***Verem*** **– magas szintű nyelvekben nem ismeretes ez a fogalom (úgy értve, hogy nem a nyelv része), viszont nélkülözhetetlen az alacsony (gépi) szinten. A fordítók pl. ennek segitségével valósítják meg a lokális változókat (amiről már szó volt).**

A gyakorlatban legtöbbször a programkódot nem egyetlen összefüggő egységként kezeljük (és ez az adatokra is vonatkozik), hanem "csoportosítjuk" a program részeit, s ennek alapján külön szegmensekbe (logikailag különálló részekbe) pakoljuk őket. Pontosan ezt csináljuk akkor, amikor az assemblernek szegmenseket definiálunk (ez tehát az a másfajta szegmensfogalom, amikor a szegmenseket logikailag különválasztható programrészeknek tekintjük). Meg lehet még említeni a Pascalt is: ő pl. a unitokat tekinti különálló programrészeknek, igy minden unit programkódja külön kódszegmensbe kerül. Adatszegmens viszont csak egy globális van.

Nézzük meg, hogy mely részeket kell eltárolnunk a programállományban!

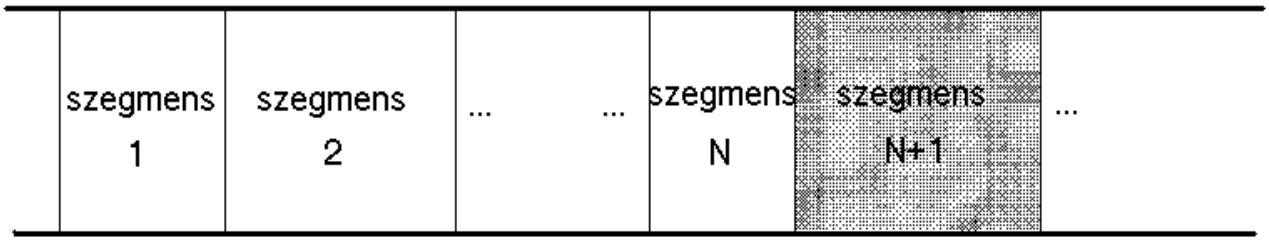
A programkódot mindenképpen, hiszen azt nem tudjuk a „semmiből" előállítani. A vermet viszont semmiképpen nem kell, hiszen a program indulásakor a verem üres (ezért mit is tárolnánk el?), ezért ezzel csak annyit kell csinálni, hogy a memóriában kijelölünk egy bizonyos nagyságú területet a verem számára, a veremmutatót a veremterület végére állitjuk (ott lesz a verem teteje). Szándékosan hagytuk a végére az adatokat, mert ennek elég csak valamely részét eltárolni. Elég csak azokat a globális változókat eltárolnunk, amelyeknek van valami kezdeti értéke a program indulásakor. Ezt úgy szokás megoldani, hogy a kezdőértékkel nem rendelkező változókat egy külön adatszegmensbe rakjuk, s előírjuk (ezt meg lehet mondani az assemblernek), hogy ez a szegmens ne kerüljön be az állományba (azaz a linker ne szerkessze be), viszont a

program indításakor az operációs rendszer „tudjon” eme szegmensről, s biztosítson neki memóriát, azaz hozza létre azt. E megoldás azért kényelmetlen, mert ilyenkor két adatszegmensünk is van, de azt mi egyben szeretnénk látni, ezért a programnak futás közben „váltogatnia” kellene a 2 között. Kényelmesebb megoldás, ha egyetlen adatszegmenst deklarálunk, az egyik felében a kezdőértékkel rendelkező változókat, a másik felében pedig az azzal nem rendelkező változókat helyezzük el, s azt írjuk elő, hogy a szegmens másik fele ne kerüljön be az állományba (ez is megoldható, hiszen a szegmenseket szakaszokban is megadhatjuk, így megmondhatjuk azt is az assemblernek, hogy a szegmens adott darabja ne foglaljon feleslegesen helyet az állományban. Ez persze csak akkor teljesülhet, ha abban a szegmensben ténylegesen úgy deklaráljuk a változókat, hogy ne legyen kezdőértéke (pl. Nev DB ?), ha ez a követelmény valahol nem teljesül, akkor mindenképpen bekerül az egész szegmensdarab az állományba).

Egy program indításakor a programállományból tehát meg kell tudnia az operációs rendszernek (mivel ő indítja a programokat), hogy hol található(ak) az állományban a kódot tartalmazó részek. Ha ez megvan, lefoglalja nekik a megfelelő mennyiségű memóriá(ka)t (ott lesznek elhelyezve a program <u>kódszegmensei,</u> majd oda betölti a programkódot. Ezután következik az adat. Hasonlóan jár el a program adataszegmenseivel (az állományban le nem tárolt szegmenseket is beleértve, ezen memóriabeli szegmensekbe nyilván nem tölt be semmit az állományból, így az itt található változóknak meghatározatlan lesz az kezdőértéke), az adott szegmensek tartalmát szintén bemásolja. A memóriában meglesznek tehát a program <u>adatszegmensei</u> is. Azután foglal memóriát a veremnek is (<u>veremszegmens</u>). A veremmutatót ráállítja a veremszegmens végére, az adatszegmens-regisztert pedig valamely adatszegmens elejére (ezzel a program elő van

készítve a futásra), majd átadja a vezérlést (ráugrik) a program első utasítására valamelyik kódszegmensben.

Mindez, amit leírtunk, csak elméleti dolog, azaz hogyan végzi el egy operációs rendszer általában egy program elindítását. Azonban mint azt látni fogjuk, a DOS ilyen nagy fokú szabadságot nem enged meg számunkra.



Egy program felépitése. Az, hogy mely szegmens mit tartalmaz (kód, adat) nem is lényeges. A "szürke" szegmens az állományban el nem tárolt szegmens, persze több ilyen is lehet

12.3 Ahogy a DOS indítja a programokat

A DOSnak kétféle futtatható állománya van (hüha!), a .COM és a .EXE formátumú. Amit ebben a fejezetben általánosságban leírunk, az mind a kettőre vonatkozik. A kettő közti részletes különbséget ezután tárgyaljuk.

Mint az látható, az, hogy a program mely szegmense tulajdonképpen mit tartalmaz, azaz hogy az kód- vagy adatszegmens-e, az teljesen lényegtelen. A DOS viszont ennél is továbbmegy: a programállományaiban letárolt szegmensekről semmit sem tud! Nem tudja, hogy hány szegmensről van szó, így pl. azt sem, hogy melyik hol kezdődik, mekkora, stb., egyetlen egy valamit tud róluk, mégpedig azt, hogy azok összmérete mennyi. Éppen ezért nem is foglal mindegyiknek külön-külön memóriát, hanem megnézi, hogy ezek mindösszesen mekkora memóriát igényelnek, s egy ekkora méretű memóriablokkot foglal le, ezen belül kap majd helyet minden szegmens. Annak, hogy a szükséges memóriát egyben foglalja le, annyi hátránya

van, hogy ha nincs ekkora méretű szabad memóriablokk, akkor a programot nem tudja futtatni (ekkor kiköpi a "Program too big to fit in memory – a program túl nagy ahhoz, hogy a memóriába férjen" hibaüzenetet), holott lehet, hogy a szükséges memória "darabokban" rendelkezésre állna. Igazából azonban a gyakorlatban ez a probléma szinte egyáltalán nem jelentkezik, mivel ez csak a memória felaprózódása esetén jelenhet meg, azaz akkor, ha egyszerre több program is fut, aztán valamely (ek) végetér(nek), az általuk lefoglalt memóriablokk(ok) szabaddá válik/válnak, igy a végén egy "lyukacsos" memóriakép keletkezik. Mivel a DOS csak egytaszkos oprendszer, azaz egyszerre csak egy program képes futni, mégpedig amelyiket a legutoljára indítottak, így nyilván az őt indító program nem tud azelőtt befejeződni, mielőtt ez befejeződne. Ebből az következik, hogy a szabad memória mindig egy blokkot alkot, a teljes memória foglalt részének mérete pedig a veremelv szerint változik: amennyit hozzávettünk a foglalt memória végéhez, azt fogjuk legelőször felszabadítani. Persze azért a DOS esetében is felaprózódhat a memória, ha egy program dinamikusan allokál magának memóriát, aztán bizonyos lefoglalt területeket tetszőleges sorrendben (és nem a lefoglalás fordított sorrendjében) szabadít fel. Ha ez a program ebben az állapotban saját maga kéri az DOS-t egy újabb program indítására, akkor már fennáll(hat) az előbbi probléma. Elkalandoztunk egy kicsit, ez már inkább az operációs rendszerek tárgy része, csak elmélkedésnek szántuk.

A DOS tehát egyetlenegy memóriablokkot foglal le minden szegmens számára, ezt az egész memóriablokkot nevezik *programszegmensnek* (Program Segment) (már megint egy újabb szegmensfogalom, de ez abból adódik, hogy a szegmens szó darabot, szeletet jelent, s ezt ugyebár sokmindenre lehet érteni). Fontos, hogy ezekkel a fogalmakkal tisztában legyünk.

Mint arról már szó volt, a programszegmens akkora méretű, amekkora a szegmensek által összesen igényelt memória. Nos, ez nem teljesen igaz, ugyanis a programszegmens LEGALÁBB ekkora (+256 bájt, ld. később), ugyanis ha a programindításkor a DOS egy olyan szabad memóriablokkot talál, ami még nagyobb is a szükségesnél, akkor is "odaadja" az egészet a programnak. Hogy ez mire jó, azt nem tudjuk, mindenesetre így van. Ebből az a kellemetlenség adódik, hogy pl. ha a program a futás során dinamikusan szeretne memóriát allokálni, akkor azt a hibaüzenetet kaphatja, hogy nincs szabad memória. Valóban nincs, mert az összes szabad memória a programszegmenshez tartozhat, a programszegmens meg ugyebár nem más, mint egy foglalt memóriablokk, s a DOS memóriafoglaláskor nem talál más szabad blokkot. Ezért ilyenkor a programnak induláskor le kell csökkentenie a programszegmensének méretét a minimálisra a már említett funkcióval!

Még egy fontos dolog, amit a DOS minden program indításakor elvégez: a programszegmens elején létrehozza a *programszegmens-prefixet* (Program Segment Prefix, PSP), ami egy 256 bájtból álló információs tömb. A +256 bájt tehát PSP miatt kell. Mire kell a PSP és mit tartalmaz?

MCB, 16 byte	PSP, 256 byte	Itt helyezkednek el a program szegmensei	a programszegmens felesleges, szabad része, ezt le lehet vágni

A programszegmenst alkotó memóriablokk felépítése

Sokmindent. Ezen 256 bájtos terület második fele pl. azt a stringet tartalmazza, amit a promptnál a program neve után írunk paraméternek. Azt, hogy az első 128 bájt miket tárol, azt

nagyon hosszú lenne itt felsorolni (inkább nézzük meg egy könyvben), számunkra nem is igazán fontos, csak egy-két érdekesebb dolog: az első két bájt az INT 20h kódja (0CDh 20h), vagy pl. a környezeti stringek (amelyeket a DOS SET parancsával állíthatunk be) számára is le van foglalva egy memóriablokk, ennek a szegmenscime (ami azt jelenti, hogy a cím offszetrésze nulla, így azt nem kell letárolni, csak a cím szegmensrészét, de ez igaz minden más memóriablokk címére is) benne van a PSP-ben. Így a program kiolvashatja ezt, s hozzáférhet a környezeti stringekhez, módosíthatja azokat, stb. A DOS saját maga számára is tárolgat információt: pl. megtalálható az adott programot elindító program PSP-jének szegmenscíme is (*visszaláncolás*=backlink), ami általában a parancsértelmezőé (COMMAND.COM), hiszen legtöbbször onnan indítjuk a programokat. Mire kell ez? Arra, hogy ha a program befejeződik, akkor a DOS tudja, hogy „kinek" kell visszaadnia a vezérlést, és ez nyilván a programot betöltő program.

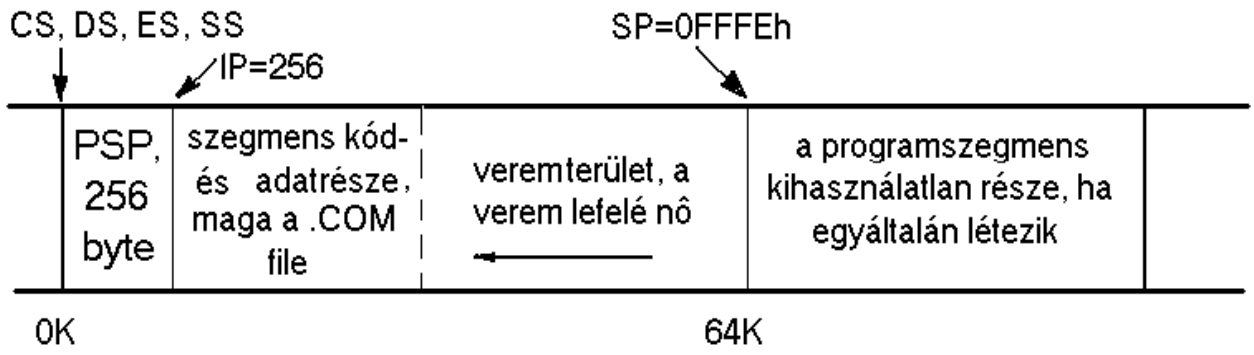
Ezek után nézzük meg a különbséget .COM és .EXE között:

12.4 .COM állományok

Ezek a fájlok voltak a DOS első futtatható állományai, nagyon egyszerű felépítésűek, ugyanis olyan programok számára tervezték, amelyek egyetlen szegmensből állnak. Erről ugyan nem volt szó, de egy szegmensnek természetesen lehet vegyes jellege is, tartalmazhat kódot is és adatot is egyszerre. Nos, .COM-ok esetében még a veremterület is ebben a szegmensben kell, hogy elhelyezkedjen. Egy programnak illene minimum 3 szegmensének lennie a három programrész számára, de itt szó sincs erről. Mint tudjuk, egy szegmens max. 64 Kbájt lehet, így egy .COM program is. Ráteszünk azonban

még egy lapáttal: a PSP is a szegmens része, indításkor a DOS a PSP mögé másolja be a .COM programot! (Ezért van az, hogy .COM programok írásakor a szegmens elejére be kell raknunk azt a bizonyos ORG 100h direktívát, ezzel jelezve az assemblernek, hogy minden szegmensbeli offszetcímhez „adjon hozzá” 100h-t, azaz az offszeteket tolja el 256 bájttal, mivel a PSP nyilván nem tárolódik el a programállományban, de mindig a "oda kell képzelnünk" a szegmens elejére). Induláskor a DOS a verembe berak egy 0000-t, ami a PSP-ben levő INT 20h utasítás offszetcíme. A kilépés tehát megoldható egy RET utasítással is, mivel a DOS a programot egy közeli eljárásnak tekinti. (Ez csak .COM-ok esetén alkalmazható, mert ehhez a CS-nek a PSP-re kell mutatnia).

Ezután programszegmensen a programszegmens csak azon részét értjük, amelyet a program ténylegesen használ, nem vesszük bele tehát a felesleges részt, mivel annak semmi szerepe. A .COM programok programszegmense mindig 64 Kbájt, eltekintve attól az (egyébként nagyon ritka) esettől, amikor már 64 Kbájt szabad memória sincs a program számára, de ezzel most nem foglalkozunk. Miért van az, hogy a szegmens mindig kibővül 64 Kbájtra? Azért, hogy minél több hely legyen a verem számára. Ugyanis a .COM fájlokban a programról semmi kisérőinformáció nem található, a fájl csak magát az egyetlen szegmenst tárolja le, így nem tudjuk megadni a DOS-nak, hogy a szegmensen belül hol a veremterület, azaz hogy mi legyen az SP kezdeti értéke. Emiatt a DOS úgy gondolkodik, hogy automatikusan maximális vermet biztosít, ezért növeli meg a szegmensméretet 64 Kbájtra, s annak végére állítja a veremmutatót. Ebből az is következik, hogy a program írásakor nekünk sem kell törődnünk a veremmel.



Egy .COM program programszegmense a program indulásakor

Ezek szerint azt sem tudjuk megadni, hogy a szegmensben hol található az első végrehajtandó utasítás. Így van, ennek mindig a .COM fájl elején kell lennie, azaz a memóriabeli szegmens 256. bájtján. Indításkor a DOS tehát a fenti ábra szerint állítja be a regisztereket.

12.5 Relokáció

Ha egy program több szegmenst is tartalmaz, akkor ahhoz, hogy el tudja érni azokat, meg kell határoznia minden egyes szegmens tényleges szegmenscímét. Azért nevezik ezt a folyamatot *relokációnak* (áthelyezésnek), mert a szegmenseknek csak a program elejéhez képesti relatív elhelyezkedését ismerhetjük (vehetjük úgy is, hogy a program legeleje a 0-n, azaz az 1 Mbájt memória elején kezdődik, ekkor a szegmensek relatív címei egyben abszolút címek is, a program futóképes lenne), azonban a program a memória szinte tetszőleges részére betöltődhet, s hogy pontosan hova, az csak indításkor derül ki. A program csak akkor lesz futóképes, ha a relatív szegmenscímeket átírjuk, azaz azokhoz hozzáadjuk a program elejének szegmenscímét. Ezzel a programot mintegy áthelyeztük az adott memóriaterületre. Lássuk ezt egy példán keresztül:

```
Kod SEGMENT PARA            ;paragrafushatáron kezdődjön
                            ;a szegmens
      ASSUME CS:Kod, DS:Adat
                            ;mondjuk meg az assemblernek,
                            ;hogy mely szegmensregiszter
                            ;mely szegmensre mutat, ő
                            ;eszerint generálja majd az
                            ;offszetcímeket

      MOV     AX,Adat  ;Az „adat" szegmens szegmenscíme
      MOV     DS,AX            ;A feltételezésnek
tegyünk is
                            ;eleget, ne vágjuk át szegény
                            ;assembler bácsit
      .
      .
      .
      MOV     AX,4C00h
      INT     21h

Kod ENDS
Adat SEGMENT PARA           ;paragrafushatáron kezdődjön
                            ;a szegmens

Valami          DB   2
Akarmi          DB   14

Adat ENDS
END
```

Ez egy szokványos program, de a lényeg most azon van, hogy amikor az elején a DS-t ráállítjuk az "adat" szegmensre, akkor a MOV AX,Adat utasítással mit kezd az assembler, hiszen az Adat értéke (ami egy szegmenscím) attól függ, hogy a program a memóriába hová töltődik majd be. A válasz az, hogy az Adat egy relatív érték, a program első szegmenséhez képest. Hogy melyik lesz az első szegmens, az csak a linkelésnél válik el, amikor a linker az egyes szegmensdarabokat összefűzi egyetlen

szegmenssé, valamint a szegmenseket egyetlen programmá, de hogy a szegmensek milyen sorrendben kövessék egymást a programban, arról ő dönt. Az assembler tehát csak bejelöli az . OBJ fájlban, hogy ezen a helyen egy relatív szegmenscím kelletik, ezt a linker észreveszi, s beírja oda azt. A fenti programnak csak két szegmense van, tegyük fel, hogy a Kod lesz az első, s hogy a Kod szegmens mérete pl. 36 bájt (mivel ez nem lenne 16-tal osztható, ezért a méret linkeléskor 16-tal oszthatóra nő, hogy az utána következő szegmens paragrafushatárra essen, így most 48 bájt lesz). Ekkor a lefordított program a következő lesz:

```
      MOV     AX,0003          ;48/16=3
      MOV     DS,AX
      .
      .
      .
      MOV     AX,4C00h
      INT     21h          ;Ezután szükség szerint
„szemét"
                     ;következik, hogy a „DB 2" a program
      .              ;elejéhez képest 16-tal osztható
cimre
      .              ;essen, hiszen azt a szegmenst
      .              ;paragrafushatáron kezdtük


      DB      2
      DB      14
```

Most a program egy relatív szegmenscímet tartalmaz (0003), de még mielőtt elindíthatnánk, azt át kell javítani a tényleges címre. Ha pl. a Kod szegmens a memória 5678h:0000h területére kerül majd, akkor a 0003-hoz hozzá kell adni az 5678h-t, hogy helyes értéket kapjunk.

Mivel ezt a program indításakor kell elvégezni, így ez a feladat a DOS-ra hárul, de ő ezt csak az .EXE fájlok esetében képes elvégezni, mivel ott el van tárolva a relokációhoz szükséges információ. Ebből világos, hogy egy .COM program nem tartalmazhat ilyen módon relatív szegmenscímeket, igy pl. a fenti programot sem lehet .COM-má fordítani, a linker is idegeskedik, ha ilyennel találkozik. A másik megoldás az, ha a program maga végzi el induláskor a szükséges relokációt. Ezzel elérhetjük azt is, hogy egy .COM program több szegmenst is tartalmazzon (amelyek összmérete nem haladhatja meg persze a 64K-t), csak a relokációt el kell végeznünk. A fenti példát átírva:

```
DOSSEG

Kod     SEGMENT PARA
        ASSUME CS:Kod, DS:Adat
        ORG 100h                ;PSP helye

@Start:
        MOV     AX,Kod:Adat_eleje
                                ;a "Kod" szegmens elejéhez
                                ;képest az adat_eleje címke,
                                ;azaz az adat szegmens eleje
                                ;hány bájtra helyezkedik el
        MOV     CL,4h
        SHR     AX,CL               ;osztva 16-tal (az előző pél-
                                ;dában közvetlenül ez az érték
                                ;volt benne az utasításban)
        MOV     BX,DS               ;DS ekkor még a PSP elejére,
                                ;azaz a kod szegmens elejére
                                ;mutat
        ADD     AX,BX
        MOV     DS,AX               ;ráállítjuk az adatszegmensre
        .                       ;innen DS már az adat
        .                       ;szegmensre mutat
        .
```

```
        MOV     AX,4C00h
        INT     21h

Kod     ENDS

Adat    SEGMENT PARA
        ORG 0h              ;Ezen szegmensen belül nincs
                            ;eltolás
Adat_eleje LABEL

Valami          DB 2
Akarmi          DB 14

Adat    ENDS
        END     @Start
```

A program elején levő DOSSEG direktívával biztosítjuk azt, hogy a szegmensek a deklarálás sorrendjében kövessék egymást az összeszerkesztett programban is, hiszen itt fontos, hogy a Kod legyen az első szegmens.

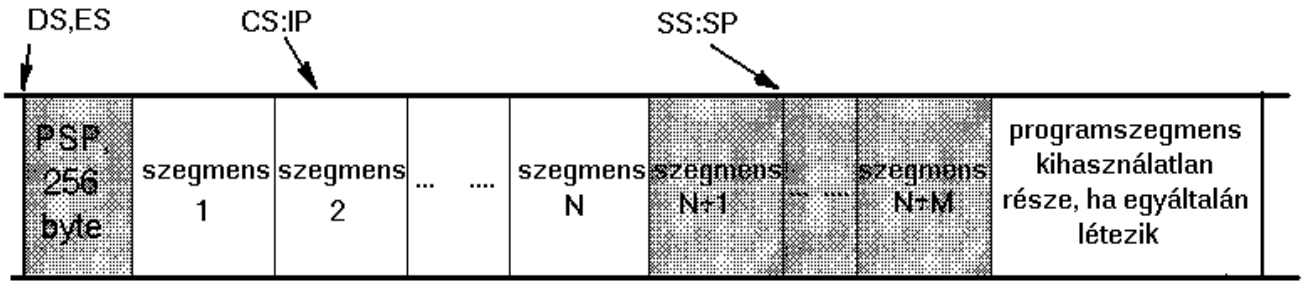
12.6 .EXE állományok

Az .EXE állományokat a bonyolultabb programok számára találták ki, amelyek tetszőleges számú szegmensből állnak. Magában az állományban ezek egymást követően, folyamatosan vannak eltárolva, ahogyan a linker egymás után fűzte őket. Fontos azonban megjegyezni, hogy nem biztos, hogy az egész állomány magát a programot tárolja, mivel az állományban a program után még tetszőleges hosszúságban tárolhatunk pl. adatokat, mint egy közönséges adatfájlban. Ezek az adatok nem tartoznak a programhoz, mégegyszer kihangsúlyozzuk. Az .EXE fájlok elején mindig található egy

fejléc, azaz egy információs tömb, amely a DOS számára nélkülözhetetlen a program elindításához.

A betöltési folyamat hasonló a .COM-okéhoz: a fejlécből a DOS megnézi, hogy a fejléc után eltárolt program milyen hosszú, azaz hol a határ a program és az előbb említett adatok között. Itt persze már nem él az a megkötés, hogy a program (azaz a szegmensek összmérete) nem haladhatja meg a 64 Kbájtot. Ezután szintén a fejlécből megnézi, hogy a programnak mennyi többletmemóriára van szüksége (ez a bejegyzés a "minimálisan szükséges többletmemória"). Ez biztositja a programban el nem tárolt szegmensek létrehozását a memóriában, amiről már beszéltünk. A programszegmens így legalább az összméret+többletmemória méretű lesz. Van azonban egy "maximálisan szükséges többletmemória"-bejegyzés is, elméletileg ennél nagyobb soha nem lehet nagyobb a programszegmens, ezzel tehát el lehetne kerülni, hogy az feleslegesen nagy legyen, de a gyakorlatban egyáltalán nem élnek ezzel a lehetőséggel, szinte mindig maximumra állitják (a linkerek, merthogy ők hozzák létre a futtatható állományokat). A DOS tehát lefoglalja a programszegmenst, bemásolja az .EXE-ből a programot, s elvégzi a program relokációját. Hogy a programban mely helyeken kell átírni a relatív szegmenscímeket, azt a *relokációs táblából* (relocation table) (ami szintén a fejléc része) tudja az oprendszer. Mivel programunk több szegmensből áll, kijelölhetjük, hogy melyik a veremszegmens (szintén relatívan), indításkor a DOS az SS:SP-t ennek megfelelően állítja be. Ugyanígy relatívan adhatjuk meg, hogy hol van a program belépési pontja, azaz mi lesz a CS:IP kezdeti értéke. Fontos megjegyezni, hogy .EXE-k esetében a PSP nem tartozik semelyik szegmenshez sem, ezért nincs szükség ilyen programok irásakor sehol sem az ORG 100h direktiva megadására. Kezdeti adatszegmenst nem tudunk kijelölni a program számára, a DOS úgy van vele, hogy ha egy programnak több szegmense is van,

akkor azok között valószínűleg egynél több adatszegmens is van, így a programnak futás közben úgyis állítgatnia kell a DS-t, akkor miért ne tegye ezt meg pl. már rögtön induláskor, ahogy az első példában már láttuk? Ez amúgy sem okoz a programnak gondot, a relokációt elvégzik helyette, nem neki kell vele szenvednie. Éppen ezért kezdetben a DS,ES a PSP elejére mutat.



Egy .EXE programszegmense induláskor. A fehér szegmensek vannak eltárolva a fájlban, mig a szürke szegmenseket a DOS allokálja a többletmemória révén. Ebben a példában az N+1. szegmens veremszegmens, annak végére mutat az SS:SP, mig a 2. szegmens kódszegmens, azon belül van az első végrehajtandó utasitás.

13 SZOFTVER-MEGSZAKÍTÁSOK

Mint említettük, szoftver-megszakításnak egy program által kiváltott megszakítást nevezünk. Hogy miért van erre szükség? Nos, néhány szoftver-megszakítás jócskán megkönnyíti a különféle hardvereszközök elérését, míg mások az operációs rendszer egyes funkcióival teremtenek kapcsolatot, de számos egyéb felhasználásuk is lehetséges.

Összesen 256 db szoftver-megszakítást kezel a 8086-os mikroprocesszor. Ebből 8 (vagy 16) db. az IRQ-k kezelésére van fenntartva, néhány pedig speciális célt szolgál, de a többi mind

szabadon rendelkezésünkre áll. Azt azért tudni kell, hogy a gép bekapcsolása, ill. az operációs rendszer felállása után néhány megszakítás már foglalt lesz, azokon keresztül használhatjuk a munkánkat segítő szolgáltatásokat.

Megszakítást az egyoperandusú INT utasítással kezdeményezhetünk. Az operandus kizárólag egy bájt méretű közvetlen adat lehet, ami a kért szoftver-megszakítást azonosítja. Az utasítás pontos működésével a következő fejezetben foglalkozunk majd. Érdemes megemlíteni még egy utasítást. Az INTO (INTerrupt on Overflow) egy 04h-s megszakítást kér, ha OF=1, különben pedig működése megfelel egy NOP-nak (azaz nem csinál semmit sem, csak IP-t növeli). Ennek akkor vehetjük hasznát, ha mondjuk az előjeles számokkal végzett műveletek során keletkező túlcsordulás számunkra nemkívánatos, és annak bekövetkeztekor szeretnénk lekezelni a szituációt.

Egy adott számú megszakítás sokszor sok-sok szolgáltatás kapuját jelenti. A kívánt szolgáltatást és annak paramétereit ilyenkor különböző regiszterekben kell közölni, és a visszatérő értékeket is általában valamilyen regiszterben kapjuk meg. Hogy melyik megszakítás mely szolgáltatása hányas számú és az milyen regiszterekben várja az adatokat, lehetetlen fejben tartani. Ezeknek sok szakkönyvben is utánanézhetünk. Ha viszont a könyv nincs a közelben, akkor sem kell csüggedni. Számos ingyenes segédprogramot kifejezetten az ilyen gondok megoldására készítettek el. A legismertebbek: HelpPC, Tech Help!, Norton Guide, Ralf Brown's Interrupt List. A programok mindegyike hozzáférhető, és tudtunkkal magáncélra ingyenesen használható. Mindegyik tulajdonképpen egy hatalmas adatbázison alapul, amiben hardveres és szoftveres témákat is találhatunk kedvünk szerint.

Most felsorolunk néhány gyakoribb, fontos megszakítást a teljesség igénye nélkül:

10h	**Képernyővel kapcsolatos szolgáltatások (Video services)**

13h	Lemezműveletek (Disk operation)
14h	Soros ki-/bemenet (Serial I/O)
16h	Billentyűzet (Keyboard)
1Ch	Időzítő (Timer tick)
20h	Program befejezés (Program terminate – DOS)
21h	DOS szolgáltatások (DOS services)
25h	Közvetlen lemezolvasás (Absolute disk read – DOS)
26h	Közvetlen lemezírás (Absolute disk write – DOS)
27h	Rezidenssé tétel (Terminate and stay resident – DOS)
2Fh	Vegyes szolgáltatások (Multiplex)
33h	Egér támogatás (Mouse support)

A következőkben néhány példán keresztül bemutatjuk a fontosabb szolgáltatások használatát.

13.1 Szövegkiíratás, billentyűzet-kezelés

Legyen a feladat a következő: a program írja ki a képernyőre a "Tetszik az Assembly?" szöveget, majd várjon egy billentyű lenyomására. Ha az "i" vagy "I" gombot nyomják le, akkor írja ki az "Igen." választ, "n" és "N" hatására pedig a "Nem." szöveget. Mindkét esetben ezután fejezze be működését. Egyéb billentyűre ne reagáljon, hanem várakozzon valamelyik helyes válaszra. Lássuk a programot:

Pelda11.ASM:

```
MODEL SMALL

.STACK
```

```
ADAT        SEGMENT
Kerdes      DB      "Tetszik az Assembly? $"
Igen        DB      "Igen.",0Dh,0Ah,'$'
Nem         DB      "Nem.",0Dh,0Ah,'$'
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            MOV     AH,09h
            LEA     DX,[Kerdes]
            INT     21h
@Ciklus:
            XOR     AH,AH
            INT     16h
            CMP     AL,'i'
            JE      @Igen
            CMP     AL,'I'
            JE      @Igen
            CMP     AL,'n'
            JE      @Nem
            CMP     AL,'N'
            JNE     @Ciklus
@Nem:
            LEA     DX,[Nem]
            JMP     @Vege
@Igen:
            LEA     DX,[Igen]
@Vege:
            MOV     AH,09h
            INT     21h
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

A program elég rövid, és kevés újdonságot tartalmaz, így megértése nem lesz nehéz.

Szöveget sokféleképpen ki lehet írni a képernyőre. Mi most az egyik DOS-funkciót fogjuk használni erre. Már láttuk, hogy a DOS-t az INT 21h-n keresztül lehet segítségül hívni, a kért szolgáltatás számát AH-ban kell megadnunk. A 4Ch sorszámú funkciót már eddig is használtuk a programból való kilépésre. A 09h szolgáltatás egy dollárjellel ($) lezárt sztringet ír ki az aktuális kurzorpozícióba. A szöveg offszetcímét DX-ben kell megadni, szegmensként DS-t használja.

A billentyűzet kezelését az INT 16h-n keresztül tehetjük meg. A szolgáltatás számát itt is AH-ba kell rakni. Ennek 00h-s funkciója egészen addig várakozik, míg le nem nyomunk egy gombot a billentyűzeten, majd a gomb ASCII-kódját visszaadja AL-ben. Hátránya, hogy néhány billentyű lenyomását nem tudjuk így megfigyelni (pl. Ctrl, Shift), míg más billentyűk AL-ben 00h-t adnak vissza, s közben AH tartalmazza a billentyűt azonosító számot (ez a *scan code*). Most nekünk csak a kicsi és nagy "I" ill. "N" betűkre kell figyelnünk, így nincs más dolgunk, mint a megszakításból visszatérés után AL-t megvizsgálnunk. Ha AL-ben 'i' vagy 'I' van, akkor a @Igen címkére megyünk. Ha AL='n' vagy AL='N', akkor a @Nem címkét választjuk célként. Különben visszamegyünk a @Ciklus címkére, és várjuk a következő lenyomandó billentyűt. Figyeljük meg, hogy a szelekció utolsó feltételében akkor megyünk vissza a ciklusba, ha AL≠'N', máskülönben "rácsorgunk" a @Nem címkére, ahová amúgy is mennünk kéne. Ezzel a módszerrel megspóroltunk egy JMP-t.

Akár a @Igen, akár a @Nem címkét választottuk , a szöveg offszetjének DX-be betöltése után a @Vege címkénél kötünk ki, ahol kiírjuk a választ, majd kilépünk a programból.

13.2 Szöveges képernyő kezelése, számok hexadecimális alakban kiírása

Következő problémánk már rafináltabb: a program indításkor törölje le a képernyőt, majd a bal felső sarokba folyamatosan írja ki egy szó méretű változó tartalmát hexadecimálisan, miközben figyeli, volt-e lenyomva billentyű. A változó értékét minden kiírást követően eggyel növelje meg, kezdetben pedig a 0000h-ról induljon. Ha volt lenyomva billentyű, akkor annak ASCII kódja szerinti karaktert írja ki a második sorba. A szóköz (space) megnyomására lépjen ki.

Pelda12.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Szamlalo    DW      0000h
HexaJegy    DB      "0123456789ABCDEF"
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

HexKiir     PROC
            PUSH    AX BX CX DX
            LEA     BX,[HexaJegy]
            MOV     CL,4
            MOV     DL,AL
            SHR     AL,CL
            XLAT
            XCHG    AL,DL
            AND     AL,0Fh
            XLAT
```

```
            MOV     DH,AL
            PUSH    DX
            XCHG    AL,AH
            MOV     DL,AL
            SHR     AL,CL
            XLAT
            XCHG    AL,DL
            AND     AL,0Fh
            XLAT
            MOV     DH,AL
            MOV     AH,02h
            INT     21h
            MOV     DL,DH
            INT     21h
            POP     DX
            INT     21h
            MOV     DL,DH
            INT     21h
            POP     DX CX BX AX
            RET
HexKiir     ENDP

Torol       PROC
            PUSH    AX BX CX DX
            MOV     AX,0600h
            MOV     BH,07h
            XOR     CX,CX
            MOV     DX,184Fh
            INT     10h
            POP     DX CX BX AX
            RET
Torol       ENDP

GotoXY      PROC
            PUSH    AX BX
            MOV     AH,02h
            XOR     BH,BH
            INT     10h
            POP     BX AX
```

```
            RET
GotoXY      ENDP

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            CALL    Torol
@Ciklus:
            XOR     DX,DX
            CALL    GotoXY
            MOV     AX,Szamlalo
            CALL    HexKiir
            INC     AX
            MOV     [Szamlalo],AX
            MOV     AH,01h
            INT     16h
            JZ      @Ciklus
            CMP     AL,20h
            JE      @Vege
            INC     DH
            CALL    GotoXY
            MOV     AH,02h
            MOV     DL,AL
            INT     21h
            XOR     AH,AH
            INT     16h
            JMP     @Ciklus
@Vege:
            XOR     AH,AH
            INT     16h
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Az adatszegmensben nincs semmi ismeretlen, bár a HexaJegy tartalma kicsit furcsának tűnhet. A homályt

rövidesen el fogja oszlatni, ha megnézzük, hogyan írjuk ki a számot.

A HexKiir az AX-ben levő előjeltelen számot írja ki négy karakteres hexadecimális alakban (tehát a vezető nullákkal együtt). Az eljárásban a szó kiírását visszavezetjük két bájt hexadecimális kiírására, azt pedig az alsó és felső bitnégyesek számjeggyé alakítására. Mivel a képernyőre először a felső bájt értékét kell kiírni, az alsó bájtot dolgozzuk fel, amit aztán berakunk a verembe, majd a felső bájt hasonló átalakításai után azt kiírjuk a képernyőre, végül a veremből kivéve az alsó bájt hexadecimális alakja is megjelenik a helyén. DX-ben fogjuk tárolni a már elkészült számjegyeket. BX és CL szerepére hamarosan fény derül.

Először a felső bitnégyest kell számjeggyé alakítanunk. Ezért AL-t elrakjuk DL-be, majd AL felveszi a felső bitnégyes értékét (ami 00h és 0Fh között lesz). Ezt az SHR AL,CL utasítással érjük el. Most CL=4 (erre állítottuk be), s így AL felső négy bitje lekerül az alsó négy helyére, miközben a felső bitnégyes kinullázódik (ez a shiftelés tulajdonsága). Ez pedig azt fogja jelenteni, hogy AL-ben ugyanakkora szám lesz, mint amekkora eredetileg a felső bitnégyesében volt.

Az operandus nélküli XLAT (transLATe byte; X=trans) utasítás az AL-ben levő bájtot kicseréli a DS:BX című fordítótáblázat AL-edik elemére (nullától számozva az elemeket), tehát szimbolikusan megfelel a MOV AL,[BX+AL] utasításnak (ami ugyebár így illegális), de AL-t előjeltelenül értelmezi. Az alapértelmezett DS szegmens felülbírálható prefixszel. Esetünkben ez most azt jelenti, hogy az AL-ben levő számot az azt szimbolizáló hexadecimális számjegyre cseréli le, hiszen BX a HexaJegy offszetcímét tartalmazza. Így már világos a HexaJegy definíciója.

Most, hogy megvan az első jegy, AL eredeti alsó bitnégyesét is át kell alakítani számjeggyé. AL-t DL-ben tároltuk el, és az elején azt mondtuk, hogy a kész számjegyeket

DX-ben fogjuk gyűjteni. Most tehát célszerűnek lenne, ha AL és DL tartalmát fel tudnánk cserélni. Erre jó a kétoperandusú XCHG (eXCHanGe) utasítás, ami a két operandusát felcseréli. Az operandusok vagy 8, vagy 16 bites regiszterek, vagy egy regiszter és egy memóriahivatkozás lehetnek. A flag-eket persze nem bántja. Speciális esetnek számít, ha AX-et cseréljük fel valamelyik másik 16 bites általános regiszterrel, ez ugyanis csak 1 bájtos műveleti kódot eredményez. Az XCHG AX,AX utasítás a NOP (No OPeration) mnemonikkal is elérhető.

AL tehát ismét az eredeti értékét tartalmazza. Most a felső bitnégyest kellene valahogy leválasztani, törölni AL-ben, hogy ismét alkalmazhassuk az XLAT-ot. Ezt most az AND utasítás végzi el, ami ugyebár a céloperandust a forrásoperandussal logikai ÉS kapcsolatba hozza, majd az eredményt a cél helyén tárolja. Most AL-t a 0Fh értékkel hozza ÉS kapcsolatba, ami annyit tesz, hogy a felső bitnégyes törlődik (hiszen Valami AND 00…0b=0h), az alsó pedig változatlan marad (mivel Valami AND 11…1b=Valami).

AL-t most már átalakíthatjuk számjeggyé (XLAT), majd miután DH-ban eltároltuk, DX-et berakjuk a verembe. AH még mindig az eredeti értékét tartalmazza, ideje hát őt is feldolgozni. Az XCHG AL,AH után (ami MOV AL,AH is lehetne) az előzőkhöz hasonlóan előállítjuk a két számjegyet DX-ben. Ha ez is megvan, akkor nincs más hátra, mint a négy számjegyet kiírni. Ezt a már jól ismert 02h számú INT 21h szolgáltatással tesszük meg.

A fenti táblázatban látható, hogy az INT 10h felelős a megjelenítéssel kapcsolatos szolgáltatásokért. Így logikus, hogy ehhez a megszakításhoz fordulunk segítségért a képernyő letörléséhez. A Torol eljárás nagyon rövid, ami azért van, mert a törlést egyetlen INT 10h hívással elintézhetjük. A 06h-s szolgáltatás egy szöveges képernyőablak felfelé görgetésére (scrolling) szolgál, és úgy működik, hogy az AL-ben megadott számú sorral felfelé csúsztatja a képernyő tartalmát, az alul

megüresedő sorokat pedig a BH-ban megadott színű (attribútumú) szóközökkel tölti fel. Ha AL=0, akkor az egész képernyőt felgörgeti, ami végül is a kép törléséhez vezet. Az ablak bal felső sarkának koordinátáit CX, míg a jobb alsó sarokét DX tartalmazza. A felső bájtok (CH és DH) a sort, az alsók (CL és DL) az oszlopot jelentik, a számozás 0-tól kezdődik. Most úgy tekintjük, hogy a képernyő 80 oszlopos és 25 soros képet jelenít meg, ezért DH-ba 24-et (18h), DL-be pedig 79-et (4Fh) töltünk. A törlés után a kurzor a képernyő legalsó sorának elejére kerül.

A GotoXY eljárás a kurzor pozícionálását teszi meg. A 02h számú video-szolgáltatás a DH-adik sor DL-edik oszlopába rakja a kurzort, a számozás itt is 0 bázisú. BH-ba 0-t rakunk, de ezzel most ne törődjünk. (Akit érdekel, BH-ban a használt képernyőlap sorszámát kell megadni.)

A főprogram nagyon egyszerűre sikeredett, hiszen a legtöbb dolgot eljáráshívással intézi el. A képernyő letörlése (CALL Torol) után belépünk a @Ciklus kezdetű hurokba. Itt a kurzort felrakjuk a bal felső sarokba, ahová kiírjuk a számláló aktuális értékét, majd megnöveljük a változót. Most jön az, hogy meg kell nézni, nyomtak-e le billentyűt. Erre a nemrég látott 00h-s INT 16 szolgáltatás nem jó, hiszen az nekiáll várakozni, ha nincs lenyomva egyetlen gomb sem. Ehelyett a 01h funkciót használjuk fel, ami a ZF-ben jelzi, volt-e billentyű lenyomva: ha ZF=1, akkor nem, különben AX tartalma megfelel a 00h-s szolgáltatás által visszaadottnak (tehát AL-ben az ASCII kód, AH-ban a scan kód). Így ha ZF=1, akkor visszamegyünk a ciklus elejére. Máskülönben megnézzük, hogy a szóközt nyomták-e meg, ennek ASCII kódja 32 (20h), és ha igen, akkor vége a bulinak, a @Vege címkén át befejezzük a ténykedést. Egyébként a kurzort a következő (második) sorba állítjuk (erre azért jó most az INC DH, mert DX végig nulla marad a @Ciklus utáni sortól kezdve), majd a megszokott INT 21h 02h számú szolgáltatással kirakjuk a lenyomott billentyű ASCII kódját, ami AL-ben van. Az utána következő két sor

(XOR AH,AH // INT 16h) feleslegesnek tűnhet, de enélkül a következő lenyomott karaktert nem fogja érzékelni a program. (Kicsit precízebben: Az INT 16h egy pufferből olvassa ki a következő lenyomott gomb kódjait. A 01h-s szolgáltatás a puffer mutatóját nem állítja át, így a következő hívás ismét ezt a billentyűt jelezné lenyomottnak, ami nem lenne okés. A 00h hívása viszont módosítja a mutatót, és ezzel minden rendben lesz.) Dolgunk végeztével visszatérünk a ciklusba.

13.3 Munka állományokkal

Az állomány-kezelés nem tartozik a könnyű dolgok közé, de egyszer érdemes vele foglalkozni, sokszor ugyanis egyszerűbben célhoz érhetünk Assemblyben, mint valamelyik magas szintű nyelvben.

A feladat nem túl bonyolult: hozzunk létre egy állományt, majd figyeljük a billentyűzetet. Minden beírt karaktert írjon ki a program a képernyőre és az állományba is folyamatosan és azonnal. Az Esc megnyomására zárja le az állományt és lépjen ki a programból. Az Esc-hez tartozó karaktert már nem kell az állományba írni. Az állomány nevét a programban konstans módon tároljuk, legyen mondjuk "TESZT.OUT". Ha nem sikerült létrehozni az állományt, akkor írjon ki egy hiba-üzenetet, majd azonnal fejezze be a működést.

Pelda13.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
```

```
FNev        DB      "TESZT.OUT",00h
Hiba        DB      "Hiba a fájl "
            DB      "létrehozásakor!$"
Karakter    DB      ?
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

@Start:
            MOV     AX,ADAT
            MOV     DS,AX
            MOV     AH,3Ch
            XOR     CX,CX
            LEA     DX,[FNev]
            INT     21h
            JC      @Hiba
            MOV     BX,AX
            MOV     CX,0001h
@Ciklus:
            XOR     AH,AH
            INT     16h
            CMP     AL,1Bh
            JE      @Lezar
            MOV     [Karakter],AL
            MOV     AH,02h
            MOV     DL,AL
            INT     21h
            MOV     AH,40h
            LEA     DX,[Karakter]
            INT     21h
            JMP     @Ciklus
@Lezar:
            MOV     AH,3Eh
            INT     21h
            JMP     @Vege
@Hiba:
            MOV     AH,09h
            LEA     DX,[Hiba]
```

```
            INT     21h
@Vege:
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Ha állományokkal, könyvtárakkal vagy lemezekkel kell dolgoznunk, azt mindenképpen a DOS szolgáltatásain keresztül érdemes tenni, hacsak valami egyéb indok (pl. a sebesség kritikus) nem indokol mást. A DOS az összes megszokott tevékenység végrehajtását lehetővé teszi az INT 21h megszakítás szolgáltatásain keresztül:

39h	**Könyvtár létrehozás (MKDIR)**
3Ah	**Könyvtár törlés (RMDIR)**
3Bh	**Könyvtárváltás (CHDIR)**
3Ch	**Állomány létrehozás/csonkítás (Create)**
3Dh	**Állomány megnyitás (Open)**
3Eh	**Állomány lezárás (Close)**
3Fh	**Olvasás megnyitott állományból (Read)**
40h	**Írás megnyitott állományba (Write)**

41h	**Lezárt állomány törlése (Delete/Unlink)**
42h	**Fájlmutató pozícionálása (Seek)**
43h	**Attribútumok beállítása/olvasása (CHMOD)**

Ezeknek az állománykezelő függvényeknek közös tulajdonsága, hogy az állományra a megnyitás és/vagy létrehozás után nem az állomány nevével, hanem egy speciális, egyedi azonosítószámmal, az ú.n. *file handle*-lel hivatkoznak. Ez egy 16 bites érték. A szabványos ki-/bemeneti eszközök (standard input és output) a 0000h..0004h értékeken érhetők el, ezekre is ugyanúgy írhatunk, ill. olvashatunk róluk, mintha

közönséges fájlok lennének. Az alábbi táblázat mutatja az ilyen eszközök handle számát:

0	**Standard input (STDIN)**
1	**Standard output (STDOUT)**
2	**Standard hiba (STDERR)**
3	**Standard soros porti eszköz (STDAUX)**
4	**Standard nyomtató (STDPRN)**

Új állomány létrehozására a 3Ch számú szolgáltatás való. Hívásakor CX-ben kell megadni a leendő állomány attribútumait a következő módon: az alsó három bit egyes bitjei kapcsolóknak felelnek meg, 1 jelenti a csak írható (Read Only) hozzáférést, 2 a rejtett fájlt (Hidden), 4 pedig a rendszerfájlt (System). Ezek különböző kombinációja határozza meg a végleges attribútumot. Most nekünk egy normál attribútumú fájlra van szükségünk, így CX nulla lesz. DS:DX tartalmazza az állomány nevét, amely egy ASCIIZ sztring, azaz a 00h kódú karakterrel kell lezárni. A szolgáltatás visszatéréskor CF-ben jelzi, hogy volt-e hiba. Ha CF=1, akkor volt, ilyenkor AX tartalmazza a hiba jellegét leíró hibakódot, különben pedig AX-ben a file handle található. Hiba esetén kiírjuk a hibaüzenetet a 09h-s szolgáltatással, majd befejezzük a programot. Állomány létrehozásánál figyelni kell arra, hogy ha az adott nevű állomány már létezik, akkor a DOS nem tér vissza hibával, hanem az állomány méretét 0-ra állítja, majd megnyitja azt.

Ha a létrehozás sikerült, akkor a handle-t átrakjuk BX-be, mivel a további szolgáltatások már ott fogják keresni.

A billentyűzetről való olvasásra a már említett 00h-s funkciót használjuk. Az Esc billentyű a 17 (1Bh) kódú karaktert generálja, így ha ezt kaptuk meg AL-ben, akkor elugrunk a @Lezar címkére. Megnyitott (ez fontos!) állomány lezárására a 3Eh szolgáltatást kell használni, ami BX-ben várja a handle-t,

visszatéréskor pedig CF ill. AX mutatja az esetleges hibát és annak okát.

Ha nem az Esc-et nyomták le, akkor AL-t berakjuk a Karakter nevű változóba, majd kiíratjuk a képernyőre a már jól ismert 02h DOS-funkcióval. Ezután a 40h számú szolgáltatást vesszük igénybe a fájlba íráshoz. A handle-t itt is BX-ben kell közölni, CX tartalmazza a kiírandó bájtok számát (ez most 1 lesz), DS:DX pedig annak a memóriaterületnek a címe, ahonnan a kiírást kérjük. Visszatéréskor AX a ténylegesen kiírt bájtok számát mutatja, ill. a hibakódot, ha CF=1. Ezek után visszatérünk a @Ciklus címkére.

13.4 Grafikus funkciók használata

Most ismét egy kicsit nehezebb feladatot oldunk meg, de a fáradozás meg fogja érni. A program először átvált a 640*480 felbontású grafikus videomódba, majd egy pattogó fehér pontot jelenít meg. A pont mindegyik falon (a képernyő szélein) rendesen vissza fog pattanni. Billentyű lenyomására pedig vissza fogja állítani a képernyőmódot szövegesre, majd ki fog lépni.

Pelda14.ASM:

```
MODEL SMALL

.STACK

KOD         SEGMENT
            ASSUME CS:KOD,DS:NOTHING

@Start:
            MOV     AX,0012h
            INT     10h
```

```
            XOR     CX,CX
            MOV     DX,CX
            MOV     SI,3
            MOV     DI,2
            XOR     BH,BH
@Ciklus:
            MOV     AX,0C00h
            INT     10h
            ADD     CX,SI
            JNS     @V1
            NEG     SI
            ADD     CX,SI
@V1:
            CMP     CX,640
            JB      @V2
            NEG     SI
            ADD     CX,SI
@V2:
            ADD     DX,DI
            JNS     @F1
            NEG     DI
            ADD     DX,DI
@F1:
            CMP     DX,480
            JB      @F2
            NEG     DI
            ADD     DX,DI
@F2:
            MOV     AL,0Fh
            INT     10h
;Várakozás...
            PUSH    CX DX
            MOV     DL,10
@Var1:
            XOR     CX,CX
@Var2:
            LOOP    @Var2
            DEC     DL
            JNZ     @Var1
```

```
            POP     DX CX
;
            MOV     AH,01h
            INT     16h
            JZ      @Ciklus
            XOR     AH,AH
            INT     16h
            MOV     AX,0003h
            INT     10h
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Adatszegmensre most nincs szükségünk, úgyhogy rögtön a videomód beállításával kezdünk. Az INT 10h 00h-s szolgáltatásával állítható be a képernyőn használt videomód, az igényelt mód számát AL-ben kell közölnünk. Sok szabványos videomódnak van előre rögzített száma, de nekünk most csak a 80*25 felbontású színes szöveges (ez a 03h számú mód), ill. a 640*480 felbontású grafikus (12h a száma) VGA-módokra lesz szükségünk. A VGA (Video Graphics Array) a videovezérlő típusára utal, azaz ennél régebbi típusú (EGA, CGA, HGC stb.) videokártyával nem tudjuk kipróbálni a programot, de ez manapság már nem túl nagy akadály.

A pattogó pont koordinátáit a DX és CX regiszter tárolja majd, DX mutatja a sort, CX pedig az oszlopot. Mindkettő 0 bázisú, az origó tehát a (0,0). BH a használt képernyőlap sorszámát tartalmazza, de ez most lényegtelen. SI lesz a vízszintes, DI pedig a függőleges lépésköz, azaz ennyi képponttal fog arrébb ugrani a pont.

A fő ciklusban először törölni kell a pont előző képét, majd ki kell számolni az új koordinátákat, ki kell rakni az új helyére a pontot, végül ellenőrizni kell a billentyűzetet.

A pont törlése annyit jelent, hogy a legutóbbi (CX,DX) koordinátákra fekete színnel kirakunk egy pontot. Pont rajzolására a 0Ch szolgáltatás lesz jó. AL-ben a rajzolt pont színét kell megadni, ehhez tekintsük a következő táblázatokat:

00h	Fekete
01h	Kék
02h	Zöld
03h	Ciánkék
04h	Vörös
05h	Bíborlila
06h	Barna
07h	Világosszürke

08h	Sötétszürke
09h	Világoskék
0Ah	Világoszöld
0Bh	Világoscián
0Ch	Világospiros
0Dh	Világosbíbor
0Eh	Sárga
0Fh	Fehér

Látjuk, hogy a fekete kódja a 00h. BH-ban a már említett képernyőlap-sorszám van, a kirakandó pont koordinátáit pedig éppen a CX, DX regiszterekben várja.

Miután töröltük a pont előző példányát, ideje, hogy az új hely koordinátáit meghatározzuk. A vízszintes és függőleges koordinátákat hasonlóan dolgozzuk fel, ezért csak az egyiket nézzük most meg. Tekintsük mondjuk az abszcissza kiszámítását. Először természetesen a lépésközt (SI) adjuk hozzá CX-hez. Ha a kapott érték negatív, akkor SF be fog állni. Eszerint végrehajtjuk a szükséges korrekciót és rátérünk a @V1 címkére, vagy pedig rögtön erre a címkére jövünk. A korrekció abban áll, hogy a lépésköz előjelét megfordítjuk (tehát negáljuk a lépésközt), majd ezt az új lépésközt ismét hozzáadjuk CX-hez. Az egyoperandusú NEG (NEGate) utasítás az operandust negálja, tehát képezi a kettes komplemensét, az aritmetikai flag-eket is beállítva. Ha ezen az ellenőrzésen túljutottunk, akkor még meg kell nézni, hogy a képernyő jobb oldalán nem mentünk-e túl. Ha igen (azaz CX>639), akkor ugyanúgy járunk el, mint az előbb, tehát negáljuk a lépésközt és

hozzáadjuk CX-hez. DX módosítása teljesen hasonló módon történik.

Most már kirakhatjuk a pontot új helyére. Mivel fehér pontot szeretnénk, AL-be 0Fh-t rakunk.

Az ez után következő néhány sor nem tartozik az eredeti célkitűzéshez, de a működés ellenőrzéséhez elengedhetetlen. A két pontosvesszővel közrefogott sorok csak a ciklus lassítását szolgálják, és nem csinálnak mást, mint 655360-szor végrehajtják az üres ciklusmagot. Erre a mostani számítógépek gyorsasága miatt van szükség, különben a pont nemhogy pattogna, de valósággal száguldana a képernyőn.

Miután letelt a késleltetés, a 01h-s INT 16h szolgáltatás hívásával megnézzük, hogy nyomtak-e le billentyűt, s ha nem, akkor irány vissza a ciklus elejére.

Kilépés előtt még egyszer kiolvassuk a billentyűzetet, majd visszaállítjuk a videomódot szövegesre.

14 MEGSZAKÍTÁS-ÁTDEFINIÁLÁS, HARDVER-MEGSZAKÍTÁSOK, REZIDENS PROGRAM, KAPCSOLAT A PERIFÉRIÁKKAL, HARDVER-PROGRAMOZÁS

Ideje, hogy kicsit mélyebben elmerüljünk a megszakítások világában. Nézzük meg először, hogy is hajtja végre a processzor az INT utasítást.

A 0000h szegmens első 1024 bájtján (a 0000h és a 03FFh offszetek közti területen) található a megszakítás-vektor tábla. Ez nem más, mint egy *ugrótábla*, mivel mindegyik bejegyzése egy 4 bájtos távoli pointer (azaz szegmens:offszet típusú memó-

riacím), nevük *megszakítás-vektor* (interrupt vector). Kicsit utánaszámolva láthatjuk, hogy 256*4=1024. Ebben a táblában található tehát mind a 256 db. szoftver- (részben hardver- is) megszakítás végrehajtó programjának kezdőcíme. Ezeket a programokat *megszakítás-kezelőnek* (interrupt handler) nevezzük. Minden megszakításnak pontosan egy kezelőprogramja van, de ugyanaz a kezelő több megszakításhoz is tartozhat.

Ha a programban az INT utasítás kódjával találkozik a processzor (ami 0CDh), akkor kiolvassa az utána levő bájtot is, ez a kért megszakítás száma. Felismerve, hogy megszakítás következik, a verembe berakja sorban a Flags, CS és IP regiszterek aktuális értékét (CS:IP az INT utasítást követő utasítás címét tartalmazza), majd az IF és TF flag-eket törli, ezzel biztosítva, hogy a megszakítás lefolyását nem fogja semmi megakadályozni. (Szimbolikusan: PUSHF // PUSH CS // PUSH IP) Jelölje a kért megszakítás számát N. Ezek után a processzor betölti a CS:IP regiszterpárba a memória 0000h:(N*4) címén levő duplaszót, azaz CS-be a 0000h:(N*4+2) címen levő, míg IP-be a 0000h:(N*4) címen található szó kerül betöltésre (emlékezzünk vissza, a processzor little-endian tárolásmódot használ). Ennek hatására az N-edik megszakítás-vektor által mutatott címen folytatódik a végrehajtás.

Ha a kezelő elvégezte dolgát, akkor egy különleges utasítás segítségével visszatér az őt hívó programhoz. Erre az IRET (Interrupt RETurn) operandus nélküli utasítás szolgál. Az IRET kiadásakor a processzor a visszatérési címet betölti a CS:IP-be (azaz leemeli először az IP-t, majd CS-t a veremből), a Flags regiszter tartalmát szintén visszaállítja a veremből, majd a végrehajtást az új címen folytatja. (Szimbolikusan: POP IP // POP CS // POPF)

Nézzünk egy példát! Tegyük fel, hogy az INT 21h utasítást adjuk ki, és legyen Flags=0F283h (azaz IF=SF=CF=1, a többi flag mind 0), CS=7000h, IP=1000h, SS lényegtelen, SP=0400h.

Az INT 21h megszakítás vektora a 0000h:0084h címen található, ennek értéke most legyen mondjuk 2000h:3000h. Az INT 21h utasítás végrehajtásának hatására a következők történnek: CS:IP értéke 2000h:3000h lesz, Flags-ben törlődik az IF flag, így az a 0F083h értéket fogja tartalmazni, valamint a verem a következő képet fogja mutatni:

	SS:0400h : ??
	SS:03FEh : 0F283h (Flags)
	SS:03FCh : 7000h (CS)
SP=>	SS:03FAh : 1000h (IP)
	SS:03F8h : ??

Mivel tudjuk, hogy hol találhatók az egyes megszakítások belépési pontjai, megtehetjük, hogy bármelyik vektort kedvünkre átírjuk. Ezzel a lehetőséggel rengeteg program él is, elég, ha a DOS-t, BIOS-t, vagy mondjuk az egeret kezelő programot (eszközmeghajtót) említjük. És mivel a hardver-megszakítások is bizonyos szoftver-megszakításokon keresztül lesznek lekezelve, akár ezeket is átirányíthatjuk saját programunkra. Mindkét esetre mutatunk most példákat.

14.1 Szoftver-megszakítás átirányítása

Első programunk a következőt fogja tenni: indításkor át fogja venni az INT 21h kezelését. Az új kezelő a régit fogja meghívni, de ha az igényelt funkció a 09h-s lesz (az a bizonyos sztringkiíró rutin), akkor DS:DX-et egy előre rögzített szöveg címére fogja beállítani, és arra fogja meghívni az eredeti szövegkiíró funkciót. A program billentyű lenyomása után vissza fogja állítani az eredeti megszakítás-kezelőt, majd be fog fejeződni.

Pelda15.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Helyes      DB      "Ha ezt látod, akkor"
            DB      " működik a dolog.$"
Teszt       DB      "Ez nem fog megjelenni!$"
ADAT        ENDS

KOD         SEGMENT
            ASSUME CS:KOD,DS:ADAT

RegiCim     DW      ?,?

UjKezelo    PROC
            PUSHF
            CMP     AH,09h
            JNE     @Nem09h
            POPF
            PUSH    DS
            PUSH    DX
            MOV     DX,ADAT
            MOV     DS,DX
            LEA     DX,[Helyes]
            PUSHF
            CALL    DWORD PTR CS:[RegiCim]
            POP     DX
            POP     DS
            IRET
@Nem09h:
            POPF
            JMP     DWORD PTR CS:[RegiCim]
UjKezelo    ENDP

@Start:
            MOV     AX,ADAT
```

```
            MOV     DS,AX
            XOR     AX,AX
            MOV     ES,AX
            CLI
            LEA     AX,[UjKezelo]
            XCHG    AX,ES:[21h*4]
            MOV     CS:[RegiCim],AX
            MOV     AX,CS
            XCHG    AX,ES:[21h*4+2]
            MOV     CS:[RegiCim+2],AX
            STI
            MOV     AH,09h
            LEA     DX,[Teszt]
            INT     21h
            MOV     AH,02h
            MOV     DL,0Dh
            INT     21h
            MOV     DL,0Ah
            INT     21h
            XOR     AH,AH
            INT     16h
            CLI
            MOV     AX,CS:[RegiCim]
            MOV     ES:[21h*4],AX
            MOV     AX,CS:[RegiCim+2]
            MOV     ES:[21h*4+2],AX
            STI
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Az első szembetűnő dolog az, hogy a kódszegmensben van definiálva a RegiCim nevű változó, ami majd az eredeti INT 21h kiszolgáló címét fogja tárolni. Ennek magyarázata a megszakítás-kezelő rutin működésében rejlik.

Azt már említettük, hogy minden eljárás elején érdemes és illendő elmenteni a használt regisztereket a verembe, hogy

azokat az eljárás végén gond nélkül visszaállíthassuk eredeti értékükre. Ez a szabály a megszakítás-kezelő rutinokra kötelezőre változik. A regiszterekbe a szegmensregisztereket és a Flags regisztert is bele kell érteni. Az operandus nélküli PUSHF (PUSH Flags) utasítás a verembe berakja a Flags regisztert, míg a POPF (POP Flags) a verem tetején levő szót a Flags-be tölti be.

A saját kezelő rutinunk működése két ágra fog szakadni aszerint, hogy AH egyenlő-e 09h-val avagy nem. A feltétel tesztelését a hagyományos CMP–JNE párossal oldjuk meg, de mivel ez az utasítás megváltoztat(hat) néhány flag-et, ezért a tesztelés előtt PUSHF áll, valamint mindkét ág a POPF-fel indul. Ez garantálja, hogy az összes flag érintetlenül marad. Erre kérdezhetné valaki, hogy miért őrizzük meg a flag-ek értékét, ha a Flags úgyis el van mentve a veremben. Nos, már láttuk, hogy néhány szolgáltatás pl. a CF-ben jelzi, hogy volt-e hiba. Ezek a szolgáltatások visszatérés során a veremben levő Flags regiszter-példányt egyszerűen eldobják (később látni fogjuk, hogyan), így módosításunk a hívó programra is visszahatna, ami kellemetlen lehetne.

Ha AH≠09h, akkor a @Nem09h talányos nevű címkén folytatódik a megszakítás kiszolgálása. Azt mondtuk, hogy minden egyéb esetben a régi funkciót fogjuk végrehajtatni. Ennek megfelelően cselekszik a programban szereplő JMP utasítás. Ezt a mnemonikot már ismerjük és használtuk is. Itt viszont két újdonságot is láthatunk: az első, hogy az operandus nem egy eljárás neve vagy címkéje mint eddig, hanem egy memóriahivatkozás (szegmensprefixszel együtt). Az ugrás eme formáját *közvetett* avagy *indirekt ugrásnak* (indirect jump) nevezzük, míg a régebbi alakot *közvetlen* vagy *direkt ugrásnak* (direct jump). Az elnevezés arra utal, hogy a cél címét nem az operandus, hanem az operandus által mutatott memóriacímen levő változóban levő pointer határozza meg. Magyarán: a JMP-t úgy hajtja végre a processzor, hogy kiszámítja az operandus

(esetünkben a CS:RegiCim változó) tényleges címét, majd az azon a címen levő pointert tölti be a megfelelő regiszterekbe (IP-be vagy a CS:IP párba). A másik újdonságot a DWORD PTR képviseli, ennek hatására az operandus típusa most DWORD lesz, ami duplaszót jelent ugyebár. Ha helyette WORD PTR állna, akkor a JMP a szokásos *közeli* (near) ugrást tenné meg, azaz csak IP-t változtatná meg. Nekünk viszont CS:IP-t kell új értékekkel feltöltenünk, és ez már *szegmensközi* vagy *távoli ugrást* jelent (intersegment vagy far jump). A DWORD helyett állhatna még a FAR is, az ugyancsak távoli pointert írna elő. Az utasítás most tehát CS-be a CS:(RegiCim+2) címen levő, míg IP-be a CS:RegiCim címen levő szót tölti be, majd onnan folytatja a végrehajtást. A szemfülesebbek rögtön keresni kezdik az említett IRET-et. Erre most nekünk nincs szükségünk, hiszen SP a régi (híváskori) állapotában van, a Flags is érintetlen, és az eredeti kiszolgálóból való visszatérés után nem akarunk már semmit sem csinálni. Ezért a legegyszerűbb módszert választjuk: a feltétlen vezérlésátadást a régi rutinra. Ha az a rutin befejezte ténykedését, a veremben az eredeti Flags, CS és IP értékeket fogja találni, és így közvetlenül a hívó programba (nem az UjKezelo eljárásba) fog visszatérni.**

Ha AH=09h, akkor el kell végeznünk DS:DX módosítását tervünknek megfelelően. Mivel nem szép dolog, ha a változás visszahat a hívóra, mindkét regisztert elmentjük a verembe. Ez a művelet azonban meggátol bennünket abban, hogy a másik esethez hasonlóan ráugorjunk a régi kezelő címére. Ha ugyanis a JMP DWORD PTR CS:[RegiCim] utasítást alkalmaznánk itt is, akkor a hívó programba visszatérés nem igazán sikerülne, mondhatni csődöt mondana. Hogy miért? A kulcs a verem. A legutolsó két veremművelettel a verem tetején a DS és DX regiszterek híváskori értéke lesz, amit a régi kezelőrutin a CS:IP regiszterekbe fog betölteni, és ez valószínűleg katasztrofális lesz (nem beszélve arról, hogy SP is meg fog változni a híváskori helyzethez képest). Ezért most ugrás helyett

hívást kell alkalmaznunk. Ez a hívás is kicsit más, mint az eddig megismert. A CALL utasítás ilyen alakját a fenti példához hasonlóan *távoli indirekt eljáráshívásnak* nevezzük. (A CALL-nak is létezik közeli és távoli alakja, és mindkettőből van direkt és indirekt változat is.) A CALL hatására CS és IP bekerül a verembe, a Flags viszont nem, miközben a régi kiszolgálórutin arra számít, hogy az SS:(SP+4) címen (ez nem szabályos címzésmód!) a Flags tükörképe van. Ezért a CALL előtt még kiadunk egy PUSHF-et. Aki kicsit jobban elgondolkodik, annak feltűnhet, hogy ez az utasításpáros tulajdonképpen egy INT utasítást szimulál. Így ha a régi kiszolgáló végez, akkor visszatér a saját kezelőnkbe, ahol mi kitakarítjuk a veremből a DS és DX értékeit, majd visszatérünk a hívó programba. Ezt most IRET-tel tesszük meg annak ellenére, hogy említettük, némelyik szolgáltatás esetleg valamelyik flag-ben adhatna vissza eredményt. Mi most viszont csak a 09h számú szolgáltatás működésébe avatkozunk be, ami nem módosítja egyik flag-et sem.

Ha egészen korrekt megoldást akarunk, akkor az IRET utasítást a RETF 0002h utasítással kell helyettesíteni. A RETF (Far RETurn) az eddig használt RET utasítástól abban tér el, hogy visszatéréskor nemcsak IP-t, de CS-t is visszaállítja a veremből (tehát működése szimbolikusan POP IP // POP CS). Az opcionális szó méretű közvetlen operandus azt az értéket jelöli, amit azután (t.i. IP és CS POP-olása után) SP-hez hozzá kell adnia. A visszatérés eme formája annyiba tér el az IRET-től, hogy a Flags eredeti értékét nem állítja vissza a veremből, a megadott 0002h érték hatására viszont SP-t úgy módosítja, mintha egy POPF-et is végrehajtottunk volna. Az eredmény: a hívóhoz gond nélkül visszatérhetünk a megszakításból úgy, hogy esetleg valamelyik flag eredményt tartalmaz.

Nem válaszoltuk még meg, hogy miért a kódszegmensben definiáltuk a RegiCim változót. A válasz sejthető: ha az adatszegmensben lenne, akkor annak eléréséhez először be

kellene állítanunk DS-t. Ez viszont ellentmond annak a követelménynek, hogy DS értékét nem (sem) szabad megváltoztatnunk abban az esetben, ha AH≠09h. Ezt csak valamilyen csellel tudnánk biztosítani, pl. így:

```
PUSH   DS DX
MOV    DX,ADAT
MOV    DS,DX
PUSH   WORD PTR DS:[RegiCim+2]
PUSH   WORD PTR DS:[RegiCim]
POP    DX DS
RETF
```

Hangsúlyozzuk, ezt csak akkor kellene így csinálni, ha az ADAT nevű szegmensben definiáltuk volna a RegiCim változót. Mivel azonban a kódszegmensbe raktuk a definíciót, a kezelő rutin közvetlenül el tudja érni a változót. (Ehhez azért az is kell, hogy az INT 21h kiadása után a saját kezelő rutinunk CS szegmense megegyezzen a RegiCim szegmensével, de ez most fennáll.)

ES-t eddig még nem sokszor használtuk, most viszont kapóra jön a megszakítás-vektor táblázat szegmensének tárolásánál.

Mielőtt nekifognánk átírni az INT 21h vektorát, a CLI (CLear Interrupt flag) utasítással letiltjuk a hardvermegszakításokat (IF=0 lesz). Erre azért van szükség, mert ha az átírás közepén bejönne egy megszakítás, akkor azt a processzor minden további nélkül kiszolgálná. Ha annak kezelőjében szerepelne egy INT 21h utasítás, akkor a rossz belépési cím miatt nem a megfelelő rutin hívódna meg, és ez valószínűleg álomba küldené a gépet. A vektor módosítása után ismét engedélyezzük a bejövő megszakításokat az IF 1-be állításával, amit az STI (SeT Interrupt flag) utasítás végez el.

A vektor módosításánál az XCHG egy csapásra megoldja mind a régi érték kiolvasását, mind az új érték beírását.

Miután sikeresen magunkra irányítottuk ezt a megszakítást, rögtön ki is próbáljuk. A helyes működés abban áll, hogy nem a Teszt címkéjű szöveg fog megjelenni, hanem a Helyes. Annak ellenőrzésére, hogy a többi szolgáltatást nem bántottuk, a 02h funkcióval egy új sor karakterpárt írunk ki.

Végül gombnyomás után visszaállítjuk az eredeti kezelőt, és kilépünk a programból.

14.2 Az időzítő (timer) programozása

A következő program már a hardver-programozás tárgykörébe tartozik. A feladat az, hogy írjunk egy olyan eljárást, ami ezredmásodperc (millisecundum) pontossággal képes meghatározott ideig várakozni. Ezzel például stoppert is megvalósíthatunk. Ez a rész kicsit magasabb szintű a szokásosnál, így ha valaki esetleg nem értené, akkor nyugodtan ugorja át. :)

Pelda16.ASM:

```
MODEL SMALL

.STACK

KOD         SEGMENT
            ASSUME CS:KOD,DS:NOTHING

Orajel      DW      ?
Oszto       DW      ?
MSec        DW      ?
RegiCim     DW      ?,?

UjKezelo    PROC
```

```
            PUSH    AX
            INC     WORD PTR CS:[MSec]
            MOV     AX,CS:[Oszto]
            ADD     CS:[Orajel],AX
            JC      @Regi
            MOV     AL,20h
            OUT     20h,AL
            POP     AX
            IRET
@Regi:
            POP     AX
            JMP     DWORD PTR CS:[RegiCim]
UjKezelo    ENDP

@Start:
            XOR     AX,AX
            MOV     DS,AX
            CLI
            LEA     AX,[UjKezelo]
            XCHG    AX,DS:[08h*4]
            MOV     CS:[RegiCim],AX
            MOV     AX,CS
            XCHG    AX,DS:[08h*4+2]
            MOV     CS:[RegiCim+2],AX
            MOV     DX,0012h
            MOV     AX,34DCh
            MOV     BX,1000
            DIV     BX
            MOV     CS:[Oszto],AX
            MOV     WORD PTR CS:[Orajel],0000h
            MOV     BL,AL
            MOV     AL,36h
            OUT     43h,AL
            JMP     $+2
            MOV     AL,BL
            MOV     DX,0040h
            OUT     DX,AL
            JMP     $+2
            MOV     AL,AH
```

```
            OUT     DX,AL
            JMP     $+2
            STI
            MOV     AH,02h
            MOV     DL,'1'
            MOV     CX,9
            MOV     BX,1000
@Ciklus:
            INT     21h
            INC     DL
            MOV     WORD PTR CS:[MSec],0000h
@Var:
            CMP     CS:[MSec],BX
            JB      @Var
            LOOP    @Ciklus
            MOV     DL,0Dh
            INT     21h
            MOV     DL,0Ah
            INT     21h
            CLI
            MOV     AX,CS:[RegiCim]
            MOV     DS:[08h*4],AX
            MOV     AX,CS:[RegiCim+2]
            MOV     DS:[08h*4+2],AX
            MOV     AL,36h
            OUT     43h,AL
            JMP     $+2
            XOR     AL,AL
            OUT     40h,AL
            JMP     $+2
            OUT     40h,AL
            JMP     $+2
            STI
            MOV     AX,4C00h
            INT     21h
KOD         ENDS
            END     @Start
```

Adatszegmensre nem lesz szükségünk, a változókat a könnyebb elérhetőség miatt a kódszegmensben definiáljuk.

A PC hardverének áttekintésekor már megemlítettük az időzítőt (timer). Ez az egység három független számlálóval (counter) rendelkezik (szemléletesen három stopperrel), és mindegyik egy időzítő-csatornához tartozik. Ebből egyik a már szintén említett memória-frissítéshez kell, egy pedig a belső hangszóróra van rákötve. A harmadik felelős a belső rendszeróra karbantartásáért, valamint a floppy meghajtó motorjának kikapcsolásáért. Bár elsőre nem látszik, a megoldást ez utóbbi fogja jelenteni. Ha ez a számláló lejár, az időzítő egy megszakítást kér, ez a megszakítás-vezérlő IRQ0-ás vonalán keresztül valósul meg. Végül a hardver-megszakítást a processzor érzékeli, és ha lehet, kiszolgálja. Nos, ha az eredeti kiszolgálót mi lecseréljük a saját rutinunkra, ami a speciális feladatok elvégzése után meghívja a régi kiszolgálót, akkor nyert ügyünk van.

Az időzítő egy kvarckristályon keresztül állandó frekvenciával kapja az áramimpulzusokat. Ez a frekvencia az 1234DCh Hz (1193180 Hz). Az időzítő mindhárom számlálójához tartozik egy 16 bites osztóérték. A dolog úgy működik, hogy az adott számláló másodpercenként 1234DCh/Osztó -szor jár le. A rendszerórához tartozó számláló (ez a 0-ás csatorna) osztója 65536, míg a memória-frissítésért felelős (ez az 1-es csatorna) számlálóé 18 alapállapotban. Ha elvégezzük az osztásokat, azt látjuk, hogy a memóriát másodpercenként kb. 66288-szor frissítik, míg a rendszerórát másodpercenként kb. 18.2-szer állítja át a kiszolgálórutin.

Tervünket úgy fogjuk megvalósítani, hogy az időzítő 0-ás csatornájának osztóját olyan értékre állítjuk be, hogy másodpercenként 1000-szer generáljon megszakítást. A megszakítást az IRQ0 vonalról a megszakítás-vezérlő átirányítja az INT 08h szoftver-megszakításra, de ettől ez még hardver-megszakítás marad, aminek fontosságát később látjuk

majd. Ha IF=1, a processzor érzékeli a megszakítást, majd meghívja az INT 08h kezelőjét, ami elvégzi a teendőket. Ezt a kezelőt cseréljük le egy saját rutinra, ennek neve UjKezelo lesz.

Nézzük meg először a változók szerepét. Az eredeti kezelő címét ismét a RegiCim tárolja. Oszto tartalmazza a számlálóhoz tartozó kiszámolt osztóértéket. MSec tartalma minden megszakítás-kéréskor eggyel fog nőni, ezt használjuk fel majd a pontos várakozás megvalósítására. Az Orajel változó szerepének megértéséhez szükséges egy dolgot tisztázni. Mikor is kell meghívni a régi kiszolgálót? Ha minden megszakítás esetén meghívnánk, akkor az óra az eredetinél sokkal gyorsabban járna, és ezt nem szeretnénk. Pontosan úgy kell működnie, mint előtte. Ehhez nekünk is üzemeltetnünk kell egy számlálót. Ha ez lejár, akkor kell meghívni a régi kiszolgálót. Számláló helyett most osztót fogunk bevezetni, ennek mikéntje mindjárt kiderül.

Az UjKezelo eljárásban csak az AX regisztert használjuk fel, ezt tehát rendesen elmentjük a verembe. Ezt követően megnöveljük eggyel az MSec értékét célunk elérése érdekében. Most következik annak eldöntése, meg kell-e hívni a régi kiszolgálót. Ehhez az osztóértéket (Oszto tartalmát) hozzáadjuk az Orajel változóhoz. Ha átvitel keletkezik, az azt jelenti, hogy Orajel értéke nagyobb vagy egyenlő lett volna 65536-nál. Mivel azt mondtuk, hogy Orajel egy osztó szerepét játssza, innen következik, hogy a régi kezelőrutint pontosan akkor kell meghívni, amikor CF=1 lesz. Gyakorlatilag az időzítő működését utánozzuk: ha az osztóértékek összege eléri vagy meghaladja a 65536-ot, és ekkor hívjuk meg a régi rutint, ez pontosan azt csinálja, mint amikor az időzítő az eredeti 65536-os osztó szerinti időközönként vált ki megszakítást. Némi számolással és gondolkozással ezt magunk is beláthatjuk.

Ha CF=1, akkor szépen ráugrunk a régi kiszolgáló címére, s az majd dolga végeztével visszatér a megszakításból.

Ha viszont CF=0, akkor nekünk magunknak kell kiadni a parancsot a visszatéréshez. Előtte azonban meg kell tenni egy fontos dolgot, és itt lesz szerepe annak a ténynek, hogy ez a rutin mégiscsak hardver-megszakítás miatt hajtódik végre. Dolgunk végeztével jelezni kell a megszakítás-vezérlőnek, hogy minden oké, lekezeltük az adott hardver-megszakítást. Ha ezt a *nyugtázásnak* (acknowledgement) nevezett műveletet elfelejtjük megtenni, akkor az esetlegesen beérkező többi megszakítást nem fogja továbbítani a processzor felé a vezérlő. A megszakítás-vezérlőt a 0020h és 0021h számú portokon keresztül érhetjük el (AT-k esetében a második vezérlő a 00A0h, 00A1h portokon csücsül). Az OUT (OUTput to port) utasítás a céloperandus által megadott portra kiírja a forrásoperandus tartalmát. A port számát vagy 8 bites közvetlen adatként (0000h..00FFh portok esetén), vagy a DX regiszterben kell közölni, míg a forrás csak AL vagy AX lehet. Ha a forrás 16 bites, akkor az alsó bájt a megadott portra, míg a felső bájt a megadott után következő portra lesz kiírva. A nyugtázás csak annyit jelent, hogy a 0020h számú portra ki kell írni a 20h értéket (könnyű megjegyezni:). Ezután IRET-tel befejezzük ténykedésünket.

Mivel nincs adatszegmens, most DS-t használjuk fel a megszakítás-vektorok szegmensének tárolására. A megszakítások letiltása után a már szokott módon eltároljuk a régi INT 08h kezelő címét, ill. beállítjuk a saját eljárásunkat. Ezt követően kiszámoljuk az Oszto értékét. Mint említettük, az időzítő alapfrekvenciája 1234DCh, és minden ezredmásodpercben akarunk majd megszakítást, így az előző értéket 1000-rel elosztva a hányados a kívánt értéket fogja adni. Aki nem értené, ez miért van így, annak egy kis emlékeztető:

1234DCh/osztó=frekvencia

Ezt az egyenletet átrendezve már következik a módszer helyessége. Az Orajel változót kinullázzuk a megfelelő

működéshez. Ezután az osztót még az időzítővel is közölni kell, a három OUT utasítás erre szolgál. Ezekkel most nem foglalkozunk, akit érdekel, az nyugodtan nézzen utána valamelyik említett adatbázisban. A JMP utasítások csak arra szolgálnak, hogy kicsit késleltessék a további bájtok kiküldését a portokra, a perifériák ugyanis lassabban reagálnak, mint a processzor. Megfigyelhetjük a JMP operandusában a cél megjelölésekor a $ szimbólum használatát. Emlékeztetőül: a $ szimbólum az adott szegmensben az aktuális sor offszetjét tartalmazza. Ez a sor egyéb érdekességet is tartogat, a használt ugrás ugyanis sem nem közeli (near), sem nem távoli (far). Ezt az ugrásfajtát *rövid* (short) ugrásnak nevezzük, és ismertetőjele, hogy a relatív cím a feltételes ugrásokhoz és a LOOP-hoz hasonlóan csak 8 bites. A 2-es szám onnan jön, hogy a rövid ugrás utasításhossza 2 bájt (egy bájt a műveleti kód, egy pedig a relatív cím). A rövid ugrást még kétféleképpen is kikényszeríthetjük: az egyik, hogy a céloperandus elé odaírjuk a SHORT operátort, a másik, hogy JMP helyett a JMPS (JuMP Short) mnemonikot használjuk.

A megszakítás működésének tesztelésére egy rövid ciklus szolgál, ami annyit tesz, hogy kiírja a decimális számjegyeket 1-től 9-ig, mindegyik jegy után pontosan 1 másodpercet várakozva. A várakozást az MSec változó figyelésével tesszük annak nullázása után. MSec-et a megszakítások bekövetkeztekor növeli egyesével az UjKezelo eljárás, ez pedig ezredmásodpercenként történik meg. Ha tehát MSec=1000, akkor a nullázás óta pontosan egy másodperc telt el. A változó nullázásakor azért kell kiírni a WORD PTR operátort, mert az assembler nem fogja tudni eldönteni az operandusok méretét (hiszen a cél egy memóriahivatkozás, a forrás pedig egy közvetlen adat).

A ciklus lejárta után egy új sor karakterpárt írunk még ki, majd végezetül visszaállítjuk az eredeti INT 08h kezelőt, valamint az időzítő eredeti osztóját. Ezután befejezzük a

program működését. Az időzítővel kapcsolatban még annyit, hogy a 0000h érték jelöli a 65536-os osztót.

A program történetéhez hozzátartozik, hogy az első verzióban az UjKezelo eljárás legutolsó sorában a DWORD helyén FAR állt. A TASM 4.0-ás verziója azonban furcsa módon nem jól fordítja le ezt az utasítást ebben a formában, csak akkor, ha DWORD-öt írunk típusként. Erre későbbi programjainkban nem árt odafigyelni, ugyanis lehet, hogy a forrás szemantikailag jó, csak az assembler hibája miatt nem megfelelő kód szerepel a futtatható állományban. Erről általában a debugger segítségével győződünk meg.

14.3 Rezidens program (TSR) készítése, a szöveges képernyő közvetlen elérése

Rezidens programon **(resident program) egy olyan alkalmazást értünk, amihez tartozó memóriaterületet vagy annak egy részét a DOS nem szabadítja fel a program befejeződésekor. A program kódja ilyenkor a memóriában bentmarad. Az inaktív programot sokszor egy megszakítás vagy valamilyen hardveresemény éleszti fel. A programok másik gyakran használt elnevezése a *TSR* (Terminate and Stay Resident). Néhány jól ismert segédprogram is valójában egy TSR, mint pl. MSCDEX.EXE, egérmeghajtók (MOUSE.COM, GMOUSE.COM stb.), különféle képernyőlopók, commanderek (Norton Commander, Volkov Commander, DOS Navigator) stb.**

TSR-t .COM programként egyszerűbb írni, így most mi is ezt tesszük. A feladat az, hogy a program rezidensen maradjon a memóriában az indítás után, és a képernyő bal felső sarkában levő karakter kódját és annak színinformációit folyamatosan növelje eggyel. (Csak szöveges módban!) Nem túl hasznos, de legalább látványos.

Pelda17.ASM:

```
MODEL TINY

KOD         SEGMENT
            ASSUME CS:KOD,DS:KOD
            ORG     0100h

@Start1:
            JMP     @Start

RegiCim     DW      ?,?

UjKezelo    PROC
            PUSHF
            CALL    DWORD PTR CS:[RegiCim]
            PUSH    DI ES
            MOV     DI,0B800h
            MOV     ES,DI
            XOR     DI,DI
            INC     BYTE PTR ES:[DI]
            INC     DI
            INC     BYTE PTR ES:[DI]
            POP     ES DI
            IRET
UjKezelo    ENDP

@Start:
            XOR     AX,AX
            MOV     ES,AX
            CLI
            LDS     SI,ES:[1Ch*4]
            MOV     CS:[RegiCim],SI
            MOV     CS:[RegiCim+2],DS
            LEA     AX,[UjKezelo]
            MOV     ES:[1Ch*4],AX
            MOV     ES:[1Ch*4+2],CS
```

```
            STI
            LEA     DX,@Start
            INT     27h
KOD         ENDS
            END     @Start1
```

A feladat megoldásához egy olyan megszakításra kell "ráakaszkodni", ami elég sokszor hívódik meg. Ez lehetne az eddig megismertek közül INT 08h, INT 16h, esetleg az INT 21h. Az időzítő által kiváltott INT 08h eredeti kezelője dolga végeztével egy INT 1Ch utasítást ad ki, aminek a kezelője alapesetben csak egy IRET-ből áll. Ezt a megszakítást bárki szabadon átirányíthatja saját magára, feltéve, hogy meghívja az eredeti (pontosabban a megszakítás-vektor szerinti) kezelőt, illetve hogy a megszakításban nem tölt el túl sok időt. Az időkorlát betartása azért fontos, mert az INT 08h ugyebár egy hardver-megszakítás, és ha a megszakítás-vezérlőt csak az INT 1Ch lefutása után nyugtázza a kezelő, akkor az eltelt időtartam hossza kritikussá válhat.

Adatszegmensünk most sem lesz, az egy szem RegiCim változót a kódszegmens elején definiáljuk, és mellesleg a rezidens részben foglal helyet.

Az UjKezelo eljárás elején hívjuk meg az előző kezelőt, s ez után végezzük el a képernyőn a módosításokat.

A képernyőre nem csak a DOS (INT 21h) vagy a video BIOS (INT 10h) segítségével lehet írni. A megjelenítendő képernyőtartalmat a videovezérlő kártyán levő videomemóriában tárolják el, majd annak tartalmát kiolvasva készül el a végleges kép a monitoron. Ennek a memóriának egy része a fizikai memória egy rögzített címtartományában elérhető. Magyarra fordítva ez azt jelenti, hogy grafikus módok esetén a 0A0000h, míg szöveges módoknál a 0B0000h (fekete-fehér) vagy a 0B8000h (színes) fizikai címeken kezdődő terület a videokártyán levő memória aktuális állapotát tükrözi, és oda beírva valamit az igazi videomemória is módosulni fog. Grafikus

módoknál a 0A000h szegmens teljes terjedelmében felhasználható, míg szöveges mód esetén a 0B000h és 0B800h szegmenseknek csak az első fele (tehát a 0000h..7FFFh offszetek közötti terület). Bennünket most csak a szöveges módok érdekelnek, azokon belül is a színes képernyők esetén használatosak. Fekete-fehér (monokróm) képet előállító videokártya esetében a 0B000h szegmenset kell használni, egyéb változtatásra nincs szükség.

A videomemóriában a képernyőn látható karakterek sorfolytonosan helyezkednek el, és minden karaktert egy újabb bájt követ, ami a színinformációt (attribútumot) tartalmazza. Ez pontosan azt jelenti, hogy a karakterek a páros, míg az attribútum-bájtok a páratlan offszetcímeken találhatók. Az attribútumot a következő módon kódolják: a 0..3 bitek adják az előtérszínt, a 4..6 bitek a hátteret, míg a 7-es bit villogást ír elő, de úgy is beállítható a videokártya, hogy a háttérszín 4. bitjét jelentse. A színkódok megegyeznek a már korábban táblázatban leírtakkal.

Az UjKezelo rutinhoz visszatérve, a bal felső sarokban levő karakter címe a 0B800h:0000h, amit az attribútum követ. Miután mindkét bájtot külön-külön inkrementáltuk, IRET-tel visszatérünk a hívóhoz.

A főprogram szokatlanul rövidre sikerült, hiszen nincs sok feladata. Mindössze a régi megszakítás-vektort menti el, majd beállítja az újat. Az egyetlen újdonságot az LDS (Load full pointer into DS and a general purpose register) utasítás képviseli, ami a DS:céloperandus regiszterpárba a forrás-operandus által mutatott memóriaterületen levő távoli mutatót (azaz szegmenst és offszetet) tölti be. Ez a mostani példában azt jelenti, hogy SI-be az ES:(1Ch*4), míg DS-be az ES:(1Ch*4+2) címeken levő szavakat tölti be. Hasonló műveletet végez az LES utasítás, ami DS helyett ES-t használja. A céloperandus csak egy 16 bites általános célú regiszter, a forrás pedig csak memóriahivatkozás lehet mindkét utasításnál.

Ezek után a főprogram be is fejezi működését, de nem a hagyományos módon. Az INT 27h egy olyan DOS-megszakítás, ami a program működését úgy fejezi be, hogy annak egy része a memóriában marad (azaz TSR-ré válik). A rezidens rész a CS:DX címig tart, kezdete pedig a PSP szegmense. A PSP-re és a .COM programok szerkezetére a következő fejezet világít rá. Mi most csak a RegiCim változót és az UjKezelo eljárást hagyjuk a memóriában (meg a PSP-t, de ez most nem lényeges).

Az INT 27h-val legfeljebb 64 Kbájtnyi terület tehető rezidenssé, és kilépéskor nem adhatunk vissza hibakódot (exit code), amit az INT 21h 4Ch funkciójánál AL-ben közölhettünk. Ezeket a kényelmetlenségeket küszöböli ki az INT 21h 31h számú szolgáltatása. Ennek AL-ben megadhatjuk a szokásos visszatérési hibakódot, DX-ben pedig a rezidenssé teendő terület méretét kell megadnunk paragrafusokban (azaz 16 bájtos egységekben), a PSP szegmensétől számítva.

Megszakítás-vektorok beállítására és olvasására a DOS is kínál lehetőséget. Az INT 21h 25h számú szolgáltatása az AL-ben megadott számú megszakítás vektorát a DS:DX által leírt címre állítja be, míg a 35h szolgáltatás az AL számú megszakítás-vektor értékét ES:BX-ben adja vissza.

15 KIVÉTELEK

A *kivétel* (exception) egy olyan megszakítás, amit a processzor vált ki, ha egy olyan hibát észlel, ami lehetetlenné teszi a következő utasítás végrehajtását, vagy egy olyan esemény történt, amiről a programoknak és a felhasználónak is értesülniük kell. Minden kivételt egy decimális sorszámmal és egy névvel azonosítanak, ezenkívül minden kivételhez tartozik egy

kettőskeresztből (#) és két betűből álló rövidítés, amit szintén mnemoniknak hívnak (ez viszont nem egy utasításra utal).

A 8086-os mikroprocesszor összesen 4 féle kivételt képes generálni. A kivétel kiváltásához szükséges feltételek teljesülése esetén a processzor a verembe berakja a Flags, CS és IP regiszterek tartalmát (szimbolikusan PUSHF // PUSH CS // PUSH IP), majd törli az IF és TF flag-eket. Ezután sor kerül a kivételt lekezelő programrész (exception handler) végrehajtására. Az Intel az INT 00h..1Fh szoftver-megszakításokat a kivétel-kezelő programok számára tartja fenn. Egy adott N sorszámú kivétel esetén az INT N megszakítás kezelője lesz végrehajtva. A teendők elvégzése után a kezelő IRET utasítással visszatérhet abba a programba, amely a kivételt okozta. Bár a későbbi processzorokon megjelentek olyan, nagyon súlyos rendszerhibát jelző kivételek, amik után a kivételt okozó program már nem indítható újra (vagy nem folytatható), a 8086-os processzoron mindegyik kivétel megfelelő lekezelése után a hibázó program futása folytatható.

A kivételek 3 típusba sorolhatók: vannak hibák (*fault*) és csapdák (*trap*). (A 80286-os processzoron egy harmadik kategória is megjelent, ezek az ú.n. *abort*-ok.) A két fajta kivétel között az különbség, hogy míg a fault-ok bekövetkeztekor a verembe mentett CS:IP érték a hibázó utasításra mutat, addig a trap-ek esetén a hibát kiváltó utasítást követő utasítás címét tartalmazza CS:IP veremben levő másolata. (Az abort kivételek esetén a CS:IP-másolat értéke általában meghatározatlan.) Ezenkívül a trap-ek (mint nevük is mutatja) a program hibamentesítését, debuggolását támogatják, míg a fault és abort típusú kivételek a kritikus hibákat jelzik, és azonnali cselekvésre szólítanak fel.

Most pedig lássuk, milyen kivételek is vannak, és ezek mely feltételek hatására jönnek létre:

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*

0	#DE	Divide Error	Trap	DIV és IDIV utasítások
1	#DB	Debug	Trap	INT 01h utasítás/TF=1
3	#BP	Breakpoint	Trap	INT 3 utasítás
4	#OF	Overflow	Trap	INTO utasítás, ha OF=1

A 0-ás kivétel az osztáskor megtörténő hibák során keletkezik. A DIV és az IDIV utasítások válthatják ki, és alapvetően két oka van: az egyik, hogy nullával próbáltunk meg osztani, ez az eset általában a hibás algoritmusokban fordul elő. A másik ok viszont gyakoribb: akkor is ezt a kivételt kapjuk, ha az osztás hányadosa nem fér el a célban. Például ha AX=0100h, BL=01h, és kiadjuk a DIV/IDIV BL utasítások valamelyikét, akkor a hányados is 0100h lenne, ez viszont nem fér el AL-ben (hiszen a hányadost ott kapnánk meg, a maradékot pedig AH-ban). Ennek eredménye a 0-ás kivétel lesz. Ezt a kivételt alapesetben a DOS kezeli le, egy hibaüzenet kiírása után egyszerűen terminálja az éppen futó programot, majd visszaadja a vezérlést a szülő alkalmazásnak (COMMAND.COM, valamilyen shell program, commander stb.).

Megjegyezzük, hogy a 0-ás kivételt a 80286-os processzortól kezdődően hibaként (fault) kezelik.

Az 1-es kivétel keletkezésének két oka lehet: vagy kiadtunk egy INT 01h utasítást (kódja 0CDh 01h), vagy bekapcsoltuk a TF flag-et.

3-ast kivételt az INT 3 utasítás tud okozni. Az INT utasításnak erre a megszakítás-számra két alakja is van: az egyik a megszokott kétbájtos forma (0CDh 03h), a másik pedig csak egy bájtot foglal el (0CCh). A két kódolás nem teljesen azonos működést vált ki, ez azonban csak a 80386-os vagy annál újabb processzorokon létező virtuális-8086 üzemmódban nyilvánul meg.

4-es kivétel akkor keletkezik, ha OF=1 és kiadunk egy INTO utasítást.

Bár nem kivétel, említést érdemel egy korábban még be nem mutatott megszakítás. Az *NMI* (NonMaskable Interrupt) olyan hardver-megszakítás, ami a megszakítás-vezérlőt kikerülve közvetlenül a processzor egyik lábán (érintkezőjén) keresztül jut el a központi egységhez, és amint neve is mutatja, nem tiltható le. A CPU az IF flag állásától függetlenül mindig ki fogja szolgálni ezt a megszakítást, mégpedig az INT 02h kezelőt meghívva. Mivel NMI csak valamilyen hardverhiba folytán keletkezik (pl. memória-paritáshiba, nem megfelelő tápfeszültség stb.), lekezelése után a számítógép nincs biztonságos állapotban a folytatáshoz, ezért az NMI-kezelőben általában hibaüzenet kiírása (esetleg hangjelzés adása) után leállítják a működést. Ez utóbbit úgy érik el, hogy kiadnak egy CLI utasítást, majd egy végtelen ciklusba engedik a processzort (ez a legegyszerűbben egy önmagára mutató JMP utasítást jelent).

A dologban az a vicces, hogy az NMI keletkezését le lehet tiltani az alaplap egy bizonyos portjára írva. (XT-k és PC esetén a 00A0h porton a legfelső bit 0-ás értéke, míg AT-knál a 0070h port ugyanazon bitjének 1-es értéke tiltja le az NMI-t.) Ez elég kockázatos dolog, hiszen ha egyszer NMI-t észlel az alaplap, és nem képes azt a CPU tudtára hozni, akkor a felhasználó mit sem sejtve folytatja munkáját, közben pedig lehet, hogy több adatot veszít el így, mintha az NMI-t látva resetelné vagy kikapcsolná a gépet.

Megjegyezzük, hogy a koprocesszor (FPU) is generálhat NMI-t. Ebben az esetben az NMI keletkezése nem hardverhibát, hanem a számítások közben előforduló numerikus hibákat (kivételeket) jelzi. Az NMI-kezelőnek a dolga, hogy eldöntse, le kell-e állítani a számítógépet avagy nem.

16 OPTIMALIZÁLÁS MÓDJAI

Kétféle szempont alapján tudjuk a programot optimalizálni:

- **sebességre: a program, vagy annak valamely része a lehető leggyorsabban fusson le**
- **méretre: a program, vagy annak valamely része a lehető legkisebb legyen**

Meg kell jegyeznünk, hogy az Assemblyre a magas szintű nyelvekkel szemben általában igaz a tömörebb-gyorsabb elv, valójában azonban a két tulajdonság nagyjából egymás rovására megy, mintha azok "forditott arányosságban állnának" egymással (nem mindig, erről még lesz egy pár szó). Manapság úgy tűnik, hogy egyikre sem optimalizálnak (legalábbis egyesek), mert az újabb gépek úgyis gyorsabbak lesznek, több memóriával, stb., azaz a szoftverhez igazitják a hardvert, és nem forditva.

16.1 Sebességre optimalizálás

Szerintünk ez az a tulajdonság, amire inkább érdemes egy programot optimalizálni (kivéve, ha valami nyomós okunk van az ellenkezőjére). A sebességre optimalizálás esetében észre kell vennünk, hogy a program futásidejének döntő többségét a ciklusok végrehajtása teszik ki. Épp ezért elég, ha csak ezekkel foglalkozunk. Tovább szűkithetjük azonban a kört, mert a ciklusok között is elég csak azokkal foglalkozni, amelyek biztosan a legtöbbször hajtódnak végre, igy az ő sebességüket kell mocsokmód megnövelni. Erre egy példa: a 3 dimenziós játékok esetén (persze a régebbiekre gondolok, nem a mai 3D-

kártyásokra) a szoftvernek kell megrajzolnia az egész 3D-s látványt, ami poligonokból épül fel. A tér megrajzolásának a lehető leggyorsabbnak kell lennie, különben a játék lassú, élvezhetetlen lesz. Mivel ilyenkor a legtöbbször a poligont megrajzoló eljárás fut, annak azon ciklusait kell a végletekig optimalizálni, amelyek pl. egy sort vagy oszlopot rajzolnak meg a poligonból.

A sebességre optimalizálás alapvetően úgy történik, hogy ismerve a gépi utasitások óraciklusban mért végrehajtási sebességét, az adott programrészt olyan utasitások felhasználásával irjuk meg, hogy az összidő minél kisebb legyen. Ne higgyük azonban, hogy ez ennyire egyszerű: az egyes utasitások tényleges végrehajtási ideje függhet attól, hogy konkrétan milyen operandusokra adjuk ki, vagy hogy pl. mennyi ideig tart az operandus kihozása a memóriából, ha az nincs benne a processzor cache-ében (büntetőóraciklusokat kaphatunk, amikor a proci nem csinál semmit), stb. Ez igaz a programkódra is, jobb ha az befér a proci kód-cache-ébe (ilyenformán méretre is optimalizálnunk kell!). De ez még mind semmi, ha egy mai processzorra akarunk optimalizálni (pl. Pentium-tól felfelé). Ezek ugyanis képesek pl. két utasitást szimultán végrehajtani, de csak erős feltételek mellett. Viszont a programrész akkor fut le a leggyorsabban, ha a lehető legtöbb utasitás hajtódik végre párban, tehát ezen bizonyos feltételeket is figyelembe kell vennünk. Ha van feltételes elágazásunk is egy cikluson belül, akkor a processzor leghatékonyabb kihasználásához nem árt ismernünk annak elágazás-előrejóslásának működési elvét. Ha mindezeket figyelembe vesszük (ami még mindig kevés, mert vannak még más nyalánkságok is ám), akkor pl. egy optimális ciklusra olyan megoldások születnek, amit vagy senki nem ért meg, vagy egyáltalán nem tűnik optimálisnak, vagy ezek lineáris kombinációja.

A sebességre való teljes optimalizálás tehát rendkivül nehéz feladat, igy csak egy-két általános tippet adunk.

1. **Hülyén hangzik, de lehetőleg kerüljük a ciklusokat! Kevés iterációszámú vagy speciális ciklusok esetén léteznek velük ekvivalens és náluk gyorsabb szekvenciális kódok.**
2. **Ne általánositsuk a ciklusokat, hanem lehetőség szerint specializáljuk. Ez alatt azt értjük, hogy ne olyan ciklust irjunk, ami valaminek többféle esetét is kezelni tudja, mert ilyenkor az esetek vizsgálata vagy az abból származó "felesleges" kód is a cikluson belül van, ami lassitja azt. Irjunk inkább egy-egy ciklust külön-külön minden esetre, s azok végrehajtása előtt döntsük el, hogy melyiket is kell majd meghivni. Ezzel az esetek vizsgálatát kivettük a ciklusokból. Pl.:**

```
MOV  CX,1000
@Ujra:
     OR      BL,BL
     JZ      @Oda
     ADD     AX,[SI]
     JMP     @Vege
@Oda:
     ADD     [SI],DX
@Vege:
     ADD     SI,2
     LOOP    @Ujra
```

helyett:

```
      MOV     CX,1000
      OR      BL,BL
      JE      @Ujra2
@Ujra1:
      ADD     AX,[SI]
      ADD     SI,2
      LOOP    @Ujra1
```

```
      JMP     @Vege
@Ujra2:
      ADD     [SI],DX
      ADD     SI,2
      LOOP    @Ujra2
@Vege:
```

Ez láhatóan hosszabb, viszont gyorsabb is.

3. Bontsuk ki a ciklusokat (roll out), azaz ismételjük meg a ciklusmagot, hogy kevesebb idő menjen el pl. a ciklus elejére való visszaugrásokra:

```
@Ujra:
     ADD     AX,[SI]
     ADD     SI,2
     LOOP    @Ujra
```

helyett:

```
SHR  CX,1
     JNC     @Ujra
     ADD     AX,[SI]
     ADD     SI,2
@Ujra:
     ADD     AX,[SI]
     ADD     AX,[SI+2]
     ADD     SI,4
     LOOP    @Ujra
```

Először osztjuk 2-vel a ciklusszámlálót, ha az páratlan volt, akkor egyszer végrehajtjuk a ciklusmagot. A tényleges ciklus viszont már csak feleannyiszor fut le, ráadásul kevesebb összeadást hatjunk végre összesen, viszont ez csak a fejlettebb procikra igaz, mivel 8086-on több óraciklusig tart kiszámolni a cimet az [SI+2] cimzésmódból, mint az [SI]-ből. Ebből is látszik, hogy a sebességre optimalizálás mennyire processzorfüggő.

4. A cikluson belül a lehető legkevesebb legyen a memóriahivatkozás, a változók értékét lehetőleg regiszterekben tároljuk, mivel azok nagyságrendekkel gyorsabban elérhetőek, mint a memória.
5. Ha nem mindent tudunk regiszterbe tenni, mert túl sok változónk van, akkor egy sokak által elitélt módszert, az önmódositó kódot is alkalmazhatjuk:

```
@Ujra:
  ADD        [SI],DX
  ADD        SI,Adando
  LOOP       @Ujra
```

Tegyük fel, hogy az adando értékét nem tudjuk regiszterbe betenni, ekkor:

```
MOV  AX,Adando
     MOV     WORD PTR CS:@Utasitas+2,AX
     JMP     $+2
@Ujra:
     ADD     [SI],DX
@Utasitas:
     ADD     SI,1234h     ;gépi kódja: 81h C6h 34h 12h
     LOOP    @Ujra
```

Azaz az ADD SI,1234h közvetlen operandusát átirtuk a megfelelő értékre a kódszegmensben. A JMP $+2 azért kell, mert a fejlettebb procikon a ciklus kódja már azelőtt bekerül (het) a kód-cache-be, mielőtt átirnánk az operandust, igy nem lenne neki semmi hatása. A JMP viszont törli a cache-t, igy a már átirt utasitás fog bekerülni oda. Ez a módszer csak akkor éri meg, ha a ciklus viszonylag sokszor iterál.

6. A ciklusra nézve konstans értékek kiszámitását vigyük ki a ciklus elé, azaz ne számoljunk ki mindig valamit a cikluson belül, ha minden iterációban ugyanazt az eredményt kapjuk.

Olyan ez, mintha egy szorzótényezőt emelnénk ki egy összegből.

7. Az összetett utasitásokat helyettesithetjük több egyszerűbbel. Idáig pl. LOOP-pal szerveztük a ciklust, viszont gyorsabb nála a DEC CX // JNZ páros, bármily meglepő! Ilyenekre akkor jöhetünk rá, ha böngészünk egy táblázatot az egyes utasitások végrehajtási idejéről.

Lehetőség szerint kerüljük az időigényes aritmetikai műveleteket, mint pl. a szorzás/osztás (a DIV/IDIV utasitások különösen sok óraciklust igényelnek), s helyettesitsük őket egyszerűbbekkel, pl. ha 2 hatványával kell osztani/szorozni.

```
      MOV     BX,10
@Ujra:
      MOV     AX,[SI]
      MUL     BX
      MOV     [SI],AX
      ADD     SI,2
      DEC     CX
      JNZ     @Ujra
```

helyett:

```
@Ujra:
      MOV     AX,[SI]
      SHL     AX,1
      MOV     BX,AX
      SHL     BX,1
      SHL     BX,1
      ADD     AX,BX
      ADD     [SI],AX
      ADD     SI,2
      DEC     DX
      JNZ     @Ujra
```

16.2 Méretre optimalizálás

Láttuk, hogy a sebességre optimalizált részek hossza megnő. A program azon részeit viszont, amelyeket nem szükséges sebességre optimalizálni, azt optimalizálhatjuk méretre. Ezzel kompenzálhatunk, s összességében megtarthatjuk a tömör-gyors tulajdonságpárost. Méretre optimalizálni már sokkal könnyebb, ilyenkor csak az utasitások hosszát kell figyelnünk. Lássunk egy-két dolgot:

1. Az általános regiszterek közül az AX-en végzett elemi műveletek kódja általában egy bájttal rövidebb, mint a többi esetében:

```
MOV     AX,Valami
ADD     AX,1234h
```

rövidebb, mint pl.

```
MOV     BX,Valami
ADD     BX,1234h
```

2. Ha AL/BL/CL/DL közül valamelyiket eggyel kell növelni, és tudjuk, hogy nem lesz túlcsordulás, valamint a jelzők állapotára sem leszünk kiváncsiak, akkor pl.

```
INC     CL
```

helyett egy bájttal rövidebb az

```
INC     CX
```

3. Olyan esetekben, amikor biztosan tudjuk, hogy AL, illetve AX nemnegativ, akkor egy bájttal nullázhatjuk az AH, illetve a DX regisztert:

```
        CBW
```

kinullázza AH-t, és

```
        CWD
```

kinullázza DX-et.

4. Rövidebb a memóriába való mozgatás, ahhoz hozzáadás, stb., ha az operandus egy regiszter, mintha közvetlenül adnánk meg, igy pl.:

```
        ADD     VALTOZO1,3
        ADD     VALTOZO2,3
        ADD     VALTOZO3,3
        ADD     VALTOZO4,3
```

helyett rövidebb

```
        MOV     AX,3
        ADD     VALTOZO1,AX
        ADD     VALTOZO2,AX
        ADD     VALTOZO3,AX
        ADD     VALTOZO4,AX
```

5. Utasitás elrejtése egy másiknak az operandusába. Pl.:

```
        JE      @OK
        STC
        JMP     @Ki
@OK:
        CLC
```

```
@Ki:  RET
```

helyett rövidebb

```
      JE      @OK
      STC
      DB      0B0h
@OK:
      CLC
      RET
```

Itt a B0h bájt a MOV AL,... utasitás kódja. Mivel a CLC egybájtos (F8h), igy az pont ennek az utasitásnak lesz az operandusa. Az STC után tehát egy MOV AL,0F8h utasitás hajtódik végre, ami nem befojásolja a jelzőket. Ez persze csak akkor alkalmazható, ha AL tartalmát el szabad rontanunk.

6. Felesleges ugrások elkerülése.Pl.:

```
      CMP     AL,7
      JE      @Oda
      MOV     BL,5
      JMP     @Ki
@Oda:
      MOV BL,9
@Ki:
```

helyett:

```
      MOV     BL,5
      CMP     AL,7
      JE      @Ki
      MOV     BL,9
@Ki:
```

7. Ha két ágnak megegyezik az eleje, akkor azt "ki lehet emelni" az elágazás elé. Ha csak hasonlitanak, akkor megvizsgálhatjuk, hogy lehet-e őket egyesiteni egy mindkettővel ekvivalens, de rövidebb kóddal.

```
        CMP     SI,DX
        JE      @Ag2
@Ag1:
        MOV     BL,6h
        OR      DX,DX
        JNE     @Oda1
        MOV     BL,13
@Oda1:
        .
        .
        .
@Ag2:
        MOV     BL,6h
        OR      DX,DX
        JNE     @Oda2
        MOV     BL,13
@Oda2:
        .
        .
        .
```

helyett:

```
        MOV     BL,6h
        OR      DX,DX
        JNE     @Oda
        MOV     BL,13
@Oda:
        CMP     SI,DX
        JE      @Ag2
@Ag1:
        .
        .
```

.

@Ag2:

.

.

.

17 ÚJABB ARCHITEKTÚRÁK

Ez a fejezet az eddig ismertetett anyagon jócskán túlmutat, ezért főleg azok olvassák el, akiket érdekel, hogyan is fejlődött a PC-k piaca az 1981-es évi megjelenéstől napjainkig (ebben a pillanatban 1999-et írunk). A 18 év alatt lezajlott összes változást és újdonságot nem lehet egyetlen fejezetben bemutatni, lehet, hogy egy könyv is kevés lenne rá. Itt mégis összegyűjtöttük a megjelent Intel-kompatibilis processzorok fontosabb ismérveit, újításait, sőt rejtett képességeit.

Ennek a világnak az összetettségét is jelzi, hogy a PC-t megálmodó IBM-en (International Business Machines) és annak processzorát gyártó Intel-en kívül számos más cég is gyártott és gyárt még manapság is az Intel termékeivel kompatibilis mikroprocesszorokat: Sony, NEC, Chips and Technologies (C&T), TI (Texas Instruments), NexGen, UMC (United Microelectronics Corporation), IDT Incorporated, Centaur Technology, National Semiconductor, Rise Technology, Cyrix, AMD Incorporated (Advanced Micro Devices). A lista valószínűleg nem teljes, de ennél több gyártóról nem volt információnk. A mostani processzorpiacon az Intel és az AMD verseng a vásárlók kegyeiért, az IBM és a Cyrix pedig főleg hordozható gépekbe és beágyazott rendszerekbe (embedded systems) gyárt olcsó, közepes teljesítményű procikat. Újdonságnak számítanak a Rise Technology alacsony teljesítmény-felvételű processzorai.

A legelső mikroprocesszort az Intel Corporation készítette el 1971-ben, amit 4004-es néven kereszteltek meg. A rákövetkező évben ezt a 8008-as, majd 1974-ben a 8080-as processzorok követték. Ez utóbbi chip két számítógép-család ősének tekinthető: belőle fejlődött ki az 1976-ban megjelent Z80-as, a Zilog Corporation által gyártott processzor (erre alapult sok későbbi mikroszámítógép, mint pl. Enterprise és a Sinclair-termékek), de ennek a mikroprocesszornak a tovább-fejlesztéséből született meg az első PC-k központi egysége, az 1978-ban kiadott Intel 8086-os is, aminek előfutára a szintén 1976-ban gyártott 8085-ös volt. A 8086-os már valódi 16 bites processzor volt, 16 bites regiszterekkel és adatbusszal, valamint 20 bites címbusszal (ez 1 Mbájt memória megcímzésére elég).

Az első IBM XT (talán az eXtended Technology rövidítése) azonban nem ezzel a processzorral, hanem az 1979-ben világra jött Intel 8088-as CPU-val került a piacra. Ez a mikroprocesszor minden belső tulajdonságában megegyezik az előbbivel, viszont a külső adatbusza csak 8 bites volt, ami olcsóbbá tette a számítógépek gyártását. A processzort kezdetben 4.77 MHz-es órajel hajtotta, amit a későbbiekben át lehetett kapcsolni 10 MHz-re is (erre szolgált a legendás Turbo-gomb a gépházon).

Ezek a gépek az 1975-ben alapított, manapság is "közkedvelt" Microsoft cég által készített MS DOS 1.0-ás operációs rendszerrel jelentek meg. A későbbi gépek is mind ennek az op. rendszernek a különböző változatait használták (a legelterjettebbek az MS DOS 3.3, 4.0, 5.0, 6.2, 6.22 verziók voltak). Az 1992-ben megjelent MS Windows 3.1, valamint az 1995-ben kiadott MS Windows 95 (igazából Windows 4.0) azonban a jó "öreg" DOS leváltásának idejét mind közelebb hozzák. Ha lehet hinni a Microsoft híreinek, akkor a 2000-ben megjelenő Windows 2000 operációs rendszer már nem a DOS-

ra fog épülni, rajta csak a Windows 9x/Windows NT rendszerekre írt programok fognak futni.

1982-ben három processzort is megjelentetett az Intel: a 80186 és 80188 nevű chipek csak néhány új utasítással és megváltoztatott külső interfésszel tértek el a korábbi két processzortól. A 80286-os mikroprocesszor megjelenésével azonban beköszöntött az AT-k (Advanced Technology) korszaka, és ez mind a mai napig tart. Ez a proci már nagyon sok újítást tartalmazott: új utasítások, új működési mód, 16 Mbájt megcímezhető memória (azaz a címbusz 24 bit széles volt) stb. Új regiszterként megjelent a gépi állapotszó (Machine Status Word–MSW), mindegyik szegmensregiszter kapott egy rejtett árnyékregisztert (segment descriptor cache), valamint a védett móddal együtt 4 rendszerregiszter is be lett vezetve (GDTR, IDTR, LDTR, TR). Órajele 12.5/16/20 MHz volt.

1985-ben igazi áttörést hozott az Intel 80386-os processzor (fedőnevén P9) piacra dobása, ami az addigi 16 bites architektúrát 32 bitesre váltotta át. Rengeteg újításával az addig megvalósíthatatlannak hitt dolgok is realizálhatóak lettek: erős védelem a többi programmal szemben, 32 bites regiszterek, 4 Gbájt (=4*1024*1024*1024 bájt) megcímezhető memória, virtuális memória kezelése lapozással (kétszintű szegmens-lapcímzés, 4 Kbájtos lapmérettel), hatékonyabb címzési módok, két új adatszegmens-regiszter, kibővített debuggolási lehetőségek (hardveres töréspontok, nyomkövető regiszterek), tesztregiszterek, vezérlőregiszterek és még sorolhatnánk. Hasonlóan a 8086-8088 pároshoz, ennek a processzornak is két változata terjedt el igazán: a 80386SX 16 bites külső adatbuszt kapott, órajele 20/25/33 MHz lehetett, míg a 80386DX 32 bites volt külsőleg is, órajele pedig 33 illetve 40 MHz volt típustól függően.

Az 1989-ben bejelentett Intel 80486-os processzor látszólag nem sok újdonsággal szolgált: a processzorra épített 8 Kbájt méretű gyorsítótár (*cache* memory), párhuzamos végrehajtás lehetősége (parallel processing/execution), egy-két új utasítás. Ezek azonban igen fontos újítások voltak, mivel a később megjelent P5 és P6 architektúrák főbb alapjait lerakták. Ez a processzor volt az első, amely belső órajelként a külső órajel többszörösét használta a gyorsabb működés érdekében. A 25, 33 vagy 40 MHz nagyságú külső jelet a processzor különböző típusai más-más értékkel szorozták fel: a 80486SX (P4S, P23) és 80486DX (P4) típusok 1-es szorzót használtak, a 80486DX2 (P24, P24S, P24D) típusok dupláztak (55, 66 ill. 80 MHz-es változatok), míg a 80486DX4 (P24C, P24CT) számú változatok háromszorozták a külső órajelet (75, 100 ill. 120 MHz-es típusok).

Eddig nem említettük, de már a 8086-os proci megjelenése óta lehetőség volt a rendszerben a központi egység mellett egy segédprocesszor (co-processor) használatára is. Mivel a CPU csak egész típusokat ismer, szükség volt egy olyan hardverre, ami hatékonyan képes számolni *lebegőpontos számokkal* (floating-point number). Ezek olyan, speciálisan ábrázolt racionális számok, ahol a tizedespont helye nincs rögzítve, ellentétben a *fixpontos számokkal* (fixpoint number). A 8086/8088 processzorokhoz a 8087, míg a 80286-os processzorhoz a 80287 típusjelzésű segédprocesszorokat lehetett használni. A 80386-os proci két fajta koprocesszort is képes volt kezelni: 80287-es és 80387-es is működött vele. A 80486 processzor legnagyobb újítása talán az volt, hogy a processzor immár magába foglalta ezt a segédprocesszort is. Mivel azonban ez igen megdrágította az amúgy sem olcsó processzort, ebből a sorozatból is készült "gyengített" változat: a 80486SX processzor nem tartalmazott segédprocesszort, viszont lehetett hozzá venni ilyet 80487SX (P23S, P23N) néven. A segédprocesszor három néven terjedt el: numerikus

segédprocesszor (numeric co-processor), numerikus adatfeldolgozó (Numeric Data Processor–NDP), valamint lebegőpontos egység (Floating-Point Unit–FPU), így bármelyik kifejezéssel találkozunk is, mindegyik ugyanarra az eszközre utal. A numerikus koprocesszor egyébként 8 regisztert tartalmaz, melyek azonban csak veremként kezelhetők (co-processor stack), közvetlenül nem érhetők el. A verem tetejét az ST vagy ST0 kifejezések jelölik, ST1 a verem teteje előtte (alatti) elemet jelzi, és így tovább, egészen ST7-ig. Mindegyik regiszter 80 bites, és tartalmuk bináris egész, BCD egész, illetve 32, 64 vagy 80 bites lebegőpontos szám lehet.

Ekkortájt jelentkezett az igény a mobil számítógépek (úgymint notebook-ok, laptop-ok) megjelenésére. Az Intel ezekbe a számítógépekbe szánta 80386SL és 80486SL Enhanced nevű processzorait, amelyek különféle, az energiatakarékosságot támogató szolgáltatást is nyújtottak. Ilyen volt a processzor automatikus leállítása (Auto Halt Powerdown), a processzor belső órajelének lelassítása (Stop Clock), illetve egy speciális karbantartó üzemmód (System Management Mode–SMM).

1993-ban jelent meg az eredetileg P5 fedőnevű Intel Pentium processzor, de P54C név is ezt a processzort takarja. Mivel az addigi Intel termékeknek sok hasonmása jelent meg más gyártóktól, és azok mindegyike magában hordozta a nevében valamelyik 86-ra végződő típusszámot (tehát 386, 486, 486SX stb.), ez eléggé legyengítette az Intel processzorok esélyét. Egy számot nem lehet márkanévként bejegyeztetni (regisztráltatni), egy szót viszont lehet. Ezért ettől az időponttól kezdve az összes gyártó valamilyen egyedi nevet ad az újdonsült processzorainak. Innen ered a Pentium név is, amely a "penta" szóval utal arra, hogy ez az 586-os sorozat tagja. Ez tehát már jogilag védett bejegyzett márkanév (registered trademark), azaz ha a "Pentium" szót halljuk, akkor az csak az eredeti Intel

Pentium valamelyik változatát jelentheti, más termékek csak a "Pentium kompatibilis" jelzőt használhatják.

A P5 architektúra ismét hozott néhány kellemes újítást: szuperskalár utasítás-végrehajtás (két utasítás órajelenként), az ezt lehetővé tevő két végrehajtó-csővezeték (u és v execution pipeline-ok), 2*8 Kbájt nagyságú cache (writeback és write-through módokban is), elágazás-előrejelzés (branch prediction); a lapok mérete immár nem csak 4 Kbájt, de 4 Mbájt is lehet; 64 bites külső adatbusz, továbbfejlesztett megszakítás-vezérlő (Advanced Programmable Interrupt Controller–APIC), valamint kétprocesszoros rendszerek támogatása (dual processing). Ezenkívül a 80386-oson bevezetett tesztregisztereket eltörölték. A 2*8 Kbájt cache 8 Kbájt utasítás-cache-t és 8 Kbájt adat-cache-t jelent. A Pentium támogatja az SMM üzemmódot is, és ez a további processzorok esetén is így lesz valószínűleg. Ezen kívül a Pentium-tól kezdve minden processzor magában foglalja a numerikus egységet is.

A 80486-os processzorok későbbi változatai már tartalmaztak néhány ú.n. modell-specifikus regisztert (Model-Specific Register–MSR), de igazán csak a Pentium processzorral terjedtek el. Ezek a 64 bites regiszterek elég változatos tartalmúak lehetnek, közülük nagyon hasznos például a Time- Stamp Counter (TSC), amely tartalma a processzor resetelésekor kinullázódik, és minden órajel eggyel növeli értékét. A MSR regiszterek olvasására és írására szolgáló RDMSR és WRMSR, valamint RDTSC utasítások, de az MSR-ek különösen elég gyengén dokumentáltak voltak. Ennek ellenére a legtöbb Pentium-kompatibilis termék is támogatta őket.

A Pentium kategóriájú processzorok immár olyan alaplapon foglaltak helyet, amire a processzor maga nincs gyárilag ráültetve. A processzort egy speciális foglalatba kell beilleszteni, ennek neve Socket 7. A processzor és az alaplapi perifériák működésének meggyorsítására a központi órajelet az

addigi 33 MHz-ről át lehetett kapcsolni 66 vagy 83 MHz-re, sőt néhány alaplapon még más értékre is. A processzor belső órajele az előzőkhöz hasonlóan ennek a külső jelnek a felszorzásával áll elő. A korábbi 80486-os processzorok gyárilag rögzített szorzószámától eltérően azonban az összes új processzor a Pentium-tól kezdve egy kívülről, az alaplapon állítható szorzóértéket használ. A Pentium processzor elég sokfajta változatban került a piacra, így voltak 66, 75, 90, 100, 133 MHz-es órajelet igénylő típusok.

A külsőleg, kézzel állítható szorzók bevezetése magával vonta egy új fogalom megjelenését is. *Túlpörgetésen* (overclocking) azt értjük, amikor olyan szorzót állítunk be, ami a hivatalos órajelnél magasabb ütemű jelet vált ki. Az, hogy egy Pentium-ot mondjuk 100 MHz-esként kínálnak, nem feltétlenül jelenti, hogy a processzor csak ekkora órajelet visel el. Így történhetett meg például, hogy egy 90 MHz-es Pentiumot mondjuk 166 MHz-en működtettek (ez 5*33 , 2.5*66 vagy 2*83 MHz-nek felel meg). A túlpörgetés elvileg károsítja a processzort, mivel a magasabb órajel miatt jobban melegszik a proci, ez viszont megfelelő hűtéssel ellensúlyozható. Így aztán nem lehet azon sem csodálkozni, hogy sokan használják számítógépüket "felhúzva". Ha már a hűtés szóba került, azt sem árt megemlíteni, hogy míg egy 80386-os proci vígan elvolt magában, egy 80486-ost már illett valamivel hűteni (hűtőbordával és/vagy ventilátorral), addig a Pentium procikat és utána megjelent társait szinte kötelező hűteni, ha nem akarjuk, hogy a chip leolvadjon az alaplapról. (Ez nem is lenne oly nehéz, hiszen egy 300 MHz-en működő proci körülbelül 20-25W-nyi teljesítményt fogyaszt, ami hűtés nélkül akár 60-70 °C-os hőmérsékletre is felmelegítené a processzor tokját, és ezt a gépházban uralkodó meleg levegő csak tovább emelné.)

A sovány pénztárcájúak számára dobta piacra az Intel a Pentium Overdrive (P24T) processzort, amit egy 80486-os

tetejére kellett ráhelyezni, ezzel az majdnem elérte egy igazi Pentium sebességét, viszont jóval olcsóbb volt annál.

Ettől a ponttól kezdve két szálon folytatjuk a történetet. Az Intel mellett ugyanis igazán csak az AMD processzorai rúghatnak labdába. Sajnos az IBM és a Cyrix processzorait nem ismerjük közelebbről, és pontosabb információnk sincs róluk, ezért itt nem foglalkozunk velük.

Az AMD is elkészítette saját Pentium-kompatibilis mikroprocesszorát K5 néven.

A következő generációt a P6-os fedőnevű Intel Pentium Pro processzor hozta, melyet 1995-ben jelentettek be. Főbb újdonságai: háromutas szuperskalár végrehajtás (azaz három utasítás végrehajtása óraciklusonként), dinamikus végrehajtás (dynamic execution), ami adatáramlás-analízist (micro-data flow analysis), soronkívüli utasítás-végrehajtást (out-of-order execution), fejlett elágazás-előrejelzést (superior branch prediction) és spekulatív végrehajtást takar (speculative execution); ezenkívül a 2*8 Kbájt *elsőszintű* (Level 1–L1) *cache* mellett 256 Kbájtos *másodszintű* (Level 2–L2) *cache*, 64 bites kapcsolat-orientált külső adatbusz (transaction-oriented external data bus), valamint 64 Gbájt címterületet biztosító 36 bites címbusz. Ez a processzor már a tokozás terén is elindul a változás felé, ugyanis speciális, Socket 8 nevű csatlakozóba illeszkedett csak, amire egy külön e célra való alaplap szolgált.

A multimédiás alkalmazások egyre gyorsabb ütemű terjedése magával hozta azt a természetes igényt, hogy a processzor is támogassa utasításszinten a különféle számolásigényes feladatokat. Ebből a célból jelentette be az Intel az MMX (MultiMedia eXtension) technológiát 1997-ben. A technológia a következő újításokból áll: új adattípusok (kvadraszó, valamint csomagolt bájt, szó és duplaszó), 8 db új

64 bites regiszter (MM0..MM7), illetve 57 db. új utasítás. Az új utasításkészlet adatmozgató, aritmetikai, logikai, shiftelő, összehasonlító és konverziós utasításokat tartalmaz. A 8 db. új regisztert sajnálatos módon a koprocesszor vermére képezték rá, ami a gyakorlat szempontjából annyit jelent, hogy ha módosítjuk mondjuk az MM2 regisztert, akkor az ST2 regiszter tartalma is módosulni fog, és viszont. Ez a tény magával hozta azt a kényelmetlenséget is, hogy ha koprocesszor- és MMX utasításokat akarunk keverni egymással, akkor az MMX utasításokat tartalmazó blokk után szükség van egy speciális utasítás, az EMMS (Empty MMX State) használatára, ami a koprocesszor-verem mindegyik bejegyzését érvénytelennek jelöli be. Bár bejelentésekor nagyon csábítónak tűnt az új technológia, igazán nem váltotta be a hozzá fűzött reményeket. Ez főleg annak a következménye, hogy az MMX csak egész vagy fixpontos számokkal képes dolgozni, lebegőpontos értékekkel nem.

Az MMX technológia nagy újítása igazán az, hogy ez az első olyan utasításkészlet-bővítés, ami az addig megszokott *SISD* elv (Single Instruction, Single Data) helyett a *SIMD* (Single Instruction, Multiple Data) elvet követi. Ennek lényege, hogy egyetlen utasítás egyszerre több adaton (operanduson) végzi el ugyanazt a műveletet. Ha például a csomagolt-bájt (packed byte) adattípust használjuk, ami 8 db független bájtot jelent egy 64 bites MMX regiszterben vagy memóriaterületen, és összeadunk két ilyen típusú operandust, az eredmény szintén egy csomagolt-bájt típusú 64 bites adat lesz. A másik fontos újdonság a szaturációs (telítő) aritmetika (saturation arithmetic). Ha mondjuk a 80h és 90h előjel nélküli bájtokat összeadjuk, akkor az eredmény kétféle lehet: a hagyományos körbeforgó (wrap-around) aritmetikával 10h, míg a telítő aritmetikával 0FFh lesz. Ennek az aritmetikának tehát az a lényege, hogy az eredményt az ábrázolható legnagyobb értékre állítja be, ha egyébként átvitel keletkezne. A szaturáció

mindhárom csomagolt típusra működik, és van előjeles és előjeltelen változata is.

Az MMX technológia bejelentése után rögvest megjelentek az őt támogató processzorok is: Intel Pentium MMX (P55C), AMD K6, Cyrix MediaGX (fedőnevén GXm és GXm2), Cyrix 6x86MX (M2), IDT/Centaur WinChip C6, IBM 6x86MX és még talán mások is. Az olcsóbb átállást könnyítette meg az Intel Pentium MMX Overdrive (P54CTB), amit egy normál Intel Pentium-mal együtt használva, azt képessé tette az MMX kihasználására.

Az AMD K6 története elég különleges. Az AMD-nak volt gyártósora, viszont nem volt elég ötlete. Ezért felvásárolta a NexGen vállalatot, aminek voltak jó ötletei és kész tervei, viszont nem tudott saját maga processzort gyártani. Az AMD K6 így a AMD K5 és a majdnem kész NexGen Nx686 processzorok alapjaiból született hibrid lett.

Az 1998-as év is elég mozgalmas volt. Ekkor jelentek meg az első Intel Pentium II processzorok (fedőnevük Klamath, majd Deschutes). Ez a proci alapvetően a Pentium Pro architektúráját használja, de már MMX képességekkel is fel van vértezve. Az Intel ezzel a processzorral kezdődően kiszállt a Socket 7-es processzorok piacáról, és valószínűleg a Pentium Pro-t sem fogja támogatni. A Pentium II újdonságai: 2*16 Kbájt L1 cache, 256 Kbájt, 512 Kbájt, 1 Mbájt vagy 2 Mbájt nagyságú L2 cache, valamint számos új üzemmód az alacsonyabb fogyasztás érdekében (AutoHALT, Stop-Grant, Sleep, Deep Sleep). A processzor teljesen új tokozást kapott, és két változatban kapható: a Slot 1 nevű tokozású processzor Pentium II néven vásárolható meg, ebben 256 vagy 512 Kbájtos L2 cache van, míg a Slot 2 tokozású Pentium II Xeon 1 Mbájt vagy 2 Mbájt méretű L2 cache-sel kerül forgalomba, és főleg nagy teljesítményű szerverekbe szánják. A sima Pentium II-esek L2 cache-je csak a processzor órajelének felével üzemel, a Xeon

viszont teljes gőzzel járatja mindkét cache-t. Mindkét processzorhoz így új alaplapra lesz szükség, mivel a Slot 1 és Slot 2 nem kompatibilis a Socket 7-tel. Az új csatolóhoz új buszsebesség dukál – gondolhatták az Intel-nél, mivel a Pentium II-ket befogadó alaplapok rendszerbusza 100 MHz-es, de ezt hamarosan fel fogják emelni 133 MHz-re is. A processzor belső órajele 266, 300, 333, 350, 400 és 450 MHz lehet típustól függően, és felpörgetni is csak kb. 500 MHz-ig lehet technikai okokból.

A Pentium II ára igen borsos volt megjelenésekor (és még ennek a jegyzetnek az írásakor is elég magas). Voltak, akiknek a Pentium II sebességére volt szükségük, de nem engedhették meg azt maguknak. Nekik szánta az Intel a Celeron néven megjelentetett (Covington fedőnevű), ugyancsak P6 magra épülő processzorát. Az eredetileg 266 vagy 300 MHz-es Celeron nem tartalmazott L2 cache-t. Hamarosan lehetett kapni azonban a Mendocino fedőnevű Celeron-okat, melyek mindegyikében volt 128 Kbájt nagyságú L2 cache is. A Mendocino-k közé tartozik a Celeron A jelzésű processzor, valamint az összes olyan Celeron, aminek órajele legalább 333 Mhz. Mendocino-kat a jelenlegi állás szerint 300, 333, 366, 400, 433 és 466 MHz-es változatban lehet kapni. Csatolót tekintve sem egyszerű a helyzet, mivel az eredetileg Slot 1-es Celeron-ok mellett már lehet kapni Socket 370-es foglalatba illeszkedő, PPGA (Plastic Pin Grid Array) tokozású Celeron-okat is. A Socket 370 nevű foglalat a Socket 7-hez hasonló, négyzet alakú, 370 tűs processzorcsatoló, és létezik olyan átalakító, amivel a Slot 1-es alaplapokon is használhatunk Socket 370-es processzort.

A L2 cache immár nem a processzormagban helyezkedik el, hanem attól elkülönítve, de ugyanabban a tokban (ez a Slot 1 tokozás lényege). Ezzel a cache-sel a processzor egy speciális buszon keresztül kommunikál, ennek neve *backside bus* (BSB), míg az alaplap központi buszát *frontside bus*-nak (FSB) hívják.

Hasonlóan a processzortokban elhelyezkedő L2 cache neve backside cache, míg az alaplapon levőt frontside L3 cache-nek nevezik. Ezek alapján mondhatjuk, hogy a Pentium II félsebességű (half clock speed, röviden half-speed) backside bus-t és L2 cache-t tartalmaz, míg a Xeon teljes sebességűt (full clock speed, röviden full-speed). A Celeron processzorok (a Mendocino-t beleértve) teljes sebességű L2 cache-t tartalmaznak.

A Pentium II-vel majdnem egyidőben, szintén 1998-ban jelent meg az AMD K6-os sorozatának következő tagja, a K6-2. A processzort az AMD egyértelműen visszavágásnak szánta (természetesen az Intel-lel szemben), és ennek érdekében sok érdekességet rakott ebbe a chipbe: 10 párhuzamos végrehajtási egység, 2 szintű elágazás-előrejelzés, spekulatív végrehajtás, soronkívüli végrehajtás, RISC86 szuperskalár architektúra (6 RISC utasítás óraciklusonként), 32 Kbájt L1 utasítás-cache, 20 Kbájt elődekódoló cache (predecode cache), 32 Kbájt L1 adat-cache, SMM támogatása, néhány üzemmód a fogyasztás csökkentésére (Halt, Stop Grant, Stop Clock állapotok), 7 db. MSR. A 321 tűs CPGA (Ceramic Pin Grid Array) tokozású processzor Socket 7 vagy Super7 foglalatba illeszkedik, típustól függően. A Super7 az AMD és üzleti partnerei által kifejlesztett, 100 MHz buszsebességet támogató, a Socket 7-tel kompatibilis foglalat márkaneve. Ez a processzor is elég sokféle változatban kapható órajel igényét tekintve: 266, 300, 333, 350, 366, 380, 400, 450 és 475 MHz-es darabok léteznek. A legalább 400 MHz-es típusok fedőneve egyébként Chomper.

A processzor az Intel MMX technológiáját is támogatja, azokat nagyon gyorsan (általában 1 óraciklus alatt) képes végrehajtani, akár párhuzamosan is. Ez nem nagy újdonság, hiszen már a K6 is nyújtott MMX-támogatást. Az MMX bevezetésére az AMD is lépett azonban, ennek eredménye lett a 3DNow! technológia. Ez a következőket foglalja magában: 8 db

64 bites regiszter, amik az MMX regiszterekre képződnek le (azok pedig a koprocesszor vermére), nevük MM0..MM7; új pakolt lebegőpontos típus, ami két 32 bites, egyszeres pontosságú lebegőpontos számot tárol egy 64 bites 3DNow!/MMX regiszterben vagy memóriaterületen; valamint 21 új utasítás. Az utasítások az aritmetikai, akkumuláló-átlagoló, szélsőérték-meghatározó, konverziós, összehasonlító és teljesítmény-növelő kategóriákat ölelik át. Az aritmetikai utasítások között a szokásos additív utasítások mellet találunk szorzást kerekítéssel és anélkül, reciprok-képzést és reciprok négyzetgyököt meghatározó utasításokat. A teljesítmény-növelő csoportba 2 utasítás tartozik: az EMMS MMX utasítást felváltó, annál gyorsabb FEMMS (Faster Entry/Exit of the MMX or floating- point State), valamint a PREFETCH és PREFETCHW mnemonikok, melyek a megadott memóriaoperandus címétől legalább 32 bájnyi területet beolvasnak az L1 adat-cache-be. A 3DNow! utasítások egy része vektoron (azaz pakolt lebegőpontos típuson), míg mások skaláron (tehát egyetlen lebegőpontos számon) tudnak dolgozni. A vektor-utasítások a SIMD elvet követik, de az MMX-től eltérően nem egész, hanem egyszeres pontosságú lebegőpontos racionális számokkal képesek számolni. Ez az igazi ereje a 3DNow! technológiának az MMX-szel szemben, ennek hatására támogatja egyre több alkalmazás használatát. A visszavágás, úgy tűnik, sikerült.

Idén, 1999-ben mutatta be az Intel legújabb erőgépét, a Pentium III-at (fedőneve Katmai). A szintén P6-os architektúrára épülő processzor egy-két jellemzője: 2*32 Kbájt L1 cache, 512 Kbájt L2 cache (lesz ez még több is), 8 új 128 bites regiszter, 70 új utasítás. Reménytelenül sok, pontosan 79 db. modell-specifikus regisztert (MSR) tartalmaz. Ezt a processzort már ellátták egy 96 bites gyári azonosító sorszámmal is, ami processzoronként egyedi, és valamilyen módon ki is olvasható. Ez a bejelentés nagy port kavart, ezért az

Intel "bekapcsolhatóvá" tette az azonosítót. A processzor 100 MHz-es (frontside) buszfrekvenciát kíván, Slot 1 csatolóba dugható, de lesz Slot 2-es Xeon változata is (fedőnevén Tanner), ami legalább 512 Kbájt L2 cache-t tartalmaz majd. Csak a Xeon-ban lesz teljes sebességű L2 cache, a Pentium III továbbra is a magsebesség felével üzemelteti ezt a cache-t. Egyelőre 450, 500 és 550 MHz-es típusai vannak. Az utasításkészlet-bővítést eredetileg MMX2-nek hívták, majd átkeresztelték KNI-re (Katmai New Instructions), végső neve pedig SSE lett (Streaming SIMD Extensions). Az elnevezés sokat sejtet: a "streaming" szó utal arra, hogy immár adatfolyamok párhuzamos feldolgozása válik lehetővé, méghozzá a SIMD elvet követve. Ezt támogatandó 8 új 128 bites regisztert kapott a processzor, nevük XMM0..XMM7, és az MMX regiszterekkel ellentétben nem a koprocesszor vermén foglalnak helyet. Szintén bevezetésre került egy új státuszregiszter, az MXCSR (neve hivatalosan: SIMD Floating-point Control/Status Register), ennek mérete 32 bit. Tartalmát az STMXCSR és FXSAVE utasításokkal tárolhatjuk memóriában, míg megváltoztatására az LDMXCSR és FXRSTOR mnemonikok szolgálnak. A regiszterekkel együtt egy új adattípus is megjelent, ez a SIMD lebegőpontos pakolt típus, ami a 128 bites regiszterben vagy memóriaterületen 4 db. 32 bites, egyszeres pontosságú lebegőpontos számot tárol. A 70 db. új utasítás számos kategóriába sorolható: vannak adatmozgató, aritmetikai, összehasonlító, konverziós, logikai, keverő (shuffle), állapot-kezelő (state management), cache-selést vezérlő (cacheability control), továbbá egészekkel operáló utasítások (MMX-bővítés).

Az Intel-t egy héttel megelőzve jelentette be az AMD legújabb processzorát, az AMD K6-III-at, fedőnevén a Sharptooth-t. Néhány főbb újítása: 100 MHz-es frontside bus, teljes sebességű, 256 Kbájtos backside L2 cache, frontside L3

cache támogatása (TriLevel Cache Design), 11 db. MSR. A processzor továbbra is 321 tűs CPGA tokozást használ, és Super7-es foglalatba illeszthető. L1 cache-ének mérete a K6-2-vel megegyező, azaz 32 Kbájt utasítás-cache, 20 Kbájt elődekódoló cache, és 32 Kbájt adat-cache. Támogatja az SMM-et, az MMX és 3DNow! technológiákat. A K6-III egyelőre 400, 450 és 500 MHz-es változatokban kapható.

Most pedig tekintsünk kicsit előre a (közeli) jövőbe!

Az Intel Katmai-sorozatának következő darabjai a Coppermine és Cascades processzorok lesznek. Az első Slot 1-es, a második pedig Slot 2-es (Xeon) processzor lesz, és mindkettőben legalább 1 Mbájt teljes sebességű L2 cache lesz. Ezek a processzorok már a 133 MHz-es frontside bus-t lehetővé tevő 0.18 mikronos csíkszélességgel fognak készülni, az eddigi 0.25 mikronnal szemben, ami 100 MHz buszsebességet és max. 500 MHz körüli magsebességet biztosított. Órajelük legalább 600 MHz lesz. Őket követi majd a P7-es architektúra két képviselője, a legalább 700 MHz-es Willamette, és a 800 MHz fölötti magórajelet használó Foster. Ez utóbbi már 200 MHz-es frontside bus-t fog támogatni, és állítólag egy új csatolót, a Slot M-et fogja használni. A nagy durranás azonban a Merced fedőnevű, teljesen 64 bites processzor lesz, ami már az IA-64 (Intel Architecture 64-bit) architektúrára fog épülni.

Az AMD a K7 architektúra fejlesztésével foglalatoskodik. Az Athlon nevű processzort 1999 júliusában jelentették be hivatalosan. (A processzor a jegyzet írásakor még nem kapható, így ezek csak előzetes információk.) A név kicsit félrevezető, mivel az Athlon az Intel processzorokhoz hasonlóan a processzormagra vonatkozik, és az ezt használó processzorok neve eltérő lehet majd. Ezek a processzorok kezdetben a

hagyományos 0.25 mikronos, a későbbiekben pedig 0.18 mikronos csíkszélességgel készülnek majd, végül pedig az alumínium vezetőt felváltja a réz. A rézalapú technológia lehetővé fogja tenni az 1 GHz fölötti processzor-órajelet. Addig is a 22 millió tranzisztort tartalmazó, 575 lábú processzor magja 500-600 MHz környékén fog ketyegni. Az FSB sebessége 133, 200, 266, majd később 400 MHz lesz. Ha már az FSB-t említettük, fontos, hogy az új AMD processzor és későbbi társai nem az Intel P6-os buszrendszerét használják, hanem a Compaq/Digital 21264-es, másnéven EV6-os processzorának buszát. Ezek a processzorok már nem támogatják a Socket 7/Super7 platformot. Háromféle csatoló képzelhető el: az asztali gépekbe szánják a Socket A-t, a munkaállomásokba a Slot A nevű csatolót, és végül a szerverekbe és a többprocesszoros rendszerekbe a Slot B-t. A Socket A-ról nem sokat tudni, csak annyi bizonyos, hogy a Socket kialakítás olcsóbb a Slot-nál. Az Athlon 2*64 Kbájt L1 cache-t fog tartalmazni, az L2 cache pedig 512 Kbájt (Socket A és Slot A), ill. 512 Kbájt-8 Mbájt (Slot B) lesz. Az L2 cache sebessége a processzormag sebességének 1/5-e, 1/3-a vagy fele lesz, a felső határ pedig a 2/3-szoros sebesség. Elképzelhető, hogy az Intel SSE (MM2/KNI) technológiáját ez a processzor át fogja venni. Az azonban biztos, hogy az eddig jól bevált 3DNow! és MMX technológiákat támogatni fogja.

Távolabbi tervek között szerepel még a K8-as projekt, ami az IA-64 nyomdokait fogja követni.

Megemlítjük, hogy a processzorok nevét rövid alakban is szokás írni. A 80186-os és 80188-as processzoroktól kezdődően a számmal jelölt processzorok nevéből elhagyhatjuk az elöl álló 80-as értéket, és így az adott processzort csak az utolsó 3 számjeggyel azonosítjuk. Ezért pl. a 386 szimbólum a 80386-os,

a 186 a 80186-os processzort jelöli. Szintén szokásos, hogy a rövidebb leírás céljából "Intel 80486" helyett "i486"-ot írunk.

17.1 Dokumentálatlan lehetőségek

Szinte mindegyik megjelent processzor tartalmaz olyan funkciókat, utasításokat stb., amiket a gyártó hivatalosan nem ismer(t) el, azaz nem hozta őket nyilvánosságra. Ezeket közös néven *dokumentálatlan lehetőségeknek* (undocumented features) nevezzük. Hogy derültek ki ezek mégis? Némelyikre elvetemült gépbuherálók jöttek rá kísérletezés útján, míg másokról a gyártótól szivárogtak ki hírek.

Ezek a lehetőségek alapvetően két táborba tartoznak: némelyek minden Intel és azzal kompatibilis processzoron léteznek (vagy legalábbis illik létezniük), míg vannak, amik egy bizonyos gyártó adott termékére vonatkoznak csak. Itt most kizárólag olyan rejtett utasításokkal foglalkozunk, ami az Intel processzorokon biztosan megtalálható, és nagy valószínűséggel a többi gyártó is átvette őket.

Fontos még tudni, hogy a kérdéses utasítás (vagy annak egy bizonyos kódolási formája) milyen típusú processzorokon érhető el. Ez alapján is lehet osztályozni az utasításokat:

- **Vannak, amik egy adott típusú processzortól kezdve minden újabb típuson használhatók. Jelölésük: a processzor neve mellett egy plusz (+) jel szerepel. Így pl. a 186+ azt jelenti, hogy a 80186/80188-as processzorokon és az annál újabb CPU-kon (mint a 80286, 80386DX, Pentium stb.) létezik az adott utasítás.**
- **Némelyik utasítás használata egy konkrét processzorcsalád tagjaira korlátozódik. Ebben az esetben annak a processzornak a nevét írjuk ki, amelyen az adott utasítás elérhető.**

- **Ritkán az is megtörténik, hogy az a bizonyos utasítás csak egy processzorcsalád néhány, bizonyos feltételeket teljesítő tagján található meg. Így például van olyan utasítás, ami csak a 80386-os processzorok korai változatán létezik.**

A fenti pontban a "korai" szó nem túl pontos meghatározást ad, de rögtön pontosítjuk. Egy processzorcsalád megjelentetésekor gyakran megtörténik, hogy az adott processzor már kapható az üzletekben, mikor a gyártó valami apró hibát vesz észre a termékben, és ezért vagy egyéb okból valamit megváltoztat abban. Az így módosított processzor ezután kikerül a piacra. Ezt nevezzük *revíziónak* (revision), vagy magyarul *felülvizsgálatnak*. Mindegyik processzor tartalmaz egy gyárilag beégetett azonosítószámot (processor identification number). Ez három részből áll: *processzorcsalád* (family), *modell* (model) és az ú.n. *stepping*. Ez utóbbit nem igazán lehet magyarra lefordítani, mindenesetre valami olyan jelölést takar, ami a revízió "szintjét" mutatja (tehát nevezhetnénk akár revízió-azonosítónak is). A processzorcsalád lehet például 486-os, P5 (Pentium), P6 (Pentium Pro, Pentium II stb.). A modell a családon belüli tagokat különíti el, így pl. a 486SX, 486DX, 486DX2 tagok mind különböző modellek. A stepping egy adott család adott modelljén belül osztályozza a processzorokat a revízió szerint. Például a 386-os család SX modelljeinek léteztek A0, B, C és D1 jelzésű darabjai. Hogy a különböző stepping-értékkel rendelkező processzorokban milyen változások történtek, azt csak a gyártó tudja, de ezeket általában nem árulja el. Sőt, a legtöbb vásárló nem is tudja, pontosan milyen processzort vásárol, mármint a revízió szempontjából.

A processzor azonosítószámát nem mindig könnyű megszerezni: Pentium-tól "felfelé" a CPUID utasítás szépen visszaadja ezt, régebbi processzorok esetén azonban nem ennyire rózsás a helyzet. Ha nincs CPUID, akkor vagy a BIOS egy kevésbé ismert szolgáltatását használjuk fel (INT 15h,

AX=0C910h), vagy a processzort szoftverből újraindítva, a reset után DX/EDX regiszterben megtalálhatjuk a keresett adatokat. Ez utóbbi, mondani sem kell, egyáltalán nem könnyű módszer, viszont létezik rá szabadon terjeszthető Assembly forráskód és leírás is.

Az alábbiakban a központi egység és a numerikus koprocesszor által ismert rejtett utasításokat ismertetjük. Mindegyik utasításnál szerepel annak mnemonikja, operandusainak típusa, kódolása (azaz gépi kódú formája), elérhetősége (tehát mely processzorokon és milyen körülmények között használható), valamint működésének, hatásának leírása. Mindegyik utasítást a következő séma szerint ismertetjük:

- **MNEMONIK operandusok {elérhetőség}**
  [gépi kódú alak(ok)]
  (mnemonik elnevezésének eredete angolul)
  Az utasítás hatásának, rejtett voltának stb. leírása.

A következő jelöléseket alkalmaztuk:

- **(no op) – nincs operandus**
- **src – forrás (source)**
- **dest – cél (destination)**
- **reg – általános célú regiszter (general purpose register)**
- **?? – tetszőleges megfelelő operandus**
- **imm8, imm16 – 8 illetve 16 bites közvetlen érték (immediate data)**
- **reg/mem8, reg/mem16 – memóriahivatkozás vagy általános célú regiszter, 8 illetve 16 bites méretben**
- **8086 – Intel 8086 vagy 8088 vagy ezekkel kompatibilis proci**
- **186 – 80186 vagy 80188**
- **187 – 80187 és kompatibilis koprocesszor**

- **286 – 80286**
- **386 – 80386**
- **486 – 80486**
- **Pent – Pentium (P5 architektúra)**
- **PROC+ – az adott PROC processzoron és attól kezdve minden későbbin létezik az utasítás**
- **csak PROC – csak az adott PROC család tagjain érvényes az utasítás**
- **csak PROC STEP – csak a PROC processzorcsalád valamely modelljeinek STEP stepping-ű tagjain érhető el az utasítás (pre-B a B revízió előtti tagokat jelöli, így pl. A-t, A0-t, A1-t stb., ha ilyenek voltak)**
- ***r – a címzési mód/operandus info bájtot jelöli (addressing mode/operand info byte); leírását lásd az "Utasítások kódolása" című fejezetben**
- ***r+N – a címzési mód/operandus info bájt középső, 'Reg' mezője (3..5 bitek) a megadott N értéket tartalmazza (0≤N≤7)**

A törtvonal (/) mindenhol vagylagosságot jelez (kizáró vagy). A gépi kódú alak leírásában az egyes összetevőket vesszővel választottuk el. A címzési mód bájtot esetlegesen követheti egy S-I-B bájt valamint egy abszolút offszet/eltolási érték (1, 2 vagy 4 bájt), ez a használt címzési módtól függ. Ezeket a plusz bájtokat nem tüntettük fel, de ha valamelyik előfordul, akkor az utasítás közvetlen operandusa (ha van ilyen) mindenképpen a címzési mód bájt(ok) és az offszet/eltolás után következik.

- **AAD (no op)/imm8 {8086+}**
  [0D5h, 0Ah/imm8]
  (ASCII Adjust before Division)
  Ez az utasítás hivatalosan is létezik, korábban is szóltunk már róla. Működése: az AX-ben levő pakolatlan BCD számot átalakítja binárissá, az eredményt AL-be teszi, AH-t pedig kinullázza.

Eközben PF, ZF és SF módosulhat, míg CF, AF és OF meghatározatlanok. A pakolatlan BCD olyan típus, ahol egy szó alsó és felső bájtja egyetlen decimális számjegyet tartalmaz (ez ASCII 00h..09h közötti értékeket jelent). Így pl. a 0104h a decimális 14-et jelöli. Hivatalosan ez az utasítás operandus nélküli, és csak BCD (azaz tízes számrendszerbeli) számokat kezel. Ezzel ellentétben az utasítás kódolása tartalmazza azt a bizonyos 10-es értéket (0AH), amit nyugodt szívvel átírhatunk bármilyen más értékre, és az ott megadott érték fogja mutatni a használt számrendszer alapját. Az utasítás ezen formája már az "ősidőktől kezdve" elérhető, és valószínűleg az is marad a jövőben is.

- **AAM (no op)/imm8 8086+}**

  [0D4h, 0Ah/imm8]

  (ASCII Adjust after Multiplication)

  Az AAD-hez hasonlóan már erről az utasításról is szóltunk korábban. Működése: AL tartalmát maradékosan elosztja 10-zel, a hányados AH-ba, a maradék AL-be kerül (azaz az AL-ben levő bináris számot pakolatlan BCD-számmá alakítja, és azt az AX-ben tárolja el). A flag-eket az AAD-hoz hasonlóan kezeli. Hasonló cselt alkalmazhatunk, mint az AAD-nél, mivel a 10-es konstanst (0Ah) itt is eltárolják az utasítás kódolásakor, amit mi szépen átírhatunk bármire. Elérhetősége is megegyezik az AAD-éval.

- **CMPXCHG dest, src reg {csak 486 pre-B step}**

  [0Fh, 0A6/A7h, *r]

  (CoMPare and eXCHanGe)

Ez az utasítás a 80486-os processzoron jelent meg, és szemaforok kezelését könnyíti meg. Működése: az akkumulátort (AL-t, AX-et vagy EAX-et) összehasonlítja a céllal (dest) a CMP utasításnak megfelelően. Ha ZF=1, akkor a forrás regiszter tartalmát betölti a célba, különben pedig a célt betölti az akkumulátorba. Ez eddig rendben is van, az utasítást is elismeri hivatalosan az Intel. Viszont a kódolással már nem minden okés. A nagyon korai, B stepping előtti 80486-osok a fenti kódolást használták, de ez ütközött a 80386-osokon korábban létezett, később viszont eltörölt IBTS és XBTS utasítások kódjával. Ezért, a visszafelé kompatibilitás érdekében az Intel megváltoztatta a CMPXCHG utasítás kódját a 0Fh, 0B0/0B1h, *r formára a B stepping-től kezdve, és a későbbi processzorok is mind ez utóbbi formát használják. Ez tehát inkább processzorhiba, mint rejtett funkció. Zavaró, hogy néhány assembler és debugger még a régi kódolást használja, így pl. a 3.1-es verziójú Turbo Debugger is hibásan fordítja le ezt a mnemonikot.

- **CPUID (no op) {486+}**
  [0Fh, 0A2h]

  (CPU IDentification)

  Ez az utasítás a processzorról szolgál különféle információkkal. Hívása előtt EAX-ban a kívánt funkció számát kell közölni. Ha EAX=00000000h, akkor az utasítás lefutása után EAX-ben kapjuk meg azt a legnagyobb értéket, amit funkciószám-ként megadhatunk, ECX:EDX:EBX pedig a gyártó nevét azonosító szöveget tartalmazza. Ez Intel esetén "GenuineIntel", az AMD processzorainál "AuthenticAMD", míg Cyrix

prociknál "CyrixInstead". Ha EAX=00000001h híváskor, akkor visszatérés után EAX tartalmazza a processzor családját, modelljét és stepping-számát, EBX és ECX meghatározatlan, míg EDX a processzor speciális tulajdonságainak jelenlétét vagy hiányát jelzi. Az utasítás hivatalosan a Pentium-on jelent meg, de néhány későbbi gyártású 80486-os is ismerte már. Ha az EFlags regiszter 21-es bitjét (ID) tudjuk módosítani, akkor a processzorunkon van CPUID utasítás.

- **FCOS (no op) {187+}**
  [0D9h, 0FFh]

  (compute COSine)

  Ez a koprocesszor-utasítás az ST-ben radiánban megadott szög koszinuszát rakja vissza ST-be. Az utasítás hivatalosan a 80387-es koprocesszoron jelent meg, de már ismerték az Intel 80187, Intel 80287xl koprocesszorok, valamint az Integrated Information Technology Incorporated cég IIT 2c87 jelű chipje is.

- **FSIN (no op) {187+}**
  [0D9h, 0FEh]

  (compute SINe)

  Az utasítás az FCOS-hoz hasonlóan működik, tehát ST tartalmát cseréli ki annak szinuszára. Elérhetősége is az előző utasításéval egyezik meg.

- **FSINCOS (no op) {187+}**
  [0D9h, 0FBh]

  (compute SINe and COSine)

  Az utasítás az ST-ben radiánban megadott szög szinuszát ST-be rakja, majd a verembe a koszinusz értékét is berakja (PUSH), így ST-ben a koszinusz, ST(1)-ben pedig a szinusz érték lesz. Elérhetősége az előző két utasításéhoz hasonló,

viszont az IIT 2c87 koprocesszor nem ismeri ezt az utasítást.

- **IBTS base, EAX/AX, CL, src reg {csak 386 A-B0 step}**
  [0Fh, 0A7h, *r]

  (Insert BiT String)

  Az utasítás nem igazán jól dokumentált, így működése nem biztos, hogy megfelel az itt leírtaknak. Tehát: a forrás regiszter (src reg) alsó CL db. bitjét átmásolja a bázis (base) című terület EAX/AX sorszámú bitjétől kezdődő bitekre, a környező bitek sértetlenek maradnak. Az utasítás története enyhén szólva is furcsa: a 80386-as processzorok nagyon korai változatán (A és B stepping-érték) jelent meg, de az Intel valamilyen megfontolásból a B0 stepping utáni változatoktól kezdve megszüntette. Így az a helyzet állt elő, hogy egyazon processzorcsalád különböző tagjai másképp viselkedtek, ha találkoztak a 0Fh, 0A7h bájtsorozattal: a korai változatok értelmezték és végrehajtották, míg a későbbi típusok 6-os kivétellel (Undefined/Invalid Opcode–#UD) reagáltak. A 80486-os processzoron szintén keveredést idézett elő ez az utasítás, ezt a CMPXCHG utasításnál tárgyaltuk. A leírások szerint hasznosnak tűnne ez az utasítás, csakúgy, mint társa, az XBTS, de sajnálatos módon mindkettőt megszűntette az Intel, így valószínűleg a többi gyártó sem támogatja őket.

- **ICEBP/SMI (no op) {386+}**
  [0F1h]

  (In-Circuit Emulator BreakPoint/System Management Interrupt)

  Ez az utasítás már igazán dokumentálatlan, annyira, hogy ahány leírás van róla, annyiféle működést

írnak. Két mnemonikja van, de egyiket sem ismerik sem az assemblerek, sem a debuggerek. Az utasítás alapvetően két célt szolgál: ha teljesül egy bizonyos feltétel, akkor a processzor az SMM (System Management Mode) üzemmódba vált át, különben pedig egy INT 01h hívást hajt végre. A feltétel az, hogy a DR7 regiszter 12-es bitje 1 legyen. Az Intel dokumentációi szerint SMM módba csak külső hardverjel hatására kerülhet a processzor, szóval ez elég furcsává teszi a dolgot. Az utasítás legfőbb haszna, hogy a 0CCh kódú INT 3 utasításhoz hasonlóan ez az egyetlen bájt egy INT 01h hívást tesz lehetővé. Egyetlen hátránya, hogy csak CPL=0 szinten adható ki, tehát vagy valós módban, vagy védett módban az operációs rendszer szintjén. Ez a korlátozás nem tudni, mi miatt van, SMM-módba ugyanis bármely üzemmódból átkerülhet a processzor. Az SMM üzemmód már néhány 80386-os és 80486-os procin megtalálható, hivatalosan pedig a P5 (Pentium) családdal vezették be.

- **LOADALL (no op) {csak 286, 386}**
  [0Fh, 04h/05h/07h]

  (LOAD ALL/every register)

  Ha lehet mondani, ez az utasítás még rejtélyesebb, mint az ICEBP/SMI. Az operandus nélküli mnemonikot a 80286-os processzorokon a 0Fh, 05h bájtokkal kódolták (egyes források szerint a 0Fh, 04h is ezt az utasítást jelöli), a 80386-os procik viszont a 0Fh, 07h formát ismerték. Működése röviden: a memória adott címén található táblázatból a processzor (majdnem) összes regiszterének tartalmát betölti, majd nem tudni, mit csinál, de a logikus az, hogy az új CS:IP

helyen folytatja a végrehajtást. A táblázat címe a 80286-os esetén rögzített, a 00000800h fizikai (vagy lineáris?) címen keresi az értékeket, míg a 80386-osoknál ES:EDI adja a táblázat címét. Kicsit különös, ha jobban belegondolunk: védett módban nem szegmensregiszterek, hanem szelektorok vannak, így nem tudni, pontosan mit is csinál ez az utasítás (főleg a 80286-os változat). A betöltött regiszterek közé értendő az összes általános célú regiszter, a szegmens-(szelektor) regiszterek, a Flags regiszter, IP, az MSW (Machine Status Word), a rendszerregiszterek, úgymint LDTR, GDTR, IDTR, TR, sőt még az árnyékregiszterek (descriptor cache) is. A 80386-oson ez kiegészül a regiszterek 32 bites alakjának betöltésével, továbbá az FS, GS, CR0, DR6 és DR7 regiszterek is módosulnak. Az utasítás csak CPL=0 esetben hajtható végre, különben 13-as kivétel (General Protection fault–#GP) keletkezik. Néhány gyártó (pl. az IBM is) az Intel-től eltérő módon valósított meg az utasítást, ami mondjuk abban áll, hogy a táblázatok felépítése kicsit más. Az utasítás működését ismerve teljesen érthetetlen, miért nem lehetett szabványosítani az utasítás használatát, illetve, ami fontosabb, miért kellett eltörölni az utasítást a 80486-osoktól kezdve?

- **MOV CS,?? {csak 8086}**
  [8Eh, *r+1]

  (MOVe data)

  Korábban már említettük a MOV utasítás első előfordulásánál, hogy célként nem szerepelhet a CS regiszter. Ez igazából csak a 80186/80188 processzoroktól van így, a 8086/8088-at kivéve

ugyanis minden processzor 6-os kivétellel válaszol erre a kódra. A 8086/8088 viszont értelmezni tudja ezt a formát is, és így egy olyan feltétel nélküli vezérlésátadást valósít meg, ahol csak CS változik. Lásd még az alábbi POP CS utasítást is.

- **POP CS {csak 8086}**
  [0Fh]

  (POP data from stack)

  A POP tárgyalásánál szóltunk róla, hogy a CS regisztert operandusként nem adhatjuk meg. A helyzet azonban egy kicsit összetettebb. A 8086/8088 gond nélkül értelmezi a 0Fh bájtot, és a MOV CS,?? -hez hasonlóan feltétel nélküli vezérlésátadást hajt végre. A 80186/80188 már nem ezt teszi, helyette 6-os kivételt vált ki. A 80286-os processzoron sok új utasítás lett bevezetve, ezek lekódolásához a 0Fh bájtot egyfajta prefixként használja a processzor. Így a 0Fh bájt önmagában semmit sem jelent a 80286-os és annál későbbi processzorok számára, viszont az utána álló bájttal együtt valamilyen új utasítást képezhet. Ez az egyik mód a 8086/8088 processzorok detektálására. (A másik módszer, hogy például CL-be 1Fh-nál nagyobb értéket rakunk, és így hajtunk végre egy shiftelést. Magyarázatát lásd a 80186-os processzor újdonságait ismertető fejezetben.)

- **RDMSR (no op) {386+}**
  [0Fh, 32h]

  (ReaD Model-Specific Register)

  Ez az utasítás az ECX-ben megadott sorszámú MSR tartalmát olvassa ki, és adja vissza EDX:EAX-ben. Ha CPL≠0, vagy a megadott MSR nem létezik, 13-as kivétel (#GP) keletkezik. Az utasítás hivatalosan a Pentium-on lett bevezetve, de az

IBM 386SLC és az Intel 486SLC processzorok már ismerték.

- **RDTSC (no op) {Pent+}**
  [0Fh, 31h]

  (ReaD Time-Stamp Counter)

  A Time-Stamp Counter egy olyan MSR, aminek tartalma a processzor resetelésekor kinullázódik, és attól kezdve minden óraütem eggyel növeli értékét. Az RDTSC utasítás a TSC értékét olvassa ki, és azt EDX:EAX-ben visszaadja. Végrehajthatósága a CR4 regiszter 2-es bitjének (Time-Stamp instruction Disable–TSD) állásától függ: ha TSD=0, akkor bármikor kiadható, különben pedig csak akkor, ha CPL=0. Ha TSD=1 és CPL>0, akkor 13-as kivétel (#GP) keletkezik. Az utasítás a Pentium-tól kezdve elérhető, és nem rejtett, egyszerűen csak nem volt túlságosan dokumentálva a kezdetekben.

- **RSM (no op) {386+}**
  [0Fh, 0AAh]

  (ReSuMe from System Management Mode)

  Az utasítás kiadásával a processzor kilép az SMM üzemmódból, a processzorállapotot visszaállítja az SMRAM-ból (System Management RAM), és visszatér a futás előző helyére (CS:IP vagy CS:EIP). Az utasítást csak az SMM módban ismeri fel a processzor, más módban kiadva azt az eredmény 6-os kivétel (#UD) lesz. Ha az állapot visszaállítása során az elmentett információban (SMM state dump) valami illegális van, akkor a processzor ú.n. shutdown állapotba kerül, ahonnan csak egy hardveres reset ébresztheti fel. Három fő hibaforrás van: az egyik, ha a CR4 regiszter tükörképében valamelyik fenntartott bit

1-es értékű, a másik, ha a CR0 érvénytelen beállításokat tartalmaz (pl. PE=0 és PG=1, CD=0 és NW=1 stb.). Szintén hibának számít, ha a mentési terület kezdőcíme nem osztható 32768-cal, de ez csak a Pentium processzorokig fontos, utána már nem. Az utasítás hivatalosan a Pentium-on lett bejelentve az SMM-mel együtt, de mivel SMM már az Intel 386SLC és 486SLC processzorokon is volt (sőt néhány más gyártó hasonló processzorcsaládján is), így ez az utasítás rejtetten már sok 80386-os processzoron elérhető.

- **SETALC (no op) {8086+} [0D6h]**

  (SET AL from Carry flag)

  Igazi gyöngyszem ez az egybájtos, operandus nélküli utasítás, mely ráadásul az ősidőktől kezdve elérhető minden Intel processzoron, és a kompatibilitás miatt sok más gyártó is implementálta termékébe. Működése majdnem megfelel az SBB AL,AL utasításnak, azaz AL-t kinullázza, ha CF=0, illetve AL-be 0FFh-t tölt, ha CF=1. Az SBB-vel ellentétben viszont nem változtatja meg egyetlen flag állását sem. Minden privilégium-szinten és minden üzemmódban végrehajtható, és mint említettük, minden teljesen Intel-kompatibilis processzornak ismernie kell. Teljesen érthetetlen, annyi év után miért nem ismeri el egyetlen gyártó sem ennek az utasításnak a létezését. (Jelenleg, 1999-ben az Intel és az AMD dokumentációja is fenntartott bájtnak nevezi ezt a kódot, de az Intel legalább azt megmondja, hogy az utasítás végrehajtása nem okoz 6-os kivételt.)

- **SHL dest,1 {8086+} [0D0h/0D1h, *r+6]**

- **SHL dest,CL {8086+}**
  [0D2h/0D3h, *r+6]
- **SHL dest,imm8 {186+}**
  [0C0h/0C1h, *r+6, imm8]

(SHift logical Left)

Ezt az utasítást már jól ismerjük, esetleg az tűnhet fel, hogy a léptetés számát 8 bites közvetlen konstansként is meg lehet adni. Ez a lehetőség a 80186/80188 processzorokon jelent meg, és azóta is elérhető. Az utasítás maga nem rejtett, viszont ez a fajta kódolása igen. A hivatalos verzió szerint a címzési mód bájt (*r) 'Reg' mezőjében (3..5 bitek) az 100b értéknek kell szerepelnie, de az 110b érték (mint fent is látható) is ugyanezt az utasítást jelenti. Néhány forrás azt állítja, hogy a mostani kódolás (*r+6) a SAL mnemonikhoz tartozó utasítást jelzi, de mivel tudjuk, hogy SHL=SAL, nem világos, miért van szükség ugyanannak az utasításnak két példányára.

- **TEST reg/mem8,imm8 {8086+}**
  [0F6h, *r+1, imm8]
- **TEST reg/mem16,imm16 {8086+}**
  [0F7h, *r+1, imm16]

(TEST)

Ez az utasítás is jól ismert. Itt azt a változatát látjuk, ami a cél memóriaterületet vagy általános regisztert teszteli egy közvetlen értékkel. Az utasítás szintén nem rejtett, az eltérés itt is a kódolásban van az SHL-hez hasonlóan. A hivatalos formában a címzési mód bájt 'Reg' mezője 000b-t kell hogy tartalmazzon, viszont a

001b érték ugyanezt a hatást váltja ki. Az utasítás duplázására itt sincs értelmes magyarázat.

- **WRMSR (no op) {386+}**
  [0Fh, 30h]

  (WRite Model-Specific Register)

  Ez az utasítás az ECX-ben megadott sorszámú MSR regiszterbe az EDX:EAX-ben megadott értéket írja. Ha CPL≠0, vagy a megadott MSR nem létezik, 13-as kivétel (#GP) keletkezik. Elérhetősége megegyezik az RDMSR utasításnál leírtakkal.

- **XBTS dest reg, base, EAX/AX,CL{csak 386 A-B0 step}**
  [0Fh, 0A6h, *r]

  (eXtract BiT String)

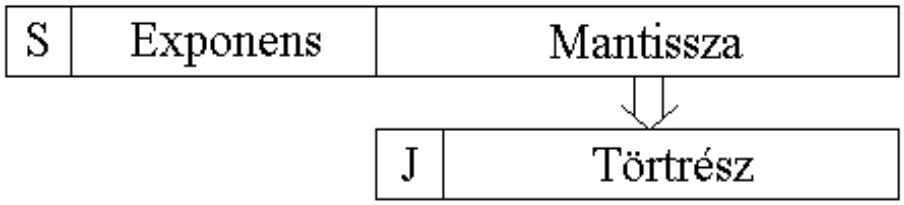
  Az IBTS utasítás párja, minden tekintetben. Működése: a bázis (base) címen elhelyezkedő terület EAX/AX sorszámú bitjétől kezdve CL db. bitet kimásol, és a cél regiszterbe (dest reg) jobbra igazítva betölti, a fennmaradó biteket pedig kinullázza. Ezt az utasítást is eltörölték a B0 stepping utáni 80386-osoktól kezdve, kódja azután "feltámadt" CMPXCHG formájában egy kis időre, majd örökre eltűnt.

17.2 A numerikus koprocesszor szolgáltatásai

Mint már korábban említettük, a valós számok közelítő ábrázolására a legelterjedtebb módszer a lebegőpontos formátum. Ezzel az adattípussal lehetővé válik a valós számok egy véges elemszámú részhalmazának megjelenítése. Bár azt írtuk, hogy valós, igazából csak racionális számok tárolhatók ilyen módon. Mivel azonban minden irracionális szám tetszőleges pontossággal megközelíthető racionális számokkal

(pontosabban alulról és felülről is korlátozható egy-egy racionális számsorozattal), a gyakorlati feladatok során ez is elegendő.

A lebegőpontos számok gépi formátumát az IEEE (Institute of Electrical and Electronics Engineers Incorporated) nevű szervezet szabványosította, ezek az IEEE-754 és IEEE-854 nevű szabványok. A számok általános alakja a következő:



Az *előjel* (S–Sign) a szokásos jelentést hordozza, tehát 1 esetén negatív, 0 esetén pozitív számot jelez. A *mantissza* (mantissa, significand) két részre osztható: egy 1 bites *egészrészre* (*J-bit*, binary integer bit), valamint *törtrészre* (fraction). Az *exponens* (exponent) azt a bináris (2-es alapú) hatványkitevőt adja meg, amire a mantisszát emelni kell, hogy az ábrázolt számot megkapjuk. (A bináris hatványkitevő most azt jelenti, hogy a hatványalap lesz 2.) A bináris egészrészt alkotó J-bitet gyakran nem tárolják el ténylegesen, hanem odaképzelik (implied J-bit). Az exponens egy bináris egész szám, és úgy tárolják, hogy valódi értékéhez hozzáadnak egy eltolást (*bias*, biasing constant), így alakul ki az ú.n. eltolt exponens (*biased exponent*). Az eltolási értéket úgy választják meg, hogy egyrészt az exponens eltolt értéke mindig pozitív legyen (mármint decimális alakban, tehát nem a kettes komplemens forma szerint), másrészt a legkisebb abszolút értékű ábrázolható szám reciproka szintén ábrázolható maradjon.

A következő típusú számok alakját rögzítették:

- **előjeles nulla (signed zeros)**
- **előjeles végtelen (signed infinities)**
- **normalizált véges szám (normalized finite numbers)**

- **denormalizált véges szám (denormalized finite numbers)**
- **érvénytelen, "nem szám" értékek (NaNs–Not a Number)**
- **meghatározatlan számok (indefinite numbers)**

Bármely előjelű nullának azonos a numerikus értéke, az előjel pedig általában azt mutatja, hogy a nulla milyen előjelű szám alulcsordulásakor, ill. milyen előjelű végtelen érték reciprokának képzése során keletkezett.

A végtelen értékek az ábrázolható legkisebb negatív ill. legnagyobb pozitív értéket képviselik, és egy aritmetikai *túlcsordulás* (overflow) következményei.

Véges értékek tárolásakor a használt formát a szám nagysága határozza meg. A koprocesszor, ha lehet, minden számot normalizált alakban tárol el. Ez azt jelenti, hogy a számot olyan alakra hozzák, ahol a 1≤mantissza<2 feltételek teljesülnek, továbbá a törtrész elején minimális számú bevezető nulla (leading zero) áll (ezt csak akkor veszi figyelembe, ha az ábrázolandó szám 1.0-nél kisebb). Ha az így képződött szám exponense a tárolható tartományon kívülre esik, akkor aritmetikai *alulcsordulás* (underflow) következik be. Alulcsordulás esetén a koprocesszor megpróbálja az eredményből a lehető legtöbb részt ábrázolhatóvá tenni, erre szolgál a denormalizáló eljárás (a használt módszer neve: gradual underflow technique). Ennek eredménye egy denormalizált szám, aminek az exponense a lehető legkisebb, az egészrészt tároló J-bit 0 lesz. Továbbá a törtrész elején nem minimális számú zéró lesz, tehát azokat itt nem lehet eliminálni, éppen ellenkezőleg, a koprocesszor mindaddig nullát szúr be a törtrész elejére, amíg az exponens ismét érvényes nem lesz. Ebből az okból a denormalizált számok már nem lesznek olyan pontosak, mint az eredeti végeredmény vagy a normalizáltak.

Látható, hogy normalizált esetben a J-bit értéke mindig 1, így azt felesleges eltárolni, és a koprocesszor két esetben (egyszeres és dupla pontosság) így is tesz.

A NaN értékek, mint nevük is mutatja, nem numerikus adatok, tehát normális esetben nem is kaphatjuk meg őket számolás útján. Kezdőértéket nem tartalmazó változó vagy memóriaterület inicializálására azonban jól használhatók, hiszen biztosan nem jelölnek egyetlen számot sem. Két típusuk van: csendes (quiet NaN–QNaN) és jelző (signaling NaN–SNaN) NaN-ok. Az SNaN-ok kivételt (exception) váltanak ki, ha megpróbálunk velük számolni, míg a QNaN-ok ezt nem teszik meg.

Meghatározatlan érték akkor keletkezik, ha egy numerikus kivételt vált ki a koprocesszor, de a kivételek maszkolva (tiltva) vannak. Meghatározatlan értékként általában a QNaN-t használja a koprocesszor.

A koprocesszor a következő 7 adattípust képes használni:

- **egyszeres pontosságú valós (single-precision real)**
- **dupla pontosságú valós (double-precision real)**
- **kiterjesztett pontosságú valós (extended-precision real)**
- **szó méretű egész (word integer)**
- **rövid egész (short integer)**
- **hosszú egész (long integer)**
- **pakolt BCD egész (packed BCD integer)**

A koprocesszoron belül csak a kiterjesztett pontosságú valós típus létezik, a többi típus pedig a memóriában előforduló adatok formátuma, amelyek betöltéskor átkonvertálódnak a kiterjesztett valós típusra, illetve tároláskor ilyen típusra vissza tudja alakítani a koprocesszor a (koprocesszor)verem tartalmát. Az egyes típusok gépi alakját mutatja a következő táblázat:

Típus	*Hossz*	*Exp.*	*J*	*Tört*	*Tartomány*
Egyszeres	32	23..30	imp.	0..22	$1.18*10^{-38}$–$3.40*10^{38}$
Dupla	64	52..62	imp.	0..51	$2.23*10^{-308}$–$1.79*10^{308}$
Kiterjesztett	80	64..78	63	0..62	$3.37*10^{-4932}$–$1.18*10^{4932}$
Szó egész	16	–	–	–	-32768–32767
Rövid egész	32	–	–	–	$-2.14*10^{9}$–$2.14*10^{9}$

Hosszú egész	**64**	**–**	**–**	**–**	**$-9.22*10^{18}$–$9.22*10^{18}$**
Pakolt BCD	**80**	**–**	**–**	**–**	**$(-10^{18}+1)$–$(10^{18}-1)$**

A "Hossz" oszlop az adattípusok bitekben elfoglalt méretét mutatja. Az "Exp." rövidítésű oszlop azt a bittartományt írja le, ahol az eltolt exponens kerül tárolásra. A "Tört" feliratú oszlop hasonló módon a törtrész kódolási tartományát mutatja. A "J" jelzésű oszlop a J-bit (tehát a bináris egészrészt tároló bit) pozícióját mutatja, míg a "Tartomány" jelzésű oszlop az adott típus számábrázolási tartományát tartalmazza.

Az előjelbit (S) mindegyik típus esetén a legfelső bitet foglalja el.

Az egyszeres és dupla pontosságú valós típusok esetén az "imp." (implied) rövidítés utal arra, hogy a J-bit értéke itt nem tárolódik el, értékét denormalizált véges számok és az előjeles nullák esetén 0-nak, minden más esetben 1-nek veszi a koprocesszor. Az ábrázolási tartomány csak közelítő értékeket mutat valós típusoknál.

Az exponens tárolásánál használt eltolási értékek a következőek: 127 (egyszeres), 1023 (dupla) és 16383 (kiterjesztett). Az exponens ábrázolási tartománya ennek megfelelően: -126–127, -1022–1023, illetve -16382–16383.

Az egész típusok tárolási alakja megfelel a CPU által használtnak, így természetes módon nincs exponens, J-bit, sem törtrész. Az előjelbitet kivéve a többi bit tárolja a szám bináris alakját. A negatív számok a szokásos kettes komplemens alakot használják. Az ábrázolási tartomány minden egész típusnál pontosan megadható: a rövid egészek -2147483648–+2147483647 (-2^{31}–$(2^{31}-1)$) között, míg a hosszú egészek -9223372036854775808 –+9223372036854775807 (-2^{63}–$(2^{63}-1)$) között ábrázolnak.

A pakolt BCD típus 18 db. decimális számjegyet tárol a következő módon: a 0..8 számú bájtok alsó és felső bitnégyese egy-egy decimális számjegyet kódol le (ASCII 0..9 karakterek).

A decimális szám legalsó számjegye a 0-ás bájt alsó felében van, míg a 8-as bájt felső fele a szám legelső, legnagyobb helyiértékű számjegyét tárolja. A 9-es bájtnak csak a legfelső, 7-es bitje van kihasználva (előjelbit), a többi bit értéke nem számít. A BCD számok NEM kettes komplemens alakban tárolódnak el, csak az előjelbit különbözteti meg az azonos abszolút értékű pozitív és negatív számokat. A pozitív és negatív nulla azonos értéket képvisel.

Az egyes számtípusok bináris kódolási formáját mutatják a következő táblázatok valós, bináris egész és pakolt BCD egész típusok esetén.

Érték típusa	*S*	*Eltolt exponens*	*J*	*Tört*
+Végtelen	0	11…11	1	00…00
+Normált	0	11…10–00…01	1	11…11–00…00
+Denormált	0	00…00	0	11…11–00…01
+Zéró	0	00…00	0	00…00
-Zéró	1	00…00	0	00…00
-Denormált	1	00…00	0	11…11–00…01
-Normált	1	11…10–00…01	1	11…11–00…00
-Végtelen	1	11…11	1	00…00
SNaN	?	11…11	1	0X…XX
QNaN	?	11…11	1	1?…??
Meghatározatlan	1	11…11	1	10…00

Érték típusa	*S*	*Értékes bitek*
Pozitív bináris egész	0	11…11–00…01
Negatív bináris egész	1	11…11–00…00
Zéró	0	00…00
Meghatározatlan	1	00…00

Érték típusa	*S*	*Spec*	*18. jegy*	*17. jegy*	*16. jegy*	…	*1. jegy*
Max. pozitív	0	0…0	1001	1001	1001	…	1001
Min. pozitív	0	0…0	0000	0000	0000	…	0001
Zéró	0	0…0	0000	0000	0000	…	0000

-Zéró	1	0…0	0000	0000	0000	…	0000
Min. negatív	1	0…0	1001	1001	1001	…	1001
Max. negatív	1	0…0	0000	0000	0000	…	0001
Meghatározatlan	1	1…1	1111	1111	????	…	????

A táblázatokban a kérdőjellel (?) jelzett bitek értéke tetszőleges lehet.

A valós típusok esetén az SNaN-ok kódolására használt szám törtrésze nem lehet nulla, különben az X-ek helyén tetszőleges érték állhat. Az SNaN-t a QNaN-tól a törtrész legfelső (legértékesebb) bitjének értéke különbözteti meg.

A bináris egészek a meghatározatlan értéket ugyanúgy kódolják, mint a legkisebb negatív számot. A koprocesszorba betöltéskor ezt az értéket számnak értelmezi az FPU, míg a memóriába tárolás esetén a numerikus kivételt jelző flag-ek (a státuszregiszterben) vizsgálatával kideríthető, melyik értékről van szó.

A harmadik táblázatban a "Spec" címkéjű oszlop a pakolt BCD számok 9-es bájtjának 0..6 bitjeire vonatkozik, de ezeknek az értékeknek csak a számok memóriába való tárolásakor van szerepük, mivel a koprocesszorba betöltéskor ezek a bitek figyelmen kívül lesznek hagyva. A meghatározatlan BCD érték betöltése a koprocesszorba megjósolhatatlan eredménnyel jár.

A koprocesszor számos regisztert tartalmaz a műveletek elvégzésére és egyéb célokra: 8 db. 80 bites adatregisztert (data registers), egy 16 bites státuszregisztert (Status register/Word–SW), egy 16 bites vezérlőregisztert (Control register/Word–CW), egy 16 bites tag-regisztert (tag word), egy 48 bites FPU utasításmutatót (FPU Instruction Pointer–IP), egy 48 bites FPU operandusmutatót (FPU Operand Pointer–OP), és végül egy 11 bites műveleti kód regisztert (opcode register).

Az adatregiszterek neve FPR0, FPR1, …, FPR7. Ezek a nevek azonban nem használhatóak közvetlenül, mivel az adatregiszterek verem szervezésűek, ami annyit jelent, hogy minden regisztert csak a verem tetejéhez (Top Of Stack–TOS) képest relatívan tudunk elérni. Ha pl. TOS=2, akkor a verem teteje a 2-es adatregiszter (FPR2), és erre az ST névvel hivatkozhatunk. A verem tetejétől számított i-edik adatregiszter elérésére az ST (i) jelölést használjuk, ahol $0 \leq i \leq 7$. ST(0) megegyezik ST-vel, tehát ugyancsak a verem tetejét jelöli ki. A példánál maradva, ST(1) az FPR3, ST(5) az FPR7, míg ST(7) az FPR1 regisztert választja ki. Ebből is látszik, hogy a regiszter sorszáma a (TOS+i) mod 8 kifejezés eredménye lesz.

A betöltő (load) utasítások hasonlóan cselekszenek a PUSH utasításhoz: csökkentik TOS-t, majd a verem új tetejére rakják a kért adatot, amit használtnak jelölnek be (tehát ezentúl már nem lesz üres). A tároló (store) utasítások a TOS által mutatott regiszter tartalmát olvassák ki. A POP művelet itt azt jelenti, hogy ST-t üresnek jelöli be, majd TOS értékét eggyel megnöveli. TOS tehát SP (Stack Pointer) szerepét játssza. Ha TOS=0, akkor egy betöltő utasítás hatására TOS értéke 7 lesz, és keletkezik egy veremtúlcsordulás kivétel (floating-point stack-overflow exception, #IS).

A státusz regiszter, nevéhez méltóan, a koprocesszor állapotát mutatja. Felépítése az alábbi ábrán látható:

B	C3	TOS	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE
15		12 11			8	7			4	3			0

Az alsó hat bit kivételek keletkezését jelzik, sorban érvénytelen művelet (Invalid operation Exception), denormalizált operandus (Denormalized operand, #D), nullával való osztás (division by Zero, #Z), túlcsordulás (Overflow, #O),

alulcsordulás (Underflow, #U), illetve a pontosságból veszítés (Precision loss, #P) tényét közlik. Ezek a bitek addig megőrzik állapotukat, míg azt valamilyen vezérlő utasítás (mint pl. FCLEX/FNCLEX, FINIT/FNINIT, FSAVE/FNSAVE, FRSTOR, FLDENV) meg nem változtatja. Mindegyik kivételt maszkolni (tiltani) lehet a vezérlőregiszterben. Ha legalább egy maszkolatlan (engedélyezett) bit be van állítva, akkor ES (Error Summary status) értéke 1, különben 0. Ha ES értéke 1 lesz, akkor a CPU a kivételt megpróbálja orvosolni a megfelelő kezelőrutin meghívásával. A B (FPU Busy) bit a 8087-es koprocesszoron játszik csak szerepet, és értéke mindig megfelel az ES bitnek.

Az érvénytelen művelet kivétel (IE=1) két dolgot jelenthet: veremhibát vagy érvénytelen aritmetikai operandust. Az SF (Stack Fault) bit jelzi, hogy keletkezett-e veremtúlcsordulás vagy -alulcsordulás. Ha SF=1, akkor veremkivétel keletkezett (Invalid Stack operation, #IS), és C1 állapota mutatja a tényleges hiba jellegét: C1=0 esetén alulcsordulás, egyébként túlcsordulás történt. Ezek a hibák összefüggnek az adatregiszterek tartalmával, és így a tag-regiszterrel is. Ez a bit a kivételt jelzőkhöz hasonlóan megőrzi értékét a következő törlésig. SF=0 esetén akkor a legutolsó aritmetikai utasítás érvénytelen operandussal lett meghívva (Invalid Arithmetic operand, #IA)

A 3 bites TOS mező a verem aktuális tetejének megfelelő adatregiszter sorszámát tartalmazza, értéke tehát 0..7 lehet.

A C0, C1, C2, C3 bitek állapotát külön-külön is állítja néhány utasítás, ilyenkor számos jelentést hordozhatnak. Az összehasonlító utasítások is ezekben a bitekben jelzik az összehasonlítás eredményét, és mivel összehasonlítást elágazások (feltételes vezérlési szerkezetek) megvalósításakor használunk, ezeket a biteket közös néven feltételjelző (condition code) biteknek hívjuk.

Ahhoz, hogy feltételes ugrást hajthassunk végre a feltételjelzők állásától függően, a legegyszerűbb módszer a következő lehet:

```
FSTSW AX
SAHF
```

Az első utasítás a státusz regiszter (SW) értékét az AX regiszterbe másolja át. Vegyük észre, hogy a C0, C1, C2 és C3 bitek pozíciója megfelel a CF, PF és ZF flag-ekének (C1-et kihagyva), így AH-t a Flags alsó bájtjába betöltve célhoz is értünk. AX operandust csak 80287-es vagy annál újabb típusú koprocesszor esetén kaphat az FSTSW utasítás, így a 8087-esen SW-t először a memóriába kell beírni, majd onnan kiolvasni AX-be (vagy rögtön AH-ba).

A Pentium Pro processzortól "fölfelé" létező FCOMI, FCOMIP, FUCOMI és FUCOMIP utasítások használatakor nincs szükség a fenti két plusz lépésre, mivel ezek az összehasonlítás eredményét közvetlenül az EFlags regiszter megfelelő flag-jeiben jelzik.

A vezérlőregiszter a koprocesszor működését befolyásoló biteket tartalmaz:

			IC	RC	PC	I		PM	UM	OM	ZM	DM	IM
15			12	11	8	7			4	3			0

Az alsó hat bit a numerikus kivételek tiltására szolgál. Ha valamelyik bit értéke 1, akkor a megfelelő kivétel nem keletkezhet, azaz a CPU nem észleli őket, és így a kivétel-kezelő rutin sem fut le.

A PC (Precision Control) bitek állapota adja meg, hogy az FPU milyen pontossággal végezze el az additív (FADD, FADDP, FSUB, FSUBP, FSUBR, FSUBRP), multiplikatív (FMUL, FMULP, FDIV, FDIVP, FDIVR, FDIVRP) és négyzetgyök-vonó (FSQRT) utasításokat. Jelentése:

PC értéke	*Pontosság*

00	Egyszeres
01	–
10	Dupla
11	Kiterjesztett

Az RC (Rounding Control) mező értéke határozza meg, milyen kerekítést alkalmazzon az FPU. Jelentése:

RC értéke	*Kerekítés módja*
00	Legközelebbi értékhez
01	Lefelé (-Inf felé)
10	Felfelé (+Inf felé)
11	Nulla felé (csonkítás)

A legközelebbi értékhez kerekítés (round to nearest, even) az eredményhez legközelebbi értéket választja, és egyenlő távolság esetén a páros értéket választja. A lefelé (round down) és felfelé (round up) kerekítések az intervallum-aritmetika megvalósítását segítik ("Inf"=Infinity–végtelen). Csonkítást (truncate, chop) általában akkor alkalmazunk, ha az eredményt valamilyen bináris egész alakban várjuk.

Az IC (Infinity Control) bitnek csak a 8087, 80187 és 80287 (a 80287xl-t kivéve) koprocesszorok esetén van jelentése. A pozitív végtelen értékét általában úgy definiáljuk, hogy az nagyobb bármely pozitív véges számnál. Ezt affin szemléletnek hívjuk, megkülönböztetve a projektív szemlélettől. IC=1 esetén affin módon kezeli a végteleneket az FPU, egyébként pedig projektívan.

Az I (Interrupt enable mask) bit csak a 8087-es koprocesszoron létezik, és a megszakítások engedélyezésére szolgál.

A tag-regiszter 8 mezőt tartalmaz, melyek mindegyike egy-egy adatregiszterhez tartozik. Értékük mutatja, hogy a megfelelő adatregiszternek mi a tartalma. Felépítése, és a mezők jelentése:

TAG7	TAG6	TAG5	TAG4	TAG3	TAG2	TAG1	TAG0

15 12 11 8 7 4 3 0

Tag értéke	*Tartalom*
00	**Érvényes szám (Valid)**
01	**Nulla (Zero)**
10	**Különleges (Special)**
11	**Üres (Empty)**

A koprocesszor inicializálása (FINIT/FNINIT vagy FSAVE/ FNSAVE útján) után mindegyik tag értéke 11b, tehát az összes adatregisztert üresnek jelzik. Különleges értéknek minősülnek a NaN-ok, érvénytelen értékek, végtelenek és a denormált számok.

Veremtúlcsordulás történik, ha TOS csökkentése után egy nem üres adatregiszterre mutat. Hasonlóan, veremalulcsordulásról akkor beszélünk, ha TOS-t növelve az egy üres regiszterre fog mutatni, vagy ha az utasítás egyik operandusa egy üres adatregisztert választ ki. Nem üresnek tekinthető minden olyan adatregiszter, aminek tag értéke 00b, 01b vagy 10b.

Az utasítás- és operandus mutatók a legutolsó, nem vezérlő utasítás címét, ill. legutóbb használt operandus címét tartalmazzák. A regiszterek tartalma egy szabályos távoli mutató: 16 bites szegmens/szelektor, valamint 32 bites offszet.

Minden koprocesszor utasítás műveleti kódjának első bájtja a 0D8..DFh értékek valamelyike. Mivel ennek a felső öt bitje állandó (11011b), ezt felesleg lenne eltárolni. Ezért csak az első bájt alsó három bitje, valamint a második bájt kerül eltárolásra, így jön ki a 11 bit. A műveleti kód regiszter 0..7 bitjei a műveleti kód második bájtját tartalmazzák, míg a 8..10

bitekben az első bájt alsó három bitje található. Ebben a regiszterben a legutolsó, nem vezérlő utasítás opcode-ját találjuk meg.

A kivételek kezeléséről is kell néhány szót ejteni. Ha a koprocesszor egy numerikus hibát észlel, akkor kétféleképpen cselekedhet. Ha az adott kivétel az FPU vezérlőregiszterében (CW) maszkolva (tiltva) van, a koprocesszor saját maga kezeli le a hibát, ami általában azt fogja jelenteni, hogy az eredmény a meghatározatlan érték (real/integer indefinite) vagy egy denormált szám (denormalized number) lesz. Ha azonban engedélyeztük a kérdéses numerikus kivétel keletkezését, a koprocesszor a kivételt tudatja a processzorral.

A 8087-es koprocesszor numerikus hiba észlelésekor aktiválja az INT# nevű kivezetését, ami a 8086/8088-as processzor NMI# lábára van közvetlenül rákötve. Egy numerikus kivétel keletkezése így az NMI megszakítás létrejöttéhez vezet, amit a processzor az INT 02h szoftver-megszakítás kezelőjének végrehajtásával intéz el. Az NMI kezelőnek tehát először el kell döntenie, FPU-hiba vagy egyéb hardveresemény okozta az NMI-t, csak ezután cselekedhet.

A 80286-os processzor már külön bemenettel rendelkezik a koprocesszor-hibák fogadására, ennek neve ERROR#. A kezdetben kitűzött cél az volt, hogy numerikus hiba keletkezése esetén a koprocesszor az azonos nevű kimenetét aktiválja. A jelet az ERROR# bemeneten érzékelve a processzor 16-os kivételt (#MF) váltott volna ki, amit kifejezetten erre a célra vezettek be ezen a processzoron. Mégsem ez történt. Korábbi ismereteinkből már tudhatjuk, hogy a kivételek az INT 00h..1Fh szoftver-megszakításokra képződnek le. A 16-os kivételnek az INT 10h felel meg, amit az IBM már lefoglalt a képernyővel kapcsolatos szolgáltatások számára. Éppen ezért az összes IBM AT és kompatibilis számítógépen a 80286-os

processzor ERROR# bemenetét állandóan magas feszültségszinten tartják, a koprocesszor ERROR# kivezetése pedig a megszakítás-vezérlő IRQ13-as vonalára van rákötve. Egy numerikus kivétel keletkezése esetén így a processzor IRQ13-at érzékel, végrehajtja annak kezelőjét (ez alapállapotban az INT 75h), ami végül egy INT 02h utasítást hajt végre. A numerikus kivételek lekezelése tehát a 80286-os processzoron is az NMI-n keresztül történik a visszafelé kompatibilitás megőrzése végett.

Az NMI-t igénybe vevő módszert *MS-DOS-kompatibilis módnak* hívják, szemben a 16-os kivételt generáló *natív móddal* (native mode). A 80286-os processzor támogatja a natív módot, de az említett okokból egyetlen IBM AT-kompatibilis számítógépen sem keletkezhet 16-os kivétel hardveres úton.

A 80386-os processzor a 80286-ossal megegyező módon kezeli a numerikus kivételeket.

A 80486-os processzor már lehetővé teszi a felhasználónak (vagy inkább az operációs rendszernek), hogy válasszon az MS-DOS-kompatibilis és a natív numerikus hibakezelés között. A CR0 regiszter NE (5-ös) bitjének állapota dönti el, mi történjen numerikus kivétel észlelésekor. Ha NE=0, akkor a processzor az MS-DOS-kompatibilis módnak megfelelően cselekszik. Ez egész pontosan azt jelenti, hogy aktiválja FERR# nevű lábát, ami a megszakítás-vezérlőn keresztül IRQ13-at vált ki, és végül ismét az NMI kezelője (INT 02h) kerül végrehajtásra. Ha NE=1, akkor is keletkezik IRQ13, de 16-os kivételt is generál a processzor.

A P5 architektúrájú processzorok (Pentium, Pentium MMX) a 80486-ossal megegyező módon kezelik a numerikus hibákat.

A 80486-os és a Pentium processzorokon előfordulhat, hogy MS-DOS-kompatibilis módban numerikus hiba keletkezése esetén nem azonnal reagál a processzor IRQ13-mal, hanem csak a következő FPU vagy WAIT utasításnál. Ez a

késleltetett kivétel-kezelés a P6-os architektúrában (Pentium Pro, Pentium II, Pentium III stb.) megszűnt, ezek a processzorok ugyanis a natív módhoz hasonlóan azonnal reagálnak a kivételre.

Az alábbiakban megtalálható a koprocesszorok utasítás-készlete, a 8087-estől egészen a Pentium Pro-ig (ezen a processzoron vezettek be legutoljára új numerikus utasítást). Mindegyik utasítás után olvasható az operandusok formája, az utasítás elérhetősége kapcsos zárójelek között (azaz milyen koprocesszorokon ill. processzorokon található meg), a mnemonik jelentése angolul, valamint az utasítás működésének rövid leírása. Nem célunk teljes részletességgel bemutatni minden utasítást, mivel erről a témáról elég sok irodalom jelent meg. Az utasítások mnemonikjának első betűje ("F") utal arra, hogy ezek FPU utasítások.

A következő jelöléseket alkalmaztuk:

- **(no op) – nincs operandus**
- **src – forrás (source)**
- **dest – cél (destination)**
- **mem – memóriaoperandus**
- **m16 – szó bináris egész memóriaoperandus (word integer)**
- **m32 – rövid bináris egész vagy egyszeres pontosságú valós memóriaoperandus (short integer, single real)**
- **m64 – hosszú bináris egész vagy dupla pontosságú valós memóriaoperandus (long integer, double real)**
- **m80 – pakolt BCD egész vagy kiterjesztett pontosságú valós memóriaoperandus (packed BCD integer, extended real)**
- **ST – ST(0) -t rövidíti, a koprocesszor verem teteje**

- **ST(i) – a TOS értékéhez relatív i-edik koprocesszor adatregiszter/verembejegyzés**
- **8087 – Intel 8087 vagy vele kompatibilis koprocesszor**
- **287 – Intel 80287, Intel 80287xl, IIT 2c87 stb.**
- **387 – Intel 80387 vagy vele kompatibilis koprocesszor**
- **PPro – Intel Pentium Pro (P6 architektúra) és vele kompatibilis processzor**
- **PROC+ – az adott PROC processzoron/koprocesszoron és attól kezdve minden későbbin létezik az utasítás**
- **csak PROC – csak az adott PROC család tagjain érvényes az utasítás**

A törtvonal (/) mindenhol vagylagosságot jelez (kizáró vagy). Ahol nem jeleztük külön egy utasítás elérhetőségét, akkor az adott utasítás a 8087-es koprocesszoron és attól kezdve minden későbbi rendszeren is megtalálható.

17.2.1 Adatmozgató utasítások

- **FLD m32/m64/m80/ST(i)**
  (LoaD real)
  Csökkenti TOS-t, majd a megadott forrásból betölti ST-be a valós számot.
- **FST m32/m64/ST(i)**
  (STore real)
- **FSTP m32/m64/m80/ST(i)**
  (STore real and Pop)
  A megadott helyre eltárolja ST tartalmát valós alakban. Az FSTP változat ezután még egy POP-ot hajt végre.
- **FILD m16/m32/m64**
  (LoaD Integer)
  Csökkenti TOS-t, majd a megadott forrásból betölti ST-be a bináris egész számot.

- **FIST m16/m32**
  (STore Integer)
- **FISTP m16/m32/m64**
  (STore Integer and Pop)
  A megadott helyre eltárolja ST tartalmát bináris egész alakban. Az FISTP változat ezután egy POP-ot hajt végre.
- **FBLD m80**
  (LoaD BCD integer)
  Csökkenti TOS-t, majd a megadott forrásból betölti ST-be a pakolt BCD egész számot.
- **FBSTP m80**
  (STore BCD integer and Pop)
  A megadott helyre eltárolja ST tartalmát pakolt BCD egész alakban, majd végrehajt egy POP-ot.
- **FXCH (no op)/ST(i)**
  (eXCHange registers)
  Megcseréli ST tartalmát ST(i)-vel, vagy ha nincs operandus megadva, akkor ST(1)-gyel.
- **FCMOVE/FCMOVZ ST,ST(i) {PPro+}**
  (MOVe if Equal/Zero)
- **FCMOVNE/FCMOVNZ ST,ST(i) {PPro+}**
  (MOVe if Not Equal/Not Zero)
- **FCMOVB/FCMOVC/FCMOVNAE ST,ST(i) {PPro+}**
  (MOVe if Below/Carry/Not Above or Equal)
- **FCMOVBE/FCMOVNA ST,ST(i) {PPro+}**
  (MOVe if Below or Equal/Not Above)
- **FCMOVNB/FCMOVNC/FCMOVAE ST,ST(i) {PPro+}**
  (MOVe if Not Below/Not Carry/Above or Equal)
- **FCMOVNBE/FCMOVA ST,ST(i) {PPro+}**
  (MOVe if Not Below or Equal/Above)
- **FCMOVU ST,ST(i) {PPro+}**
  (MOVe if Unordered)

- **FCMOVNU ST,ST(i) {PPro+}**
  (MOVe if Not Unordered)
  Ez a 8 utasítás valamilyen feltétel teljesülése esetén ST (i) tartalmát ST-be mozgatja. Az utasítások az EFlags bitjeit veszik figyelembe, nem pedig a státuszregiszterben levő feltételbiteket (condition code bits). Az unordered feltétel akkor teljesül, ha PF=1 (C2=1). Egy előző összehasonlítás alapján akkor áll fent az unordered feltétel, ha a két operandus közül legalább egy NaN vagy érvénytelen érték volt. A mnemonikban a CMOV szó utal arra, hogy feltételes adatmozgatás történik (conditional move).

17.2.2 Alap aritmetikai utasítások

- **FADD (no op)/m32/m64**
- **FADD ST,ST(i)**
- **FADD ST(i),ST**
  (ADD real)
- **FADDP (no op)**
- **FADDP ST(i),ST**
  (ADD real and Pop)
  Az FADD a valós forrás értéket hozzáadja a célhoz, az FADDP pedig ezután még végrehajt egy POP-ot. Az operandus nélküli változatok megfelelnek az FADDP ST(1),ST utasításnak mindkét esetben.
- **FIADD m16/m32**
  (ADD Integer)
  A bináris egész forrás tartalmát hozzáadja ST-hez.
- **FSUB (no op)/m32/m64**
- **FSUB ST,ST(i)**
- **FSUB ST(i),ST**
  (SUBtract real)

- **FSUBP (no op)**
- **FSUBP ST(i),ST**
  (SUBtract real and Pop)
  Az FSUB kivonja a valós forrást a célból, az eredményt a cél helyén eltárolja, ami után az FSUBP végrehajt egy POP-ot is. Az operandus nélküli változatok az FSUBP ST(1),ST utasítást rövidítik.
- **FSUBR (no op)/m32/m64**
- **FSUBR ST,ST(i)**
- **FSUBR ST(i),ST**
  (SUBtract real Reversed)
- **FSUBRP (no op)**
- **FSUBRP ST(i),ST**
  (SUBtract real Reversed)
  Az FSUBR a cél tartalmát kivonja a valós forrásból, az eredményt a célban eltárolja, ezután az FSUBRP végrehajt egy POP-ot. Az op. nélküli formák az FSUBRP ST(1),ST utasításnak felelnek meg.
- **FISUB m16/m32**
  (SUBtract Integer)
  A bináris egész forrást kivonja ST-ből, majd az eredményt ST-be rakja.
- **FISUBR m16/m32**
  (SUBtract Integer Reversed)
  ST tartalmát kivonja a memóriában levő bináris egész forrásból, majd az eredményt ST-be rakja.
- **FMUL (no op)/m32/m64**
- **FMUL ST,ST(i)**
- **FMUL ST(i),ST**
  (MULtiply real)
- **FMULP (no op)**
- **FMULP ST(i),ST**
  (MULtiply real and Pop)

Az FMUL megszorozza a célt a valós forrással, majd az FMULP végrehajt egy POP-ot. Az operandus nélküli változatok megegyeznek az FMULP ST(1), ST utasítással.

- **FIMUL m16/m32**
  (MULtiply Integer)
  ST-t megszorozza a bináris egész forrás operandussal.
- **FDIV (no op)/m32/m64**
- **FDIV ST,ST(i)**
- **FDIV ST(i),ST**
  (DIVide real)
- **FDIVP (no op)**
- **FDIVP ST(i),ST**
  (DIVide real and Pop)
  Az FDIV elosztja a célt a valós forrással, az eredményt a cél helyén eltárolja, ami után az FDIVP végrehajt egy POP-ot is. Az operandus nélküli változatok az FDIVP ST(1),ST utasítást rövidítik.
- **FDIVR (no op)/m32/m64**
- **FDIVR ST,ST(i)**
- **FDIVR ST(i),ST**
  (DIVide real Reversed)
- **FDIVRP (no op)**
- **FDIVRP ST(i),ST**
  (DIVide real Reversed)
  Az FDIVR a valós forrást elosztja a céllal, az eredményt a célban eltárolja, ezután az FDIVRP végrehajt egy POP-ot. Az operandus nélküli formák az FDIVRP ST(1),ST utasításnak felelnek meg.
- **FIDIV m16/m32**
  (DIVide Integer)
  ST-t elosztja a bináris egész forrással, a hányadost ST-ben tárolja el.

- **FIDIVR m16/m32**
  (DIVide Integer Reversed)
  A bináris egész forrást elosztja ST-vel, majd az eredményt ST-ben eltárolja.
- **FPREM (no op)**
  (Partial REMainder)
- **FPREM1 (no op) {387+}**
  (IEEE 754 Partial REMainder)
  Ez a két utasítás ST-t elosztja ST(1)-gyel, majd a maradékot ST-be rakja. A hányados legalsó három bitjét ugyancsak eltárolják C0:C3:C1-ban (tehát C0 a 2-es, C3 az 1-es, C1 pedig a 0-ás bit). A maradék az osztandó (ST) előjelét örökli. Az FPREM1 utasítás az IEEE 754-es szabvány szerint adja meg a maradékot.
- **FABS (no op)**
  (ABSolute value)
  ST-t kicseréli annak abszolút értékére.
- **FCHS (no op)**
  (CHange Sign)
  ST-t negálja (előjelét megfordítja).
- **FRNDINT (no op)**
  (RouND to INTeger)
  ST-t a vezérlőregiszter RC mezője szerint egész értékre kerekíti.
- **FSCALE (no op)**
  (SCALE by a power of 2)
  ST értékét megszorozza 2-nek ST(1)-ben megadott hatványával, majd az eredményt ST-be rakja. ST (1) tartalmát egésznek értelmezi, amit a benne levő érték csonkolásával ér el. Teljesülnie kell, hogy $-2^{15} \leq ST(1) < 2^{15}$. Ha $0 < ST(1) < 1$, a művelet eredménye megjósolhatatlan.
- **FSQRT (no op)**

(SQuare RooT)
ST-t lecseréli annak négyzetgyökére. ST-nek nagyobb vagy egyenlőnek kell lennie (-0)-nál. A -0 négyzetgyöke önmaga lesz.

- **FXTRACT (no op)**
  (eXTRACT exponent and significand)
  Legyen N=ST. ST-t lecseréli N valódi (nem eltolt) exponensére, csökkenti TOS-t, és az új ST-be berakja N mantisszáját. Végül tehát az exponens ST(1)-ben, a mantissza ST-ben lesz, és mindkettőt érvényes valós számként tárolja el. Ha a kiinduló operandus nulla, akkor exponensként negatív végtelent, mantisszaként pedig nullát tárol el, aminek az előjele megegyezik a forráséval. Továbbá ebben az esetben egy nullával való osztás numerikus kivétel (#Z) is keletkezik.

17.2.3 Összehasonlító utasítások

- **FCOM (no op)/m32/m64/ST(i)**
  (COMpare real)
- **FCOMP (no op)/m32/m64/ST(i)**
  (COMpare real and Pop)
- **FCOMPP (no op)**
  (COMpare real and Pop twice)
  Összehasonlítják ST-t a valós forrással (tehát ST-ből kivonják a forrást), majd az eredménynek megfelelően beállítják a feltételjelző biteket a státuszregiszterben. Az FCOMP az összehasonlítás után egy, az FCOMPP pedig két POP-ot hajt végre. Az operandus nélküli változatok minden esetben ST(1)-et használják. A nulla előjele lényegtelen, azaz -0=+0. Ha legalább az egyik operandus NaN vagy érvénytelen érték,

akkor a bitek az *unordered* beállítást fogják mutatni, és egy érvénytelen operandus kivétel (#IA) keletkezik. Ha a kivétel keletkezését el akarjuk kerülni, akkor használjuk az FUCOM, FUCOMP és FUCOMPP utasításokat. A feltételjelző bitek lehetséges állapotát mutatja az alábbi táblázat:

Eredmény	*C0*	*C2*	*C3*
ST>forrás	0	0	0
ST<forrás	1	0	0
ST=forrás	0	0	1
Unordered	1	1	1

- **FICOM m16/m32**
  (COMpare Integer)
- **FICOMP m16/m32**
  (COMpare Integer and Pop)
  ST-t összehasonlítják a bináris egész forrással, majd az eredmény alapján beállítják a feltételjelző biteket. Az FICOMP ezután még egy POP-ot is végrehajt.
- **FUCOM (no op)/ST(i) {387+}**
  (COMpare real Unordered)
- **FUCOMP (no op)/ST(i) {387+}**
  (COMpare real Unordered and Pop)
- **FUCOMPP (no op) {387+}**
  (COMpare real Unordered and Pop twice)
  Az FCOM, FCOMP, FCOMPP utasításhoz hasonlóan működnek, tehát a valós forrást összehasonlítják ST-vel, majd az FUCOMP egy, az FUCOMPP pedig két POP-ot hajt végre. Az operandus nélküli esetben ST(1) a forrás. Ha az operandusok közül legalább az egyik SNaN vagy érvénytelen érték, akkor érvénytelen operandus kivétel (#IA)

keletkezik. QNaN-ok esetén viszont nincs kivétel, a feltételjelző bitek pedig az unordered beállítást fogják mutatni.

- **FCOMI ST,ST(i) {PPro+}**
  (COMpare real and set EFlags)
- **FCOMIP ST,ST(i) {PPro+}**
  (COMpare real, set EFlags, and Pop)

  Összehasonlítják ST(i)-t ST-vel, majd az eredménynek megfelelően beállítják a CF, PF és ZF flag-eket az EFlags regiszterben. Az FCOMIP ezután végrehajt egy POP-ot is. A nulla előjele nem számít, tehát -0=+0. Ha az operandusok közül legalább az egyik NaN vagy érvénytelen érték, a flag-eket az unordered jelzésre állítják be, majd egy érvénytelen operandus kivételt (#IA) váltanak ki. Ha ezt el akarjuk kerülni, használjuk az FUCOMI és FUCOMIP utasításokat. A flag-ek állapotának jelentése az alábbi táblázatban látható:

Eredmény	*CF*	*PF*	*ZF*
ST>forrás	0	0	0
ST<forrás	1	0	0
ST=forrás	0	0	1
Unordered	1	1	1

- **FUCOMI ST,ST(i) {PPro+}**
  (COMpare real Unordered and set EFlags)
- **FUCOMIP ST,ST(i) {PPro+}**
  (COMpare real Unordered, set EFlags, and Pop)

  Teljesen hasonlóan működnek az FCOMI és FCOMIP utasításokhoz, azzal a különbséggel, hogy QNaN operandus(ok) esetén nem generálnak kivételt.
- **FTST (no op)**

(TeST for zero)
ST-t összehasonlítja 0.0-val, majd az eredménynek megfelelően beállítja a feltételjelző biteket a státuszregiszterben. A bitek állapotát az alábbi táblázat szerint értelmezhetjük:

Eredmény	*C0*	*C2*	*C3*
ST>0.0	0	0	0
ST<0.0	1	0	0
ST=±0.0	0	0	1
Unordered	1	1	1

- **FXAM (no op)**
  (eXAMine)
  ST-t megvizsgálja, majd a benne levő adatnak megfelelően beállítja a feltételjelző biteket. C1 mindig az előjelet tartalmazza (0=pozitív, 1=negatív), a többi bit jelentése az alábbi táblázatban látható:

Tartalom	*C3*	*C2*	*C0*
Nem támogatott	0	0	0
NaN	0	0	1
Normalizált véges szám	0	1	0
Végtelen	0	1	1
Zéró	1	0	0
Üres	1	0	1
Denormalizált szám	1	1	0

17.2.4 Transzcendentális utasítások

- **FSIN (no op)** **{387+}**
  (calculate SINe)

ST-t lecseréli annak szinusz értékére. Az operandus radiánban értendő, és teljesülnie kell, hogy $|ST| < 2^{63}$. Ha érvénytelen operandust adunk meg, akkor ST érintetlen marad, és a C2 feltételjelző bitet 1-re állítja.

- **FCOS (no op) {387+}**
  (calculate COSine)
  ST-t lecseréli annak koszinusz értékére. Az operandus radiánban értendő, és teljesülnie kell, hogy $|ST| < 2^{63}$. Ha érvénytelen operandust adunk meg, akkor ST érintetlen marad, és a C2 feltételjelző bitet 1-re állítja.
- **FSINCOS (no op) {387+}**
  (calculate SINe and COSine)
  Legyen N=ST. ST-t lecseréli N szinusz értékére, csökkenti TOS-t, majd az új ST-be berakja N koszinusz értékét. Végül tehát ST(1)-ben lesz a szinusz, míg ST-ben a koszinusz érték. Az operandus radiánban értendő, és teljesülnie kell, hogy $|ST| < 2^{63}$. Ha érvénytelen operandust adunk meg, akkor ST és TOS érintetlen marad, és a C2 feltételjelző bitet 1-re állítja.
- **FPTAN (no op)**
  (calculate Partial TANgent)
  ST-t lecseréli annak tangens értékére, csökkenti TOS-t, majd az új ST-be 1.0-t rak be, így ST(1)-ben lesz a tangens érték. Az operandus radiánban értendő, és teljesülnie kell, hogy $|ST| < 2^{63}$ (a 80387 előtti koprocesszoroknál $0 \le ST \le Pi/4$ legyen). Ha érvénytelen operandust adunk meg, akkor ST és TOS érintetlen marad, és a C2 feltételjelző bitet 1-re állítja. Ha utána kiadunk egy FDIVR vagy FDIVRP utasítást, akkor ST-ben a kotangens értéket kapjuk meg.

- **FPATAN (no op)**
  (calculate Partial ArcTANgent)
  Kiszámítja az ST(1)/ST hányados arkusztangensét, az eredményt berakja ST(1)-be, majd végrehajt egy POP-ot. Az ST-ben levő eredményt radiánban kapjuk meg, és -Pi≤ST≤+Pi, valamint a szög az eredeti ST(1) előjelét örökli. A 80287-es és régebbi koprocesszorokon a kiinduló operandusokra teljesülnie kell a következőnek: 0≤ |ST(1)| < |ST| <+∞.
- **F2XM1 (no op)**
  (compute 2 to the power of X minus one)
  Kiszámítja a 2ST-1 kifejezés értékét, majd azt visszarakja ST-be. A kiinduló operandusnak 80287-es és korábbi FPU-n teljesítenie kell a 0<ST<0.5 egyenlőtlenségeket, a 80387-es és újabb koprocesszorokon pedig a -1.0<ST<+1.0 feltételeket.
- **FYL2X (no op)**
  (compute Y*Log_2 X)
  Kiszámítja ST-nek (X) kettes alapú (bináris) logaritmusát, azt megszorozza ST(1)-gyel (Y), az eredményt visszaírja ST(1)-be, majd végrehajt egy POP-ot. Ha X negatív, akkor érvénytelen operandus kivétel (#IA) generálódik. Ha X=±0, akkor két eset lehetséges: ha a vezérlőregiszterben ZM=1, akkor nullával való osztás numerikus kivétel (#Z) generálódik, különben az eredmény végtelen lesz, aminek az előjele Y előjelének ellentettje.
- **FYL2XP1 (no op)**
  (compute Y*Log_2 (X+1.0))
  Veszi ST-t (X), hozzáad 1.0-t, kiszámítja ennek a kettes alapú logaritmusát, ezt megszorozza ST(1)-gyel

(Y), az eredményt eltárolja ST(1)-ben, majd végrehajt egy POP-ot. A kiinduló operandusokra teljesülniük kell a következőknek: $-(1-\sqrt{2}/2) \leq ST \leq 1-\sqrt{2}/2$, továbbá $-\infty \leq ST(1) \leq \infty$. Ha érvénytelen operandus(ok)ra próbáljuk meg végrehajtani az utasítást, a reakció kiszámíthatatlan, és nem fog feltétlenül kivétel generálódni.

17.2.5 Konstansbetöltő utasítások

- **FLD1 (no op)**
  (LoaD 1.0)
- **FLDZ (no op)**
  (LoaD 0.0)
- **FLDPI (no op)**
  (LoaD PI)
- **FLDL2E (no op)**
  (LoaD Log_2 E)
- **FLDLN2 (no op)**
  (LoaD LN 2=Log_e 2)
- **FLDL2T (no op)**
  (LoaD Log_2 10)
- **FLDLG2 (no op)**
  (LoaD LG 2=Log_{10} 2)
  Ezek az utasítások csökkentik TOS-t, majd az új ST-be sorban az 1.0, 0.0, Pi, $\log_2$ e, ln 2, $\log_2$ 10 és lg 2 konstansokat töltik be.

17.2.6 Koprocesszor-vezérlő utasítások

- **FINCSTP (no op)**
  (INCrement floating-point STack Pointer)
  Hozzáad egyet a TOS-hoz. Ha TOS értéke 7 volt, akkor az 0-vá válik. Ez NEM egyezik meg a POP-

pal, mivel a tag-ek nem módosulnak (tehát nem jelöli be ST-t üresnek TOS növelése előtt).

- **FDECSTP (no op)**
  (DECrement floating-point STack Pointer)
  Kivon egyet a TOS-ból. Ha TOS értéke előtte 0 volt, akkor az új értéke 7 lesz. Az utasítás NEM egyezik meg a PUSH/LOAD utasításokkal, mivel nem módosítja a tag-eket.
- **FFREE ST(i)**
  (FREE data register)
  A megadott adatregiszter tag értékét üresre (empty, 11b) jelöli be, a regiszter ill. a TOS értékét nem változtatja meg.
- **FINIT (no op)**
  (INITialize FPU)
- **FNINIT (no op)**
  (INITialize FPU, No wait)
  A koprocesszort alaphelyzetbe hozzák. Ez pontosan annyit jelent, hogy a vezérlőregiszterbe a 037Fh (a legközelebbi értékhez kerekítés, kiterjesztett pontosság, minden numerikus kivétel letiltva), a státuszregiszterbe a 0000h (minden kivétel jelzőbitje törölve, TOS=0, ES=B=0), míg a tag-regiszterbe a 0FFFFh (mindegyik adatregiszter üres) értékeket töltik. A 80486 és újabb processzorokon az utasítás és operandus mutatókat is törlik. Az FINIT utasítás ezek előtt végrehajt egy FWAIT utasítást is az esetleges függő numerikus kivételek kezelésére, míg az FNINIT ezt nem teszi meg.
- **FCLEX (no op)**
  (CLear EXceptions)
- **FNCLEX (no op)**
  (CLear EXceptions, No wait)

Törlik a státuszregiszterben az összes kivétel jelzőbitjét, valamint ES-t és B-t. Az FCLEX egy FWAIT utasítást is kiad mindezek előtt, az FNCLEX viszont nem.

- **FENI (no op) {csak 8087}**
  (ENable Interrupts)
- **FNENI (no op) {csak 8087}**
  (ENable Interrupts, No wait)
  A vezérlőregiszter I bitjét állítják 0-ra, ezáltal az FPU generálhat megszakítást. Az FENI egy FWAIT utasítás kiadásával biztosítja az esetleges függő kivételek lekezelését, míg az FNENI ezt nem teszi meg.
- **FDISI (no op) {csak 8087}**
  (DISable Interrupts)
- **FNDISI (no op) {csak 8087}**
  (DISable Interrupts, No wait)
  A vezérlőregiszter I bitjét állítják 1-re, ezáltal az FPU nem generálhat megszakítást. Az FDISI egy FWAIT utasítás kiadásával biztosítja az esetleges függő kivételek lekezelését, míg az FNDISI ezt nem teszi meg.
- **FSETPM (no op) {csak 287}**
  (SET Protected Mode)
  A CPU ezzel az utasítással jelzik az FPU-nak, hogy védett módba lépett át. A 80387-es és annál újabb koprocesszorok figyelmen kívül hagyják ezt az utasítást.
- **FRSTPM (no op) {csak 287}**
  (ReSTore/resume Protected Mode)
  Valós módba visszatéréskor kell kiadni ezt az utasítást, ezzel jelezve az FPU-nak a változást. Csak a

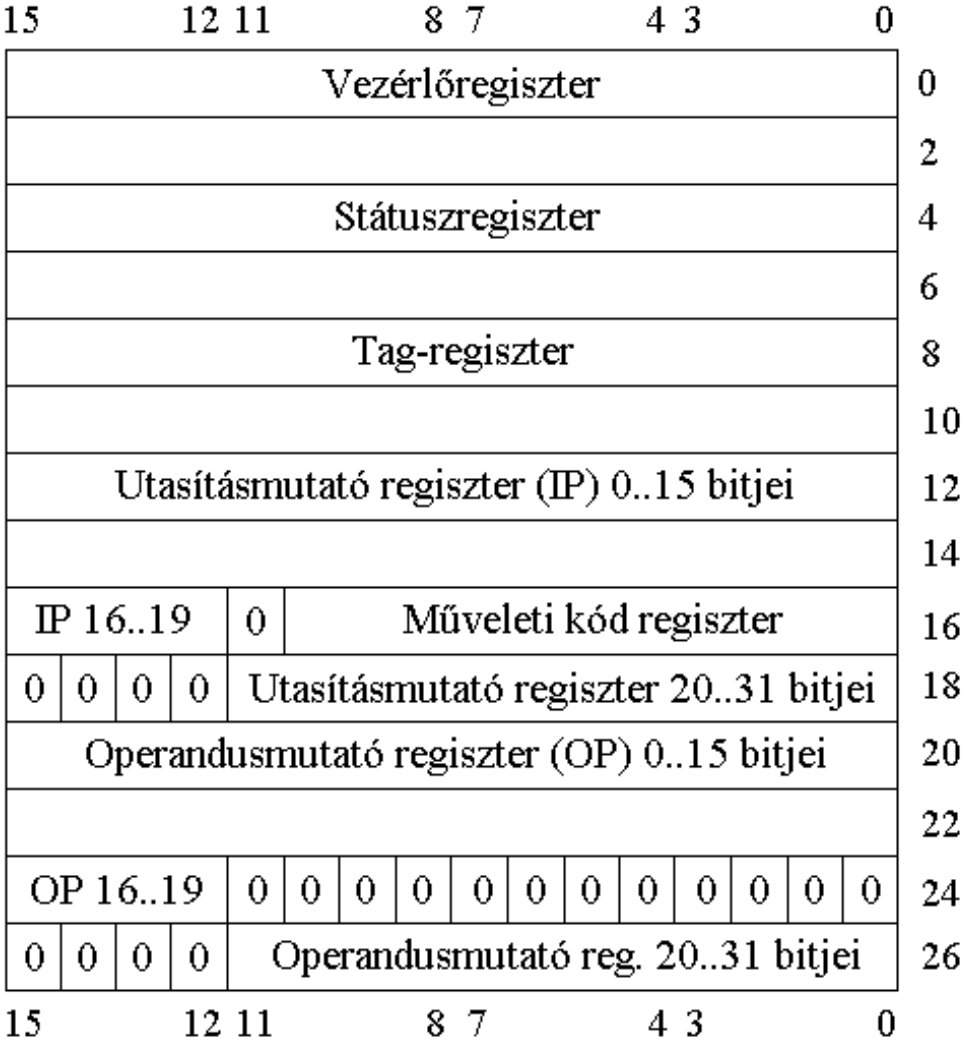
80287-es család esetén van jelentősége, az FSETPM-hez hasonlóan.

- **FLDCW m16**
  (LoaD Control Word)
  A megadott forrást betölti a vezérlőregiszterbe. Ha a státuszregiszterben valamelyik kivétel jelzőbitje be volt állítva, akkor ez problémákat okozhat. Ennek elkerülésére végrehajtása előtt érdemes kiadni egy FCLEX utasítást.
- **FSTCW m16**
  (STore Control Word)
- **FNSTCW m16**
  (STore Control Word, No wait)
  A megadott címre eltárolják a vezérlőregiszter tartalmát. Az FSTCW utasítás egy FWAIT utasítást hajt végre a tárolás előtt, az FNSTCW viszont nem teszi ezt meg.
- **FSTSW m16**
- **FSTSW AX {287+}**
  (STore Status Word)
- **FNSTSW m16**
- **FNSTSW AX {287+}**
  (STore Status Word, No wait)
  A megadott memóriacímre vagy az AX regiszterbe eltárolják a státuszregisztert. Az FSTSW utasítás egy FWAIT utasítást is végrehajt a tárolás előtt, az FNSTSW viszont nem.
- **FRSTOR source mem**
  (ReSTORe saved FPU state)
  Az előzőleg FSAVE vagy FNSAVE utasítással elmentett állapotot állítja vissza a megadott forráscímről. A terület mérete 16 bites kód-szegmens esetén 94 bájt, 32 bites kódszegmens használatakor pedig 108 bájt.

- **FSAVE dest mem**
  (SAVE FPU state)
- **FNSAVE dest mem**
  (SAVE FPU state)

  Eltárolják a koprocesszor állapotát a megadott címre, majd újrainicializálják az FPU-t az FINIT, FNINIT utasításhoz hasonlóan. Az FSAVE egy FWAIT utasítást is kiad először, ezt az FNSAVE nem teszi meg. 16 bites kódszegmens esetén 94, míg 32 bites kódszegmens esetén 108 bájtnyi információ kerül eltárolásra.

  A következő ábra valós módban, 32 bites operandusméret esetén mutatja a tárolás formátumát. Jobb oldalon a terület kezdetétől mért offszet látható bájtokban.



Védett módban, 32 bites operandusméret esetén az alábbi a helyzet:

15 12 11 8 7 4 3 0						
Vezérlőregiszter						0
						2
Státuszregiszter						4
						6
Tag-regiszter						8
						10
Utasításmutató regiszter (IP) offszet 0..15 bitjei						12
Utasításmutató regiszter offszet 16..31 bitjei						14
Utasításmutató regiszter szelektor						16
0	0	0	0	0	Műveleti kód regiszter	18
Operandusmutató regiszter (OP) offszet 0..15 bitjei						20
Operandusmutató regiszter offszet 16..31 bitjei						22
Operandusmutató regiszter szelektor						24
						26
15 12 11 8 7 4 3 0						

Valós módban, 16 bites operandusméretet használva így épül fel a táblázat:

15 12 11 8 7 4 3 0													
Vezérlőregiszter													0
Státuszregiszter													2
Tag-regiszter													4
Utasításmutató regiszter (IP) 0..15 bitjei													6
IP 16..19	0	Műveleti kód regiszter											8
Operandusmutató regiszter (OP) 0..15 bitjei													10
OP 16..19	0	0	0	0	0	0	0	0	0	0	0	0	12
15 12 11 8 7 4 3 0													

Végül védett módban, 16 bites operandusméret használatakor a tábla a következő alakot ölti:

15 12 11 8 7 4 3 0	
Vezérlőregiszter	0
Státuszregiszter	2
Tag-regiszter	4
Utasításmutató regiszter (IP) offszet	6
Utasításmutató regiszter szelektor	8
Operandusmutató regiszter (OP) offszet	10
Operandusmutató regiszter szelektor	12
15 12 11 8 7 4 3 0	

A processzor üzemmódjától függően más lesz az elmentett információ. A fenti képek pontosan azt mutatják, amit az FLDENV utasítás betölt, illetve az FSTENV és FNSTENV utasítások elmentenek. Az üresen hagyott mezők értéke definiálatlan, azokat ne használjuk. Az FSAVE és FNSAVE utasítások ezen kívül a koprocesszor verem tartalmát is kiírják a fenti adatokon kívül: először a ST(0) értéke tárolódik el (10 bájt), majd ST(1) következik, stb. Összesen tehát 14+80=94 bájtot

tárolnak el 16 bites kódszegmens esetén, valamint 28+80=108 bájtot 32 bites kódszegmenst használva.

- **FLDENV source mem**
  (LoaD FPU ENVironment state)
  Az előzőleg FSTENV vagy FNSTENV utasítással elmentett állapotot állítja vissza a megadott forráscímről. A terület mérete 16 bites kód-szegmens esetén 14 bájt, 32 bites kódszegmens használatakor pedig 28 bájt.
- **FSTENV dest mem**
  (STore FPU ENVironment state)
- **FNSTENV dest mem**
  (STore FPU ENVironment state, No wait)
  Eltárolják a koprocesszor állapotát a megadott címre, majd az összes koprocesszor-kivételt letiltják a vezérlőregiszterben. Az FSTENV egy FWAIT utasítást is kiad először, ezt az FNSTENV nem teszi meg. 16 bites kódszegmens esetén 14, míg 32 bites kódszegmens esetén 28 bájtnyi információ kerül eltárolásra. A terület tartalmát, formátumát lásd az FSAVE, FNSAVE utasítások leírásánál.
- **FWAIT/WAIT (no op)**
  (FPU WAIT)
  Addig megállítja a CPU-t, amíg a koprocesszoron be nem fejeződik minden utasítás lefutása. Segítségével a CPU és az FPU munkáját szinkro-nizálhatjuk össze, amire a Pentium processzorok előtti processzorok és koprocesszorok esetén szükség is van. Az utasítás gépi kódja 9Bh, mindkét mnemonik erre a kódra vonatkozik. Ezt az utasítást a CPU hajtja végre, tehát nem koprocesszor utasítás.
- **FNOP (no op)**

(No OPeration)

A CPU NOP utasításához hasonlóan nem végez semmi műveletet, csak az utasításmutató regiszterek értékét változtatja meg a CPU-n és az FPU-n is.

17.3 Az Intel 80186-os processzor újdonságai

17.3.1 Változások az utasításkészletben

- **IMUL dest reg16,source reg/mem16,imm8/imm16**

  Az IMUL utasítás immáron háromoperandusú formában is használható. Működése: a második (forrás) operandust megszorozza a harmadik operandussal, az eredményt pedig az első (cél) operandusban tárolja el. A cél 16 bites általános regiszter, a forrás általános regiszter vagy memóriahivatkozás lehet, a harmadik operandus pedig egy bájt vagy szó méretű közvetlen érték.
- **IMUL dest reg16,imm8/imm16**

  Ez a forma az IMUL reg16,reg16,imm8/imm16 utasítást rövidíti, ahol a cél és a forrás is ugyanaz az általános regiszter.
- **MOV CS,reg/mem16**

  Ennek az utasításnak a végrehajtása 6-os kivételt eredményez.
- **PUSH SP**

  Az utasítás működése a 8086/8088-os processzorokon szimbolikusan ilyen volt:

```
SUB     SP,0002h
MOV     SS:[SP],SP
```

Mostantól viszont az elvárt műveletet végzi el, azaz:

```
MOV     TEMP,SP
SUB     SP,0002h
MOV     SS:[SP],TEMP
```

A TEMP itt egy 16 bites ideiglenes változót jelöl. Ha mindenképpen olyan kódot akarunk, ami bármely processzoron ugyanazt az értéket helyezi a verembe (ami SP csökkentett értéke lesz), használjuk a következő utasításokat a PUSH SP helyett:

```
PUSH    BP
MOV     BP,SP
XCHG    BP,[BP]
```

- **PUSH imm8/imm16**
  A PUSH immár képes közvetlen értéket is a verembe rakni. A bájt méretű operandusnak a 16 bitesre előjelesen kiterjesztett értéke kerül a verembe.
- **POP CS**
  Ez az utasítás (melynek kódja 0Fh) 6-os kivételt okoz a 80186/80188 processzorokon.
- **SHL/SAL dest,imm8**
- **SHR dest,imm8**
- **SAR dest,imm8**
- **ROL dest,imm8**
- **ROR dest,imm8**
- **RCL dest,imm8**
- **RCR dest,imm8**
  Mindegyik forgató és léptető utasítás most már képes 1-től eltérő közvetlen értékkel léptetni a

céloperandust. A lépésszám értelmezését illetően lásd a következő bekezdést.

- SHL/SAL dest,imm8/CL
- SHR dest,imm8/CL
- SAR dest,imm8/CL
- ROL dest,imm8/CL
- ROR dest,imm8/CL
- RCL dest,imm8/CL
- RCR dest,imm8/CL

A forgató és léptető utasítások azon formája, ahol a lépésszámot 8 bites közvetlen értékként vagy CL-ben adjuk meg, a második operandusnak csak az alsó 5 bitjét veszi figyelembe, szemben a 8086/8088-os processzorokkal, amik CL-nek mind a 8 bitjét értelmezték, közvetlen értékként pedig csak az 1-et fogadták el. Ez az eltérés lehetőséget ad rá, hogy ezeket a processzorokat megkülönböztethessük a 8086-os és 8088-as prociktól.

17.3.2 Új utasítások

- BOUND dest reg16,source mem
  (check array index against BOUNDs)

  A cél regiszter előjeles tartalmát mint tömbindexet hasonlítja össze a memóriában levő tömbhatárokkal. A forrás egy olyan memóriaoperandus, ami 2 szót tartalmaz. Ha a cél kisebb mint az első érték (alsó határ), vagy nagyobb mint a második érték (felső határ), 5-ös kivétel (#BR) keletkezik.
- ENTER imm16,imm8
  (ENTER new stack frame)

  A magas szintű nyelvek blokkstruktúrájának megvalósítását könnyíti meg, mivel egy megadott

tulajdonságú új veremkeretet készít elő. Mindkét operandus rendhagyó módon közvetlen érték. Az első, szó méretű operandus neve legyen LOCALS, a második, bájt méretű operandusé pedig NESTING. Az utasítás működését a következő pszeudokód mutatja be:

```
Push BP
TEMP:=SP
NESTING:=NESTING Mod 32
While (NESTING>0) DO
    NESTING:=NESTING-1
    BP:=BP-2
    Push SS:[BP]
BP:=TEMP
SP:=SP-LOCALS
```

A TEMP egy 16 bites ideiglenes változót jelöl. A NESTING operandusnak csak az alsó 5 bitjét veszi figyelembe. Az utasítás párja a LEAVE, ami a legutóbbi ENTER által létrehozott veremkeretet szabadítja fel.

- **INS dest,DX**
  (INput String from port)
- **INSB (no op)**
  (INput String Byte from port)
- **INSW (no op)**
  (INput String Word from port)

  Ez egy újabb sztringkezelő utasítás. A többihez hasonlóan az INSB és INSW mnemonikok operandus nélküliek, míg az INS két operandust kap: az első egy áloperandus, amely egy memóriacímet jelöl ki, a második pedig a DX regiszter kell legyen. Az utasítás a DX portról beolvas egy bájtot vagy egy szót, ezt az értéket

eltárolja az ES:DI címre, majd DI értékét a DF flag állásától függően növeli vagy csökkenti az olvasott adat méretével. A REP prefix ezzel az utasítással is használható.

- **LEAVE (no op)**
  (LEAVE stack frame)
  A legutolsó ENTER utasítás tevékenységét szünteti meg a következő utasítások végrehajtásával:

```
MOV     SP,BP
POP     BP
```

- **OUTS DX,source**
  (OUTput String to port)
- **OUTSB (no op)**
  (OUTput String Byte to port)
- **OUTSW (no op)**
  (OUTput String Word to port)
  Az INS/INSB/INSW utasítás párja. Az OUTSB és OUTSW mnemonikok operandus nélküliek, míg az OUTS két operandust kap: az első a DX regiszter, a második pedig egy áloperandus, amely egy memóriacímet jelöl ki. Az utasítás a DS:SI címen levő bájtot vagy szót kiírja a DX portra, majd SI értékét a DF flag állásától függően növeli vagy csökkenti az olvasott adat méretével. A DS szegmens felülbírálható prefixszel. A REP prefix ezzel az utasítással is használható.
- **POPA (no op)**
  (POP All general purpose registers)
  A következő utasítássorozatot hajtja végre:

```
POP     DI
POP     SI
POP     BP
```

```
ADD     SP,0002h
POP     BX
POP     DX
POP     CX
POP     AX
```

A negyedik sorban szereplő ADD SP,2 utasítás csak szimbolikusan néz így ki, mivel a flag-ek tartalmát NEM befolyásolja a POPA utasítás.

- **PUSHA (no op)**
  (PUSH All general purpose registers)
  A következő utasításokat hajtja végre:

```
PUSH    AX
PUSH    CX
PUSH    DX
PUSH    BX
PUSH    SP
PUSH    BP
PUSH    SI
PUSH    DI
```

Az ötödik sorban szereplő utasítást úgy hajtja végre, ahogyan a "hagyományos" PUSH SP esetén is teszi, tehát SP aktuális értéke lesz eltárolva az SS:(SP-2) címen, majd SP értéke csökken 2-vel.

17.3.3 Új kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
5	#BR	Bound Range	Trap	BOUND utasítás
6	#UD	Undefined Opcode	Fault	Érvénytelen utasítás

5-ös kivétel akkor keletkezik, ha a BOUND utasítás céloperandusában levő előjeles tömbindex a megadott határokon kívül esett.

A 6-os kivételt akkor generálja a processzor, ha olyan bájtsorozattal találkozik, amit nem képes utasításként értelmezni. A későbbi processzoroktól kezdve ez a kivétel nagyon "univerzális", mivel elég sok dolog kiválthatja (pl. privilegizált utasítások kiadása valós módban, LOCK prefix helytelen használata, egy MMX utasítás kiadása úgy, hogy a CR0 regiszter EM bitje be van állítva, stb.). A következő feltételek valamelyike okoz 6-os kivételt:

- **érvénytelen vagy fenntartott műveleti kódot vagy kódolást (pl. érvénytelen 'Mod' vagy 'Reg' mező a címzési mód/regiszter info bájtban) próbáltunk meg végrehajtani**
- **egy utasítás adott műveleti kódját érvénytelen operandussal próbáltuk meg végrehajtani**

17.4 Az Intel 80286-os processzoron megjelent új szolgáltatások

17.4.1 Változások az utasításkészletben

- **BOUND dest reg16,source mem**
  A 80186/80188-as processzor csapdaként (trap) kezelte az 5-ös kivételt, azaz az elmentett CS:IP a BOUND utasítást követő utasításra mutatott. Mostantól hibaként (fault) kezeli ezt a kivételt a 80286-os processzor, tehát a BOUND utasításra mutató CS:IP lesz eltárolva.

- **CLI (no op)**
- **IN AL/AX,imm8/DX**
- **INS dest,DX**
- **LOCK**
- **OUT imm8/DX,AL/AX**
- **OUTS DX,source**
- **STI (no op)**

  Ezek az utasítások (ill. a LOCK prefix) IOPL-érzékenyek. Ha a Flags regiszter IOPL mezőjének (12..13 bitek) tartalma nagyobb vagy egyenlő, mint a CPL (Current Privilege Level), akkor az utasítás végrehajtható, különben pedig 13-as kivétel (#GP) keletkezik.
- **HLT (no op)**

  Ez a jó öreg utasítás védett módban privilegizált lett, tehát csak akkor hajtható végre, ha CPL=0, különben 13-as kivétel (#GP) keletkezik.
- **POP CS**

  Végrehajtása nem feltétlenül eredményez 6-os kivételt. Értelmezéséhez lásd még az "Új prefixek" alfejezetet.
- **POPF (no op)**

  Valós módban vagy védett módban CPL=0 esetén minden flag módosítható Flags-ben. Védett módban, ha CPL>0, az IOPL mező nem módosítható, változatlan marad.

Védett módban a JMP, CALL, INT, INTO, IRET és RET utasítások működése sokkal összetettebb, de erről a védett módról szóló fejezetben szólunk majd bővebben.

17.4.2 Új prefixek

- **0Fh**

(Extended instruction prefix)

A hajdani POP CS utasítás gépi kódját ezentúl az új utasítások lekódolásához használják fel. Azt jelöli, hogy az utána következő bájttal együtt kapott kétbájtos érték adja meg az utasítás műveleti kódját (opcode). Nem igazi prefix, az elnevezés csak az elhelyezkedésén alapul. Bővebb leírását lásd az "Utasítások kódolása" c. fejezetben.

17.4.3 Új utasítások

- **ARPL dest selector,source selector reg16**
  (Adjust RPL field of destination selector)
  Ha a cél szelektor RPL mezője (0..1 bitek) kisebb értéket tartalmaz, mint a forrás regiszter RPL mezője, akkor a ZF flag-et 1-re állítja, és a cél RPL értékét beállítja a forrás RPL értékére. Különben ZF-et kinullázza. Ez az utasítás nem privilegizált.
- **CLTS (no op)**
  (CLear Task Switched flag)
  A gépi állapotszó regiszter (MSW) TS (3-as) bitjét törli.
- **LAR access rights reg16,source selector**
  (Load Access Rights)
  A forrás szelektorhoz tartozó deszkriptor 32..47 bitjeinek értékét 0FF00h-val maszkolva (logikai ÉS kapcsolatba hozva) a cél általános regiszterbe tölti. Sikeres olvasás esetén ZF-et beállítja, különben ZF értéke 0 lesz. Ez az utasítás nem privilegizált.
- **LGDT source mem**
  (Load Global Descriptor Table register)

A megadott memóriaoperandus hat bájtját betölti a GDTR regiszterbe. Az alsó szó (0..1 bájtok) a határt, a 2..4 bájtok pedig a bázis lineáris címét tartalmazzák. A 5-ös bájtnak nincs szerepe.

- **LIDT source mem**
  (Load Interrupt Descriptor Table register)
  A megadott memóriaoperandus hat bájtját betölti az IDTR regiszterbe. Az alsó szó (0..1 bájtok) a határt, a 2..4 bájtok pedig a bázis lineáris címét tartalmazzák. A 5-ös bájtnak nincs szerepe.
- **LLDT source selector**
  (Load Local Descriptor Table register)
  A megadott operandusban levő szelektorhoz tartozó deszkriptort betölti az LDTR regiszterbe. A szelektornak a GDT-be kell mutatnia, és az általa megjelölt deszkriptor csak egy LDT-hez tartozhat. Az operandus a nullszelektor is lehet, ebben az esetben érvénytelennek jelöli be az LDTR tartalmát.
- **LMSW source status word**
  (Load Machine Status Word)
  Az általános regiszterből vagy memóriaoperandusból betölti a gépi állapotszót, amit az MSW regiszterbe tesz. A PE (0-ás) bitet ezzel a módszerrel nem törölhetjük.
- **LSL limit reg16,selector reg/mem16**
  (Load Segment Limit)
  A forrás operandusban levő szelektor által mutatott deszkriptor határ (limit) mezőjét betölti a cél általános regiszterbe. Az eltárolt érték mindig bájtokban értendő. Ha a deszkriptor látható a CPL-en, valamint megfelelő a deszkriptor típusa (kódszegmens vagy adatszegmens), a sikeres

betöltést a ZF=1 jelzi, különben ZF-et kinullázza. Ez az utasítás nem privilegizált.

- **LTR source selector**
  (Load Task Register)
  Az operandusban levő szelektor által mutatott TSS deszkriptort betölti a taszkregiszterbe (TR), majd a TSS-t foglaltnak (busy) jelöli be.
- **SGDT dest mem**
  (Store Global Descriptor Table register)
  A GDTR regiszter tartalmát beírja a megadott memóriaoperandus hat bájtjába. Az alsó szó (0..1 bájtok) a határt, a 2..4 bájtok pedig a bázis lineáris címét tartalmazzák. A 5-ös bájtnak nincs szerepe, azt 0FFh-val tölti fel az utasítás. Ez az utasítás nem privilegizált.
- **SIDT dest mem**
  (Store Interrupt Descriptor Table register)
  Az IDTR regiszter tartalmát beírja a megadott memóriaoperandus hat bájtjába. Az alsó szó (0..1 bájtok) a határt, a 2..4 bájtok pedig a bázis lineáris címét tartalmazzák. A 5-ös bájtnak nincs szerepe, azt 0FFh-val tölti fel az utasítás. Ez az utasítás nem privilegizált.
- **SLDT dest selector**
  (Store Local Descriptor Table register)
  A megadott operandusba eltárolja az LDTR regiszter szelektor részét. Ez az utasítás nem privilegizált.
- **SMSW dest status word**
  (Store Machine Status Word)
  A céloperandusba eltárolja a gépi állapotszó regiszter (MSW) tartalmát. Ez az utasítás nem privilegizált.
- **STR dest selector**
  (Store Task Register)

A taszkregiszter (TR) tartalmát a céloperandusba tölti. Ez az utasítás nem privilegizált.

- **VERR source selector**
  (VERify segment for Readability)
- **VERW source selector**
  (VERify segment for Writability)

  A forrásba levő szelektor által mutatott szegmenst vizsgálják, hogy az elérhető-e, továbbá lehet-e olvasni (VERR) ill. lehet-e írni (VERW). Ha a feltételek teljesülnek, ZF-et beállítják, különben ZF értéke 0 lesz. A szelektornak egy érvényes, GDT-beli vagy LDT-beli kódszegmens vagy adatszegmens deszkriptorra kell mutatnia. Ha kódszegmenst vizsgálunk, akkor a VERR esetén a következő feltételek esetén lesz ZF=1: ha az adott szegmens nem illeszkedő (nonconforming), akkor a DPL≥CPL és DPL≥RPL egyenlőtlenségeknek kell teljesülniük; ha a szegmens illeszkedő (conforming), DPL értéke tetszőleges lehet. Mindkét esetben a kódszegmensnek természetesen olvashatónak kell lennie. Ezek az utasítások nem privilegizáltak.

A felsorolt utasítások közül a CLTS, LGDT, LIDT, LLDT, LMSW és LTR privilegizáltak (privileged), tehát csak CPL=0 esetén hajthatók végre. Ezen kívül az ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR és VERW utasítások csak védett módban adhatók ki, valós üzemmódban 6-os kivételt (#UD) váltanak ki.

17.4.4 Új üzemmódok

A 8086/8088-os és 80186/80188-as processzorok csak egyféle üzemmódban voltak képesek működni. A 80286-os

processzor alapállapotban szintén ezt az üzemmódot használja, aminek neve *valós címzésű mód* vagy röviden *valós mód* (real-address mode, real mode). Ez a processzor azonban átkapcsolható egy úgynevezett *védett módba* is (protected mode).

A védett mód, amint neve is mutatja, erős védelmi szolgáltatásokat nyújt. Ezek alapja a privilegizálási rendszer (privilege system), ami annyit tesz, hogy bármely program (taszk) a 4 privilegizálási szint bármelyikén futhat. A 0-ás szint jelenti a legerősebb, míg a 3-as a leggyengébb védelmi szintet. Ezért a 0-ás szinten általában csak az operációs rendszer magja (kernel) fut, míg a felhasználói programok a 3-as, esetleg 2-es privilégium szintet használják.

A szegmensregiszterek új neve *szegmens szelektor regiszter* (segment selector register) a védett módban. A *szelektor* (selector) nem a szegmens báziscímét jelöli, hanem csak egy index egy táblázatba. Három ilyen rendszertáblázat létezik: globális deszkriptor tábla (Global Descriptor Table–GDT), lokális deszkriptor tábla (Local Descriptor Table–LDT) és megszakítás deszkriptor tábla (Interrupt Descriptor Table–IDT). A táblázatban találhatók a *deszkriptorok* (descriptor), amik egy adott szegmens típusát, báziscímét, méretét, elérhetőségét stb. tartalmazzák.

Védett módban sok kivétel egy 16 bites hibakódot is berak a verembe az IP-másolat után.

Védett módba a gépi állapotszó (MSW) regiszter PE (0-ás) bitjének 1-be állításával juthatunk. Ezt a műveletet egy távoli vezérlésátadásnak kell követnie. A valós módba való visszatérés sokkal körülményesebb, mivel az egyetlen utasítással, ami képes az MSW módosítására (LMSW), nem törölhetjük a PE bitet. Ezért a processzort szoftverből resetelni kell, amit pl. tripla hibával oldhatunk meg.

A védett módról szól egy külön fejezet is, ott bővebb leírást találhatunk erről az összetett problémakörről.

17.4.5 Változások a kivétel-kezelésben

Védett módban (valósban sosem) sok kivétel az IP után egy 16 bites hibakódot is berak a verembe. A hibakódok alakja az alábbi táblázatban látható:

Bitpozíció	*Név*
0	**EXT**
1	**IDT**
2	**TI**
3..15	**Index**

Ha az EXT (EXTernal event) bit értéke 1, akkor egy külső esemény (pl. hardver-megszakítás) okozta a kivételt.

Az IDT bit 1 értéke jelzi, hogy a szelektor az IDT egy elemét választja ki.

Ha IDT=0, akkor a TI (Table Indicator) bit értéke határozza meg, mire vonatkozik az index: ha TI=0, akkor a GDT, különben az LDT egy deszkriptorát választja ki a szelektor.

Az Index mező tartalmazza az adott táblázat egy elemének sorszámát (ez maga a szelektor).

Az eddig bevezetett kivételek (0, 1, 3, 4, 5, 6) közül egy sem rak hibakódot a verembe. Az új kivételek közül a 8, 10, 11, 12 és 13 számúak tesznek hibakódot a verembe.

A 0-ás kivételt (#DE) mostantól hibaként (fault) kezeli a processzor:

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
0	**#DE**	**Divide Error**	**Fault**	**DIV és IDIV utasítások**

Az 1-es kivétel (#DB) mostantól akkor is keletkezhet, ha taszkváltáskor az új taszk TSS-ében a T bit értéke 1. A kivétel típusa ebben az esetben is csapda (trap).

Az 5-ös kivétel (#BR) mostantól hibaként (fault) kezelődik:

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
5	**#BR**	**Bound Range**	**Fault**	**BOUND utasítás**

6-os kivétel a következő feltétel teljesülése esetén is keletkezik:

- **az ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR és VERW utasítások valamelyikét próbáltuk meg végrehajtani valós módban**

17.4.6 Új kivételek

A 10 és 11 számú kivételek csak védett módban keletkezhetnek.

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
7	**#NM**	**No Math Coprocessor**	**Fault**	**FP vagy WAIT utasítás**
8	**#DF**	**Double Fault**	**Abort**	**Sokféle**
9	**–**	**Coprocessor Segment Overrun**	**Abort**	**FP utasítás**
10	**#TS**	**Invalid TSS**	**Fault**	**Taszkváltás vagy TSS elérése**
11	**#NP**	**Segment Not Present**	**Fault**	**Szegmensregiszter betöltés vagy rendszerreg. elérés**
12	**#SS**	**Stack-Segment Fault**	**Fault**	**Veremműveletek, SS betöltése**
13	**#GP**	**General Protection**	**Fault**	**Sokféle**
16	**#MF**	**Math Fault**	**Fault**	**FP vagy WAIT utasítás**

7-es kivétel az alábbi három esetben történik:

- **egy lebegőpontos (FP) utasítást hajtottunk végre úgy, hogy az MSW regiszter EM (2-es) bitje be volt állítva**

- **egy lebegőpontos (FP) utasítást hajtottunk végre úgy, hogy az MSW regiszter TS (3-as) bitje be volt állítva**
- **egy WAIT/FWAIT utasítást hajtottunk végre úgy, hogy az MSW regiszter TS (3-as) és MP (1-es) bitjei be voltak állítva**

Az első feltétel azt jelenti, hogy a számítógépben nincs numerikus koprocesszor, de az FPU utasításokat az operációs rendszer emulálni tudja, ha ezt a kivételt észleli. A második eset akkor áll fenn, ha egy taszkváltás következik be, és az FPU így jelzi az operációs rendszernek, hogy a koprocesszor aktuális állapotát jó lenne elmentenie. Végül a harmadik feltétel a taszkváltáskor szükséges FPU-állapot elmentését terjeszti ki a WAIT/FWAIT utasításra, mivel ez nem FPU hanem CPU utasítás. A kivétel másik elnevezés alatt is ismeretes: Device Not Available. Hibakód nem kerül a verembe. Az elmentett CS:IP érték a hibázó utasításra mutat.

Ha egy kivétel-kezelő meghívása során egy újabb kivétel keletkezik, akkor a processzor általában megpróbálja lekezelni először az utóbbi kivételt, majd visszatér az eredeti kivételhez. Néhány esetben azonban ez nem működik. A processzor ilyenkor 8-as kivételt generál, és hibakódként 0000h-t rak a verembe. Ezt a helyzetet nevezzük *dupla hibának*, amiről a kivétel a nevét kapta. Ez a kivétel abort típusú, ennek megfelelően a veremben levő CS- és IP-másolatok értéke meghatározatlan, és a kivétel lekezelése után a hibázó taszk nem indítható újra. Ha a 8-as kivétel-kezelő meghívása során egy újabb kivétel keletkezik, akkor *tripla hibáról* beszélünk. Tripla hiba esetén a processzor *shutdown* állapotba kerül, ahonnan csak egy NMI vagy hardveres reset ébreszti fel. Ha az NMI kezelő (INT 02h megszakítás) végrehajtása közben történik tripla hiba, akkor csak egy hardveres reset indíthatja újra a processzort.

Dupla hiba akkor történik, ha az első és a második kivétel is a 0, 10, 11, 12, 13 kivételek valamelyike.

A 9-es kivétel azt jelzi, hogy egy koprocesszor utasítás memóriában levő operandusa szegmenshatár-sértést követett el. A kivétel abort típusú, és bár az elmentett CS:IP érték a hibázó utasításra mutat, a kivétel keletkezésekor a koprocesszor és a program (taszk) állapota definiálatlan. Hibakód nem kerül a verembe. Lásd még a 80486-os processzor fejezetében levő megjegyzést is.

10-es kivétel akkor keletkezik, ha taszkváltás során a processzor érvénytelen információt talál az új taszk TSS-ében. A következő feltételek valamelyike okozhatja ezt a hibát:

- **a TSS szegmens határa túl kicsi (index=TSS szelektor index)**
- **érvénytelen vagy nem jelenlevő LDT (index=LDT szelektor index)**
- **a veremszegmens szelektor a megfelelő deszkriptor tábla határán túlmutat (index=veremszegmens szelektor index)**
- **a veremszegmens nem írható (index=veremszegmens szelektor index)**
- **veremszegmens DPL≠CPL (index=veremszegmens szelektor index)**
- **veremszegmens szelektor RPL≠CPL (index=verem-szegmens szelektor index)**
- **a kódszegmens szelektor a megfelelő deszkriptor tábla határán túlmutat (index=kódszegmens szelektor index)**
- **a kódszegmens nem végrehajtható (index=kódszegmens szelektor index)**
- **a nem illeszkedő (nonconforming) kódszegmens DPL≠ CPL (index=veremszegmens szelektor index)**

- **illeszkedő kódszegmens DPL>CPL (index=veremszegmens szelektor index)**
- **az adatszegmens szelektor a megfelelő deszkriptor tábla határán túlmutat (index=adatszegmens szelektor index)**
- **az adatszegmens nem olvasható (index=adatszegmens szelektor index)**

A hiba észlelés történhet a taszkváltást kezdeményező taszk állapotában vagy az új taszk környezetében is. Az elmentett CS:IP érték ennek megfelelően vagy a taszkváltást okozó utasításra, vagy az új taszk első utasítására mutat. A kivétel hibakódjának index mezője a hibát okozó szelektor indexét tartalmazza.

11-es kivétel akkor keletkezik, ha az elért szegmens vagy kapu deszkriptor "jelenlevő" (present) bitje törölve van. A következő esetekben keletkezik ez a kivétel:

- **CS, DS vagy ES szegmensregiszterek betöltésekor**
- **LLDT utasítás esetén**
- **LTR utasítás esetén, ha a TSS nem jelenlevőnek van megjelölve**
- **kapu deszkriptor vagy TSS használatakor, ha az nem jelenlevőnek van bejelölve, de egyébként érvényes információt tartalmaz**

A hibakód index mezője a hibát okozó deszkriptor szelektor indexét tartalmazza. Ha a kivétel taszkváltás közben, a TSS-ben levő szelektorokhoz tartozó deszkriptorok betöltésekor történt, akkor a veremben levő CS:IP az új taszk első utasítására mutat. Minden egyéb esetben azonban a hibázó utasítás címét tartalmazza CS:IP veremben levő példánya. Ezt a kivételt a szegmens szintű virtuális memória megvalósítására használhatja az operációs rendszer.

12-es kivétel a következő esetekben következik be:

- **szegmenshatár-sértés történt a POP, PUSH, CALL, RET, IRET, ENTER, LEAVE utasításoknál, vagy egyéb, a vermet megcímző utasítás végrehajtásakor**
- **SS-be egy nem jelenlevő szegmenst próbáltunk meg betölteni**

Ha a hiba nem jelenlevő szegmens miatt történt, vagy taszkváltáskor az új taszk verme túlcsordult (határt sértett), a hibakód index mezője a hibát okozó deszkriptor szelektor indexét tartalmazza. Egyéb esetben azonban 0000h a hibakód. Ha a kivétel taszkváltás közben történt, akkor a veremben levő CS:IP az új taszk első utasítására mutat. Minden egyéb esetben a hibázó utasítás címét tartalmazza CS:IP veremben levő példánya.

13-as kivételt okoz minden olyan érvénytelen, a védelmet megsértő művelet, ami más védelmi kivételt (9, 10, 11, 12) nem okoz. Ha a kivétel egy szegmens deszkriptor betöltésekor keletkezett, a hibakód index mezője a hivatkozott deszkriptorra mutató szelektor indexét tartalmazza. Egyéb esetekben a hibakód 0000h lesz. A verembe mentett CS:IP mindig a hibázó utasításra mutat. A következő feltételek valamelyikének teljesülése esetén keletkezik 13-as kivétel:

- **szegmenshatársértés a CS, DS, ES szegmensek használatakor**
- **deszkriptor tábla határának megsértése (taszkváltást és veremváltást kivéve)**
- **vezérlésátadás egy nem végrehajtható szegmensre**
- **kódszegmensbe vagy írásvédett (read-only) adatszegmensbe próbáltunk meg írni**
- **egy csak végrehajtható (execute-only) kódszegmensből próbáltunk meg olvasni**
- **SS-be egy csak olvasható szegmens szelektorát próbáltuk meg betölteni (kivéve, ha ez taszkváltáskor történik, mert ilyenkor 10-es kivétel keletkezik)**

- **egy rendszerszegmens szelektorát próbáltuk meg betölteni a DS, ES, SS regiszterekbe**
- **egy csak végrehajtható (execute-only) kódszegmens szelektorát töltöttük be a DS, ES regiszterekbe**
- **SS-be a nullszelektort vagy egy csak végrehajtható szegmens szelektorát töltöttük be**
- **CS-be a nullszelektort vagy egy adatszegmens szelektorát töltöttük be**
- **nullszelektort tartalmazó DS, ES regiszterekkel történő memóriaelérés**
- **foglalt (busy) taszkra történő ugrás vagy annak meghívása**
- **egy szabad (nonbusy) taszkra történő visszatérés az IRET utasítással**
- **taszkváltáskor egy olyan szegmens szelektor használata, ami az aktuális LDT-beli TSS-re mutat**
- **valamelyik privilegizálási védelmi szabály megsértése**
- **megszakítás vagy kivétel esetén az IDT-nek egy olyan bejegyzését használtuk, ami nem megszakítás-, csapda-, vagy taszkkapu (interrupt, trap, task gate)**
- **egy privilegizált utasítást próbáltunk meg végrehajtani, amikor CPL$\neq$0**
- **egy nullszelektort tartalmazó kaput értünk el**
- **egy INT utasítást hajtottunk végre úgy, hogy a cél deszkriptor DPL-je kisebb mint a CPL**
- **egy call-, megszakítás- vagy csapdakapu (call, interrupt, trap gate) szegmens szelektora nem kódszegmensre mutat**
- **az LLDT utasítás operandusa nem GDT-beli szelektor, vagy nem egy LDT deszkriptorára mutat**
- **az LTR utasítás operandusa LDT-beli, vagy egy foglalt (busy) taszk TSS-ére mutat**

- **egy hívás (call), ugrás (jump) vagy visszatérés (return) alkalmával a cél kódszegmens szelektora nullszelektor**

Bár a 80286-os processzor támogatja a 16-os kivételt, ez a kivétel nem keletkezhet hardveres úton ezen a processzoron. A 16-os kivétel egyébként azt jelzi, hogy egy numerikus kivétel (#IA, #IS, #D, #Z, #O, #U vagy #P) keletkezett a legutolsó FPU utasítás végrehajtásakor. Hibakód nem kerül a verembe. A CS:IP-másolat a hibát okozó utasítás után következő FPU utasításra mutat, a tényleges címet a koprocesszor utasítás-mutató regisztere tartalmazza. Lásd még a 80486-os processzor fejezetében levő megjegyzést is.

17.4.7 Változások a regiszterkészletben

A Flags regiszter két új mezőt kapott:

Bitpozíció	*Név*
12..13	**IOPL**
14	**NT**

Az IOPL (Input/Output Privilege Level) egy kétbites mező. Tartalma néhány olyan utasítás végrehajtását befolyásolja, amelyek korlátlan használata védett módban különös veszélyt jelenthetne az operációs rendszerre. Ilyen pl. a CLI, IN vagy OUT utasítás. Ha az aktuális privilegizálási szint (CPL) nagyobb, mint az IOPL, az IOPL-érzékeny utasítások végrehajtását megtagadja a processzor, és 13-as kivétel (#GP) keletkezik.

Az NT (Nested Task) bit jelzi, ha az aktuális taszkot CALL utasítással, megszakítás vagy kivétel miatt hívták meg.

Mindkét bitnek védett módban, multitaszkos környezetben van szerepe.

A szegmensregiszterek neve védett módban szegmens szelektor regiszter lett. Ezek a regiszterek ilyenkor szegmens-báziscím helyett szelektort (selector) tartalmaznak:

Bitpozíció	*Név*
0..1	**RPL**
2	**TI**
3..15	**Index**

A kétbites RPL (Requested Privilege Level) mezőnek a deszkriptorok és azokon keresztül a szegmensek eléréséné szükséges védelmi ellenőrzések során van szerepe.

Ha a TI (Table Indicator) értéke 0, akkor a GDT, különben az aktuális LDT egy deszkriptorát választja ki a szelektor.

Az Index mező tartalmazza a megfelelő deszkriptor tábla kívánt elemének sorszámát.

17.4.8 Új regiszterek

Mindegyik szegmensregiszterhez (CS, DS, ES, SS) be lett vezetve egy *szegmens deszkriptor cache-regiszter* (segment descriptor cache register). Ezek a speciális regiszterek nem érhetők el programból, ezért sokszor hívják őket *árnyékregisz-tereknek* (shadow register) is. Feladatuk, hogy az adott szegmensregiszter által mutatott szegmens báziscímét, méretét, hozzáférési jogait tárolják, ezzel felgyorsítva a memória elérését. Fontos, hogy tartalmuk NEM törlődik a processzor módváltása közben, így pl. a valós módból védettbe átkapcsolva tartalmuk megmarad. (Ennek ellentmond az a tény, hogy az MSW regiszter PE bitje nem törölhető az LMSW utasítással, így a processzor kényszerű resetelése mégiscsak törli a szegmens deszkriptor cache regisztereket.)

A gépi állapotszó regiszter (Machine Status Word register–*MSW*) egy 16 bites rendszerregiszter. Felépítése az alábbi táblázatban látható:

Bitpozíció	*Név*
0	PE
1	MP
2	EM
3	TS
4..15	–

A PE (Protection Enable) bitet 1-be állítva juthatunk védett módba. A bitet csak reset segítségével törölhetjük ezután.

Az MP (Monitor coProcessor, vagy Math Present) bit állásától függ, hogy a WAIT/FWAIT utasítás végrehajtása generál-e 7-es (#NM) kivételt, ha az MSW regiszter TS bitje be van állítva. Ha MP=1, akkor keletkezik kivétel az említett feltétel esetén, különben nem.

Ha az EM (EMulation) bit be van állítva, akkor bármely FPU utasítás végrehajtása 7-es kivételt vált ki.

A TS (Task Switched) bitet minden taszkváltáskor beállítja a CPU. Lásd még fentebb a 7-es kivétel leírását.

A 4..15 bitek fenntartottak. Az MSW regisztert az LMSW privilegizált utasítással írhatjuk, ill. tartalmát bármikor kiolvashatjuk az SMSW utasítással. A PE bitet csak beállítani lehet az LMSW-vel, a törléshez resetelni kell a processzort.

A GDTR (Global Descriptor Table Register) és IDTR (Interrupt Descriptor Table Register) regiszterek a GDT ill. IDT báziscímét és méretét (határát) tartalmazzák:

Bitpozíció	*Tartalom*
0..15	Határ
16..39	Báziscím
40..47	–

A 0..15 bitek a 16 bites határt (azaz a táblázat szegmensének legutolsó megcímezhető offszetjét), míg a 16..39 bitek a tábla szegmensének 24 bites fizikai báziscímét tartalmazzák. A 40..47 bitek fenntartottak.

Az LDTR (Local Descriptor Table Register) az aktív LDT szelektorát tartalmazó 16 bites regiszter. A szelektornak a GDT-be kell mutatnia.

Az TR (Task Register) az aktív taszk TSS-ének (Task State Segment) szelektorát tartalmazó 16 bites regiszter. A szelektornak a GDT-be kell mutatnia.

Az LDTR és a TR regiszterek igazából az adott szegmens deszkriptorának minden fontos adatát tárolják, de ebből csak a 16 bites szelektor rész látható és érhető el a programok számára.

A GDTR, IDTR, LDTR és TR regisztereknek védett módban van jelentőségük.

17.5 Az Intel 80386-os processzor újításai

17.5.1 Változások az utasítások értelmezésében

Mostantól az utasítások maximális hossza gépi kódú alakban (beleértve a prefixeket, közvetlen operandusokat, eltolási értékeket is) 15 bájt lehet, az ennél hosszabb utasítások 13-as kivételt (#GP) váltanak ki. Ezt a határt csak redundáns prefixekkel sérthetjük meg (azaz ha ugyanabból a kategóriából

többféle prefixet használunk, pl. két szegmensprefixet adunk meg egyetlen utasítás előtt).

A mnemonikok elnevezésében bevezettek egy új konvenciót. Egy utasítás mnemonikja után "W" betűt írva az adott utasítás 16 bites, míg "D" betűt írva a 32 bites operandusméretű változatot kaphatjuk. A régi, kiegészítés nélküli mnemonik használata esetén az assemblerre bízzuk, az utasításnak mely változatát fogja generálni. Nem mindegyik assembler támogatja ezeket a mnemonikokat. Egy jó példa az ENTER-ENTERW-ENTERD trió. ENTERW esetén a 16 bites, ENTERD esetén a 32 bites változatot kapjuk, míg az ENTER mnemonik a kódszegmens operandusméret jellemzője (attribútuma) alapján jelentheti a 16 vagy a 32 bites változatot is.

17.5.2 Változások az utasításkészletben

- **CLI (no op)**
- **IN AL/AX/EAX,imm8/DX**
- **INS dest,DX**
- **INT imm8**
- **INTO (no op)**
- **IRET (no op)**
- **OUT imm8/DX,AL/AX/EAX**
- **OUTS DX,source**
- **POPF (no op)**
- **PUSHF (no op)**
- **STI (no op)**

  Ezek az utasítások IOPL-érzékenyek. Az INT, INTO, IRET, POPF, PUSHF utasítások csak a virtuális-86 (V86) módban, az IN, INS, OUT, és OUTS utasítások csak védett módban (V86-ban nem), míg a CLI és STI utasítások védett és virtuális

módban is érzékenyek. Ha a Flags regiszter IOPL mezőjének (12..13 bitek) tartalma nagyobb vagy egyenlő, mint a CPL (Current Privilege Level), akkor az utasítás végrehajtható, különben pedig 13-as kivétel (#GP) keletkezik.

- ARPL dest selector,source selector reg16
- LAR access rights reg16,source selector
- LLDT source selector
- LSL limit reg16,selector reg/mem16
- LTR source selector
- SLDT dest selector
- STR dest selector
- VERR source selector
- VERW source selector

Ezek az utasítások virtuális üzemmódban is 6-os kivételt (#UD) váltanak ki.

- ENTER imm16,imm8
- IRET (no op)
- LEAVE (no op)
- POPA (no op)
- POPF (no op)
- PUSHA (no op)
- PUSHF (no op)

Ha az operandusméret 32 bit, az utasítások működése eltérő. Lásd az ENTERD, IRETD, LEAVED, POPAD, POPFD, PUSHAD, PUSHFD utasítások leírását.

- JCXZ relative address
- LOOP relative address
- LOOPE/LOOPZ relative address
- LOOPNE/LOOPNZ relative address

Ha a címméret 32 bit, az utasítások működése eltérő. Lásd a JECXZ, LOOPD, LOOPDE/ LOOPDZ, LOOPDNE/LOOPDNZ utasítások leírását.

- **CMPS source string,dest string**
- **INS dest string,DX**
- **LODS source string**
- **MOVS dest string,source string**
- **OUTS DX,source string**
- **SCAS dest string**
- **STOS dest string**

  Ha a címméret 32 bites, a sztringutasítások SI és DI helyett ESI-t és EDI-t használják index-regiszterként.
- **BOUND dest reg16/32,source mem**

  Az utasítás céloperandusa 32 bites regiszter is lehet, a memóriaoperandus ilyenkor két duplaszóból fog állni.
- **CALL address**

  Az utasítás mostantól támogatja a 32 bites offszetet használó címeket is.
- **DIV source reg/mem32**

  Ha az operandus 32 bites, EDX:EAX-et osztja el a forrással, a hányadost EAX-be, a maradékot pedig EDX-be teszi.
- **IDIV source reg/mem32**

  Ha az operandus 32 bites, EDX:EAX-et osztja el a forrással, a hányadost EAX-be, a maradékot pedig EDX-be teszi.
- **IMUL source reg/mem32**

  Ha az operandus 32 bites, EAX-et megszorozza a forrással, a szorzatot pedig EDX:EAX-ban eltárolja.
- **IMUL dest reg32,source reg/mem32,imm8/imm32**

  A cél és a forrás is lehet 32 bites, a harmadik közvetlen operandus ilyenkor bájt vagy duplaszó méretű kell legyen. Működése egyébként megfelel a korábbinak.

- **IMUL dest reg32,imm8/imm32**

  Ez a forma az IMUL reg32,reg32,imm8/imm32 utasítást rövidíti, ahol a cél és a forrás is ugyanaz a 32 bites általános regiszter.
- **IMUL dest reg16,source reg/mem16**
- **IMUL dest reg32,source reg/mem32**

  Az IMUL utasításnak van egy új kétoperandusú formája is. A cél egy 16 vagy 32 bites általános regiszter lehet, míg a forrás szintén 16 vagy 32 bites általános regiszter vagy memóriahivatkozás kell legyen. Működése: a forrást összeszorozza a céllal, az eredményt pedig a célban eltárolja.
- **IN EAX,imm8/DX**

  A cél a 32 bites akkumulátor is lehet.
- **INT 3**

  Az egybájtos INT 3 utasítás (0CCh) nem IOPL-érzékeny virtuális módban. A kétbájtos alakra (0CDh, 03h) ez nem vonatkozik, az továbbra is IOPL-érzékeny marad.
- **Jccc relative address**

  A célutasítás relatív címe most már 16 ill. 32 bites is lehet. A cím méretét a kódszegmens címméret jellemzője (attribútuma) határozza meg. A "ccc" a szokásos feltételek valamelyikét jelentheti (pl. JE=JZ, JO, JP, JAE, JNGE stb.)
- **JMP address**

  Az utasítás mostantól támogatja a 32 bites offszetet használó címeket is.
- **LAR access rights reg32,source selector**

  Ha a cél 32 bites, a forrás szelektorhoz tartozó deszkriptor 32..63 bitjeinek értékét 00F0FF00h-val maszkolva (logikai ÉS kapcsolatba hozva) a cél általános regiszterbe tölti. Sikeres olvasás

esetén ZF-et beállítja, különben ZF értéke 0 lesz. Ez az utasítás nem privilegizált.

- **LDS dest reg32,source**

  Ha a cél regiszter 32 bites, a forrást 16:32 alakú távoli pointernek értelmezi. Az alsó duplaszót (0..3 bájtok) a cél általános regiszterbe, a felső szót (4..5 bájtok) pedig a DS-be tölti.

- **LEA dest reg16/32,source**

  A LEA utasítás működése az operandusméret és címméret jellemzők függvényében más és más:

Op.	Cím	Hatás
16	16	a 16 bites EA kerül a 16 bites célba
16	32	a 32 bites EA alsó 16 bitje kerül a 16 bites célba
32	16	a 16 bites EA zérókiterjesztve kerül a 32 bites célba
32	32	a 32 bites EA kerül a 32 bites célba

A cél lehet 32 bites általános regiszter is.

- **LES dest reg32,source**

  Ha a cél regiszter 32 bites, a forrást 16:32 alakú távoli pointernek értelmezi. Az alsó duplaszót (0..3 bájtok) a cél általános regiszterbe, a felső szót (4..5 bájtok) pedig az ES-be tölti.

- **LGDT source mem**
- **LIDT source mem**

  Ez a két utasítás az operandusméret aktuális állapota szerint különböző módon működik. Ha az operandusméret 16 bit, a szokásos módon figyelmen kívül hagyják az operandus legfelső bájtját, egyébként pedig a teljes 32 bites báziscímet betöltik.

- **LMSW source status word**

A 80286-ossal való kompatibilitás megőrzése végett a CR0/MSW regiszter ET (4-es) bitjét nem módosítja az LMSW utasítás.

- **LOCK**

  Ez a prefix mostantól NEM IOPL-érzékeny.
- **LOCK**

  Ez a prefix mostantól csak a ADC, ADD, AND, BTC, BTR, BTS, DEC, INC, NEG, NOT, OR, SBB, SUB, XCHG, XOR utasításokkal használható, különben 6-os (#UD) kivétel keletkezik.
- **LSL limit reg32,selector reg/mem32**

  A cél lehet 32 bites regiszter is, ilyenkor a szelektor által kijelölt szegmens határának bájtokban mért 32 bites értékékét tölti be a célba. Ha a határ 32 bit méretű, de a cél csak 16 bites regiszter, a határ alsó szavát tölti a cél regiszterbe.
- **MOV FS,source reg/mem16**
- **MOV GS,source reg/mem16**
- **MOV dest reg/mem16,FS**
- **MOV dest reg/mem16,GS**

  A két új szegmensregiszter (FS és GS), valamint egy 16 bites általános regiszter vagy memóriaoperandus közötti adatcserét bonyolítják le a MOV utasítás új formái.
- **MOV dest reg32,CRx**
- **MOV dest reg32,DRx**
- **MOV dest reg32,TRx**
- **MOV CRx,source reg32**
- **MOV DRx,source reg32**
- **MOV TRx,source reg32**

  A MOV utasítás új alakjai képesek a vezérlő-(control), nyomkövető (debug) és tesztregiszterek (test) és valamelyik 32 bites általános regiszter között adatot mozgatni. Ezek az utasítások

privilegizáltak, így vagy valós módban, vagy védett módban CPL=0 mellett hajthatók végre. CR0 alsó szavának olvasására használjuk inkább az SMSW utasítást, mivel az nem privilegizált.

- **MUL source reg/mem32**

  Ha az operandus 32 bites, EAX-et megszorozza a forrással, a szorzatot pedig EDX:EAX-ban eltárolja.
- **NOP (no op)**

  Védett módban, ha a kódszegmens operandusméret jellemzője 32 bit, ez a mnemonik az XCHG EAX,EAX utasításra lesz lefordítva, aminek gépi kódja ilyenkor ugyancsak 90h. 16 bites kód-szegmens esetén az XCHG EAX,EAX utasítás a 66h, 90h bájtokra fordul le.
- **OUT imm8/DX,EAX**

  A forrás a 32 bites akkumulátor is lehet.
- **POP FS**
- **POP GS**
- **POP reg/mem32**

  A POP utasítás képes a két új szegmensregisztert (FS és GS), továbbá bármelyik 32 bites általános regisztert a verem tetejéről kiszedni. Duplaszó méretű memóriaoperandus szintén betölthető a veremből.
- **POP DS/ES/FS/GS/SS**

  Ha a veremszegmens címméret jellemzője 32 bit, a kiolvasott duplaszónak az alsó szava lesz betöltve az adott szegmensregiszterbe. A felső szót figyel-men kívül hagyja az utasítás.
- **POP dest**

  Ha a veremszegmens címméret jellemzője 32 bit, a POP utasítás SP helyett ESP-t használja.
- **POPF (no op)**

Virtuális módban az IOPL mező nem módosítható Flags-ben, hanem változatlan marad. Lásd még a korábbi megjegyzést az IOPL-érzékenységről.

- **PUSH FS**
- **PUSH GS**
- **PUSH reg/mem32**
- **PUSH imm8**
- **PUSH imm32**

A PUSH utasítás képes a két új szegmensregisztert (FS és GS), továbbá bármelyik 32 bites általános regisztert a verembe rakni. Duplaszó méretű memóriaoperandus vagy 32 bites közvetlen érték szintén verembe tehető. 32 bites címméretű verem esetén a PUSH imm8 utasítás a bájt méretű operandust előjelesen kiterjeszti duplaszóvá, majd ez utóbbit rakja a verem tetejére.

- **PUSH CS/DS/ES/FS/GS/SS**

Ha a veremszegmens címméret jellemzője 32 bit, olyan duplaszó lesz a verembe rakva, amelynek az alsó szava tartalmazza az adott szegmensregisztert, a felső szó értéke pedig meghatározatlan.

- **PUSH source**

Ha a veremszegmens címméret jellemzője 32 bit, a PUSH utasítás SP helyett ESP-t használja.

- **PUSH ESP**

Az utasítás működése hasonló a PUSH SP működéséhez:

```
MOV     TEMP,ESP
SUB     SP/ESP,00000004h
MOV     SS:[SP/ESP],TEMP
```

A TEMP itt egy 32 bites ideiglenes változót jelöl.

- **REP**
- **REPE/REPZ**

- **REPNE/REPNZ**

  Ha a címméret jellemző 32 bit, a prefixek CX helyett ECX-et használják.
- **SGDT source mem**
- **SIDT source mem**

  Ez a két utasítás az operandusméret aktuális állapota szerint különböző módon működik. Ha az operandusméret 16 bit, figyelmen kívül hagyják az operandus legfelső bájtját, de a 80286-tól eltérően 00h-t tárolnak a helyére. Ha az operandusméret 32 bit, a teljes 32 bites báziscímet eltárolják.
- **XCHG dest,source**

  Ha valamelyik operandus memóriahivatkozás, az XCHG utasítás automatikusan alkalmazza a buszlezárást a LOCK prefix használatától függetlenül.
- **XLAT (no op)**
- **XLAT table address**

  Ha a címméret 32 bites, az utasítás a DS:(EBX+AL) címen levő bájtot tölti be AL-be. AL-t továbbra is előjeltelenül értelmezi, és az alapértelmezett DS szegmens is felülbírálható.

Az összes vezérlésátadó utasítás eltérően viselkedik, ha az operandusméret 32 bites. Ez a következő dolgokat jelenti:

- **IP helyett EIP változik meg mindegyik utasítás esetében**
- **a CALL, JMP utasítások 32 bites relatív vagy abszolút offszetű címet és 16:32 alakú távoli mutatót használnak operandusként**
- **a vermet használó utasítások IP és Flags helyett EIP és EFlags tartalmát rakják a verembe vagy veszik ki onnan**

Virtuális módban a JMP, CALL, INT, INTO, IRET, IRETD és RET utasítások működése sokkal összetettebb, de erről a védett módról szóló fejezetben szólunk majd bővebben.

A következő utasítások esetében is lehetséges 32 bites operandusok (közvetlen érték, memóriahivatkozás vagy általános regiszter) használata: ADC, ADD, AND, CMP, DEC, INC, MOV, NEG, NOT, OR, RCL, RCR, ROL, ROR, SAL, SAR, SBB, SHL, SHR, SUB, TEST, XCHG, XOR.

17.5.3 Új prefixek

- **64h**
  (FS Segment override prefix)
- **65h**
  (GS Segment override prefix)
  A többi szegmensfelülbíráló prefixhez hasonlóan az alapértelmezett DS szegmens helyett az FS ill. GS szegmensregiszter használatát írják elő.
- **66h**
  (Operand size prefix)
  Az utasítás (ill. a kódszegmens) alapértelmezett operandusméretét állítja át 16 bitesről 32 bitesre, vagy fordítva. Csak arra az utasítással van hatással, ami előtt alkalmazzuk.
- **67h**
  (Address size prefix)
  Az utasítás (ill. a kódszegmens) alapértelmezett címméretét állítja át 16 bitesről 32 bitesre, vagy fordítva. A használt címzési módot és néhány egyéb dolgot befolyásol. Csak arra az utasítással van hatással, ami előtt alkalmazzuk.

17.5.4 Új utasítások

- **BSF dest reg16/32,source**
  (Bit Scan Forward)
  A forrásban megkeresi azt a legalacsonyabb helyiértékű bitet, ami be van állítva. Ha mindegyik bit 0 volt a forrásban, ZF-et 1-re állítja, a cél tartalma meghatározatlan. Különben pedig ZF-et törli, majd a bit sorszámát a cél általános regiszterbe teszi. CP, PF, AF, SF és OF tartalma meghatározatlan az utasítás végrehajtása után. A forrás 16 vagy 32 bites általános regiszter vagy memóriaoperandus lehet.
- **BSR dest reg16/32,source**
  (Bit Scan Reverse)
  A forrásban megkeresi azt a legmagasabb helyiértékű bitet, ami be van állítva. Ha mindegyik bit 0 volt a forrásban, ZF-et 1-re állítja, a cél tartalma meghatározatlan. Különben pedig ZF-et törli, majd a bit sorszámát a cél általános regiszterbe teszi. CP, PF, AF, SF és OF tartalma meghatározatlan az utasítás végrehajtása után. A forrás 16 vagy 32 bites általános regiszter vagy memóriaoperandus lehet.
- **BT dest reg/mem,bit mask reg16/32**
- **BT dest reg/mem,bit mask imm8**
  (Bit Test)
- **BTC dest reg/mem,bit mask reg16/32**
- **BTC dest reg/mem,bit mask imm8**
  (Bit Test and Complement)
- **BTR dest reg/mem,bit mask reg16/32**
- **BTR dest reg/mem,bit mask imm8**
  (Bit Test and Reset)
- **BTS dest reg/mem,bit mask reg16/32**

- **BTS dest reg/mem,bit mask imm8**
  (Bit Test and Set)
  A cél operandus megadott sorszámú bitjét CF-be másolják. A forrás operandus tartalmazza a kiolvasandó bit számát. A kiolvasás után a BTC komplementálja (negálja), a BTR törli (0-ra állítja), míg a BTS beállítja a célban az adott bitet. A forrás bájt méretű közvetlen érték, ill. 16 vagy 32 bites általános regiszter lehet, míg célként 16 vagy 32 bites általános regisztert vagy memóriaoperandust írhatunk.
- **CDQ (no op)**
  (Convert Doubleword to Quadword)
  Az EAX-ban levő számot előjelesen kiterjeszti EDX:EAX-be.
- **CMPSD (no op)**
  (CoMPare String Doubleword)
  A CMPS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.
- **CWDE (no op)**
  (Convert byte to doubleword)
  Az AL-ben levő értéket előjelesen kiterjeszti EAX-be. (A mnemonik a CWD utasítás nevéből származik.)
- **ENTERD imm16,imm8**
  (ENTER new stack frame, 32 bit)
  Az ENTER utasítás 32 bites alakja. Az első, szó méretű operandus neve legyen LOCALS, a második, bájt méretű operandusé pedig NESTING. Az utasítás működését a következő pszeudokód mutatja be:

```
Push EBP
TEMP:=ESP
NESTING:=NESTING Mod 32
While (NESTING>0) DO
```

```
    NESTING:=NESTING-1
    EBP:=EBP-4
    Push SS:[EBP]
EBP:=TEMP
ESP:=ESP-LOCALS
```

Az utolsó sorban a kivonást a LOCALS zérókiterjesztett értékével végzi el. A TEMP egy 32 bites ideiglenes változót jelöl. A NESTING operandusnak csak az alsó 5 bitjét veszi figyelembe. Az utasítás párja a LEAVED, ami a legutóbbi ENTERD által létrehozott veremkeretet szabadítja fel.

- **INSD (no op)**
  (INput String Doubleword)
  Az INS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.
- **IRETD (no op)**
  (Interrupt RETurn, 32 bit)
  Az IRET utasítás 32 bites alakja. Működése valós módban szimbolikusan: POP EIP // POP CS // POPFD. Védett és virtuális módban az IRET-hez hasonlóan sokkal több dolgot csinál mint valós módban, de erről a védett módról szóló fejezetben szólunk majd.
- **JECXZ relative address**
  (Jump if ECX is/equals Zero)
  A JCXZ utasítás 32 bites alakja. Ha ECX=00000000h, az operandusban megadott címre ugrik, különben a JECXZ-t követő utasítás végrehajtásával folytatja.
- **LEAVED (no op)**
  (LEAVE stack frame, 32 bit)

A LEAVE utasítás 32 bites alakja. A legutolsó ENTERD utasítás tevékenységét szünteti meg a következő utasítások végrehajtásával:

```
MOV     ESP,EBP
POP     EBP
```

- **LFS dest reg16/reg32,source**
  (Load full pointer into FS and a general purpose register)
  Ha a cél regiszter 16 bites, a forrást a megszokott 16 bites szelektorból (vagy szegmensből) és 16 bites offszetből álló távoli mutatónak értelmezi. Az alsó szót (0..1 bájtok) ilyenkor a cél általános regiszterbe, a felső szót (2..3 bájtok) pedig FS-be tölti. Ha a cél regiszter 32 bites, a forrást 16:32 alakú távoli pointernek értelmezi. Ekkor az alsó duplaszót (0..3 bájtok) a cél regiszterbe, a felső szót (4..5 bájtok) pedig FS-be tölti.
- **LGS dest reg16/reg32,source**
  (Load full pointer into GS and a general purpose register)
  Ha a cél regiszter 16 bites, a forrást a megszokott 16 bites szelektorból (vagy szegmensből) és 16 bites offszetből álló távoli mutatónak értelmezi. Az alsó szót (0..1 bájtok) ilyenkor a cél általános regiszterbe, a felső szót (2..3 bájtok) pedig GS-be tölti. Ha a cél regiszter 32 bites, a forrást 16:32 alakú távoli pointernek értelmezi. Ekkor az alsó duplaszót (0..3 bájtok) a cél regiszterbe, a felső szót (4..5 bájtok) pedig GS-be tölti.
- **LODSD (no op)**
  (LOaD String Doubleword)
  A LODS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.

- **LOOPD relative address**
  (LOOP, 32 bit)
- **LOOPDE/LOOPDZ relative address**
  (LOOP while Equal/Zero, 32 bit)
- **LOOPDNE/LOOPDNZ relative address**
  (LOOP while Not Equal/Not Zero, 32 bit)

  A LOOP, LOOPE/LOOPZ és LOOPNE/ LOOPNZ utasítások 32 bites alakjai. Működésük csak abban tér el elődeikétől, hogy CX helyett ECX-et használják számlálóként.
- **LSS dest reg16/reg32,source**
  (Load full pointer into SS and a general purpose register)

  Ha a cél regiszter 16 bites, a forrást a megszokott 16 bites szelektorból (vagy szegmensből) és 16 bites offszetből álló távoli mutatónak értelmezi. Az alsó szót (0..1 bájtok) ilyenkor a cél általános regiszterbe, a felső szót (2..3 bájtok) pedig SS-be tölti. Ha a cél regiszter 32 bites, a forrást 16:32 alakú távoli pointernek értelmezi. Ekkor az alsó duplaszót (0..3 bájtok) a cél regiszterbe, a felső szót (4..5 bájtok) pedig SS-be tölti.

  Az LSS SP/ESP,? utasítást ajánlott használni a MOV SS,? // MOV SP/ESP,? utasításpár helyett olyankor, amikor a verem címe egy memóriában levő változóban található.
- **MOVSD (no op)**
  (MOVe String Doubleword)

  A MOVS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.
- **MOVSX dest reg16/32,source**
  (MOVe with Sign eXtension)

A bájt vagy szó méretű operandust előjelesen kiterjeszti 16 vagy 32 bitesre, majd az értéket eltárolja a cél általános regiszterben. A forrás csak általános regiszter vagy memóriaoperandus lehet.

- **MOVZX dest reg16/32,source**
  (MOVe with Zero eXtension)
  A bájt vagy szó méretű operandust zérókiterjesztéssel 16 vagy 32 bitesre egészíti ki, majd az értéket eltárolja a cél általános regiszterben. A forrás csak általános regiszter vagy memóriaoperandus lehet.
- **OUTSD (no op)**
  (OUTput String Doubleword)
  Az OUTS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.
- **POPAD (no op)**
  (POP All general purpose registers, 32 bit)
  A POPA utasítás 32 bites alakja. A következő utasítássorozatot hajtja végre:

```
POP     EDI
POP     ESI
POP     EBP
ADD     SP/ESP,00000004h
POP     EBX
POP     EDX
POP     ECX
POP     EAX
```

A negyedik sorban szereplő ADD SP/ESP,4 utasítás csak szimbolikusan néz így ki, mivel a flag-ek tartalmát NEM befolyásolja a POPAD utasítás.

- **POPFD (no op)**
  (POP EFlags)

A POPF utasítás 32 bites alakja. A verem tetejéről leemel 4 bájtot, majd az értéket az EFlags regiszterbe tölti. A processzor aktuális üzemmódjától és a CPL-től függően az utasítás másképp viselkedik. Valós módban és védett módban CPL=0 esetén az összes flag módosítható, kivéve a fenntartott biteket és a VM (17-es) flag-et. A VM flag állapota változatlan marad. Védett módban, ha 0<CPL≤IOPL, az IOPL mezőt és a VM flag-et kivéve bármelyik flag módosítható. Az IOPL és VM bitek változatlanok maradnak. Védett módban, ha CPL>0 és CPL>IOPL, nem keletkezik kivétel, és az említett bitek is változatlanok maradnak. Virtuális módban, ha IOPL=3, a VM (17-es), RF (16-os) flag-ek és az IOPL mező (12..13 bitek) változatlanok maradnak, ezek nem módosíthatók. A POPF utasításhoz hasonlóan ez az utasítás is IOPL-érzékeny virtuális módban.

- **PUSHAD (no op)**
  (PUSH All general purpose registers, 32 bit)
  A PUSHA utasítás 32 bites alakja. A következő utasításokat hajtja végre:

```
PUSH    EAX
PUSH    ECX
PUSH    EDX
PUSH    EBX
PUSH    ESP
PUSH    EBP
PUSH    ESI
PUSH    EDI
```

Az ötödik sorban szereplő utasítást úgy hajtja végre, ahogyan a "hagyományos" PUSH ESP esetén is

teszi, tehát ESP aktuális értéke lesz eltárolva az SS:(SP-4) vagy SS:(ESP-4) címen, majd SP vagy ESP értéke csökken 4-gyel.

- **PUSHFD (no op)**
  (PUSH EFlags)
  A PUSHF utasítás 32 bites alakja. Az EFlags regisztert a verem tetejére teszi. A PUSHF-hez hasonlóan virtuális módban IOPL-érzékeny ez az utasítás.
- **SCASD (no op)**
  (SCAn String Doubleword)
  A SCAS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.
- **SETccc dest reg/mem8**
  (SET byte on condition)
  Ha a "ccc" által megadott feltétel teljesül, a célba 01h-t tölt, különben pedig 00h lesz a cél tartalma. A feltételek ugyanazok lehetnek, mint a feltételes ugrásoknál, a CXZ-t kivéve. A célnak egy bájt méretű általános regiszternek vagy memória-operandusnak kell lennie.
- **SHLD dest,source reg16/32,CL/imm8**
  (Double precision SHift Left logical)
  A céloperandusban és a forrásban levő értékeket egyetlen számként kezeli úgy, hogy a forrásoperandus lesz az alacsonyabb helyiértékű. Ezután ezt a számot balra lépteti a harmadik operandusban megadott számszor, majd a végeredménynek a felső (magasabb helyiértékű) felét beírja a céloperandusba. A forrás tartalma változatlan marad. A flag-eket a többi shiftelő utasításhoz hasonlóan kezeli, de itt OF is meghatározatlan lesz. A forrásnak egy 16 vagy 32 bites általános regiszternek, a célnak 16 vagy 32 bites általános regiszternek vagy memória-

operandusnak kell lennie, a lépésszámot pedig egy bájt méretű közvetlen érték vagy a CL regiszter határozza meg. A lépésszámot modulo 32 veszi, így az a 0..31 értékek valamelyike lesz.

A következő két utasítással pl. könnyedén balra léptethetjük az EDX:EAX-ban levő 64 bites számot:

```
SHLD    EDX,EAX,CL
SHL     EAX,CL
```

- **SHRD dest,source reg,CL/imm8**
  (Double precision SHift Right logical)

  A céloperandusban és a forrásban levő értékeket egyetlen számként kezeli úgy, hogy a forrásoperandus lesz a magasabb helyiértékű. Ezután ezt a számot jobbra lépteti a harmadik operandusban megadott számszor, majd a végeredménynek az alsó (alacsonyabb helyiértékű) felét beírja a céloperandusba. A léptetés logikai lesz, tehát az operandusokat előjeltelen értékekként kezeli. A forrás tartalma változatlan marad. A flag-eket a többi shiftelő utasításhoz hasonlóan kezeli, de itt OF is meghatározatlan lesz. A forrásnak egy 16 vagy 32 bites általános regiszternek, a célnak 16 vagy 32 bites általános regiszternek vagy memóriaoperandusnak kell lennie, a lépésszámot pedig egy bájt méretű közvetlen érték vagy a CL regiszter határozza meg. A lépésszámot modulo 32 veszi, így az a 0..31 értékek valamelyike lesz.

  A következő két utasítással pl. könnyedén jobbra léptethetjük az EDX:EAX-ban levő 64 bites, előjel nélküli számot:

```
SHRD    EAX,EDX,CL
```

```
SHR     EDX,CL
```

- **STOSD (no op)**
  (STOre String Doubleword)
  A STOS utasítás rövid alakja. A sztringet duplaszó méretű elemekből állónak tekinti.

17.5.5 Változások az architektúrában

Fontos újdonságnak számít az *igény szerinti lapozáson alapuló virtuális memória* (demand-paged virtual memory). A virtuális memória egy olyan címterületet jelent, amit a felhasználói és rendszerprogramok felhasználhatnak, de annak mérete esetenként sokszorta nagyobb lehet a fizikai memóriánál. Multitaszkos és többfelhasználós operációs rendszerek ezért előszeretettel alkalmazzák. A 80286-os processzor védett módban már lehetővé tette a virtuális memória használatát, de csak a szegmensek szintjén. Mivel a védett módú szegmensek általában különböző méretűek, ez nem valami kényelmes és hatékony módszer, ezen kívül nem biztosít elég védelmet sem.

A 80386-os processzor egy új technikát, a *lapozást* (paging) is biztosítja. Ennek az a lényege, hogy a fizikai címterületet (ami ennél a processzornál 4 Gbájt) azonos méretű, összefüggő területekre osztja fel. Ezeket *lapoknak* (page) hívjuk, méretük 4 Kbájt, és mindegyik lap báziscíme 4096-tal osztható címen található (azaz a lapok a szegmensekkel ellentétben nem átfedőek). Lapozás csak a védett (és így a virtuális) módban működik. Engedélyezéséhez a CR0 regiszter PG (31-es) bitjét be kell állítanunk.

A lapozás megvalósításához új rendszerelemeket vezettek be. A *laptáblák* (page-table) 1024 db. 4 Kbájtos lapnak tartalmazzák a báziscímét, elérhetőségét és egyéb információit. A *lapcímtár* (page-directory) 1024 db. laptábla báziscímét és

egyéb adatát tartalmazza. (Báziscímen itt fizikai címet értünk.) Ez így összesen 1024*1024=1048576 db. lapot jelent, és mivel a lapok mérete 4 Kbájt=4096 bájt, ezzel kijön a 4 Gbájt=4096*1024*1024 bájtos elérhető címterület.

A laptáblák és a lapcímtár egyaránt 4 bájtos bejegyzéseket használnak, így az 1024 db. bejegyzés pont belefér egy lapba.

A lapcímtár báziscímére ugyancsak teljesülnie kell a 4096-tal való oszthatóságnak. A báziscím felső 20 bitjét a CR3 regiszter 12..31 bitjei tartalmazzák, ezért ezt a regisztert hívják még lapcímtár bázisregiszternek is (Page-Directory Base Register–PDBR).

A memória gyorsabb elérése céljából a leggyakrabban (legutóbb) használt lapcímtár-bejegyzések (Page-Directory Entry–PDE) és laptábla-bejegyzések (Page-Table Entry–PTE) speciális cache-ben, az ú.n. *Translation Lookaside Buffer*-ekben (TLBs) tárolódnak el a processzoron belül. Ezek tartalma nem érhető el a programok számára, de tartalmuk CPL=0 esetén törölhető. A TLB-k tartalma a CR3 regiszter írásakor, valamint taszkváltáskor automatikusan érvénytelenítődik.

A 32 bites lineáris cím a lapozás használatakor már nem közvetlenül a fizikai címet jelenti. A lineáris cím 22..31 bitjei választják ki a lapcímtár bejegyzését, ami egy laptáblára mutat. A 12..21 bitek határozzák meg, ennek a laptáblának mely lapjára vonatkozik a kérés. Végül a 0..11 bitek tartalmazzák a lapon belüli offszetet.

A lapozás a felhasználói- (virtuális módú) és a rendszer-programok (CPL=1 vagy 2) számára teljesen átlátszó. Csak az operációs rendszer magjában (kernel) található, a memória kezelésével foglalkozó rutinoknak és a kivétel-kezelőknek kell tudniuk a lapozás folyamatáról.

A TLB-k tesztelésére is lehetőséget ad a 80386-os a tesztregiszterek biztosításával. A TR6 a teszt parancsregiszter

(test command register), TR7 pedig a teszt adatregiszter (test data register).

A TLB egy négyutas készlet-asszociatív memória (four-way set-associative memory). Ez azt jelenti, hogy 4 bejegyzés-készlet van a processzorban, és mindegyik készlet 8 bejegyzést tartalmaz. Egy bejegyzés két részre bontható: tag és adat (data) részekre. A tag 24 bites, ebből 20 bit a 32 bites lineáris cím felső bitjeit tartalmazza, 1 bit jelzi, az adott bejegyzés érvényes-e (valid–V bit), valamint 3 attribútum-bitet is tartalmaz. Az adat tartalmazza a 32 bites fizikai cím felső 20 bitjét.

Egy bejegyzés TLB-be írását a következő módon tehetjük meg:

1. **A TR7 regiszterbe töltsük be a kívánt fizikai címet tartalmazó értéket. A HT bitnek 1-nek kell lennie, míg a REP mező értéke azt a készletet választja ki, ahová az új bejegyzés kerülni fog.**
2. **A TR6 regiszterbe töltsünk olyan értéket, amely a kívánt lineáris címet, valamint V, D, U és W jellemzőket tartalmazza. A C bitnek 0-nak kell lennie.**

Ha a TLB-bejegyzések között keresni (lookup) akarunk, az alábbi lépéseket kell végrehajtani:

1. **A TR6 regiszterbe töltsünk olyan értéket, amely a kívánt lineáris címet, valamint V, D, U és W jellemzőket tartalmazza. A C bitnek 1-nek kell lennie. Szintén ajánlott a V bitet is 1-re állítani.**
2. **Olvassuk ki, majd tároljuk el a TR7 regiszter tartalmát. Ha HT=1, a keresés sikeres volt, egyébként a regiszter többi bitjének állapota definiálatlan.**

A TLB tesztelésének ez a módszere a 80486-os processzoron még elérhető, a későbbi processzorok (a Pentium-tól kezdve) viszont modell-specifikus regisztereken (MSR)

keresztül biztosítják ezeket a lehetőségeket, sőt még ezeknél többet is.

A nyomkövetést (debugging) már az eddigi processzorok is támogatták. Ilyen lehetőségeket jelentett a Flags regiszter TF (8-as) flag-je, a TSS-ek T bitje, valamint az INT 3 utasítás. A 80386-os processzoron azonban lehetőség van ezeknél fejlettebb, *hardveres töréspontok* (hardware breakpoint) megadására. A következő feltételek mindegyike szerepelhet egy töréspont kiváltó okaként:

- **tetszőleges utasítás-végrehajtás (instruction execution), más néven utasítás-lehívás (instruction fetch)**
- **adott címen levő tetszőleges utasítás végrehajtása**
- **adott címen levő bájt, szó, duplaszó olvasása vagy írása**
- **valamelyik nyomkövető regiszter tartalmának megváltoztatására tett kísérlet (general detect)**

Bármelyik feltétel teljesülése esetén, persze ha engedélyeztük az adott töréspontot, 1-es kivétel (#DB) keletkezik. Összesen 4 független töréspontot adhatunk meg, ezek külön-külön engedélyezhetők is. A DR0..DR3 regiszterek tartalmazzák az egyes töréspontok 32 bites lineáris (nem feltétlenül fizikai) címét. A DR6 regiszter mutatja, mely töréspontok miatt keletkezett kivétel, míg a DR7 regiszter szolgál a töréspontot kiváltó feltételek beállítására.

Az új, 32 bites regiszterek és a 32 bites címzési módok bevezetése szükségessé tette, hogy a szegmensek méretét ennek megfelelően meg lehessen növelni, továbbá meg kellett adni a választási lehetőséget az új és a régi címzési módok között. Ennek megfelelően a deszkriptorok formátuma eltérő a 80286-oson használttól:

- **a szegmens határát tartalmazó mező mérete 20 bites lett**
- **a szegmens báziscíme immáron 32 bites lehet**
- **a szegmens méretének alapegysége (granularity) bájt vagy lap lehet, ennek megfelelően a határ maximum vagy 1 Mbájt lehet bájtos lépésekben, vagy 4 Gbájt 4096 bájtos lépésekben**

Szintén újdonság, hogy mindegyik szegmensnek van egy operandusméret és egy címméret jellemzője is (operand size és address size attribute). Valós és virtuális módban ez mindegyik szegmenstípusnál (adat, kód, verem) 16-16 bit. Védett módú 32 bites szegmens esetén az operandusméret és a címméret is egyaránt 32 bit. Ezek a jellemzők mindhárom szegmenstípusnál mást jelentenek.

A kódszegmensek esetén mindkét jellemző döntő hatással van a szegmens tartalmának (azaz az utasításoknak) értelmezésére. Az operandusméret az utasítások operandusának alapértelmezett méretét befolyásolja: 16 bites operandusméret esetén 8 vagy 16 bites, míg 32 bites operandusméret megadásakor 8 vagy 32 bites operandus(oka)t használnak az utasítások. A címméret az effektív cím (offszet) hosszát és a használt címzési mód jellegét befolyásolja: 16 bites címméret esetén 16 bites effektív címet és 16 bites címzési módokat használ a processzor, míg a 32 bites címméret 32 bites effektív cím és a 32 bites címzési módok használatát írja elő. Szintén teljesül, hogy 16 bites címméret esetén IP-t, különben pedig EIP-t használja utasításmutatóként a processzor.

Az adatszegmensekre csak a címméret értéke van hatással. A védett módú lefelé bővülő adatszegmensek (expand-down data segment) esetén a címméret határozza meg a szegmens felső határát. 16 bites címméret esetén a határ 0FFFFh, egyébként pedig 0FFFFFFFFh lesz. Mivel a verem egy speciális adatszegmens, ez a tulajdonság a veremre is érvényes lehet.

16 bites operandusméret szavakból álló vermet jelent, 32 bites operandusméret esetén pedig duplaszavakból fog állni a verem. 16 bites címméret esetén a verem SP-t, míg 32 bites címméretnél ESP-t használja veremmutatóként. Ezek a kijelentések csak a vermet implicite használó utasításokra vonatkoznak, úgymint PUSH, PUSHF/PUSHFD, PUSHA/ PUSHAD, POP, POPF/POPFD, POPA/POPAD, INT, INTO, IRET/IRETD, CALL, RET, BOUND, ENTER/ENTERD, LEAVE/LEAVED. Kivételek és hardver-megszakítások kezelőjének meghívásakor szintén figyelembe veszi a processzor a verem ezen jellemzőit.

Az alapértelmezett operandus- vagy címméretet felülbírálhatjuk egy-egy prefixszel (66h ill. 67h). Az assemblerek általában automatikusan alkalmazzák ezeket a prefixeket a szükséges helyeken, de néha jól jön, ha mi is meg tudjuk adni ezeket a jellemzőket egy adott utasításhoz.

17.5.6 Új üzemmódok

Az eddig létező valós címzésű és védett módok mellett a 80386-os processzor egy új üzemmódot is kínál, melynek neve *virtuális-8086 mód* vagy röviden *virtuális mód* (virtual-8086 mode, virtual-86 mode, V86 mode, virtual mode). Ennek az üzemmódnak az a fő jellemzője, hogy általa lehetségessé válik a korábban valós módra írt alkalmazások futtatása a védett módú környezetben, sőt egynél több ilyen alkalmazás (taszk) is futhat egymás mellett és egymástól függetlenül. Mindegyik virtuális taszknak megvan a saját 1 Mbájtos memóriaterülete, saját regiszterei. Az I/O és a megszakítással kapcsolatos utasítások működése taszkonként szabályozható.

Ebben a módban CPL=3, így sok utasítás végrehajtása korlátozott vagy tiltott az operációs rendszer védelmében.

Virtuális módban a címek kiszámítása teljesen ugyanúgy megy végbe, mint valós módban. Ez a következőket jelenti:

- **a szegmensregiszterek tartalmát nem szelektornak értelmezi a processzor, így sem a GDT-t sem az LDT-t nem használja a címek lefordításához**
- **a memóriacímek meghatározása úgy történik, hogy a 16 bites szegmensértéket balra lépteti 4-gyel (megszorozza 16-tal), majd ehhez a 20 bites értékhez hozzáadja a 16 bites offszet zérókiterjesztett értékét, s így jön ki a 21 bites lineáris cím**
- **a szegmenshatár 64 Kbájt, ennek megsértése 12-es (#SS) vagy 13-as (#GP) kivétel keletkezéséhez vezet**

A virtuális mód engedélyezéséhez az EFlags regiszter VM (17-es) bitjét 1-be kell állítani. Ezt a POPFD utasítással nem tehetjük meg, csak egy taszkváltással vagy megszakítás- ill. kivétel-kezelőből való visszatéréssel (IRETD utasítással) érhetjük el, hogy védett módból virtuális módba váltson a processzor. A virtuális módot csak egy szoftver- vagy hardvermegszakítás, ill. egy kivétel útján hagyhatjuk el (és persze a processzor resetelésével).

A virtuális módról bővebben a védett módot tárgyaló későbbi fejezetben lesz szó.

17.5.7 Új címzési módok

Mivel az általános regiszterek méretét 32 bitre növelték, szükség volt olyan új címzési módokra, amikkel ez a növekedés hatékonyan kihasználható. A következő 32 bites címzési módok mindegyike használható a processzor bármelyik üzemmódjában, 16 és 32 bites kódszegmens esetén is.

17.5.7.1 Közvetlen címzés

A címzés alakja [offs32], ahol offs32 egy 32 bites abszolút, szegmensen belüli offszetet jelöl.

17.5.7.2 Báziscímzés

A lehetséges alakok: [EAX], [EBX], [ECX], [EDX], [ESI], [EDI], [ESP], [EBP]. A cél bájt offszetcímét a használt általános regiszterből fogja venni.

17.5.7.3 Bázis+relatív címzés

Ide a [bázis+rel8], [bázis+rel32] formájú címmegadások tartoznak. Bázis bármelyik 32 bites általános regiszter lehet, rel8 és rel32 a bájt ill. duplaszó méretű eltolást (displacement) jelölik.

17.5.7.4 Bázis+index címzés

A következő formákat öltheti: [bázis+index]. Bázis bármelyik, index pedig ESP kivételével bármelyik 32 bites általános regiszter lehet.

17.5.7.5 Bázis+index+relatív címzés

Általános alakja [bázis+index+rel8] és [bázis+index+rel32] lehet.

*17.5.7.6 Bázis+index*skála címzés*

Alakja [bázis+index*skála], ahol bázis és index a szokásos 32 bites általános regiszterek lehetnek, míg skála értéke 1, 2, 4 vagy 8 lehet.

*17.5.7.7 Bázis+index*skála+relatív címzés*

Alakja [bázis+index*skála+rel8] és [bázis+index*skála+rel32] lehet.

*17.5.7.8 Index*skála+relatív címzés*

Alakja [index*skála+rel32].

Minden címzési módhoz tartozik egy alapértelmezett (default) szegmensregiszter-előírás. Ez a következőket jelenti:

- **a bázisként ESP-t vagy EBP-t használó címzési módok SS-t használják**
- **a többi címzési mód DS-t használja**

Az a feltétel, hogy index nem lehet az ESP regiszter, a gyakorlatban annyit jelent, hogy az

[ESP+ESP]
[ESP+ESP+rel8/rel32]
[ESP+ESP*skála]
[ESP+ESP*skála+rel8/32]

alakú címzések illegálisak. Azonban, ha skála=1, akkor az

[ESP*skála+rel8/rel32]=[ESP*1+rel8/rel32]=[ESP+rel8/rel32]

címzés megengedett, mivel ESP itt a bázis szerepét tölti be.

Bár valós és virtuális módban vigyáznunk kell, hogy az effektív cím ne haladja meg a szegmenshatárt (0FFFFh), ez alól a LEA utasítás kivételt élvez. Ezek a címzésmódok a LEA

utasítással ugyanis nagyon hatékony módszert adnak szorzások gyors elvégzésére. A következő 2 egyszerű utasítás

```
LEA     EAX,[EAX+8*EAX]
LEA     EAX,[EAX+4*EAX]
```

az EAX-ben levő előjeles vagy előjeltelen értéket megszorozza +45-tel, s mindezt igen gyorsan (4-6 óraciklus) teszi. Ugyanez ADD és MUL/IMUL, esetleg SHL/SAL utasítások sorozatával legalább 3-szor lassabb lenne.

17.5.8 Változások a kivétel-kezelésben

Az összes kivétel nemcsak a védett, de a virtuális módban is keletkezhet.

32 bites verem esetén a hibakódok mérete is 32 bit lesz. A felső 16 bit ilyenkor kihasználatlanul kerül a verembe.

A hibakódot virtuális módban is berakják a kivételek a verembe.

Az új 14-es (#PF) kivétel hibakódjának formátuma eltér az eddig használttól:

Bitpozíció	*Név*
0	P
1	R/W
2	U/S
3..15	–

Ha a P bit értéke 0, akkor a laphiba egy nem jelenlevő lap miatt keletkezett, különben egy hozzáférési védelmi sértés az ok.

Ha az R/W bit értéke 0, akkor olvasási, különben írási művelet a bűnös.

Ha U/S=1, akkor felhasználói szintű (user), egyébként pedig rendszerszintű (supervisor) lapon történt a hiba.

A 3..15 bitek fenntartottak.

1-es kivétel keletkezésének okait a nyomkövető (debug) regiszterek bevezetésével kibővítették, és a kivétel immár lehet csapda (trap) és hiba (fault) típusú is:

- **utasítás-lehívás (instruction fetch) töréspont – fault**
- **adat olvasás vagy írás töréspont – trap**
- **bármelyik nyomkövető regiszter írása (general detect) – fault**

6-os kivétel a következő valamelyik feltétel teljesülésekor is keletkezik:

- **az ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR és VERW utasítások valamelyikét próbáltuk meg végrehajtani virtuális módban**
- **a LOCK prefixet egy érvénytelen utasítás előtt alkalmaztuk, vagy az utasítás céloperandusa nem memóriahivatkozás**

A dupla hiba (8-as, #DF) kivétel keletkezésének oka két új feltétellel bővült:

- **az első kivétel a laphiba (14-es), a második pedig a 0, 10, 11, 12, 13 kivételek valamelyike**
- **mindkét kivétel laphiba (14-es, #PF)**

Az eddig létező feltétel, vagyis ha az első és a második kivétel is a 0, 10, 11, 12, 13 kivételek valamelyike, továbbra is érvényes marad.

13-as kivétel (#GP) a következő feltételek esetén is keletkezik:

- **szegmenshatársértés a FS, GS szegmensek használatakor**
- **egy rendszerszegmens szelektorát próbáltuk meg betölteni a FS, GS regiszterekbe**
- **egy csak végrehajtható (execute-only) kódszegmens szelektorát töltöttük be az FS, GS regiszterekbe**

- **nullszelektort tartalmazó FS, GS regiszterekkel történő memóriaelérés**
- **az utasítások maximális hosszának határát (15 bájt) megsértettük**
- **CR0-ba olyan értéket töltöttünk be, ahol PG=1 (31-es bit) és PE=0 (0-ás bit)**
- **virtuális módban egy olyan megszakítás- vagy kivétel-kezelőt próbáltunk meg elérni megszakítás- vagy csapdakapun keresztül, amely kódszegmensének DPL-je nagyobb mint 0**

17.5.9 Új kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
14	**#PF**	**Page Fault**	**Fault**	**Memóriahivatkozás**

14-es kivétel akkor keletkezik, ha a CR0 regiszter PG (31-es) bitjének 1-es állása mellett a következő feltételek valamelyike teljesül:

- **a hivatkozott cím lefordításához szükséges lapcímtár vagy laptábla bejegyzés "jelenlevő" (Present–P) bitje törölve van**
- **egy felhasználói (user) szinten futó taszk megpróbál elérni egy rendszerszintű (supervisor) lapot**
- **egy felhasználói (user) szintű taszk egy írásvédett (read-only) lapra próbál meg írni**

A verembe egy speciális formátumú hibakód kerül, ennek leírását lásd az előző alfejezetben. Ezen kívül a kivétel-kezelő meghívásakor a CR2 regiszter tartalmazza a hibát okozó 32 bites lineáris címet. A veremben levő CS:IP/EIP példány általában a hibát okozó utasításra mutat. Ha azonban a hiba taszkváltás közben történt, előfordulhat, hogy az új taszk első utasításának címét tartalmazza a veremben levő másolat.

17.5.10 Változások a regiszterkészletben

A 8 általános regiszter (AX, BX, CX, DX, SI, DI, SP, BP), az utasításmutató (IP) regiszter, a Flags regiszter és az MSW regiszter méretét 32 bitesre bővítették ki. Ezek a regiszterek új név alatt érhetők el, leírásukat ezért a következő alfejezetben lehet megtalálni.

A GDTR és IDTR regiszterek immár a táblák teljes 32 bites fizikai báziscímét tárolni tudják:

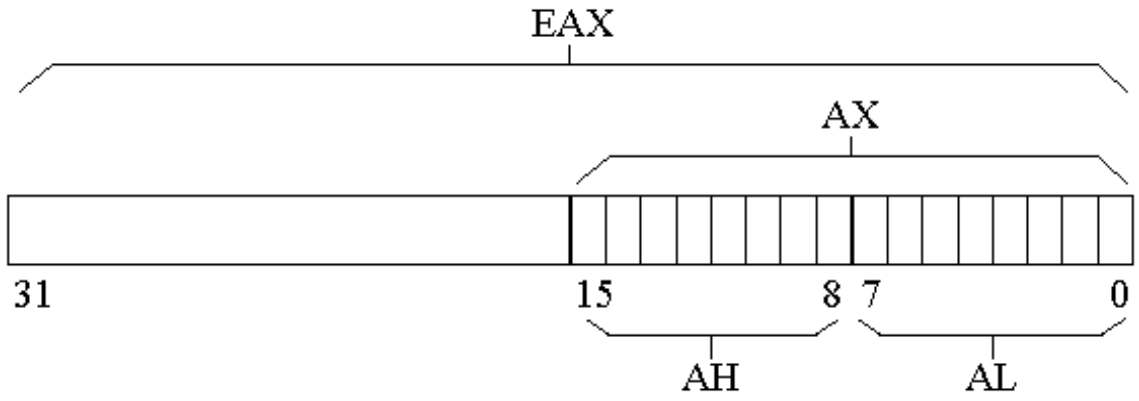
Bitpozíció	*Tartalom*
0..15	Határ
16..47	Báziscím

17.5.11 Új regiszterek

Két új szegmensregisztert kapott a 80386-os processzor. Az FS és GS regiszterek funkciója, felépítése és tartalma ugyanaz, mint az ES regiszteré, tehát egy harmadlagos és negyedleges adatszegmenst definiálnak. A processzor bármely üzemmódjában, bármely CPL esetén használhatjuk őket.

Az FS és GS szegmensregiszterekhez egy-egy szegmens deszkriptor cache-regiszter (árnyékregiszter) is be lett vezetve.

8 új 32 bites általános regisztert is bevezettek, ezek nevei EAX, EBX, ECX, EDX, ESI, EDI, ESP és EBP. Amint az alábbi ábrán is látható, az új regiszterek alsó szavát a megszokott AX, BX stb. regiszterek foglalják el, ezeket szintén használhatjuk. A felső szót nem érhetjük el külön néven. Az AX, BX, CX és DX regiszterek alsó és felső bájtjai továbbra is használhatók AL/AH, BL/BH, CL/CH és DL/DH neveken.



A régi 16 bites regiszterek fizikailag vannak ráképezve az újakra, így pl. ha az EAX regiszterben 12345678h van, akkor AX tartalma 5678h, AH-é 56h, AL-ben pedig 78h-t találunk. Ha ezután AX-be 9ABCh-t töltünk, akkor EAX új tartalma 12349ABCh lesz.

Az új általános regisztereket csakúgy, mint a régieket is, a processzor bármely üzemmódjában, tetszőleges CPL mellett elérhetjük és bármilyen értéket tölthetünk beléjük. (ESP kivételes eset, de ez a funkciójából következik.)

A korábban 16 bites utasításmutató regiszter mérete mostantól 32 bit, új neve EIP. A regiszter alsó szavában a korábbi IP regiszter helyezkedik el.

A korábban státuszregiszternek vagy Flags-nek hívott regiszter mérete ugyancsak 32 bit lett, új neve EFlags. Alsó szavában a korábbi Flags regiszter található meg. Az EFlags-ben levő bitek (flag-ek) funkcióját a következő táblázat mutatja:

Bitpozíció	*Jelölés*	*Név*
0	**CF**	**Carry Flag**
1	**–**	**–**
2	**PF**	**Parity Flag**
3	**–**	**–**
4	**AF**	**Auxilliary/Adjust Flag**
5	**–**	**–**

6	ZF	Zero Flag
7	SF	Sign Flag
8	TF	Trap Flag
9	IF	Interrupt Flag
10	DF	Direction Flag
11	OF	Overflow Flag
12..13	IOPL	Input/Output Privilege Level
14	NT	Nested Task
15	–	–
16	RF	Resume Flag
17	VM	Virtual-8086 Mode
18..31	–	–

A CF, PF, AF, ZF, SF, TF, IF, DF, OF, IOPL és NT flag-ek funkciója és működése megfelel az eddigieknek.

Az RF flag 1-es értéke az érvényben levő utasítás-töréspontok keletkezését tiltja le a következő utasítás idejére. Ha az az utasítás végrehajtódott, RF értéke automatikusan törlődik. A processzor RF-et mindig beállítja, ha egy hiba (fault) vagy abort típusú kivétel keletkezik. A kezelő meghívásakor a verembe mentett EFlags-másolatban RF tehát 1 értékű lesz. Az egyetlen kivétel ez alól az utasítás-töréspontok miatt generált 1-es kivétel, ahol RF értéke változatlan marad az EFlags-másolatban. Azonban RF-et mindig törli a processzor, mielőtt egy megszakítás vagy kivétel kezelőjét megkezdené végrehajtani. (Ez történik a TF és az IF flag-ekkel is, bár ha csapdakaput használunk, IF értéke változatlan marad.)

A VM flag által kapcsolhatjuk a processzort virtuális-8086 üzemmódba. Ha VM=0, akkor a processzor valós vagy védett módban van, különben pedig a virtuális módot használja. Ezt a bitet a POPFD utasítás NEM képes módosítani. Ahhoz, hogy beállíthassuk, CPL=0 esetén kell végrehajtani egy IRETD utasítást (ekkor a veremben levő EFlags-tükörképben VM=1), vagy egy olyan taszkra kell ugranunk, amelynek TSS-ében levő

EFlags-másolatban VM=1. VM-et mindig törli a processzor, mielőtt egy megszakítás vagy kivétel kezelőjét megkezdené végrehajtani.

Az 1, 3, 5, 15 és 18..31 számú bitek fenntartottak, ami most annyit jelent, hogy értékük nem módosítható, és nem is definiált.

Magának az EFlags regiszternek az elérhetősége nincs korlátozva, viszont az őt kezelő PUSHF, PUSHFD, POPF, POPFD, IRET és IRETD utasítások használata megkötésekkel jár. Ezen kívül néhány flag tartalma nem minden üzemmódban és nem minden CPL érték mellett olvasható és írható. Erről bővebbet az előbb említett utasításokhoz fűzött megjegyzésekben olvashatunk.

3 *vezérlőregisztert* (control register) is találunk a 80386-oson. Ezeket sorban CR0-nak, CR2-nek és CR3-nak nevezik. Ezek a regiszterek csak CPL=0 esetén érhetők el, így valós és védett módokban használhatjuk őket. Mindegyik vezérlő-regiszter 32 bites.

Vezérlőregiszterből elvileg 8 db. lehetne, legalábbis a kódolás alakjából erre lehet következtetni. A CR1 és CR4..CR7 regiszterek azonban nem érhetők el, ezeket a későbbi processzorok számára tartják fenn.

A CR0 regiszter a korábbi MSW (Machine Status Word) regiszter kibővített alakja. Alsó szavában az említett MSW-t találjuk, meg egy új bitet (ET). Az MSW-t továbbra is írhatjuk az LMSW utasítással, illetve tartalma az SMSW-vel kiolvasható. Mivel az SMSW utasítás nem privilegizált, ezt érdemes használni, ha csak arra vagyunk kíváncsiak, hogy az MSW 0..3 bitjei (PE, MP, EM, TS) közül valamelyik be van-e állítva.

A CR0 felépítése a következő táblázatban látható:

Bitpozíció	*Jelölés*	*Név*
0	PE	Protection Enable
1	MP	Monitor coProcessor
2	EM	EMulation
3	TS	Task Switched
4	ET	Extension Type
5..30	–	–
31	PG	PaGing enable

A PE, MP, EM és TS bitek funkciója megfelel az MSW hasonló bitjeiének.

Az új ET bit állapota azt mutatja, milyen koprocesszor típus van a számítógépben. Ha ET=0, akkor a 80287-es, különben a 80387-es koprocesszor utasításkészletével kompatibilis koprocesszort használunk.

Ha PG=1, akkor a lapozás engedélyezett, különben a szokásos módon történik a címszámítás (azaz a lineáris cím közvetlenül a fizikai címet jelenti). Ezt a bitet csak védett módban kapcsolhatjuk be, mert különben 13-as kivétel (#GP) keletkezik.

A CR2 regiszter laphiba (#PF, 14-es) kivétel keletkezése esetén a hibát okozó lineáris címet tartalmazza. Szerepe csak a lapozás használatakor fontos.

Szintén a lapozás mechanizmusában játszik szerepet a CR3 regiszter. Felépítését a következő táblázat mutatja:

Bitpozíció	*Jelölés*	*Név*
0..11	–	–
12..31	PDB	Page-Directory Base

A PDB mező a lapcímtár 31 bites fizikai báziscímének felső 20 bitjét tartalmazza, a báziscím alsó 12 bitjét pedig 0-nak veszi a processzor. A 0..11 bitek fenntartottak, értékük ezért 0. A CR3 regiszter másik elnevezése (utalva tartalmára) a lapcímtár bázisregiszter (Page-Directory Base Register–*PDBR*).

A korábbi processzorok már támogatták a nyomkövetés (debugging) néhány fajtáját, de igazán hatékony módszert nem adtak. A 80386-os processzor 6 db *nyomkövető regiszterrel* (debug register) támogatja a sikeresebb hibakeresést és programfejlesztést. A DR0, DR1, DR2, DR3, DR6 és DR7 nevű 32 bites regiszterekben a nyomkövetéshez szükséges minden fontosabb dolog beállítható. Ezek a regiszterek is privilegizáltak a vezérlőregiszterekhez hasonlóan.

A DR0, DR1, DR2 és DR3 regiszterek a 4 töréspont lineáris címét tárolják. Nevük ezért nyomkövető címregiszter (debug address register).

A DR6 regiszter tartalma azt mutatja, miért keletkezett 1-es kivétel (#DB). Neve ezért nyomkövető státuszregiszter (debug status register). Felépítése a következő:

Bitpozíció	*Név*
0	B0
1	B1
2	B2
3	B3
4..11	–
12	?
13	BD
14	BS
15	BT
16..31	–

A fenntartott bitek értéke mindig 1 kell hogy legyen.

A B0..B1 mezők jelzik, a definiált töréspont-feltételek közül melyik vagy melyek keletkeztek be. Ha valamelyik értéke 1, a DR7 regiszterben a hozzá tartozó LEN és R/W mező által megadott töréspont-feltétel teljesül.

A BD bit értéke 1, ha valamelyik nyomkövető regiszterbe írni próbáltak úgy, hogy a DR7 regiszterben GD=1.

Ha BS=1, a töréspont kivételt a lépésenkénti végrehajtási mód (single-step execution mode) váltotta ki. Ez akkor következik be, ha a Flags/EFlags regiszterben TF=1.

Ha BT=1, akkor taszkváltás miatt következett be a kivétel. Ezt egy olyan taszk váltja ki, melynek TSS-ében T=1.

A DR7 regiszter szolgál az egyes töréspontok engedélyezésére, valamint az őket kiváltó (definiáló) feltételek megadására. Neve ezért nyomkövető vezérlőregiszter (debug control register). Értelmezését az alábbi táblázat mutatja:

Bitpozíció	*Név*
0	L0
1	G0
2	L1
3	G1
4	L2
5	G2
6	L3
7	G3
8	LE
9	GE
10	–
11	–
12	?
13	GD

14..15	–
16..17	R/W0
18..19	LEN0
20..21	R/W1
22..23	LEN1
24..25	R/W2
26..27	LEN2
28..29	R/W3
30..31	LEN3

A 11-es, 14-es és 15-ös bitek értéke fenntartott, ezek értéke mindig 0. A 10-es bit szintén fenntartott, de ennek értéke 1 kell hogy legyen.

Az L0, L1, L2 és L3 bitek az egyes töréspontokat engedélyezik az aktuális taszkra vonatkozóan, azaz a töréspontokat lokálissá teszik. Ha valamelyik bit értéke 1, az adott töréspont keletkezése lehetséges. Ezeket a biteket a processzor automatikusan törli taszkváltáskor.

A G0, G1, G2 és G3 bitek az egyes töréspontokat engedélyezik az összes taszkra vonatkozóan, azaz a töréspontokat globálissá teszik. Ha valamelyik bit értéke 1, az adott töréspont keletkezése lehetséges.

Az LE és GE bitek 1-es értéke arra utasítja a processzort, hogy az adattörésponton (data breakpoint) elakadó utasítás címét pontosan meghatározzák. Ha ezeket nem állítjuk be, a töréspont kivétel (#DB) rendesen generálódni fog, de a veremben levő CS:IP/EIP-tükörkép nem feltétlenül a kivételt kiváltó utasításra fog mutatni. Ezeknek a biteknek a szerepe a 80486-os processzortól kezdve megváltozik.

Az R/W0, R/W1, R/W2 és R/W3 mezők határozzák meg, az adott töréspont milyen művelet hatására generáljon kivételt:

R/W	Töréspont feltétele
00	Csak utasítás-végrehajtás

01	**Csak adat írása**
10	**–**
11	**Adat olvasása vagy írása, de nem utasítás-végrehajtás**

Végül a LEN0, LEN1, LEN2 és LEN3 mezők az adott töréspont memóriacímén kezdődő kritikus terület hosszát tartalmazzák:

LEN érték	*Tartomány hossza*
00	**1 bájt**
01	**2 bájt**
10	**–**
11	**4 bájt**

Utasítástöréspont esetén a megfelelő LEN érték csak a 00 beállítás lehet (tehát 1 bájt hosszú memóriaterület). Egy utasítástöréspontot csak akkor ismer fel a processzor, ha a töréspont címe az utasítás legelső bájtjára mutat (ami vagy a legelső prefix, vagy a műveleti kód első bájtja).

Két bájt hosszú területek esetén a memóriacímnek 2-vel, míg négy bájtos terület esetén 4-gyel kell oszthatónak lennie (tehát a címnek szó- vagy duplaszóhatárra kell illeszkednie).

Debug kivétel (#DB, 1-es) akkor keletkezik, ha az elért memóriacím a megfelelő DR0..DR3 regiszter, ill. a DR7 regiszter adott LEN mezőjével meghatározott tartomány tetszőleges bájtjának címével megegyezik.

Bár nem említettük, a DR4 és DR5 regiszterekre való hivatkozás egyenértékű a DR6 ill. DR7 regiszterek használatával. Mégsem ajánlott a DR4 és DR5 regiszternevek használata, mivel a Pentium processzortól kezdve inkompatibilitás léphet fel (lásd a Debugging Extensions szolgáltatást).

A DR6 és a DR7 regiszter 12-es bitjét külön-külön be lehet 1-re állítani, de ezen bitek funkciója nem ismert pontosan. (A dokumentálatlan ICEBP/SMI utasításnál azonban említettük, hogy az SMM módba kapcsolásnak feltétele, hogy a DR7 regiszter ezen bitje 1-es értékű legyen.)

Az architektúrában történt változások leírásánál már említettük, hogy a 80386-os processzoron lehetőség van a lapozással együtt bevezetett TLB-k működésének tesztelésére. Ezt a *tesztregisztereken* (test register) keresztül tehetjük meg. Két 32 bites tesztregiszter érhető el, nevük TR6 és TR7. Ezek a regiszterek is privilegizáltak, így csak valós módban vagy védett módban CPL=0 esetén használhatók.

A TR6 regisztert teszt parancsregiszternek (test command register) nevezzük, mivel tartalma határozza meg, a TLB-ből olvasni vagy abba írni szeretnénk-e. Csak ezt a két műveletet támogatja a processzor. A regiszter felépítését a következő táblázat mutatja:

Bitpozíció	*Jelölés*	*Név*
0	C	Command select
1..4	–	–
5	W#	R/W bit complement
6	W	R/W bit
7	U#	U/S bit complement
8	U	U/S bit
9	D#	Dirty bit complement
10	D	Dirty bit
11	V	Valid bit
12..31	–	Linear address

Az 1..4 számú bitek fenntartottak, értékük mindig legyen 0.

A C bit állapota határozza meg a TLB-n végrehajtott műveletet. Ha C=0, akkor a TLB-be írás, különben onnan való olvasás fog történni.

A V bit 1-es értéke azt jelzi, hogy az adott bejegyzés érvényes értékeket tartalmaz.

A D, U és W bitek a megfelelő lapcímtár- vagy laptábla-bejegyzés D, U/S és R/W bitjeinek felelnek meg. A D#, U# és W# jelölésű bitek értéke a társuk értékének ellentettje. Ezek a bitpárok másképp funkcionálnak a TLB-be íráskor és az onnan való olvasáskor. A következőekben jelölje X a D, U, W bitek valamelyikét, míg X# az adott bit komplemens párját.

A TLB-be íráskor az X és X# bitek az új bejegyzés X bitjének értékét határozzák meg:

X	*X#*	*Az új bejegyzés X értéke*
0	0	–
0	1	0
1	0	1
1	1	–

A TLB-ben kereséskor, ill. egy TLB-bejegyzés olvasásakor az X és X# bitek a keresés eredményét befolyásolják:

X	*X#*	*Találat, ha…*
0	0	–
0	1	X=0
1	0	X=1
1	1	–

A lineáris címet tartalmazó mezőnek is más a szerepe a művelettől függően. A TLB-be íráskor az új bejegyzés ehhez a lineáris címhez lesz rendelve. Ha a TLB-ben keresünk, akkor a TLB-bejegyzéseket ezzel a mezővel hasonlítja össze, és egyetlen találat esetén a TR7 regiszter megfelelően fel lesz töltve.

A TR7 regiszter neve teszt adatregiszter (test data register), mivel ez tartalmazza a TLB-bejegyzés adat részének tartalmát, a lap fizikai címét. Felépítése alább látható:

Bitpozíció	*Jelölés*	*Név*
0..1	–	–
2..3	REP	block Report
4	HT	HiT
5..11	–	–
12..31	–	Physical address

A 0..1 és 5..11 bitek fenntartottak, értékük mindig legyen 0.

A REP mező funkciója is változik a művelettől függően. A TLB-be íráskor azt a készletet (blokkot) választja ki, ahová az új bejegyzésnek kerülnie kell. A TLB-ben kereséskor, ha az eredményben HT=1, akkor REP mutatja, melyik készletben helyezkedik el a találat. Ha HT=0, értéke meghatározatlan.

A HT bit 1-es értéke jelzi, hogy sikeres volt a keresés, és a TR7 regiszter többi mezője érvényes információt tartalmaz a találattal kapcsolatban. A TLB-be íráskor ennek a bitnek 1-et kell tartalmaznia.

Végül a fizikai címet tartalmazó mező tartalmát is másképp kell értelmezni a TLB írásakor és olvasásakor. Íráskor ez tartalmazza az új bejegyzés adat részének tartalmát. Olvasáskor, ha HT=1, ez a mező tartalmazza a találat adat mezőjét. Ha HT=0, értéke meghatározatlan.

17.6 Az Intel 80486-os processzoron bevezetett újdonságok

17.6.1 Változások az utasításkészletben

- **LOCK**
  Ez a prefix a CMPXCHG és XADD utasításokkal is használható, előttük nem okoz 6-os (#UD) kivételt.

17.6.2 Új utasítások

- **BSWAP reg32**
  (Byte SWAP)
  Az operandusában levő értéket little-endian tárolásról big-endian-re konvertálja, azaz megfordítja a bájtok sorrendjét. Csak egy 32 bites általános regiszter lehet az operandus. Az utasítás hatására a regiszter 0. és 3., valamint 1. és 2. bájtjai megcserélődnek.
- **CMPXCHG dest,source reg**
  (CoMPare and eXCHanGe)
  Az akkumulátort (AL-t, AX-et vagy EAX-et) összehasonlítja a céllal a CMP utasításnak megfelelően. Ha ZF=1 (azaz a cél megegyezik az akkumulátorral), akkor a forrás regiszter tartalmát betölti a célba, különben pedig a célt betölti az akkumulátorba. Az operandusok mérete 8, 16 és 32 bit is lehet. Céloperandusként memóriahivatkozást és általános regisztert, forrásként pedig csak általános regisztert írhatunk. Mind a hat aritmetikai flag-et módosítja. Ez az utasítás a szemaforok kezelését könnyíti meg.

- **INVD (no op)**
  (INValiDate caches)
  A belső cache-memóriákat kiüríti, majd a processzor a külső (alaplapi) cache-eket is arra készteti, hogy semmisítsék meg tartalmukat. A cache-ben levő, a rendszermemóriába még nem visszaírt adatok nem kerülnek frissítésre, így adatvesztés fordulhat elő. Ez az utasítás privilegizált utasítás.
- **INVLPG mem**
  (INVaLidate TLB-entry of given PaGe)
  A memóriaoperandust tartalmazó lap TLB-bejegyzését érvényteleníti. Ez az utasítás privilegizált utasítás.
- **WBINVD (no op)**
  (Write Back and INValiDate caches)
  A belső cache-ekben levő módosított adatokat visszaírja a rendszermemóriába, és kiüríti a cache-eket. Ezután a külső cache-eket arra utasítja, hogy azok is írják vissza a módosításokat, majd semmisítsék meg tartalmukat. Ez az utasítás privilegizált utasítás.
- **XADD dest,source reg**
  (eXchange and ADD)
  A forrás és a cél tartalmát felcseréli, majd a célhoz hozzáadja a forrást. Az operandusok mérete 8, 16 és 32 bit is lehet. Céloperandusként memória-hivatkozást és általános regisztert, forrásként pedig csak általános regisztert írhatunk. Az utasítás hatása megegyezik az

```
XCHG    dest,source
ADD     dest,source
```

vagy az

```
ADD     source,dest
XCHG    dest,source
```

utasítások végrehajtásával. Az aritmetikai flag-eket (CF, PF, AF, ZF, SF, OF) az ADD utasítás eredménye szerint állítja be. (A fenti két utasítás-sorozat végrehajtása igazából egy plusz dolgot is jelent: az XCHG utasítás végrehajtásakor a buszt automatikusan lezárja a processzor, míg ezt az XADD esetében nem teszi meg.)

A felsorolt utasítások közül az INVD, INVLPG és WBINVD privilegizáltak, tehát csak CPL=0 esetén hajthatók végre.

17.6.3 Változások az architektúrában

A 80486-os processzorba egy 8 Kbájtos, egyesített utasítás-/adat-cache-t is beépítettek. A cache csak write-through módban képes üzemelni. A cache-sor (cache line) mérete 16 bájt.

A 80486-os a TLB mellett lehetővé teszi a cache tesztelését is. Ennek támogatására 3 új tesztregiszter lett bevezetve. A TR3 a cache-teszt adatregiszter (cache test data register), a TR4 a cache-teszt státuszregiszter (cache test status register), a TR5 pedig a cache-teszt vezérlőregiszter (cache test control register). Megjegyezzük, hogy ezeket a regisztereket (a TR6 és TR7 regiszterekkel együtt) a Pentium processzor már nem támogatja.

17.6.4 Változások a kivétel-kezelésben

A 9-es kivétel megszűnt a 80486-os processzoron, helyette mindig 13-as (#GP) kivételt generál a processzor.

13-as kivétel (#GP) a következő feltételek teljesülésekor is keletkezik:

- **CR0-ba olyan értéket töltöttünk be, ahol NW=1 (29-es bit) és CD=0 (30-as bit)**
- **minden olyan művelet, ami a 80386-oson 9-es kivételt okozott volna**

A 14-es (#PF) kivétel mostantól akkor is keletkezik, ha egy rendszerszintű (supervisor) taszk egy írásvédett (read-only), felhasználói (user) szintű lapra próbál meg írni úgy, hogy a CR0 regiszter WP (16-os) bitje be van állítva.

17.6.5 Új kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
17	**#AC**	**Alignment Check**	**Fault**	**Bármely adathivatkozás**

A 17-es kivétel akkor keletkezik, ha a processzor egy nem megfelelően illeszkedő (unaligned) memóriaoperandust talált, miközben az illeszkedés-ellenőrzés (alignment check) funkció engedélyezve volt. Az engedélyezés a következő feltételek mindegyikének teljesülését jelenti:

- **a CR0 regiszter AM (18-as) bitjének értéke 1**
- **az EFlags AC (18-as) flag-jének értéke szintén 1**
- **CPL=3**

A processzor csak adat- és veremszegmensekben figyeli az illeszkedést, kód- és rendszerszegmensekben nem foglalkozik vele. A verembe mindig egy nullás hibakód (0000h vagy 00000000h) kerül. CS:IP/EIP veremben levő példánya mindig a hibát okozó utasításra mutat.
A következő táblázat mutatja a különféle adattípusok esetén szükséges illeszkedés típusát. A második oszlopban levő érték azt adja meg, az adott típusú operandus címének mennyivel kell oszthatónak lennie.

Adat típusa	*Illeszkedés bájtokban*
Szó	2
Duplaszó	4
Egyszeres valós	4
Dupla valós	8
Kiterjesztett valós	8
Szegmens szelektor	2
32 bites távoli pointer	2
48 bites távoli pointer	4
32 bites közeli pointer (offszet)	4
GDTR, IDTR, LDTR, TR tartalom	4
FLDENV, FSTENV terület	2 vagy 4
FRSTOR, FSAVE terület	2 vagy 4
Bitsztring	2 vagy 4

Az utolsó 3 esetben az illeszkedés az alkalmazott operandusméretettől függ: az első érték 16 bites, a második 32 bites operandusméretre vonatkozik.

17.6.6 Változások a regiszterkészletben

Az EFlags regiszter egy új flag-et kapott:

Bitpozíció	*Név*

18	AC

Az AC (Alignment Check enable) flag 1-es értéke engedélyezi az illeszkedés-ellenőrzést. A 17-es (#AC) kivétel keletkezéséhez még két feltételnek teljesülnie kell: CPL=3, és a CR0 regiszter AM (18-as) bitjének is be kell lennie állítva.

A CR0 regiszterben 5 új bit lett bevezetve:

Bitpozíció	*Név*
5	NE
16	WP
18	AM
29	NW
30	CD

Az NE (Numeric Error handling) bit állása választja ki a koprocesszorban keletkező numerikus hibák lekezelésének módját. Az NE=0 állapot a régi, MS-DOS-kompatibilis módot jelenti, míg NE=1 esetén a natív módnak megfelelően történik a numerikus kivételek jelzése. Erről a témáról bővebben a koprocesszort tárgyaló fejezetben írunk.

A WP (Write Protect) bit védett módban az írásvédett felhasználói (user) szintű lapok elérésére van hatással. WP=0 esetén a rendszerszintű (supervisor) taszkok írhatnak ezekre a lapokra, míg WP=1 esetén az ilyen próbálkozások lapkivételt (#PF, 14-es) okoznak.

Az AM (Alignment check Mask) bit 1-es értéke esetén engedélyezett az illeszkedés-ellenőrzés. A 17-es (#AC) kivétel keletkezéséhez még két feltételnek teljesülnie kell: CPL=3, és az EFlags regiszter AC (18-as) flag-jének is be kell lennie állítva.

A CD (Cache Disable) és NW (cache Not Write-through) bitek a cache működését vezérlik a következő módon:

CD	*NW*	*Következmény*
0	0	Cache engedélyezve, WT
0	1	–
1	0	Cache korlátozva, WT

1	1	Cache korlátozva

A WT jelölés a Write-Through kifejezést rövidíti. Ez a cache optimális működésére utaló fogalom, és annyit takar, hogy a módosított adatok a cache-be és a rendszermemóriába is bekerülnek rögvest egymás után. A cache korlátozása annyit jelent, hogy a cache bizonyos fokig kevesebb szerepet játszik a memóriaelérések lebonyolításában.

A CD=1, NW=1 esetben a cache olyan szempontból kiiktatódik, hogy ha nincs olvasási vagy írási találat (azaz read miss vagy write miss fordul elő), a megfelelő cache-sor (cache line) nem kerül betöltésre a cache-be. Ha a cache-t ebben a beállításban kiürítjük az INVD vagy a WBINVD utasítással, az immáron üres cache tényleg ki lesz iktatva a memóriaelérés folyamatából.

A CD=1 esetek a CR3 regiszterben, a lapcímtár- és a laptábla-bejegyzésekben található PWT (3-as) és PCD (4-es) bitek működésére is hatással vannak.

A CD=0, NW=1 beállítás érvénytelen, használata 13-as (#GP) kivétel keletkezését vonja maga után.

A CR3 regiszterben két új bit lett bevezetve, melyek a lapcímtár-bejegyzések cache-selését vezérlik:

Bitpozíció	*Név*
3	PWT
4	PCD

A PWT (Page-level Write-Through) és PCD (Page-level Cache Disable) bitek állása határozza meg, a lapcímtár bejegyzéseit (PDE) cache-seli-e a processzor:

PCD	*PWT*	*Következmény*
0	0	Nincs PDE cache-selés
0	1	WT PDE cache-selés
1	0	Nincs PDE cache-selés
1	1	Nincs PDE cache-selés

Ezen bitek állását figyelmen kívül hagyja a processzor, ha a CR0 regiszterben PG=0 vagy CD=1.

A lapcímtár- és laptábla-bejegyzésekben ugyanezen a bitpozíciókban megtalálható mindkét bit, és működésük is megegyezik az itt megadottakkal. A lapcímtár-bejegyzésekben levő hasonló bitek a laptáblák, míg a laptábla-bejegyzések azonos bitjei a lapok cache-selését szabályozzák.

A DR7 nyomkövető regiszter LE (8-as) és GE (9-es) bitjeinek állását figyelmen kívül hagyja a processzor, és a töréspontón fennakadó utasítást mindig pontosan jelzi ezek értékétől függetlenül.

17.6.7 Új regiszterek

3 új tesztregiszter áll a felhasználó rendelkezésére a cache tesztelése céljából. A TR3, TR4 és TR5 regiszterek felépítése sajnos nem ismert számunkra, így ezek ismertetésétől eltekintünk.

17.7 Az Intel Pentium bővítései

17.7.1 Változások az utasításkészletben

- **INT 3**

  Az egybájtos INT 3 utasításra (0CCh) virtuális módban nincs hatással a megszakítás-átirányítás (interrupt redirection), ami akkor történhetne, ha a CR4 regiszter VME (0-ás) bitje be van állítva. Az utasítás által kiváltott 3-as kivételt (#BP)

mindig a védett módú kezelő fogja elintézni. A kétbájtos alakra (0CDh, 03h) ez nem vonatkozik.

- **LOCK**

  Ez a prefix a CMPXCHG8B utasítással is használható, előtte nem okoz 6-os (#UD) kivételt.
- **MOV dest reg32,TRx**
- **MOV TRx,source reg32**

  A MOV utasítás ezen formái mostantól érvénytelenek, mivel a tesztregisztereket eltörölték a Pentium (P5 architektúrájú) processzortól kezdve. Kiadásuk 6-os (#UD) kivétel keletkezését vonja maga után.
- **POPFD (no op)**

  Valós módban és védett módban CPL=0 esetén a VIF (19-es) és a VIP (20-as) flag-ek mindig kitörlődnek. Védett módban, ha 0<CPL≤IOPL, a VIF és a VIP flag-ek ugyancsak mindig kitörlődnek. Védett módban, ha CPL>0 és CPL>IOPL, nem keletkezik kivétel, és az említett bitek is változatlanok maradnak. Virtuális módban, ha IOPL=3, a VIF és a VIP flag-ek változatlanok maradnak.
- **PUSH CS/DS/ES/FS/GS/SS**

  Ha a veremszegmens címméret jellemzője 32 bit, olyan duplaszó lesz a verembe rakva, amelynek az alsó szava tartalmazza az adott szegmensregisztert, a felső szó értéke pedig 0000h.

17.7.2 Új utasítások

- **CMPXCHG8B source mem64**

  (CoMPare and eXCHanGe 8 Bytes)

  Az EDX:EAX regiszterpárban levő értéket összehasonlítja a 64 bites memóriaoperandussal. Ha a két érték megegyezik, az ECX:EBX

regiszterpár tartalmát beírja a memóriaoperandusba, és ZF-et 1-re állítja. Különben a memóriaoperandust eltárolja az EDX:EAX regiszterpárban, és ZF=0 lesz. A CF, PF, AF, SF és OF flag-ek értéke változatlan marad. Ha operandusként nem memóriahivatkozás szerepel, 6-os (#UD) kivétel keletkezik.

- **CPUID (no op)**
  (CPU IDentification)
  A processzor pontos típusának és az általa támogatott szolgáltatások azonosítására szolgál ez az utasítás. Az utasítás operandus nélküli, de az EAX regiszter végrehajtás előtti értéke határozza meg, milyen információra van szükségünk. Az utasítás végrehajtása után az EAX, EBX, ECX és EDX regiszterek tartalmazzák a kért információkat.
  EAX=00000000h bemenet esetén EAX tartalmazza majd az a legmagasabb értéket, amit inputként a CPUID utasítás az EAX regiszterben elfogad. Az ECX:EDX:EBX regiszterhármas a gyártó nevét azonosító ASCII szöveget tartalmazza. Ez Intel esetén "GenuineIntel", az AMD processzorainál "AuthenticAMD", míg Cyrix prociknál "CyrixInstead".
  EAX=00000001h bemenet esetén EAX a processzor pontosabb típusát tartalmazza. A 0..3 bitek a stepping-értéket (revíziószám), a 4..7 bitek a modellt, míg a 8..11 bitek a processzorcsalád azonosítóját tartalmazzák. A 12..31 bitek fenntartottak. Az EBX és ECX regiszterek tartalma meghatározatlan. Az EDX regiszter a processzor által támogatott, modell-specifikus szolgáltatásokat (features) sorolja fel.

Ez a két funkció minden P5 és későbbi architektúrájú processzor esetén megtalálható, a többi létezése viszont gyártótól függ. (A dokumentálatlan lehetőségeket tárgyaló fejezetben említettük, hogy néhány 80486-os processzor is támogatta ezt az utasítást. Ezek esetében is mindkét említett funkciónak kötelezően működnie kell.)

A CPUID utasítással részletesebben a "Processzorok detektálása" című fejezetben foglalkozunk.

- **RDMSR (no op)**

  (ReaD Model-Specific Register)

  Az ECX regiszterben megadott sorszámú modell-specifikus regiszter tartalmát az EDX:EAX regiszterpárba tölti. Ez egy privilegizált utasítás. Ha egy nem létező MSR-t próbálunk meg kiolvasni, 13-as kivétel (#GP) keletkezik.

- **RDTSC (no op)**

  (ReaD Time-Stamp Counter)

  A Time-Stamp Counter regiszter értékét az EDX:EAX regiszterpárba tölti. A TSC a 10h számú MSR, melynek tartalma resetkor nullázódik, és minden óraciklus 1-gyel növeli értékét. Ha a CR4 regiszter TSD (2-es) bitje törölve van, az utasítás bármikor végrehajtható. Ha azonban TSD=1, az utasítás csak CPL=0 esetén adható ki.

- **RSM (no op)**

  (ReSuMe from System Management Mode)

  Az utasítás kiadásával a processzor kilép az SMM üzemmódból, a processzorállapotot visszaállítja az SMRAM-ból (System Management RAM), és visszatér a futás előző helyére (CS:IP vagy CS:EIP). Az utasítást csak az SMM módban ismeri fel a processzor, más módban kiadva azt az eredmény 6-os kivétel (#UD) lesz. Ha az állapot

visszaállítása során az elmentett információban (SMM state dump) valami illegális van, akkor a processzor shutdown állapotba kerül, ahonnan csak egy hardveres reset ébresztheti fel. Három fő hibaforrás van: az egyik, ha a CR4 regiszter tükörképében valamelyik fenntartott bit 1-es értékű, a másik, ha a CR0 érvénytelen beállításokat tartalmaz (pl. PE=0 és PG=1, CD=0 és NW=1 stb.). Szintén hibának számít, ha a mentési terület kezdőcíme nem osztható 32768-cal.

- **WRMSR (no op)**
  (WRite Model-Specific Register)
  Az EDX:EAX regiszterpár tartalmát az ECX-ben megadott sorszámú MSR-be tölti. Ez az utasítás privilegizált. Ha egy nem létező MSR-be próbálunk meg írni, vagy egy létező MSR valamelyik fenntartott bitjét beállítjuk, 13-as kivétel (#GP) keletkezik.

Az RDMSR és WRMSR utasítások privilegizáltak, azaz csak CPL=0 esetén hajthatók végre. Az RDTSC utasítás szintén privilegizáltnak viselkedik, ha a CR4 regiszter TSD (2-es) bitjének értéke 1. Az RSM utasítás csak SMM módban adható ki.

17.7.3 Változások az architektúrában

A Pentium-ban már 8 Kbájt utasítás-cache és külön 8 Kbájt adat-cache van. A cache-sor mérete 32 bájt. A cache-ek wtrite-through és writeback módban is képesek dolgozni.

A 4 Kbájtos lapméret nem ideális, ha egy nagyobb összefüggő területen helyezkedik el mondjuk a programok kódja (ahogy ez általában szokásos is), mivel a sok lap lemezre kivitele vagy onnan beolvasása sok ideig tart. A Pentium processzorok ezért támogatják a lapméret 4 Mbájtra növelését. Ennek a technikának *Page Size Extension* (PSE) a neve. A CPUID utasítással eldönthető, hogy az adott processzor ismeri-e a PSE-t.

Először az új CR4 regiszter PSE (4-es) bitjét be kell állítanunk. Ha ez megvan, akkor a lapcímtár-bejegyzések eddig fenntartott 7-es (Page Size–PS) bitje határozza meg, mi történik. Ha PS=0, akkor a bejegyzés egy laptáblára mutat, aminek mindegyik bejegyzése egy 4 Kbájtos lap adatai tartalmazza. Ha azonban PS=1, a lapcímtár-bejegyzés közvetlenül egy 4 Mbájtos lapra fog mutatni. Mivel 1024 lapcímtár-bejegyzés van, így ugyanennyi db. 4 Mbájtos lap lehet, ami ismételten kiadja a 4 Gbájtos címterületet.

Mivel a lapcímtár bejegyzéseiben külön-külön állíthatjuk a PS bitet, 4 Kbájtos és 4 Mbájtos lapokat felváltva is használhatunk. A lapozás ezen fajtája továbbra is átlátszó marad a programok számára.

A 32 bites lineáris cím a 4 Mbájtos lapok használatakor más felépítésű. A 22..31 bitek továbbra is a lapcímtár egy bejegyzését választják ki, ami viszont most egy 4 Mbájtos lapra mutat. A 0..21 bitek tartalmazzák a lapon belüli offszetet.

A *Virtual-8086 Mode virtual interrupt Extension* (VME) szolgáltatás a megszakítás-kezelés módszereit bővíti ki a virtuális-8086 módban. Ezt engedélyezni a CR4 regiszter VME (0-ás) bitjének beállításával lehet. Ez a szolgáltatás a maszkolható hardver-megszakítások és a szoftver-megszakítások kiszolgálására vonatkozik. A hardver által

kiváltott megszakítások és a kivételek továbbra is a védett módú kezelőhöz futnak be.

Először nézzük, hogy cselekszik a processzor a maszkolható hardver-megszakítások (IRQ0..IRQ15) esetén:

Ha VME=0, akkor a megszakításokat a megszokott módon kezeli a processzor. Ez azt jelenti, hogy ha IOPL=3, akkor a CLI, STI, POPF, POPFD, PUSHF és PUSHFD utasítások közvetlenül az IF flag-re vannak hatással. Ha IOPL<3, 13-as (#GP) kivétel keletkezik.

Ha VME=1, a lehetőségek köre tovább bővül. Ha IOPL=3, nincs változás, az említett utasítások szintén az IF-et befolyásolják. Ha azonban IOPL<3, az EFlags regiszter VIF (19-es) és VIP (20-as) flag-jei életre kelnek. A fent felsorolt utasítások VIF-et fogják módosítani ill. kiolvasni. A virtuális módú taszkokat felügyelő védett módú megszakítás-kezelő ekkor a VIF flag-et megvizsgálva dönthet, mi legyen. Ha VIF=1, akkor a hardver-megszakítást a virtuális taszk vagy a CPL=0 szintű operációs rendszer kezelőjét meghívva lekezelheti. Ha VIF=0, akkor a VIP flag beállításával jelzi, hogy egy függő hardver-megszakítás keletkezett.

Ha VIF=0, VIP=1, valamint VME=1, és a virtuális módú taszkban kiadunk egy STI utasítást, a processzor 13-as kivételt (#GP) generál, ezzel lehetőséget adva a függő megszakítás lekezelésére. Ha VIF=1, VIP=1 és VME=1 egy utasítás végrehajtása előtt, akkor is ez történik.

A VIP flag-et a processzor csak olvassa, de sosem módosítja tartalmát közvetlenül. (Kivételes eset a POPFD utasítás, ami kitörölheti a VIF és VIP flag-eket, ha CPL≠3.)

A szoftver-megszakítások (tehát az INT utasítás által kiváltott megszakítások) kezelése sokkal összetettebb. A processzor 6 különböző módon képes ezeket a megszakításokat kiszolgálni. Hogy melyik módszert választja, az három dologtól függ. Először is a CR4 regiszter VME bitjének állása határozza meg, a klasszikus vagy az új módszert kell-e alkalmazni.

Másrészt az EFlags regiszter IOPL mezőjének értéke befolyásolja, mit tegyen a processzor. A harmadik tényező egy újdonság, neve pedig *megszakítás-átirányító bittérkép* (software interrupt redirection bit map). Ez egy 32 bájtos terület, ami a virtuális taszkok I/O-engedélyező bittérképe (I/O permission bit map) előtt helyezkedik el közvetlenül. A 32 bájtnak mindegyik bitje egy-egy szoftver-megszakításra van hatással. A 0. bájt 0. bitje az INT 00h-hoz tartozik, míg a 31. bájt 7. bitje az INT 0FFh lekezelését befolyásolja. Ha a kérdéses megszakításhoz tartozó bit értéke 1 a megszakítás-átirányító bittérképben, a megszakítást a megszokott módon, a védett módú kezelő szolgálja ki az IDT-n keresztül, ami azután meghívhatja a virtuális taszk saját kezelőjét is. Ha azonban ez a bizonyos bit törölve van, a megszakítás a védett módú kezelőt kihagyva közvetlenül a virtuális módú kezelőhöz irányítódik.

A következő táblázat mutatja az említett bitek és flag-ek kombinációjából eredő módszereket:

VME	*IOPL*	*Átir. bit*	*Módszer*
0	3	–	1
0	<3	–	2
1	<3	1	3
1	3	1	4
1	3	0	5
1	<3	0	6

Az egyes módszerek a következőket teszik:

1) A megszakítást a védett módú kezelő szolgálja ki:

- **0-ás szintű veremre vált**
- **a 0-ás verembe rakja a GS, FS, DS, ES regisztereket**
- **törli a GS, FS, DS, ES regiszterek tartalmát**
- **a 0-ás verembe rakja SS, ESP, EFlags, CS, EIP tartalmát**
- **törli a VM és TF flag-eket**

- **ha megszakításkapun (interrupt gate) keresztül történik a kiszolgálás, IF-et törli**
- **CS:EIP-t beállítja a kezelő címére a kapuból**

2) A 13-as kivétel (#GP) védett módú kezelője hívódik meg.

3) A 13-as kivétel (#GP) védett módú kezelője hívódik meg. A VIF és VIP flag-eket használni lehet a maszkolható hardver-megszakítások kiszolgálására.

4) Megegyezik az 1-es módszerrel.

5) A megszakítást a virtuális módú kezelő szolgálja ki:
- **a verembe rakja az EFlags tartalmát úgy, hogy NT=0 és IOPL=0 a másolatban**
- **a verembe rakja a CS, IP regiszterek tartalmát**
- **törli az IF és TF flag-eket**
- **CS:IP-t beállítja a virtuális taszk megfelelő megszakítás-vektorából**

6) A megszakítást a virtuális módú kezelő szolgálja ki, és a VIF és VIP flag-eket is használni lehet a maszkolható hardver-megszakítások kiszolgálására:
- **a verembe rakja az EFlags tartalmát úgy, hogy IOPL=3 és IF=VIF a másolatban**
- **a verembe rakja a CS, IP regiszterek tartalmát**
- **törli az VIF és TF flag-eket**
- **CS:IP-t beállítja a virtuális taszk megfelelő megszakítás-vektorából**

A *Protected mode Virtual Interrupt extension* (PVI) szolgáltatás a maszkolható hardver-megszakítások (IRQ0.. IRQ15) kezelését bővíti ki az előbb ismertetett módszerrel a védett módban. Ezt engedélyezni a CR4 regiszter PVI (1-es) bitjének beállításával lehet. Fontos, hogy itt csak a védett módról van szó, a virtuális módra ez a szolgáltatás nem vonatkozik. Ezért feltételezzük, hogy VM=0 az EFlags-ben. Szintén fontos tudni, hogy ez a szolgáltatás csak a maszkolható

hardver-megszakítások kiszolgálására van hatással. A szoftver-megszakítások, a hardver által kiváltott megszakítások és a kivételek az eddig megszokott módon lesznek lekezelve.

Ha PVI=0, IOPL=3, vagy CPL<3, akkor a CLI és STI utasítások a megszokott módon működnek. Tehát ha CPL=0, az utasítások átállítják IF-et. Ha CPL>0 és CPL≤IOPL, ugyancsak IF értéke változik meg. Ha CPL>0 és CPL>IOPL, 13-as kivétel (#GP) keletkezik.

Ha PVI=1, CPL=3 és IOPL<3, a helyzet hasonló a VME=1 és virtuális módú taszkok esetén leírtakhoz. Ekkor a CLI és STI utasítások a VIF flag-et módosítják IF helyett. Továbbá a megszakítás-kezelőnek lehetősége van a VIF flag alapján a VIP beállítására (ha VIF=0). Ha VIF=1 és VIP=1, vagy VIF=0, VIP=1 és a 3-as szintű taszk kiadja az STI utasítást, 13-as kivétel keletkezik. A kivétel-kezelő ilyenkor kiszolgálhatja a függőben levő megszakítást.

A PUSHF, PUSHFD, POPF, POPFD, IRET és IRETD utasításokra ez a szolgáltatás nincs hatással, a VME-től eltérően.

A *Debugging Extensions* (DE) szolgáltatás lehetővé teszi I/O töréspontok megadását. A CR4 regiszter DE (3-as) bitjét bekapcsolva engedélyezhetjük ezt. A DR4 és DR5 regiszterekre való hivatkozás ilyenkor illegálisnak számít. Cserébe viszont a DR7 regiszter R/W mezőjében az eddig fenntartott 10b beállítással egy olyan töréspontot adhatunk meg, ami portok megadott tartományára íráskor vagy onnan olvasáskor 1-es kivételt (#DB) vált ki. A tartomány kezdetét a DR0..DR3 regiszterek valamelyikének alsó 16 bitje tartalmazza, míg a tartomány hosszát a DR7 regiszter megfelelő LEN mezőjének értéke határozza meg.

A Pentium processzor tudása már a korábbi processzorok tudásának sokszorosa. A rengeteg funkció egy részét speciális regisztereken, a *modell-specifikus regisztereken* (Model-Specific Register–MSR) keresztül érhetjük el, ill. ezeken keresztül szabályozhatjuk azok működését. Ezek olyan 64 bites tárolóhelyek, amiket csak rendszerutasítások segítségével olvashatunk és írhatunk. MSR-be írni a WRMSR, onnan olvasni az RDMSR utasítással lehet.

Az Intel úgy definiálta az MSR-eket, hogy ezek ténylegesen egy adott processzormodellhez tartoznak, és semmi (senki) sem garantálja, hogy mondjuk egy processzorcsalád későbbi modelljei ismerni fogják az elődök MSR-jeit.

Az MSR-ek célja igen sokféle lehet: néhány MSR a cache és a TLB tesztelését segíti, mások a nyomkövetést (debugging) támogatják. Néhány speciális célú MSR-t mi is bemutatunk a következő bekezdésekben.

Mivel az MSR-ek nagy része a rendszerprogramozóknak, hardverfejlesztőknek szól, nem foglalkozunk velük részletesebben.

A *Performance-Monitoring Counters* (PMCs) olyan MSR-ek, amikkel a processzoron belül előforduló események, folyamatok előfordulásainak számát vagy időtartamukat mérhetjük, számlálhatjuk meg. A kapott adatokat a programok vagy a rendszer teljesítményének növelésére használhatjuk fel például. 3 MSR támogatja ezt a szolgáltatást: 1 MSR segítségével állíthatók be a figyelni kívánt események, míg 2 független számláló a kívánt események előfordulási számát avagy órajelekben mért időtartamát méri. A Pentium 42 különféle eseményt képes figyelni, a Pentium MMX pedig még 18 egyéb eseményt képes számolni vagy mérni.

A *Time-Stamp Counter* (TSC) egy olyan MSR, amelynek tartalmát resetkor törli a processzor, majd onnantól kezdve értéke minden órajel-ciklus alatt 1-gyel nő. Az Intel dokumentációja garantálja, hogy a 64 bites TSC értéke annak resetelése után 10 éven belül nem fog túlcsordulni. A TSC olvasására az RDTSC utasítás használható. Bár a Pentium-on MSR-ként van megvalósítva ez a számláló (méghozzá a 10h számú MSR-rel), az Intel szerint nem ajánlott a TSC-t az RDMSR utasítással olvasni, mivel elképzelhető, hogy a későbbi processzorok nem fogják támogatni ezt. A biztos módszer az RDTSC használata, miután a CPUID utasítással megbizonyosodtunk a TSC támogatottságáról. Ha a TSC-be mint MSR-be a WRMSR utasítással adatot írunk, a TSC tartalma kinullázódik.

A Pentium processzor sok szolgáltatásának célja a többprocesszoros rendszerek (multiple-processor systems) biztonságos, hatékony működésének biztosítása. Ilyen például a buszlezárás megfelelő alkalmazása, a különböző processzorokban levő cache-sek tartalmának összehangolása, szabályozott memóriahozzáférések, valamint a megszakítások központosított lekezelése és szétosztása a processzorok között.

Az utóbbi problémakört fedi le a *továbbfejlesztett programozható megszakítás-vezérlő* (Advanced Programmable Interrupt Controller–APIC). Az APIC két alapvető feladatot lát el:

- **Feldolgozza a kívülről érkező és a szoftver által generált megszakításokat.**
- **Többprocesszoros rendszerek esetén az alaplapon található külső APIC eszközzel kommunikál, ami azután a rendszertől és a processzoroktól származó megszakításokat összegyűjti, majd szétosztja azokat a processzorok között.**

Megkülönböztetés céljából a processzoron levő APIC egységet *helyi APIC*-nak (local APIC), az alaplapon levő külső egységet pedig *I/O APIC*-nak nevezi a szakirodalom.

A helyi APIC jelenlétét a CPUID utasítással ellenőrizhetjük, míg az I/O APIC a több processzort támogató alaplapok chipkészletében biztosan benne van. A helyi APIC az I/O APIC-kal egy külön e célra dedikált buszon (APIC bus) kommunikál. A helyi APIC-ot a memória bizonyos címein elhelyezkedő, ú.n. memóriába leképezett regisztereken (memory-mapped registers) keresztül programozhatjuk. Alapesetben ez a 0FEE00000h-0FEE01000h tartományt jelenti, de a regiszterek áthelyezhetők egy tetszőleges olyan címre, ami 4096-tal osztható.

A processzorban és az alaplapi eszközökben, erőforrásokban keletkező hardverhibák jelzésére és megfelelő lekezelésére a Pentium processzor támogatja a Machine Check kivételt (#MC, 18-as). Ha valamely eszköz vagy maga a processzor egy súlyos hibát észlel, akkor két dolgot tehet. Ha a CR4 regiszter MCE (6-os) bitje 1-es értékű, akkor 18-as kivétel keletkezik. Ez a kivétel abort típusú, így az éppen futó program vagy taszk állapota nem állítható vissza. Ha MCE=0, a processzor shutdown állapotba kerül (ez történik pl. a tripla hiba keletkezése esetén is). Az #MC kivétel lekezelése és annak kiváltó okai implementáció-specifikusak, így a későbbi processzorok nem fogják feltétlenül azonos módon kezelni ezt a szolgáltatást. A P6 architektúrában az MCE továbbfejlesztésének tekinthető az MCA (Machine Check Architecture).

17.7.4 Új üzemmódok

A *System Management Mode* (SMM) egy olyan speciális célú üzemmód, amit kifejezetten rendszerfunkciók, mint pl. teljesítmény-szabályozás (power management), hardver-vezérlés (hardware control), működés felfüggesztése (suspend) stb. megvalósítására fejlesztettek ki. Az SMM mód a processzor többi üzemmódjától függetlenül létezik, és feladatának megfelelően az alkalmazások, sőt még az operációs rendszer számára is teljesen átlátszóan teszi lehetővé az említett tevékenységek elvégzését.

Az egyetlen lehetőség az SMM módba váltásra az, hogy a processzor az SMI# kivezetésén vagy az APIC-buszon keresztül egy speciális megszakítást, *System Management Interrupt*-ot (SMI) kap. Az SMI egy nem maszkolható, külső hardver-megszakítás, és nagyobb precedenciájú mint az NMI vagy a maszkolható hardver-megszakítások. Ha SMM módban van a processzor, a további SMI-k fogadása le van tiltva, tehát az SMM nem újrabeléphető (nonreentrant) üzemmód.

Az SMM-be lépéskor a processzor egy új működési környezetbe vált át. Ezt a területet *System Management RAM*-nak (SMRAM) nevezik. Az aktuálisan futó taszk vagy program állapotát elmenti ennek a területnek egy kijelölt részére (ez az SMM state dump), majd elkezdi az SMI-kezelő program kódjának végrehajtását. Ha ez befejezte feladatát, az RSM utasítás végrehajtásával a processzor visszaállítja a korábbi környezetet, majd folytatja a megszakított program vagy taszk végrehajtását. Fontos megjegyezni, hogy bármilyen üzemmódban is legyen a processzor (valós, védett vagy virtuális-8086), az SMI hatására mindig SMM-be vált át.

Az SMM módot csak az RSM utasítás kiadásával lehet elhagyni. Ezt az utasítást más üzemmódban nem ismeri fel a processzor, így 6-os kivétellel (#UD) reagál végrehajtására.

Az SMRAM területét a fizikai címterület egy adott tartományára képezik rá. Az SMRAM nagysága minimum 32 Kbájt, maximum 4 Gbájt lehet, alapesetben pedig 64 Kbájt. A

terület kezdőcíme (amit SMBASE-nek neveznek) változtatható, alapértéke pedig a 00030000h fizikai cím. A processzor állapota az (SMBASE+0FE00h)..(SMBASE+0FFFFh) területre lesz elmentve, míg az SMI-kezelőt az (SMBASE+8000h) címtől kezdi végrehajtani.

Az SMRAM-ba sok mindent elment a processzor az SMM-be lépéskor, de a következőt állapotokat nem:

- **DR0..DR3 regiszterek**
- **FPU regiszterek**
- **CR2 regiszter**
- **MTRR-ek**
- **MSR-ek**
- **tesztregiszterek (TR3..TR7)**
- **APIC megszakítás-állapota**

Szintén nem lesznek elmentve egyéb, modell-specifikus funkciók állapotai, mint pl. a Pentium Pro MCA-regiszterei. A TSC (Time-Stamp Counter) és a PMC-k (Performance-Monitoring Counter) tartalmát természetüknél fogva nem lehet visszaállítani az SMM-ből visszatéréskor.

Az SMM-módban a futási környezet hasonló a valós-címzésű módhoz, mivel a CR0 regiszterben PE=0 és PG=0 beállítások lesznek. A következők a főbb eltérések a valós módhoz képest:

- **a megcímezhető memória területe 4 Gbájt (a P6 architektúrában támogatott PAE nem működik)**
- **a szegmensek határa a megszokott 64 Kbájt helyett 4 Gbájtra áll be**
- **az operandusméret és a címméret is 16 bit, viszont prefixekkel lehetővé válik 32 bites operandusok, 32 bites effektív cím és 32 bites címzési módok használata**
- **a maszkolható hardver-megszakítások, NMI, kivételek, SMI, töréspontok, single-step csapdák keletkezése tiltott**

Ha szükséges, az IDT megfelelő felépítésével, majd az IDTR regiszter beállításával a szoftver-megszakítások, a kivételek és a maszkolható hardver-megszakítások használhatóvá válnak. Az NMI-k kezelése ugyan tiltva van, de az IRET, IRETD utasítások végrehajtásakor a függő NMI kéréseket kiszolgálja a processzor.

17.7.5 Változások a kivétel-kezelésben

1-es kivétel (#DB) keletkezésének okait a Debugging Extensions szolgáltatás által a következő feltétellel kibővítették:

- **I/O olvasás vagy írás töréspont – trap**

A feltétel csak akkor érvényes, ha a CR4 regiszter DE (3-as) bitje be van állítva.

6-os kivétel (#UD) a következő feltételek valamelyikének fennállásakor is keletkezik:

- **a DR4 vagy DR5 regisztert úgy próbáltuk meg elérni, hogy a CR4 regiszter DE (3-as) bitje be volt állítva**
- **az RSM utasítást úgy próbáltuk meg végrehajtani, hogy nem SMM módban voltunk**
- **egy MMX utasítást próbáltunk meg végrehajtani egy olyan processzoron, ami nem támogatja az Intel MMX technológiát**
- **egy MMX utasítást próbáltunk meg végrehajtani úgy, hogy a CR0 regiszter EM (2-es) bitje be volt állítva**

Az utóbbi feltétel csak a Pentium MMX processzorokra vonatkozik.

7-es kivétel (#NM) Pentium MMX processzorok esetén a következő feltétel esetén is keletkezik:

- **egy MMX utasítást hajtottunk végre úgy, hogy a CR0 regiszter TS (3-as) bitje be volt állítva**

13-as kivétel (#GP) a következő feltételek esetén is generálódik:

- **valamely speciális célú regiszter (mint pl. CR4, CR3/PDBR, MXCSR stb.) egy fenntartott bitjébe 1-et próbáltunk meg írni**
- **nem létező MSR-be akartunk írni vagy onnan olvasni**
- **egy létező MSR fenntartott bitjébe próbáltunk meg 1-et írni**

A laphiba (14-es, #PF) kivétel hibakódja egy új bitet kapott:

Bitpozíció	*Név*
0	P
1	R/W
2	U/S
3	RSVD
4..15	–

Az új RSVD bit 1 értéke jelzi, ha a hiba azért keletkezett, mert a lapcímtár (page-directory) vagy egy laptábla (page-table) egyik bejegyzésében valamelyik fenntartott bit 1-es értékű volt, miközben a CR4 regiszter PSE (4-es) bitje be volt állítva.

17.7.6 Új kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
18	#MC	Machine Check	Abort	Modelltől függő

A 18-as kivétel keletkezése gépi hiba, buszhiba vagy valamely külső, alaplapi eszköz által észlelt súlyos hiba detektálását jelenti. A kivétel kiváltó okai és a kivétel lekezelésének módja processzormodelltől függőek, ezek implementációja

esetleg más lehet a későbbi architektúrákban. Hibakód nem kerül a verembe, a hiba jellemzését e célra fenntartott MSR-ek tartalmazzák. A kivétel abort típusú, a veremben levő CS:IP/EIP másolat által mutatott utasítás pedig általában nincs kapcsolatban a hibával. A megszakított program vagy taszk általában NEM indítható újra.

17.7.7 Változások a regiszterkészletben

Az EFlags regiszter 3 új flag-et kapott:

Bitpozíció	*Név*
19	VIF
20	VIP
21	ID

A VIF (Virtual Interrupt Flag) és VIP (Virtual Interrupt Pending) flag-ek a processzor VME és PVI szolgáltatásának használatakor játszanak szerepet, működésüket ezért lásd azok leírásánál.

Ha az ID (processor IDentification) flag állapota állítható, az adott processzor támogatja a CPUID utasítás használatát.

A CR0 regiszter CD és NW bitjeit a 80486-ostól eltérően értelmezi a Pentium:

CD	*NW*	*Következmény*
0	0	Cache engedélyezve, WB
0	1	–
1	0	Cache korlátozva, WB
1	1	Cache korlátozva

A WB jelölés a WriteBack kifejezést rövidíti, és annyit takar, hogy a módosított adatok először mindig a cache-be kerülnek be. Ha a rendszerbusz szabaddá válik, a processzor elvégzi a korábban csak a cache-be írt adatok memóriában való

frissítését, méghozzá nem egyenként, hanem csoportosan. Ez a WT típusú cache-kezelésnél optimálisabb megoldást nyújt.

A CR3 regiszterben, a lapcímtár- és a laptábla-bejegyzésekben levő PWT és PCD biteket szintén eltérően értelmezi a processzor:

PCD	*PWT*	*Következmény*
0	0	WB cache-selés
0	1	WT cache-selés
1	0	Nincs cache-selés
1	1	Nincs cache-selés

Ha a CR4 regiszter DE (3-as) bitje be van kapcsolva, néhány nyomkövetési funkció másképpen működik:

- **a DR4 és DR5 regiszterekre való hivatkozás 6-os (#UD) kivételt okoz**
- **a DR0..DR3 regiszterek tartalmát egy 16 bites portszámnak értelmezi a processzor, ha az adott töréspont típusa szerint I/O-töréspont (a felső szónak 0000h-nak kell lennie)**
- **a DR7 regiszter R/W0, R/W1, R/W2 és R/W3 mezőit eltérően értelmezi a processzor, amint ezt a következő táblázat is mutatja**

R/W	Töréspont feltétele
00	Csak utasítás-végrehajtás
01	Csak adat írása
10	Port olvasása (input) vagy írása (output)
11	Adat olvasása vagy írása, de nem utasítás-végrehajtás

A DR6 és a DR7 regiszter 12-es bitjeit ezentúl nem állíthatjuk be, ezek tartalma mindig 0 lesz.

A tesztregisztereket (TR3..TR7) eltörölték a Pentium processzortól kezdve, helyüket néhány MSR vette át. A rájuk való hivatkozás 6-os kivételt (#UD) vált ki.

17.7.8 Új regiszterek

Az architektúrában történt sok újítás, funkció engedélyezésére és vezérlésére szolgál egy új vezérlőregiszter, a CR4 regiszter. A többi társához hasonlóan ennek használata is privilegizált. Felépítése a következő táblázatban látható:

Bitpozíció	*Jelölés*	*Név*
0	VME	Virtual Mode virtual int. Extension
1	PVI	Protected mode Virtual Interrupts
2	TSD	Time-Stamp instruction Disable
3	DE	Debugging Extensions
4	PSE	Page Size Extension
6	MCE	Machine Check exception Enable
5,7..31	–	–

A VME, PVI, DE, PSE és az MCE bitek a már említett, azonos nevű szolgáltatásokat engedélyezik, ha értékük 1, ill. tiltják le, ha törölve vannak.

Ha TSD=0, az RDTSC utasítás bármikor kiadható. Ha viszont TSD=1, az utasítás privilegizálttá válik, azaz csak CPL=0 esetén hajtható végre, mert különben 13-as (#GP) kivétel fog keletkezni. Reset után ez a bit törölve van, erre érdemes odafigyelni.

Az 5 és 7..31 számú bitek fenntartottak. Ezeket a biteket ne állítsuk be, mert ezzel a jövő processzoraival való kompatibilitást veszélyeztetjük. Az RSM utasítás különösen érzékeny ezeknek a biteknek az állapotára.

A Pentium MMX processzoron 8 új MMX regiszter is megjelent, ezek az MM0..MM7 neveken érhetők el. Ezekről a következő, az MMX technológiát tárgyaló fejezetben lesz szó.

17.8 Az Intel MMX technológiája

Az MMX technológia egynél több egész operanduson képes egyszerre ugyanazt a műveletet elvégezni. Ennek támogatására új adatszerkezetek jelentek meg: pakolt bájtok, szavak, duplaszavak (packed bytes, words, doublewords), valamint kvadraszó (quadword). Ezek megérésére nézzünk egy példát! Ha összeadunk két, pakolt bájtokat tartalmazó operandus, akkor az eredmény szintén pakolt bájtokat fog tartalmazni. Így tehát egyszerre 8 számpárt adtunk össze, és a 8 eredményt egyetlen utasítással kaptuk meg.

A 64 bites kvadraszó típust eddig még nem használtuk, mivel mindegyik processzor regiszterei 16 vagy 32 bitesek. Az MMX technológia ezért 8 db. 64 bites adatregisztert is bevezetett, ezeket sorban MM0, MM1, …, MM7-nek nevezzük. Mindegyik regiszter közvetlenül elérhető, és csak adattárolásra és MMX műveletek operandusaként használható. A visszafele kompatibilitás miatt az MMX regiszterek a koprocesszor adatregisztereinek mantissza részére (0..63 bitek) vannak ráképezve. Ez azt jelenti, hogy az MM0 regiszter tartalma a fizikailag 0-adik FPU regiszterben (FPR0) van, ami azonban csak akkor egyezik meg ST(0)-val, ha TOS=0.

Az MMX utasítások a következő módon befolyásolják a koprocesszor állapotát:

- **az EMMS utasítást kivéve mindegyik MMX utasítás törli az FPU tag-regiszter tartalmát, ezzel mindegyik FPU adatregisztert érvényesnek jelölve be**

- **az EMMS utasítás 0FFFFh-t tölt az FPU tag-regiszterbe, ezzel mindegyik FPU adatregisztert üresnek jelöl be**
- **mindegyik MMX utasítás TOS értékét 000b-ra állítja**
- **ha egy MMX utasítás adatot tölt be egy MMX regiszterbe (MM0..MM7), az adott regiszterhez tartozó fizikai FPU adatregiszter 64..79 bitjeit (exponens és előjelbit) csupa 1-re állítja**

Az MMX regiszterek ilyen elhelyezése valóban leegyszerűsíti az operációs rendszer dolgát, mivel annak elég csak az FPU állapotát elmentenie (az FSAVE vagy FNSAVE utasításokkal), mert ezzel egyúttal az MMX állapotot is elmenti. Azonban problémát vagy inkább kényelmetlenséget is okozhat ez a megoldás (mármint az MMX regiszterek keverése az FPU regiszterekkel), mivel már említettük, hogy az EMMS-t kivéve az összes MMX utasítás érvényesnek jelöli be a koprocesszor adatregisztereit. Éppen e miatt nem ajánlott koprocesszor utasítások keverése MMX utasításokkal. A javasolt megoldás az lehet, hogy az MMX utasításokat tartalmazó blokkot egy EMMS utasítással zárjuk le, utána már nyugodtan használhatjuk az FPU-t.

Az új adattípusok felépítése az alábbi ábráról leolvasható:

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
Bájt 7	Bájt 6	Bájt 5	Bájt 4	Bájt 3	Bájt 2	Bájt 1	Bájt 0
Szó 3		Szó 2		Szó 1		Szó 0	
Duplaszó 1				Duplaszó 0			
Kvadraszó							

Egy MMX regiszterben tehát 8 db. bájt, 4 db. szó vagy 2 db. duplaszó operandus lehet egymástól függetlenül, de tekinthetjük az egész regisztert egyetlen kvadraszónak is.

Az MMX utasításokra jellemző, hogy a művelet eredményét vagy eredményeit nem jelzik a flag-ekben, így

például átvitel sem keletkezik a CF-ben. Aritmetikai kivételt szintén nem generál egyetlen MMX utasítás sem.

Az aritmetika típusairól is szóljunk néhány mondatot. A wrap-around aritmetika ugyanazt csinálja, amit az egész aritmetikás utasítások (ADD, ADC, SUB, SBB, INC, DEC stb.), tehát csak az eredmény átvitel nélküli részét tartják meg. A telítő aritmetika annyiban különbözik ettől, hogy ha a művelet elvégzése után átvitel keletkezik, akkor az eredmény a lehető legkisebb avagy legnagyobb szám értéke lesz attól függően, hogy melyik határon csúszott át az eredeti eredmény. A határok a számok szokásos ábrázolási tartományainak alsó és felső korlátai: előjeles esetben -128..+127, -32768..+32767 és -2147483648..+2147483647, míg előjeltelen operandusok alkalmazásakor 0..+255, 0..+65535 és 0..+4294967295 attól függően, hogy bájtosak, szavasak vagy duplaszavasak az operandusok.

Sok utasítás képes ugyanazt a műveletet elvégezni pakolt bájtokon, szavakon vagy duplaszavakon, az operandusok típusára ilyenkor a mnemonik utal (B–Byte, W–Word, D–Doubleword, Q–Quadword). Szintén a mnemonik mutatja a használt aritmetika jellegét is: előjeles körbeforgó (signed wrap-around), előjeltelen telítő (Unsigned Saturating–US), előjeles telítő (Signed Saturating arithmetic–SS, S).

Negatív értékek tárolására a már megszokott kettes komplemens alakot használják az MMX utasítások, ami lehetővé teszi, hogy ugyanazt a gépi kódot alkalmazzuk előjeles és előjeltelen wrap-around aritmetika esetén is.

A kétoperandusú utasításokra fennáll az a már megszokott tulajdonság, hogy az első operandus a cél (néhány esetben forrás is), míg a második operandus a forrás. Az adatmozgató MMX utasításokat leszámítva minden más esetben teljesülnie kell az operandusokra, hogy a cél csak egy MMX regiszter lehet, míg a forrás helyére MMX regiszter vagy memóriahivatkozás kerülhet. Az adatmozgató utasításoknál

azonban mind a cél, mind a forrás lehet egy 32 bites általános célú regiszter, MMX regiszter vagy memóriahivatkozás is.

Prefixeket csak a következő megkötésekkel használhatunk MMX utasításokkal: szegmensfelülbíráló (segment override; 26h, 2Eh, 36h, 3Eh, 64h, 65h) és címhossz (address size; 67h) prefix bármelyik MMX utasítás előtt megengedett, az operandushossz (operand size; 66h) és a sztringutasítást ismétlő (string instruction repeat; 0F2h, 0F3h) prefixek használata fenntartott, míg a buszlezáró (bus lock; 0F0h) prefix minden esetben 6-os kivételt (Invalid/Undefined Opcode–#UD) generál.

Ha egy MMX utasítás kiadásakor a CR0 regiszter EM bitje be van állítva, szintén 6-os kivétel keletkezik.

Az alábbiakban olvasható az MMX utasítások jellemzése: a mnemonik után látható az operandusok típusa, a mnemonik jelentése angolul, majd az utasítás működésének leírása következik. A PACK… kezdetű mnemonikokat leszámítva az első "P" betű utal arra, hogy az adott utasítás pakolt típusú operandusokat használ.

A következő jelöléseket alkalmaztuk:

- **(no op) – nincs operandus**
- **source – forrás; mmreg/mem64 lehet**
- **dest – cél (destination); mmreg lehet**
- **reg/mem32 – 32 bites általános regiszter vagy memóriaoperandus**
- **mmreg – az MM0, MM1, …, MM7 regiszterek valamelyike**
- **mmreg/mem64 – MMX regiszter vagy 64 bites memóriaoperandus**

17.8.1 Adatmozgató utasítások

- **MOVD mmreg,reg/mem32**
- **MOVD reg/mem32,mmreg**
  (MOVe Doubleword to/from MMX register)

A forrást átmásolja a cél területére. Ha a cél az MMX regiszter, akkor a forrás zéró-kiterjesztéssel lesz átmásolva. Ha a forrás az MMX regiszter, akkor annak az alsó duplaszavát másolja a célba.

- **MOVQ mmreg,mmreg/mem64**
- **MOVQ mmreg/mem64,mmreg**
  (MOVe Quadword to/from MMX register)
  A forrást bemásolja a célba.

17.8.2 Konverziós utasítások

- **PACKSSWB dest, source**
  (PACK Words into Bytes, Signed with Saturation)
- **PACKSSDW dest, source**
  (PACK Doublewords into Words, Signed with Saturation)
- **PACKUSWB dest, source**
  (PACK Words into Bytes, Unsigned with Saturation)
  A PACKSSWB a forrásban és a célban levő 8 db. előjeles szót 8 db. bájtra telíti, majd azokat a célban eltárolja. A PACKSSDW a forrásban és a célban levő 4 db. előjeles duplaszót 4 db. szóra telíti, majd azokat a célban eltárolja. Végül a PACKUSWB a forrásban és a célban levő 8 db. előjeltelen szót 8 db. bájtra telíti, majd azokat a célban eltárolja. A forrás operandus telített bájtjai/szavai az eredmény alsó részébe kerülnek. Ez pl. a PACKSSWB esetében a következőket jelenti: az eredmény 0..7 bitjeibe a forrás 0..15 bitjeinek telített értéke kerül, a 8..15 bitekbe a forrás 16..31 bitjeinek telített értéke íródik, …, a 32..39 bitek a cél 0..15 bitjeinek telített értékét tartalmazzák, stb.
- **PUNPCKHBW dest,source**

(UNPaCK High-order Bytes of Words)

- **PUNPCKHWD dest,source**
  (UNPaCK High-order Words of Doublewords)
- **PUNPCKHDQ dest,source**
  (UNPaCK High-order Doublewords of Quadwords)

  A forrás és a cél felső bájtjait, szavait vagy duplaszavait felváltva az eredménybe írják, majd azt a cél területén eltárolják. A sorrend a következő: egy elem a célból, egy a forrásból, a következő elem a célból, majd egy a forrásból, stb. Így pl. a PUNPCKHBW esetén az eredmény 0..7 bitjeibe a cél 32..39 bitjei kerülnek, a 8..15 bitek a forrás 32..39 bitjeit tartalmazzák, a 16..23 bitekbe a cél 40..47 bitjei íródnak stb.
- **PUNPCKLBW dest,source**
  (UNPaCK Low-order Bytes of Words)
- **PUNPCKLWD dest,source**
  (UNPaCK Low-order Words of Doublewords)
- **PUNPCKLDQ dest,source**
  (UNPaCK Low-order Doublewords of Quadwords)

  A forrás és a cél alsó bájtjait, szavait vagy duplaszavait felváltva az eredménybe írják, majd azt a cél területén eltárolják. A sorrend a következő: egy elem a célból, egy a forrásból, a következő elem a célból, majd egy a forrásból, stb. Így pl. a PUNPCKLBW esetén az eredmény 0..7 bitjeibe a cél 0..7 bitjei kerülnek, a 8..15 bitek a forrás 0..7 bitjeit tartalmazzák, a 16..23 bitekbe a cél 8..15 bitjei íródnak stb.

17.8.3 Pakolt aritmetikai utasítások

- **PADDB dest,source**
  (ADD Bytes, wrap-around)

- **PADDW dest,source**
  (ADD Words, wrap-around)
- **PADDD dest,source**
  (ADD Doublewords, wrap-around)
- **PADDSB dest,source**
  (ADD Bytes, Signed with saturation)
- **PADDSW dest,source**
  (ADD Words, Signed with Saturation)
- **PADDUSB dest,source**
  (ADD Bytes, Unsigned with saturation)
- **PADDUSW dest,source**
  (ADD Words, Unsigned with Saturation)

  Összeadják a forrásban és a célban levő elemeket egyenként, az eredményeket egy 64 bites típusba pakolják, majd eltárolják a cél regiszterbe. Az összeadás történhet előjeles és előjeltelen operandusokon is, és mindkét esetben lehet körbeforgó és telítő aritmetikát is használni.
- **PSUBB dest,source**
  (SUBtract Bytes, wrap-around)
- **PSUBW dest,source**
  (SUBtract Words, wrap-around)
- **PSUBD dest,source**
  (SUBtract Doublewords, wrap-around)
- **PSUBSB dest,source**
  (SUBtract Bytes, Signed with saturation)
- **PSUBSW dest,source**
  (SUBtract Words, Signed with saturation)
- **PSUBUSB dest,source**
  (SUBtract Bytes, Unsigned with saturation)
- **PSUBUSW dest,source**
  (SUBtract Words, Unsigned with saturation)

  Kivonják a célban levő elemekből a forrásban levő elemeket egyenként, az eredményeket egy 64 bites

típusba pakolják, majd eltárolják azt a cél regiszterbe. Az összeadás történhet előjeles és előjeltelen operandusokon is, és mindkét esetben lehet körbeforgó és telítő aritmetikát is használni.

- **PMULHW dest,source**
  (MULtiply signed Words, and store High-order words of results)
  Összeszorozza a forrásban és a célban levő előjeles szavakat, az eredmények felső szavát bepakolja egy 64 bites típusba, majd eltárolja azt a célba.
- **PMULLW dest,source**
  (MULtiply signed Words, and store Low-order words of results)
  Összeszorozza a forrásban és a célban levő előjeles szavakat, az eredmények alsó szavát bepakolja egy 64 bites típusba, majd eltárolja azt a célba.
- **PMADDWD dest,source**
  (Multiply signed Words, and ADD the resulting Doublewords)
  Összeszorozza a forrásban és a célban levő előjeles szavakat, ennek eredménye négy előjeles duplaszó lesz. Ezután a felső négy szóból képződött két duplaszót összeadja, és az összeget bepakolja az eredmény felső felébe. Hasonlóan az alsó négy szó két szorzatát összeadja, az összeg az eredmény alsó felébe kerül. Végül az eredményt eltárolja a cél regiszterbe.

17.8.4 Összehasonlító utasítások

- **PCMPEQB dest,source**
  (CoMPare Bytes for EQual)
- **PCMPEQW dest,source**
  (CoMPare Words for EQual)

- **PCMPEQD dest,source**
  (CoMPare Doublewords for EQual)
- **PCMPGTB dest,source**
  (CoMPare signed Bytes for Greater Than)
- **PCMPGTW dest,source**
  (CoMPare signed Words for Greater Than)
- **PCMPGTD dest,source**
  (CoMPare signed Doublewords for Greater Than)

  Összehasonlítják a forrásban levő elemeket a célban levőkkel (tehát a forrás elemeket kivonják a cél elemekből), majd az eredményeknek megfelelően kitöltik a cél elemeket: ha a megadott feltétel teljesül (azaz a két elem egyenlő, vagy a cél elem nagyobb a forrás elemnél), akkor a cél elembe 11…1b kerül, különben pedig 00…0b lesz a cél elem értéke. Így pl. ha a cél operandus értéke 0EEEEEEEE22222222h, a forrás operandusé pedig 3333333322222222h, akkor a PCMPEQD és a PCMPGTD utasítások eredménye sorban 00000000FFFFFFFFh és 0000000000000000h lesz.

17.8.5 Logikai utasítások

- **PAND dest,source**
  (bitwise logical AND)
- **PANDN dest,source**
  (bitwise logical AND plus NOT)
- **POR dest,source**
  (bitwise logical OR)
- **PXOR dest,source**
  (bitwise logical eXclusive OR)

  A forrás és a cél operandusok mint 64 bites számok között végeznek bitenként valamilyen logikai műveletet, majd az eredményt a cél operandusba

írják. A PANDN kicsit szokatlan utasítás, ugyanis a cél egyes komplemensét hozza a forrással logikai ÉS kapcsolatba.

17.8.6 Shiftelő utasítások

- **PSLLW/PSLAW dest,count**
  (Shift Words Logical/Arithmetical Left)
- **PSLLD/PSLAD dest,count**
  (Shift Doublewords Logical/Arithmetical Left)
- **PSLLQ/PSLAQ dest,count**
  (Shift Quadword Logical/Arithmetical Left)
- **PSRLW dest,count**
  (Shift Words Logical Right)
- **PSRLD dest,count**
  (Shift Doublewords Logical Right)
- **PSRLQ dest,count**
  (Shift Quadword Logical Right)
- **PSRAW dest,count**
  (Shift Words Arithmetical Right)
- **PSRAD dest,count**
  (Shift Doublewords Arithmetical Right)

A cél elemeit a megadott lépésszámmal balra vagy jobbra léptetik, az eredmény(eke)t bepakolják egy 64 bites típusba, majd azt a cél regiszterbe visszaírják. A léptetés előjelesen és előjeltelenül is történhet, az elemek pedig szavak, duplaszavak lehetnek, az aritmetikai jobbra léptetést leszámít-va pedig kvadraszó is lehet a céloperandus. A forrásoperandus itt kivételesen közvetlen bájt érték (immediate byte) is lehet.

17.8.7 Állapot-kezelő utasítások

- **EMMS (no op)**
  (Empty MMX State)
  A koprocesszor tag-regiszterbe 0FFFFh-t tölt, így mindegyik FPU adatregisztert üresnek jelöli be.

17.9 Az Intel Pentium Pro friss tudása

17.9.1 Új utasítások

- **CMOVccc dest reg16/32,source**
  (Conditional MOVe data)
  A forrás tartalmát a céloperandusba másolja, ha a megadott feltétel teljesül. A "ccc" egy olyan szimbólumot jelöl, ami a használandó feltételt határozza meg. Ezek a feltételek megegyeznek a feltételes ugrásoknál (Jccc utasítások) használhatókkal, de a CXZ feltétel itt nem adható meg. Az operandusok mérete 16 vagy 32 bit lehet. Céloperandusként csak általános regisztert, forrásként pedig általános regisztert vagy memóriaoperandust adhatunk meg. Az utasítás az EFlags megfelelő flag-jeit vizsgálja.
- **FCMOVccc ST,ST(i)**
  (Floating-point Conditional MOVe)
  Ezek az utasítások valamilyen feltétel teljesülése esetén ST(i) tartalmát ST-be mozgatják. Az utasítások az EFlags bitjeit veszik figyelembe, nem pedig az FPU státuszregiszterében levő feltételbiteket (condition code bits). Feltételként az E/Z, NE/NZ, B/C/NAE, BE/NA, NB/NC/AE, NBE/A, U és NU szimbólumok használhatók. Ez egy koprocesszor utasítás, így bővebb leírása megtalálható a "A

numerikus koprocesszor szolgáltatásai" című fejezetben.

- **FCOMI ST,ST(i)**
  (COMpare real and set EFlags)
- **FCOMIP ST,ST(i)**
  (COMpare real, set EFlags, and Pop)
- **FUCOMI ST,ST(i)**
  (COMpare real Unordered and set EFlags)
- **FUCOMIP ST,ST(i)**
  (COMpare real Unordered, set EFlags, and Pop)

  Az utasítások a forrás tartalmát összehasonlítják a céllal, majd ennek megfelelően beállítják a CF, PF és ZF flag-eket. Az FCOMIP és FUCOMIP utasítások az összehasonlítás után egy POP-ot hajtanak végre. Az FUCOMI és FUCOMIP utasítások QNaN operandusok esetén nem generálnak érvénytelen operandus kivételt (#IA), a másik két utasítás viszont igen. Az utasítások részletesebb leírása a koprocesszort tárgyaló fejezetben található meg.
- **RDPMC (no op)**
  (ReaD Performance-Monitoring Counter)

  Az ECX regiszterben megadott számú PMC tartalmát az EDX:EAX regiszterpárba tölti. ECX értéke csak 00000000h vagy 00000001h lehet, különben 13-as kivétel (#GP) keletkezik. Az utasítás végrehajthatósága a CR4 regiszter PCE (8-as) bitjének állásától függ. Ha PCE=1, bármilyen privilegizálási szinten és üzemmódban végrehajtható, különben pedig csak CPL=0 esetén.
- **UD2 (no op)**
  (UnDefined instruction)

Az utasítás egyetlen feladata, hogy garantáltan 6-os kivételt (#UD) váltson ki a processzor bármely üzemmódjában, bármely CPL érték mellett. Hatása egyébként megfelel egy NOP-nak, mivel egyetlen regiszter vagy flag értékét sem változtatja meg, persze IP-t/EIP-t kivéve.

A felsorolt utasítások közül egy sem privilegizált, de az RDPMC privilegizáltként viselkedik, ha a CR4 regiszterben PCE=0.

17.9.2 Változások az architektúrában

Nem kifejezetten újítás, hanem inkább hiba az, hogy a P6 architektúrájú processzorok néhány ismételt sztringutasítás esetén nem képesek a töréspontón fennakadó utasítást pontosan megjelölni. A REP INS, REP MOVS, REP OUTS és REP STOS utasítások esetén az adattöréspont elérése esetén az adott iterációs lépés végrehajtása után keletkezik csak 1-es kivétel (#DB). (Tehát a processzor végrehajtja egyszer az adott sztringutasítást az aktuális regiszterértékekkel, ezután generál egy 1-es kivételt, majd a REP prefix leállási feltételeit kezdi vizsgálni.) Ezen kívül a REP INS és REP OUTS utasítások esetén, ha az utasítások által használt portszám egy töréspont kiváltó feltétele, akkor a legelső iterációs lépés végrehajtása után keletkezik csak a debug kivétel.

Ha a CR3 regiszter tartalmát gyakran módosítjuk, mint pl. taszkváltáskor, a rendszeresen használt memórialapok (mint pl. a kernel területe) TLB-bejegyzéseit feleslegesen érvényteleníti a processzor, mivel azok mindvégig állandóak maradnak. Ezt az ellentmondást oldja meg a *globális lapok* használata. A laptábla- és lapcímtár-bejegyzésekben a 8-as bit

eddig fenntartott volt, mostantól viszont funkciója van, neve G (Global page). Ha G=1, az adott lap globális. Csak 2 Mbájtos vagy 4 Mbájtos lapra mutató lapcímtár-bejegyzés esetén létezik ez a bit, a 4 Kbájtos lapokat a laptábla-bejegyzés G bitjével jelölhetjük be globálisnak. A CR4 regiszter 7-es, PGE (Page Global Enable) bitjét beállítva a processzor a TLB-k érvénytelenítésekor figyelembe veszi a G bit állását, és ha G=1, az adott TLB-bejegyzés változatlan marad. Ha PGE=0, a G bit állásától függetlenül megtörténik az érvénytelenítés.

Ha valakinek a 4 Gbájtos megcímezhető virtuális memória kevés lenne, az ne csüggedjen. A Pentium Pro processzor újdonságként lehetővé teszi a 32 bites címterület 36 bitesre való kiterjesztését. A technika neve *Physical Address Extension* (PAE). A CPUID utasítással eldönthető, hogy az adott processzor ismeri-e a PAE-t. Használatához a CR4 regiszter PAE (5-ös) bitjét be kell állítanunk.

A PAE használatakor a lapozás az eddigiektől eltérő módon történik. A laptáblák és a lapcímtár megmaradtak, de a bejegyzések mérete immáron 64 bit lett. Ezáltal "csak" 512 bejegyzés fér el egy 4 Kbájtos lapban. A lapok mérete 4 Kbájt vagy 2 Mbájt lehet. (Az Intel dokumentációja szerint 4 Mbájtos lap is létezik, de ez elég furcsa lenne, mivel a lineáris címben az offszet csak 21 bites…) A laptáblák bejegyzései továbbra is 4 Kbájtos lapokra mutatnak. Lapcímtárból ezentúl már 4 db. van, és ezek függetlenek egymástól. A lapcímtár bejegyzései vagy 2 Mbájtos lapokra, vagy 4 Kbájtos laptáblákra mutatnak. Hogy melyikre, azt a CR4 regiszter PSE (4-es) és a lapcímtár-bejegyzés PS (7-es) bitjének állása dönti el.

A 4 db. lapcímtár báziscímét egy új táblázat, a *lapcímtár-mutató tábla* (Page-Directory-Pointer Table–PDPT) tartalmazza. A PDPT báziscímének 32-vel oszthatónak kell lennie. A 32 bites PDPT-báziscím felső 27 bitjét a CR3 regiszter 5..31

bitjei tárolják, ezért ezt a regisztert ilyenkor lapcímtár-mutató tábla bázisregiszternek is hívják (Page-Directory-Pointer Table Base Register–PDPTR vagy PDPTBR). (A báziscím itt is fizikai címet jelent, a PDBR-hez hasonlóan.)

A lineáris cím felépítése különböző lesz 4 Kbájtos és 2 Mbájtos lapok használata esetén. A cím 30..31 bitjei választják ki a használni kívánt lapcímtárat. A 21..29 bitek határozzák meg a lapcímtár-bejegyzést.

Ha ez a bejegyzés 4 Kbájtos lapra mutat (azaz a CR4 regiszterben PSE=0, vagy PSE=1 és PS=0), akkor a 12..20 bitek választják ki a lapcímtár-bejegyzés által mutatott laptábla valamely bejegyzését. Végül a 0..11 bitek tartalmazzák a lapon belüli offszetet.

Ha a lapcímtár-bejegyzés 2 Mbájtos lapra mutat (azaz PSE=1 és PS=1), a 0..20 bitek adják meg a lapon belüli offszetet.

Mindkét esetben 4 Gbájt méretű terület címezhető meg egyszerre. A lapcímtár-mutató tábla, a lapcímtárak és a laptáblák bejegyzéseiben viszont az adott tábla ill. lap 36 bites báziscímének megfelelő bitjei találhatók meg, így lehetővé válik a fizikai 4 Gbájt feletti címek kihasználása is.

Néhány P6 architektúrájú processzor egy másik olyan technikát is támogat, ami lehetővé teszi a címterület 36 bitesre való kiterjesztését de úgy, hogy a jól megszokott 4 bájtos bejegyzéseket és 4 Mbájtos lapokat használjuk. Ennek neve *36-bit Page Size Extension* (PSE-36). A CPUID utasítással eldönthető, hogy az adott processzor ismeri-e a PSE-36-ot. Használatához a CR4 regiszter PSE (4-es) bitjét be kell állítanunk, PAE (5-ös) bitjét viszont törölni kell.

A lapozási hierarchia a szokásos marad, tehát az egyetlen lapcímtár bejegyzései 4 Mbájtos lapokra mutatnak (PS=1). A lapcímtár báziscímét továbbra is CR3 (PDBR) tárolja, ennek a címnek 4096-tal oszthatónak kell lennie. A 4 Mbájtos lapok

báziscíme 36 bites lett, ennek a felső 24 bitjét a lapcímtár bejegyzések tárolják, az alsó 12 bitet 0-nak veszi a processzor. A lapcímtár-bejegyzések eddig fenntartott 13..16 bitjei tárolják a lap báziscímének felső 4 bitjét.

A számítógépben levő hardvereszközök és perifériák különböző típusú memóriákat tartalmazhatnak. Így például sok videovezérlő kártya támogatja a lineáris lapkeretet (Linear Frame Buffer–LFB), ami annyit jelent, hogy a videomemória teljes területét elérhetővé teszi a fizikai címterület kihasználatlan részén (pl. a 0E0000000h címen, ahol általában nincs alaplapi memória). A processzor minden memóriaelérést a cache-sen keresztül bonyolít le. Az LFB tartalma viszont elég gyakran változik (lévén hogy videomemóriáról van szó), így felesleges lenne a cache-ben levő, esetleg sokkal fontosabb adatokat felülírni az LFB-ből származókkal. Másrészt, ha mégis szükség van ezekre a videoadatokra, teljesen mindegy, hogy a processzor milyen sorrendben végzi el a cache-ből a videomemóriába írást, előbb-utóbb úgy is meg fog jelenni annak tartalma a monitoron. A P5 architektúrától kezdve lehetőség van arra, hogy a processzor nem feltétlenül a megadott sorrendben végezze el a memóriába írásokat az optimális sebesség céljából. Ilyen és ehhez hasonló, a memória cache-selését érintő probléma megoldására vezették be a *Memory Type Range Register*-eket (MTRRs).

Az MTRR-ek olyan MSR-ek (MSR=Model-Specific Register), amik lehetővé teszik, hogy a memória 96 db. különböző összefüggő területére előírjuk, az adott helyeken a cache-selés szempontjából milyen memóriatípus van. A 96-ból 88 db. tartomány rögzített báziscímű és méretű, a maradék 8 tartományra viszont mi írhatjuk elő, hol kezdődik, meddig tart, és hogyan kezelje az ottani memóriaeléréseket a processzor.

A 0..7FFFFh tartományt 8 db. 64 Kbájtos, a 80000h..0BFFFFh tartományt 16 db. 16 Kbájtos, míg a 0C0000h..0FFFFFh tartományt 64 db. 4 Kbájtos "szegmensre" osztották, ezek mindegyikére csak a memória típusa adható meg. A szabadon állítható 8 db. MTRR-ben az adott tartomány 36-bites báziscímének felső 24 bitjét, a tartomány hosszát, valamint a memória típusát állíthatjuk be.

A memóriatípust mindegyik MTRR-ben egy 8 bites mezőben adhatjuk meg, ennek a következő a jelentése:

Típ.	*Érték*	*Cache*	*WB*	*Spek. olv.*	*Ordering Model*
UC	0	N/N	N	N	Erős
WC	1	N/N	N	I	Gyenge
WT	4	I/I	N	I	Spekulatív
WP	5	I/N	N	I	Spekulatív
WB	6	I/I	I	I	Spekulatív
–	2,3,7..255	–	–	–	–

A második oszlop tartalmazza az MTRR adott mezőjének tartalmát, az első oszlop pedig a mező értéke által kiválasztott memóriatípus nevét mutatja. A harmadik oszlopban azt láthatjuk, ezt a területet cache-seli-e a processzor olvasáskor (első válasz) ill. íráskor (második válasz). A negyedik oszlopban azt láthatjuk, az adott terület visszaíró (WriteBack) módon cache-selhető-e. Az ötödik oszlop mutatja, megengedettek-e a spekulatív olvasások erről a területről. Az "I" válasz "igen"-t, az "N" pedig "nem"-et jelent.

Az Ordering Model fogalom arra a sorrendre vonatkozik, amilyenben a processzor a memóriakéréseket (olvasás és írás) a rendszerbuszra helyezi. Ennek típusát a hatodik oszlop mutatja. A szigorú (strict), más néven erős (strong) modell a programban való előfordulási sorrendnek felel meg. A gyenge (weak) modell lehetővé teszi, hogy az egymást követő sorozatos írásokat összevonva, de esetleg más sorrendben végezze el a processzor.

A spekulatív (speculative) modell pedig a módosított memóriaterületek cache-selését is lehetővé teszi, így a módosítások csak az írást követően jóval később kerülnek be a rendszermemóriába, s hogy mikor, azt a processzor dönti el (ettől spekulatív).

A memóriatípust megadó értékek közül a 2, 3 és a 7..255 értékek fenntartottak, ezek alkalmazása 13-as (#GP) kivétel keletkezését vonja maga után.

A memóriatípusok elnevezése: UnCacheable, Write Combining, Write-Through, Write-Protected, WriteBack. Velük nem foglalkozunk részletesebben, bővebb információért az Intel dokumentációját ajánlott forgatni.

A MTRR-ekhez hasonló célt szolgál a *Page Attribute Table* (PAT) technika. Ennek lényege, hogy az egyes memóriatípusokat a lapok szintjén definiálhatjuk, és ez nagyobb rugalmasságot biztosít az operációs rendszernek. A CPUID utasítással ellenőrizhető, hogy az adott processzor támogatja-e ezt a szolgáltatást. A PAT technikát nem lehet sem engedélyezni sem tiltani. Ha az adott processzor támogatja, akkor bármelyik lapozási típusnál (normál, PSE, PAE vagy PSE-36) használni fogja.

A laptábla-bejegyzések eddig fenntartott 7-es bitjét, ill. a lapcímtár-bejegyzések eddig szintén fenntartott 12-es bitjét ezentúl *PAT Index* (PATi) bitnek nevezik. A PDBR, PDPTBR regiszterek, ill. a lapcímtár-mutató tábla-, a lapcímtár- és a laptábla-bejegyzések eddig meglevő PWT (3-as) és PCD (4-es) bitjei a PATi bittel együtt egy 3 bites bináris számot alkotnak, ennek alakja PATi:PCD:PWT (PWT a legalsó bit).

Az újonnan bevezetett 00000277h számú MSR-t hívják PAT-nak. Ebben a 64 bites regiszterben mindegyik bájt egy-egy attribútum-mező, de ezekben csak az alsó 3 bitet lehet használni, a maradék 5 bit fenntartott (0). A mezőkben az MTRR-eknél már említett memóriatípus-kódolásokat

használhatjuk azzal a különbséggel, hogy a 7-es érték az UC (UnCacheable) egy olyan formáját jelenti, amit a megfelelő MTRR WC (Write Combining) beállítása felülbírálhat. Ezt az UC- szimbólum jelöli.

A korábban említett PATi:PCD:PWT szám választja ki, az adott lap vagy tábla a PAT regiszter melyik mezőjét használja a memóriatípus megjelölésére. A következő táblázat mutatja a PAT regiszter mezőinek tartalmát a processzor resetelése után, valamint a PATi, PCD és PWT értékek által így meghatározott memóriatípust:

PATi	*PCD*	*PWT*	*PAT mező száma*	*Mem. típus*
0	0	0	0	WB
0	0	1	1	WT
0	1	0	2	UC-
0	1	1	3	UC
1	0	0	4	WB
1	0	1	5	WT
1	1	0	6	UC-
1	1	1	7	UC

A korábbiak alapján világos, hogy a PAT mezők tartalma a fenti táblázat szerint sorban 6, 4, 7, 0, 6, 4, 7, 0.

A PAT technikáról és az MTRR-ekről bővebb információt találhatunk az Intel dokumentációjában (pl. Intel Architecture Software Developer's Manual Volume 3, Chapter 9, "Memory cache control").

A Pentium processzorban bevezetett, gépi hibák jelzésére szolgáló Machine Check Exception (#MC, 18-as) a P6 architektúrájú processzorok esetében is elérhető. Az új, *Machine Check Architecture* (MCA) sokkal többféle hiba

jelzését teszi lehetővé, mint amire a Pentium képes volt. Szintén újdonság, hogy az MCA-t támogató processzor egy ilyen hiba keletkezésekor először megpróbálja azt orvosolni. Ha a kijavítás sikeres volt, nem keletkezik 18-as kivétel. Ha a hiba természete lehetővé teszi, a megszakított program vagy taszk esetleg mégis újraindítható a kivétel-kezelőből, de ilyenkor általában a program állapotában már olyan változás állt be, ami hibás működést fog okozni.

17.9.3 Változások a kivétel-kezelésben

A 6-os kivétel (#UD) mostantól az UD2 utasítás hatására is keletkezhet.

13-as kivétel (#GP) a következő feltétel esetén is kiváltódik:

- **a lapcímtár-mutató tábla (page-directory-pointer table) egyik bejegyzésének valamelyik fenntartott bitje 1-es értékű, miközben a CR4 regiszter PAE (5-ös) bitje is be van állítva**

A laphiba (#PF, 14-es) kivétel hibakódjában a RSVD (3-as) bit értéke akkor is 1 lehet, ha a CR4 regiszter PAE (5-ös) bitje be volt állítva, és a hivatkozott lapcímtár és/vagy laptábla bejegyzésben egy fenntartott bit helyén 1 állt.

17.9.4 Változások a regiszterkészletben

Ha a CR4 regiszterben PAE=1 (5-ös bit), és a lapozás engedélyezett, a CR3 regiszter a lapcímtár helyett a lapcímtár-mutató tábla fizikai báziscímét tárolja. A regiszter felépítése ilyenkor a következőképpen fest:

Bitpozíció	*Jelölés*	*Név*

0..2	–	–
3	PWT	Page-level Write-Through
4	PCD	Page-level Cache Disable
5..31	PDPTB	Page-Directory-Pointer Table Base

A PWT és PCD bitek ekkor a lapcímtár-mutató tábla bejegyzéseinek cache-selését vezérlik.

A PDPTB mező az említett tábla 32 bites fizikai báziscímének felső 27 bitjét tárolja. A báziscím alsó 5 bitjét 0-nak veszi a processzor.

A CR3 regiszter neve tartalma miatt ekkor PDPTR vagy PDPTBR lesz.

A CR4 regiszterben 3 új bit lett bevezetve:

Bitpozíció	*Név*
5	PAE
7	PGE
8	PCE

A PAE bit az azonos nevű szolgáltatást engedélyezi 1-es érték esetén.

A PGE (Page Global Enable) bit 1-es állása engedélyezi a globális lapok használatát.

A PCE (Performance-Monitoring Counter instruction Enable) bit értéke vezérli az RDPMC utasítás végrehajthatóságát. Ha PCE=1, a kérdéses utasítás bármikor végrehajtható, különben csak CPL=0 esetén. Reset után ez a bit törölve van, erre érdemes odafigyelni az utasítás használatakor.

17.10 Az Intel Pentium II-n megjelent újdonságok

17.10.1 Új utasítások

- **FXRSTOR mem**
  (ReSTORe Floating-Point and MMX state, and SSE state)
- **FXSAVE mem**
  (SAVE Floating-Point and MMX state, and SSE state)
  Ez a két utasítás a Pentium III processzor SSE technológiáját egészíti ki. Feladatuk, hogy az FPU/MMX állapotot, valamint az SSE állapotot elmentsék (FXSAVE) ill. visszaállítsák (FXRSTOR). Az utasításokra hatással van a CR4 regiszter OSFXSR (9-es) bitjének állása. Komplett leírásuk megtalálható az SSE technológiát ismertető fejezetben.
- **SYSENTER (no op)**
  (transition to SYStem call ENTRY point)
- **SYSEXIT (no op)**
  (transition from SYStem call entry point)
  A CS, EIP, SS és ESP regisztereket a megadott értékre állítják be, és az új környezetben folytatják a végrehajtást. A SYSENTER kizárólag csak 0-ás privilégium szintre tudja átadni a vezérlést. A SYSEXIT viszont csak 0-ás szintről 3-as szintre képes átugrani. Fontos, hogy egyik utasítás sem menti el a processzor aktuális állapotát, így a módosított regisztereket, EFlags-et stb. sem, erről a felhasználói programnak és az operációs rendszernek kell gondoskodnia. Az utasítások tehát nem használhatók közvetlenül a CALL-RET páros felváltására. Ezt mutatja az is, hogy igen sok megkötés van. Mindkét utasítás csak védett módban adható ki, de a SYSEXIT csak CPL=0

esetén (tehát virtuális módban nem). Különben 13-as kivétel (#GP) fog keletkezni.

A SYSENTER először kitörli az IF és VM flag-eket az EFlags-ben. Ezután az említett regisztereket a következő értékre állítja be:

Regiszter	*Tartalom*
CS	SYSENTER_CS_MSR
EIP	SYSENTER_EIP_MSR
SS	8+SYSENTER_CS_MSR
ESP	SYSENTER_ESP_MSR

Végül CPL-t 0-ra állítja, és megkezdi a kód végrehajtását az új címtől.

A SYSENTER_CS_MSR, SYSENTER_EIP_MSR és SYSENTER_ESP_MSR szimbólumok sorban a 174h, 175h és 176h számú MSR-eket jelentik.

A CS-be töltött szelektornak egy 32 bites, 00000000h bázisú, 4 Gbájt határú kódszegmensre kell mutatnia, futtatási és olvasási jogokkal.

Az SS-be töltött szelektor egy 32 bites, 00000000h bázisú, 4 Gbájt határú adatszegmensre kell hogy mutasson, írási, olvasási és elért (accessed) jogokkal.

A SYSEXIT nem bántja a flag-eket, rögtön a regisztereket állítja be:

Regiszter	*Tartalom*
CS	16+SYSENTER_CS_MSR
EIP	EDX regiszter

SS	**24+SYSENTER_CS_MSR**
ESP	**ECX regiszter**

Ezután CPL-t 3-ra állítja, és megkezdi a kód végrehajtását az új címtől.

A CS-be töltött szelektornak egy 32 bites, 00000000h bázisú, 4 Gbájt határú, nem illeszkedő (nonconforming) kódszegmensre kell mutatnia, futtatási és olvasási jogokkal.

Az SS-be töltött szelektor egy 32 bites, 00000000h bázisú, 4 Gbájt határú, felfelé bővülő (expand-up) adatszegmensre kell hogy mutasson, írási és olvasási jogokkal.

A szegmensekre vonatkozó feltételek (mint a bázis, határ és elérési jogok) teljesülését feltételezi mindkét utasítás, és a szelektorok betöltésekor a szegmens deszkriptor cache-regiszterekbe ezeket a rögzített értékeket írják be. Ellenőrzés nem történik ezek fennállására vonatkozóan, az illető operációs rendszer feladata, hogy a deszkriptorok tartalma tükrözze a helyes értékeket.

Ha a SYSENTER_CS_MSR regiszter tartalma 0, 13-as kivétel (#GP) keletkezik mindkét utasítás esetén.

A szelektorok beállított értéke biztosítja, hogy a szükséges szegmensek deszkriptorát kiválasztó indexek a következő értékeket veszik fel: ha IND jelöli a 0-ás szintű CS indexét, akkor a 0-ás szintű SS indexe IND+1, a 3-as szintű CS indexe IND+2, míg a 3-as szintű SS indexe IND+3 lesz.

A SYSENTER és SYSEXIT utasítások jelenléte ellenőrizhető a CPUID utasítással. Arra azonban

figyelni kell, hogy ezeket az utasításokat legelőször a Pentium II processzor támogatta. A Pentium Pro például jelzi a CPUID kimenetében, hogy támogatja az említett utasításokat, de ez nem fedi a valóságot.

Az említett utasítások közül a SYSEXIT csak védett, a SYSENTER pedig csak védett vagy virtuális-8086 módban hajtható végre. A SYSEXIT ezen kívül csak CPL=0 esetén érvényes.

A SYSENTER és SYSEXIT utasításokhoz hasonló feladatot látnak el az AMD K6-2 SYSCALL és SYSRET utasításai.

17.10.2 Változások a regiszterkészletben

A CR4 regiszterben egyetlen új bit lett bevezetve:

Bitpozíció	*Név*
9	OSFXSR

Az SFENCE, PREFETCHccc, MASKMOVQ, MOVNTQ, FXRSTOR, FXSAVE és az új SIMD MMX utasítások kivételével az összes többi SSE utasítás csak akkor hajtható végre, ha az OSFXSR (Operating System supports FXRSTOR/FXSAVE) bit értéke 1. Ez a bit a Pentium III processzor SSE utasításkészletének támogatására született meg.

17.11 Az AMD K6-2 újításai

17.11.1 Új prefixek

- **0Fh 0Fh**
  (3DNow! instruction prefix)
  Ez a speciális bájtsorozat az Intel processzorokon fenntartott utasítást jelöl, az AMD processzorain K6-2-től kezdődően azonban a 3DNow! utasítások kódolásához használják. Azt jelöli, hogy az utasítás tényleges műveleti kódját ez a prefix és a címzési mód bájt(ok) és az esetleges eltolási érték után álló bájt méretű közvetlen érték együttesen jelentik. Az utóbbi bájt neve *szuffix* (suffix). Nem igazi prefix. Bővebb leírását lásd az "Utasítások kódolása" c. fejezetben.

17.11.2 Új utasítások

- **SYSCALL**
  (CALL operating SYStem)
- **SYSRET**
  (RETurn from operating SYStem)
  A CS, EIP és SS regisztereket a megadott értékre állítják be, és az új környezetben folytatják a végrehajtást. A SYSCALL kizárólag csak 0-ás privilégium szintre tudja átadni a vezérlést. A SYSRET viszont csak 0-ás szintről 3-as szintre képes átugrani. Fontos, hogy a SYSCALL utasítás csak EIP-t menti el, a SYSRET pedig csak EIP-t állítja vissza. A processzor állapotának (főleg EFlags és ESP) elmentéséről ill. visszaállításáról a felhasználói programnak és az operációs rendszernek kell gondoskodnia. Az utasítások tehát nem használhatók közvetlenül a CALL-RET páros felváltására. Ezt mutatja az is, hogy igen

sok megkötés van. Mindkét utasítás valós és védett módban is kiadható, de a SYSRET csak CPL=0 esetén (tehát virtuális módban nem). Különben 13-as kivétel (#GP) fog keletkezni. Valós módban viszont nem sok értelme van használatuknak, mivel az utasítások szelektorokkal és deszkriptorokkal dolgoznak.

A SYSCALL először kitörli az IF és VM flag-eket az EFlags-ben. Ezután EIP-t átmásolja ECX-be, majd az említett regisztereket a következő értékre állítja be:

Regiszter	*Tartalom*
CS	STAR 32..47 bitjei
EIP	STAR 0..31 bitjei
SS	8+(STAR 32..47 bitjei)

Végül CPL-t 0-ra állítja, és megkezdi a kód végrehajtását az új címtől.

A STAR regiszter a 0C0000081h című MSR-t jelenti.

A CS-be töltött szelektornak egy 32 bites, 00000000h bázisú, 4 Gbájt határú kódszegmensre kell mutatnia, futtatási és olvasási jogokkal.

Az SS-be töltött szelektor egy 32 bites, 00000000h bázisú, 4 Gbájt határú felfelé bővülő (expand-up) adatszegmensre kell hogy mutasson, írási és olvasási jogokkal.

A SYSRET IF-et 1-re állítja, majd a regiszterekbe a következő értékeket tölti:

Regiszter	*Tartalom*

CS	**STAR 48..63 bitjei**
EIP	**ECX regiszter**
SS	**8+(STAR 48..63 bitjei)**

Ezután CPL-t és SS RPL mezőjét 3-ra állítja, majd megkezdi a kód végrehajtását az új címtől.

A CS-be töltött szelektornak egy 32 bites, 00000000h bázisú, 4 Gbájt határú kódszegmensre kell mutatnia, futtatási és olvasási jogokkal.

A szegmensekre vonatkozó feltételek (mint a bázis, határ és elérési jogok) teljesülését feltételezi mindkét utasítás, és a szelektorok betöltésekor a szegmens deszkriptor cache-regiszterekbe ezeket a rögzített értékeket írják be. Ellenőrzés nem történik ezek fennállására vonatkozóan, az illető operációs rendszer feladata, hogy a deszkriptorok tartalma tükrözze a helyes értékeket.

A szelektorok beállított értéke biztosítja, hogy a szükséges szegmensek deszkriptorát kiválasztó indexek a következő értékeket veszik fel: ha IND_0 jelöli a 0-ás szintű CS indexét, akkor a 0-ás szintű SS indexe IND_0+1 lesz; hasonlóan, ha IND_3 jelöli a 3-as szintű CS indexét, akkor a 3-as szintű SS indexe IND_3+1 lesz.

A SYSCALL és SYSRET utasítások jelenléte ellenőrizhető a CPUID utasítással. Arra azonban figyelni kell, hogy ezeket az utasításokat egyelőre csak az AMD processzorai támogatják, így a CPUID-t is ennek megfelelően kell használni.

Mindkét utasítás csak akkor hajtható végre, ha az EFER MSR (0C0000080h) SCE (0-ás) bitje 1-es értékű, különben 6-os (#UD) kivétel keletkezik.

A SYSRET utasítás csak védett, a SYSCALL pedig csak védett vagy virtuális-8086 módban hajtható végre. A SYSRET ezen kívül csak CPL=0 esetén érvényes.

A SYSCALL és SYSRET utasításokhoz hasonló feladatot látnak el az Intel Pentium II SYSENTER és SYSEXIT utasításai.

17.11.3 Változások a kivétel-kezelésben

6-os kivétel (#UD) a következő feltételek esetén is keletkezik:

- **egy 3DNow! utasítást próbáltunk meg végrehajtani egy olyan processzoron, ami nem támogatja az AMD 3DNow! technológiát**
- **egy 3DNow! utasítást próbáltunk meg végrehajtani úgy, hogy a CR0 regiszter EM (2-es) bitje be volt állítva**

7-es kivétel (#NM) a következő feltétel esetén is keletkezik:

- **egy 3DNow! utasítást hajtottunk végre úgy, hogy a CR0 regiszter TS (3-as) bitje be volt állítva**

17.11.4 Változások a regiszterkészletben

Az MMX regiszterek (MM0..MM7) 3DNow! regiszterként funkcionálnak a 3DNow! utasítások végrehajtásakor. Ezek tartalmáról és használatáról a következő, a 3DNow! technológiáról szóló fejezetben lesz szó.

17.11.5 Új regiszterek

Kivételesen ismertetjük két modell-specifikus regiszter (MSR) felépítését, mivel ezek a SYSCALL és SYSRET utasítás működését közvetlenül befolyásolják.

A 0C0000080h című MSR neve *EFER* (Extended Feature Enable Register). A regiszter felépítése alább látható:

Bitpozíció	*Jelölés*	*Név*
0	SCE	System Call/return Extension
1..63	–	–

Ha SCE=1, a SYSCALL és SYSRET utasítások használata engedélyezett, különben ezek végrehajtása 6-os (#UD) kivételt vált ki.

A 0C0000081h című MSR neve *STAR* (SYSCALL/ SYSRET Target Address Register). Tartalma a következő táblázatban látható:

Bitpozíció	*Név*
0..31	SYSCALL target ring 0 EIP address
32..47	SYSCALL CS and SS selector base
48..63	SYSRET CS and SS selector base

A 0..31 biteken található mező a SYSCALL utasítás esetén az EIP új értékét tartalmazza, ez lesz az operációs rendszer belépési pontja.

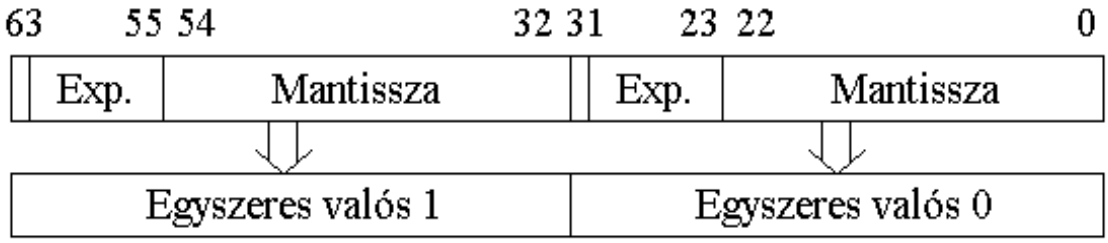
A 32..47 biteken található értéket a SYSCALL utasítás CS-be, míg ennek tartalmát plusz 8-at SS-be tölti. Ez a mező

jelöli ki az operációs rendszer kódszegmensét és veremszegmensét.

A 48..63 biteken található értéket a SYSRET utasítás CS-be, míg ennek tartalmát plusz 8-at SS-be tölti. Ez a mező jelöli ki a felhasználói program kódszegmensét és veremszegmensét.

17.12 Az AMD 3DNow! technológiája

Az AMD 3DNow! nevű technológiája az MMX által lerakott alapokra épül. Ezért nem meglepő, hogy elég sok közös tulajdonságot találunk a két utasításkészletben. Rögtön itt vannak például a regiszterek: a 3DNow! technológia szintén az MMX-ben bevezetett MM0..MM7 regisztereket használja. Említettük, hogy a 3DNow! legfőbb újítása az volt az MMX-szel szemben, hogy egész (fixpontos) számok helyett lebegőpontos értékekkel képes SIMD-elven számolni. Ezt támogatja a 3DNow! saját adattípusa, a pakolt lebegőpontos típus (packed floating-point type). Az alábbi ábra mutatja ennek felépítését:



A 64 bites típusban tehát két, egyszeres pontosságú lebegőpontos valós (single precision floating-point real) szám van bepakolva. A 0-ás valós mantisszája a 0..22 biteken, exponense a 23..30 biteken, előjele pedig a 31-es biten található. A másik valós a 64 bites szám felső duplaszavában helyezkedik el ugyanilyen módon.

A fentebb említettük, hogy a 3DNow! regiszterek megegyeznek az MMX regiszterekkel, amik viszont a koprocesszor adatregisztereire vannak ráképezve. Szinte mindegyik 3DNow! utasítás végrehajtása után érvényesnek jelöli be az FPU regisztereket a tag-regiszterben. Ezért a 3DNow! technológia alkalmazása során is megjelenik a probléma, hogy a 3DNow!/MMX utasítások blokkja után törölni kell a tag-eket, azaz 0FFFFh-t kell a tag-regiszterbe írni. Erre az EMMS MMX utasítás szolgál, de a 3DNow! technológia is kínál erre megoldást az FEMMS személyében. A PREFETCH és PREFETCHW utasítás nem módosítja a tag-eket.

A pakolt valós típusban található valós számok formátuma megfelel az IEEE-754-es szabvány egyszeres pontosságú valós lebegőpontos szám-formátumának. Az exponens 127-es eltolási értéket használva eltolva tárolódik (biased exponent). Minden szám normáltként kerül eltárolásra. Denormált számokat, NaN-okat és végtelen értékeket nem támogat a 3DNow! technológia forrásoperandusként. Alulcsordulás esetén nem képződik denormált szám, és kivétel sem generálódik, az eredményt pedig nullának veszi a processzor. Túlcsordulás után két eset lehetséges: az eredmény vagy helyes előjelű végtelen lesz, vagy pedig a maximális abszolút értékű pozitív vagy negatív szám (ahol az exponens és a mantissza is maximális értékű). Az adott implementációtól függ, hogy mi fog történni. Az AMD K6-2 processzor ez utóbbi megoldást választja, tehát végtelen értéket sosem kaphatunk eredménynek.

Az IEEE ajánlással ellentétben a 3DNow! technológia csak egyféle kerekítési módot ismer. Ez implementációtól függően a legközelebbi értékhez kerekítés (round to nearest) vagy a csonkolás (truncation, round to zero) lehet. Az AMD K6-2 processzor a legközelebbi értékhez kerekítést támogatja. A PF2ID és PI2FD konverziós utasítások mindig a csonkolást használják.

Ha valamilyen nem támogatott számformátumot használunk operandusként (pl. NaN-t, végtelent, denormált számot, érvénytelen értéket), a műveletek eredménye kiszámíthatatlan. Numerikus kivételt egyik utasítás sem vált ki, a flag-eket sem állítja, így a hibák elkerüléséről a felhasználónak kell gondoskodnia.

Az utasítások két csoportra oszthatók: egy részük valódi SIMD-elvű utasítás, ezeket vektor-utasításoknak (vector instructions) hívjuk, mivel egyszerre két valós számpáron hajtják végre dolgukat. A másik társaság a 64 bites típusnak csak az alsó duplaszavát használja forrásként, ezek a skalár-utasítások (scalar instructions).

Néhány utasítás operandusként pakolt egész típusokat használ. Ezek a pakolt bájtok, szavak és duplaszavak, és formátumuk megfelel az MMX technológiánál leírtaknak.

Prefixek tekintetében a 3DNow! megengedőbb az MMX-nél: a szegmensfelülbíráló és címhossz prefixek befolyásolják a rákövetkező 3DNow! utasítást, az operandushossz és a sztring-utasítást ismétlő prefixeket figyelmen kívül hagyja a processzor. A buszlezáró prefix viszont itt is 6-os kivételt okoz a 3DNow! utasítások előtt.

Ha egy 3DNow! utasítás kiadásakor a CR0 regiszter EM bitje be van állítva, az MMX-hez hasonlóan 6-os kivétel keletkezik.

A kétoperandusú utasítások esetén mindig igaz a következő szabály: a céloperandus egy 3DNow!/MMX regiszter kell legyen, míg a forrás lehet memóriahivatkozás vagy 3DNow!/MMX regiszter is.

Az alábbiakban olvasható a 3DNow! utasítások listája: a mnemonik után látható az operandusok típusa, a mnemonik jelentése angolul, majd az utasítás működésének leírása következik. A PREFETCH és PREFETCHW mnemonikokat leszámítva az első "P" betű utal arra, hogy az adott utasítás pakolt típusú operandusokat használ.

A következő jelöléseket alkalmaztuk:

- **(no op) – nincs operandus**
- **source – forrás; 3DNow!/MMX regiszter vagy 64 bites memóriaoperandus lehet**
- **dest – cél (destination); 3DNow!/MMX regiszter lehet**
- **mem8 – 8 bites memóriaoperandus**

17.12.1 Teljesítmény-növelő utasítások

- **FEMMS (no op)**
  (Faster Entry/Exit of the MMX or floating- point State)
  Az EMMS utasításhoz hasonlóan a tag-regiszterbe a 0FFFFh értéket tölti, ezzel mindegyik koprocesszor adatregisztert üresnek jelöli be. Az EMMS utasítással ellentétben viszont az adatregiszterek tartalmát nem feltétlenül őrzi meg, így gyorsabban valósítja meg ugyanazt a funkciót.
- **PREFETCH mem8**
- **PREFETCHW mem8**
  (PREFETCH processor cache line into L1 data cache)
  Az operandusban megadott memóriacímtől beolvasnak egy cache-sort, aminek a hossza legalább 32 bájt, és a későbbi (t.i. K6-3 processzor utáni) processzorokon változhat. A PREFETCHW utasítás a beolvasott cache-sor MESI állapotát (Modified, Exclusive, Shared, Invalid) módosítottra (modified) állítja be, a PREFETCH viszont általában a kizárólagos, egyedi (exclusive) beállítást alkalmazza. Ha a beolvasandó adaton változtatni is fogunk, akkor a PREFETCHW utasítás gyorsabb működést garantál. Az AMD K6-2 processzor mindkét mnemonikot azonosan hajtja végre, a későbbi

processzorok viszont meg fogják különböztetni az egyes változatokat. Az utasítás kódolása a következő: 0Fh 0Dh *r+i, ahol i=000b a PREFETCH, valamint i=001b a PREFETCHW utasítások esetén. A többi érték fenntartott a későbbi utasítás-bővítésekre, de végrehajtásuk nem okoz kivételt. Ha az operandus nem memóriahivatkozás, 6-os kivétel (#UD) keletkezik.

17.12.2 Egész aritmetikai utasítások

- **PAVGUSB dest,source**
  (AVeraGe of UnSigned Bytes)
  A forrásban és a célban levő 8 db. előjeltelen elempár kerekített átlagát számítja ki, az eredményeket egy 64 bites típusba belepakolja, majd a végleges értéket a cél regiszterbe eltárolja. Az átlagolás a szokásos módon történik: például az eredmény 0..7 bitjeit úgy kapjuk meg, hogy a forrás és a cél ugyanezen bitjein levő bájtokat összeadja, a 9 bites értékhez hozzáad 001h-t, majd a képződő összeget eggyel jobbra lépteti előjeltelenül (logikai léptetés).
- **PMULHRW dest,source**
  (MULtiply signed Words with Rounding, and store High-order words of results)
  Összeszorozza a forrásban és a célban levő előjeles szavakat, a 32 bites szorzatokhoz hozzáad 00008000h-t (kerekítés), az eredmények felső szavát bepakolja egy 64 bites típusba, majd azt eltárolja a célba. Pontosabb eredményt szolgáltat a kerekítés miatt mint a PMULHW MMX utasítás, ami csonkolja a szorzatokat.

17.12.3 Lebegőpontos aritmetikai utasítások

- **PFADD dest,source**
  (Floating-point ADDition)
  A forrásban és a célban levő valós elempárokat összeadja, az összegeket bepakolja az eredménybe, amit a cél területére eltárol.
- **PFSUB dest,source**
  (Floating-point SUBtraction)
  A célban levő elemekből kivonja a forrásban levő megfelelő elemet, a különbségeket bepakolja az eredménybe, amit a cél területére eltárol.
- **PFSUBR dest,source**
  (Floating-point Reversed SUBtraction)
  A forrásban levő elemekből kivonja a célban levő megfelelő elemet, a különbségeket bepakolja az eredménybe, amit a cél területére eltárol.
- **PFACC dest,source**
  (Floating-point ACCumulate)
  Az eredmény alsó felébe a célban levő két lebegőpontos valós szám összege kerül, míg a felső duplaszó a forrásban levő két valós érték összegét fogja tartalmazni. Végül az eredményt eltárolja a cél helyén.
- **PFRCP dest,source**
  (Floating-point ReCiProcal approximation)
  Ez a skalár-utasítás a forrás alsó duplaszavában található valós szám reciprokának 14 bitre pontos közelítését tölti be a cél alsó és felső duplaszavába is. A teljes, 24 bitre pontos reciprok értékhez a PFRCPIT1 és PFRCPIT2 utasításokat is használni kell.
- **PFRSQRT dest,source**

(Floating-point Reciprocal SQuare Root approximation)

Ez a skalár-utasítás a forrás alsó duplaszavában található valós szám reciprok négyzetgyökének 15 bitre pontos közelítését tölti be a cél alsó és felső duplaszavába is. A negatív operandusokat úgy kezeli, mintha azok pozitívak lennének. Az eredmény előjele megegyezik a forrás előjelével. A teljes, 24 bitre pontos reciprok értékhez a PFRSQIT1 és PFRCPIT2 utasításokat is használni kell.

- **PFRCPIT1 dest,source**

  (Floating-point ReCiProcal, first ITeration step)

  Ez az utasítás az előzőleg PFRCP utasítással meghatározott reciprok közelítést finomítja tovább egy Newton-Raphson iterációs lépés végrehajtásával. A célnak meg kell egyeznie a PFRCP utasítás forrásával, a forrásoperandusnak pedig az előbbi PFRCP utasítás eredményének kell lennie. Egyéb operandus-kombinációk esetén az utasítás hatása, eredménye definiálatlan. A PFRCPIT1 eredményét a PFRCPIT2 utasításnak kell továbbadni, hogy a 24 bit pontosságú közelítést megkaphassuk. Az utasítás vektor-elven üzemel, tehát az operandusok mindkét felén elvégzi a számolást.

- **PFRSQIT1 dest,source**

  (Floating-point Reciprocal SQuare root, first ITeration step)

  Ez az utasítás az előzőleg PFRSQRT utasítással meghatározott reciprok négyzetgyök közelítést finomítja tovább egy Newton-Raphson iterációs lépés végrehajtásával. A célnak meg kell egyeznie a PFRSQRT utasítás forrásával, a forrás-

operandusnak pedig az előbbi PFRSQRT utasítás eredménye négyzetének kell lennie. Egyéb operandus-kombinációk esetén az utasítás hatása, eredménye definiálatlan. A PFRSQIT1 eredményét a PFRCPIT2 utasításnak kell továbbadni, hogy a 24 bit pontosságú közelítést megkaphassuk. Az utasítás vektor-elven üzemel, tehát az operandusok mindkét felén elvégzi a számolást.

- **PFRCPIT2 dest,source**
  (Floating-point ReCiProcal/reciprocal square root, second ITeration step)
  Ez az utasítás az előzőleg PFRSQIT1 utasítással meghatározott reciprok négyzetgyök, vagy a PFRCPIT1 utasítással kiszámolt reciprok közelítést finomítja tovább a második Newton-Raphson iterációs lépés végrehajtásával. A célnak meg kell egyeznie a PFRCPIT1 vagy a PFRSQIT1 utasítás eredményével, a forrásoperandusnak pedig az előbbieket megelőző PFRCP vagy PFRSQRT utasítás eredményének kell lennie. Egyéb operandus-kombinációk esetén az utasítás hatása, eredménye definiálatlan. Az utasítás hatására a célban előáll a kívánt reciprok vagy reciprok négyzetgyök érték 24 bit pontos közelítése. Az utasítás vektor-elven üzemel, tehát az operandusok mindkét felén elvégzi a számolást.
- **PFMUL dest,source**
  (Floating-point MULtiplication)
  A forrásban és a célban levő valós elempárokat összeszorozza, a szorzatokat bepakolja az eredménybe, amit a cél területére eltárol.
- **PFMAX dest,source**
  (Floating-point MAXimum)

A forrásban és a célban levő megfelelő elemek közül kiválasztja a nagyobbat, azokat bepakolja az eredménybe, amit a cél területére eltárol. Nulla és egy negatív szám esetén, vagy ha mindkét elem nulla, akkor pozitív nullát ad vissza.

- **PFMIN dest,source**
  (Floating-point MINimum)
  A forrásban és a célban levő megfelelő elemek közül kiválasztja a kisebbet, azokat bepakolja az eredménybe, amit a cél területére eltárol. Nulla és egy pozitív szám esetén, vagy ha mindkét elem nulla, akkor pozitív nullát ad vissza.

17.12.4 Összehasonlító utasítások

- **PFCMPGE dest,source**
  (Floating-point CoMParision for Greater or Equal)
- **PFCMPGT dest,source**
  (Floating-point CoMParision for Greater Than)
- **PFCMPEQ dest,source**
  (Floating-point CoMParision for EQual)
  A forrásban levő elemeket összehasonlítja a célban levő megfelelő elemmel. Ha a megadott feltétel teljesül, akkor az eredménybe 0FFFFFFFFh-t rak be, különben pedig 00000000h kerül az eredmény duplaszavába. Ezután a pakolt eredményt visszarakja a cél regiszterbe.

17.12.5 Konverziós utasítások

- **PI2FD dest,source**
  (Doubleword Integer To Floating-point conversion)
  A forrást pakolt duplaszavaknak tekintve átkonvertálja a benne levő két bináris egész

előjeles számot lebegőpontos értékké, azokat bepakolja az eredménybe, amit azután a cél regiszterben eltárol. Ha a konvertált érték nem férne el az egyszeres pontosságú valós típusban, csonkolást hajt végre.

- **PF2ID dest,source**
  (Floating-point To Doubleword Integer conversion)
  A célt pakolt lebegőpontos típusnak tekintve átkonvertálja a két valós számot előjeles dupla-szavakká, azokat bepakolja az eredménybe, amit azután a cél regiszterben eltárol. A konverziót csonkolással végzi el, azaz nulla felé kerekítéssel. Az 1-nel kisebb abszolút értékű számokból nulla lesz. Ha a szám kisebb mint -2147483648.0, akkor a konverzió eredménye 80000000h lesz, míg ha a forrás érték nagyobb mint +2147483647.0, akkor a 7FFFFFFFh értékre telítődik az adott elem.

A következő algoritmus és programrészlet mutatja, hogy lehet gyorsan meghatározni 1/b és a/b értékét a 3DNow! technológiával:

$$X_0=PFRCP(b)$$
$$X_1=PFRCPIT1(b, X_0)$$
$$X_2=PFRCPIT2(X_1,X_0)$$
$$q=PFMUL(a,X_2)$$

```
MOVD        MM0,[b]
PFRCP       MM1,MM0
PFRCPIT1    MM0,MM1
MOVD        MM2,[a]
PFRCPIT2    MM0,MM1              ;MM0=1/b
PFMUL       MM2,MM0              ;MM2=a/b
MOVD        [q],MM2
```

Az alábbi pszeudokód és programrészlet azt szemlélteti, milyen módon határozható meg $1/\sqrt{b}$ és $\sqrt{b}$ értéke egyszerre:

X_0=PFRSQRT(b)
X_1=PFMUL(X_0,X_0)
X_2=PFRSQIT1(b,X_1)
X_3=PFRCPIT2(X_2,X_0)
X_4=PFMUL(b,X_3)

```
MOVD        MM0,[b]
MOVQ        MM3,MM0
PFRSQRT     MM1,MM0
MOVQ        MM2,MM1
PFMUL       MM1,MM1
PFRSQIT1    MM0,MM1
PFRCPIT2    MM0,MM2              ;MM0=1/√b
PFMUL       MM3,MM0              ;MM3=√b
```

17.13 Az Intel Pentium III-n megjelent újdonságok

17.13.1 Új prefixek

- **0F3h**
  (SSE instruction prefix)
  Az SSE utasítások között sok olyan utasításpár van, amelyek egyik tagja skaláron, másik tagja pedig pakolt lebegőpontos típusú (SIMD single-FP) operandusokon végzi el ugyanazt a műveletet. Ilyen utasításpárt képeznek pl. az ADDPS-ADDSS, CMPPS-CMPSS utasítások. A párok mindkét tagjának minden tekintetben megegyezik a műveleti kódja, de a skalárral dolgozó utasítás opcode-ját egy REP/REPE/REPZ prefix előzi meg. Mivel a sztringutasítást ismétlő prefixet ilyen

utasításra nem lehetne alkalmazni, ezért az 0F3h bájt itt a műveleti kód első bájtjaként funkcionál. A példánál maradva, az ADDPS utasítás kódolási formája 0Fh, 58h, *r, míg az ADDSS utasításé 0F3h, 0Fh, 58h, *r. Nem igazi prefix, bővebb leírása pedig megtalálható az "Utasítások kódolása" c. fejezetben.

17.13.2 Új utasítások

- **LDMXCSR mem32**
  (LoaD SIMD Control/Status Register)
  A megadott címről betölti az MXCSR regiszter új tartalmát. Ha valamelyik fenntartott bitpozícióban 1 áll, 13-as kivétel (#GP) keletkezik.
- **STMXCSR mem32**
  (STore SIMD Control/Status Register)
  A megadott címre eltárolja az MXCSR regiszter tartalmát, a fenntartott bitek helyére 0-t írva.

17.13.3 Változások a kivétel-kezelésben

6-os kivétel (#UD) a következő feltételek esetén is keletkezik:

- **egy SSE utasítást próbáltunk meg végrehajtani egy olyan processzoron, ami nem támogatja az Intel SSE technológiát, azaz a CPUID kimenetében az XMM (25-ös) bit törölve van**
- **egy SSE utasítást próbáltunk meg végrehajtani úgy, hogy a CR0 regiszter EM (2-es) bitje be volt állítva (ez a PREFETCHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA és SFENCE utasításokra nem vonatkozik)**

- **egy SSE utasítást próbáltunk meg végrehajtani úgy, hogy a CR4 regiszter OSXMMEXCPT (10-es) bitje törölve volt, és a kérdéses utasítás numerikus kivételt okozott**
- **egy SSE utasítást próbáltunk meg végrehajtani úgy, hogy a CR4 regiszter OSFXSR (9-es) bitje törölve volt (ez az új MMX utasításokra, továbbá a PREFETCHccc, SFENCE, MASKMOVQ, MOVNTQ, FXRSTOR és FXSAVE utasításokra nem vonatkozik)**

7-es kivétel (#NM) a következő feltétel esetén is keletkezik:

- **egy SSE utasítást hajtottunk végre úgy, hogy a CR0 regiszter TS (3-as) bitje be volt állítva**

Az SFENCE, PREFETCHT0, PREFETCHT1, PREFETCHT2 és PREFETCHNTA utasítások a TS bit 1-es állapota mellett sem generálnak 7-es kivételt.

A 17-es kivétel (#AC) az illeszkedés-ellenőrzés engedélyezése és a szokásos feltételek teljesülése esetén akkor is keletkezik, ha a következő táblázatban felsorolt adattípusok nem illeszkednek megfelelően:

Adat típusa	*Illeszkedés bájtokban*
SIMD pakolt lebegőpontos	16
32 bites skalár egyszeres valós	16
16 bites MMX pakolt egész	16
32 bites MMX pakolt egész	16
64 bites MMX pakolt egész	16

Ezek a típusok az új SSE utasítások (SIMD lebegőpontos és SIMD MMX utasítások) használata esetén érvényesek. Néhány utasítás (mint pl. a MOVUPS, FXSAVE, FXRSTOR) kivételesen kezeli az illeszkedések ilyen fajtáját, erről az adott utasítás ismertetésénél és az Intel dokumentációjában lehet többet találni.

17.13.4 Új kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
19	**#XF**	**SSE FP Error**	**Fault**	**SIMD lebegőpontos ut.**

19-es kivétel akkor keletkezik, ha a processzor egy SSE lebegőpontos utasítás végrehajtása közben numerikus hibát észlel, az adott kivétel nincs maszkolva az MXCSR regiszterben, valamint a CR4 regiszter OSXMMEXCPT (10-es) bitje is be van állítva. Ha keletkezik numerikus hiba, de az adott kivétel maszkolva van, a processzor kijavítja a hibát, nem generál kivételt, és folytatja a végrehajtást. Ha az adott numerikus kivétel nincs maszkolva, de OSXMMEXCPT=0, 6-os kivétel (#UD) keletkezik 19-es helyett. Hibakód nem kerül a verembe, az MXCSR regiszter ill. az XMM0..XMM7 regiszterek tartalma bőséges információt szolgáltat a hibával kapcsolatban. A veremben levő CS:IP/EIP másolat mindig a hibát okozó SSE utasításra mutat.

17.13.5 Változások a regiszterkészletben

A CR4 regiszterben egyetlen új bit lett bevezetve:

Bitpozíció	*Név*
10	**OSXMMEXCPT**

Az OSXMMEXCPT (Operating System supports SIMD floating-point unMasked EXCePTions) bit az SSE utasítások által kiváltott numerikus hibák lekezelését befolyásolja. Ha egy maszkolatlan numerikus kivétel keletkezik, akkor a processzor ennek a bitnek az állása alapján cselekszik. Ha OSXMMEXCPT=1, akkor 19-es kivételt (#XF), egyébként pedig 6-os kivételt (#UD) generál.

17.13.6 Új regiszterek

Az SSE utasítások támogatására néhány új regisztert kapott a processzor.

A *SIMD vezérlő-/státuszregiszter* (SIMD Floating-point Control/Status Register) olyan 32 bites regiszter, ami az SSE utasítások működését vezérli, továbbá az azok végrehajtása utáni állapotot tartalmazza. A regiszter neve MXCSR.

A SIMD lebegőpontos adatregiszterek az SSE utasítások operandusaként szolgálnak. A 8 regisztert az XMM0, XMM1, …, XMM7 neveken érhetjük el.

Az említett regiszterekről bővebb leírás található a következő fejezetben, ami az SSE utasításokat is bemutatja.

17.14 Az Intel SSE technológiája

Az Intel Pentium III (Katmai) processzorral együtt jelentette meg az Intel a második nagy utasításkészlet-bővítést. Az első ilyen változtatás az Intel x86-os architektúrában az MMX technológia és annak új utasításai voltak, ami azonban nem bizonyult elég sikeresnek, mivel csak egész (fixpontos) számokkal tudott SIMD-elven műveletet végezni. Ezt a hiányosságot igyekszik kijavítani a 70 új utasításból álló készlet. A technológia 3 elnevezés alatt ismeretes: kezdetben MMX2-nek hívták, ezzel utalva elődjére. Később átkeresztelték KNI-re (Katmai New Instructions), amely név magában hordozza a szülő processzor fedőnevét. A végleges név mégis az SSE (Streaming SIMD Extensions) lett. Ez a név ugyanis jobban kifejezi ennek a technológiának a főbb jellemzőit: vele képesek

leszünk hatékonyan dolgozni adatfolyamokkal, valamint SIMD végrehajtási modell alapján egyszerre több független elemen végezhetjük el párhuzamosan ugyanazt a tevékenységet.

A hatékonyabb számolás érdekében új adattípust definiál az SSE technológia. Ez a típus ugyancsak pakolt lebegőpontos adattípus (packed floating-point data type – Packed FP), de ennek mérete a 3DNow! technológia hasonló típusától eltérően nem 64 hanem 128 bit. Az új típust ezentúl SIMD lebegőpontos típusként emlegetjük, hogy elkerüljük a 3DNow!-val való összetévesztést.

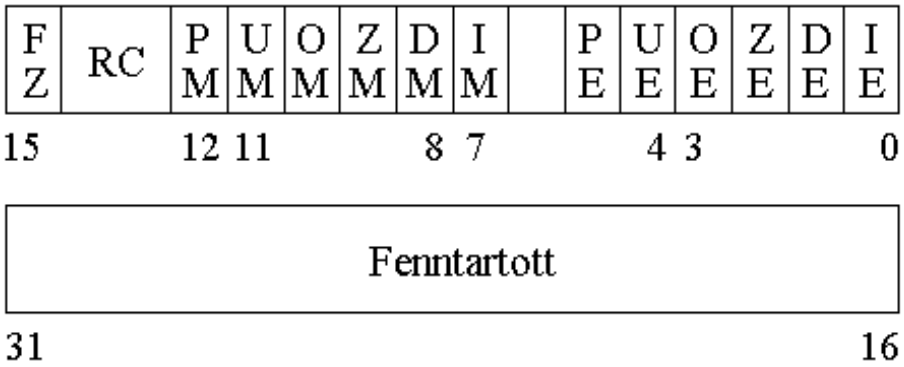
A 128 bites adatba négy darab egyszeres pontosságú lebegőpontos valós érték (Single-FP) van bepakolva:

127 96	95 64	63 32	31 0
Valós 3	Valós 2	Valós 1	Valós 0

A lebegőpontos valósok formátuma teljesen megfelel az IEEE-754 szabványnak. Ebbe a kerekítési módokat, a NaN-ok és denormalizált értékek kezelését, numerikus kivételek generálását is bele kell érteni.

Az új adattípus tárolására 8 db. új, 128 bites adatregisztert vezettek be. A SIMD lebegőpontos regiszterek neve XMM0..XMM7. Ezek a regiszterek az alap regiszter-készlettől teljesen függetlenül helyezkednek el, tehát az MMX regiszterekkel ellentétben NEM a koprocesszor adatregisztereit foglalják el. Ez a megoldás egyrészt nagy programozási szabadságot ad, mivel az MMX és SSE kód könnyedén keverhetők egymással. Másrészt viszont az operációs rendszernek támogatnia kell ezeket az új regisztereket, mivel a taszkváltás esetén szükséges regiszter-elmentés és -visszaállítás folyamata során az FSAVE, FNSAVE, FRSTOR utasítások elmentik ugyan az FPU regisztereket, és azzal együtt az MMX regisztereket is, a SIMD regisztereket viszont nem.

Szintén új a SIMD vezérlő-/státuszregiszter (SIMD Floating-point Control/Status Register). Ennek a 32 bites regiszternek a neve MXCSR, felépítése pedig a következő ábrán látható:



Az MXCSR 6-os és 16..31 számú bitjei fenntartottak, ami annyit jelent, hogy tartalmuk nulla, és ha bármelyiket megpróbáljuk beállítani, 13-as kivétel (#GP) keletkezik. A regisztert memóriába tárolni az STMXCSR, beállítani pedig az LDMXCSR utasítással lehet. A SIMD lebegőpontos állapotot elmentő ill. visszaállító FXSAVE és FXRSTOR utasításokkal szintén manipulálható az MXCSR tartalma.

A 0..5 bitek az SSE utasítások végrehajtása során keletkező numerikus kivételeket jelzik. A kivételek fajtái megegyeznek az FPU által felismert/generált kivételekkel, ezért leírásukat lásd a numerikus koprocesszort tárgyaló fejezetben. Eltérést az érvénytelen művelet (Invalid operation–#I) kivétel jelent, ami csak az operandus hibás voltát jelzi, mivel veremhiba nem keletkezik SSE utasítások esetén (így nincs #IS kivétel sem). Ez a hat bit "ragadós" (sticky), mivel állásukat egészen addig megőrzik, amíg ki nem töröljük őket az FXRSTOR vagy LDMXCSR utasítással. A 7..12 bitek ezeket a kivételeket maszkolják (tiltják) le egyenként, ha be vannak állítva. Működésük az FPU vezérlőregiszterében található hasonló bitek működésével egyezik meg. Alapállapotban az összes kivétel maszkolva van.

A 13, 14 biteken található RC (Rounding Control) mező határozza meg a számítások során alkalmazandó kerekítés

fajtáját. A mező jelentése megegyezik az FPU vezérlő-regiszterének azonos mezőjének funkciójával. Az alapbeállítás a 00b, tehát a legközelebbi értékhez kerekít a processzor.

A 15-ös bit neve FZ (Flush-to-Zero). Ha egy SSE utasítás végrehajtása során alulcsordulás (underflow) történik, akkor két eset lehetséges: ha az MXCSR 11-es UM bitje 0 értékű, akkor alulcsordulás numerikus kivétel (#U) keletkezik. Ha az #U kivétel tiltva van, akkor ismét két eset lehetséges: ha FZ=0, akkor denormalizált eredményt kapunk, aminek következménye lehet a pontosság elvesztése is (#P kivétel). Ha azonban FZ=1 és UM=1, akkor eredményként helyes előjelű nullát kapunk, és UE és PE értéke is 1 lesz.

A prefixek használhatósága megegyezik az MMX utasításoknál leírtakkal, tehát a szegmensfelülbíráló és a címhossz prefixek megengedettek, az operandushossz és a sztringutasítást ismétlő prefixek fenntartottak, míg a buszlezáró prefix minden esetben 6-os kivételt (#UD) generál. Fontos tudni, hogy néhány SSE utasítás gépi kódja implicite tartalmazza a REP/REPE/REPZ prefixet (0F3h), ezért óvatosan bánjunk ezzel a prefixszel!

Az SSE utasítások végrehajthatóságának alapfeltétele, hogy a CPUID utasítás EAX=00000001h értékével való kiadása után EDX 25-ös bitje (XMM) 1-es értékű legyen. Ez a feltétel már elegendő, hogy az SFENCE, PREFETCHT0, PREFETCHT1, PREFETCHT2 és PREFETCHNTA utasí-tásokat használhassuk. Az új MMX utasítások (SIMD integer instructions) akkor hajthatók végre, ha a CR0 regiszter EM (2-es) bitje törölve van (ez vonatkozik a MASKMOVQ, MOVNTQ, FXRSTOR és FXSAVE utasításokra is). Az összes többi SIMD lebegőpontos utasítás lefutásához a CR4 regiszter OSFXSR (9-es) bitjének 1-es értékűnek kell lennie. Ha ezek a feltételek nem teljesülnek, 6-os kivétel keletkezik. A következő táblázat ezeket a feltételeket foglalja össze:

Utasítás	*CPUID.XMM*	*CR0.EM*	*CR4.OSFXSR*
SFENCE	**1**	**–**	**–**
PREFETCHccc	**1**	**–**	**–**
MASKMOVQ	**1**	**0**	**–**
MOVNTQ	**1**	**0**	**–**
FXRSTOR, FXSAVE	**1**	**0**	**–**
SIMD MMX	**1**	**0**	**–**
Többi SSE	**1**	**0**	**1**

Szintén fontos, hogy azok az SSE utasítások, amelyek numerikus kivételt képesek kiváltani, érzékenyek a CR4 regiszter OSXMMEXCPT (10-es) bitjének állására. Ha egy SIMD lebegőpontos kivétel keletkezik, akkor OSXMMEXCPT=1 esetén 19-es (#XF), különben pedig 6-os (#UD) kivétel generálódik.

A 3DNow! technológiához hasonlóan itt is két nagy csoportba sorolhatók az SSE utasításai. A skalár-utasítások (scalar instructions) az operandusok alsó duplaszavában levő lebegőpontos értékekkel dolgoznak, és a felső három duplaszó értékét a forrásból az eredménybe átírják. A másik csoportot a 3DNow! esetében a vektor-utasítás alkotják, itt pedig pakolt-utasításoknak (packed instructions) hívják őket. A pakolt-utasítások mind a 4 lebegőpontos számpáron végrehajtanak valamilyen műveletet, a négy eredményt bepakolják a 128 bites típusba, majd ezt a pakolt végeredményt a cél helyére írják.

A kétoperandusú utasításokra fennáll az a már megszokott tulajdonság, hogy az első operandus a cél (néhány esetben forrás is), míg a második operandus a forrás. Az adatmozgató, konverziós, cache-selést befolyásoló és az új MMX utasításokat leszámítva minden más esetben teljesülnie kell az operandusokra, hogy a cél csak egy SIMD lebegőpontos regiszter lehet, míg a forrás helyére SIMD regiszter vagy memóriahivatkozás kerülhet.

Ha az operandusok valamelyike NaN, és SNaN esetén IM=1 is teljesül (az MXCSR regiszterben), akkor az eredmény négy esetet kivéve mindig QNaN lesz. A kivételeket a MINPS, MINSS, MAXPS és MAXSS SSE utasítások jelentik: ha csak az egyik operandus NaN, akkor ez a két utasítás a forrás tartalmát (tehát a második operandust) adja vissza eredményként, ami normált vagy denormált véges szám, nulla, végtelen, QNaN vagy SNaN lehet.

Az SSE utasítások az FPU utasításoktól eltérően figyelmen kívül hagyják a CR0 regiszter NE (5-ös) bitjének állását, és maszkolatlan numerikus kivétel keletkezése esetén mindig 19-es (#XF) kivételt generálnak (és persze beállítják az MXCSR regiszterben a megfelelő biteket). Ehhez az is szükséges, hogy a CR4 regiszter OSXMMEXCPT (10-es) bitje be legyen állítva.

Az alábbiakban olvasható az SSE utasítások jellemzése: a mnemonik után látható az operandusok típusa, a mnemonik jelentése angolul, majd az utasítás működésének leírása következik. A mnemonikban a "PS" betűcsoport a "Packed Single-FP", míg az "SS" betűk a "Scalar Single-FP" kifejezésekre utalnak.

A következő jelöléseket alkalmaztuk:

- **(no op) – nincs operandus**
- **source – forrás; xmmreg/mem128 lehet**
- **dest – cél (destination); xmmreg lehet**
- **mem – memóriaoperandus**
- **imm8 – bájt méretű közvetlen operandus**
- **mem8 – bájt méretű memóriaoperandus**
- **reg32 – 32 bites általános célú regiszter**
- **reg32/mem16 – 32 bites általános célú regiszter vagy szó méretű memóriaoperandus**
- **mem32 – duplaszó méretű memóriaoperandus**

- **reg/mem32 – 32 bites általános célú regiszter vagy memóriaoperandus**
- **mmreg – az MM0, MM1, …, MM7 regiszterek valamelyike**
- **mem64 – 64 bites memóriaoperandus**
- **mmreg/mem64 – MMX regiszter vagy 64 bites memóriaoperandus**
- **xmmreg – az XMM0, XMM1, …, XMM7 regiszterek valamelyike**
- **mem128 – 128 bites memóriaoperandus**
- **xmmreg/mem32 – SIMD lebegőpontos regiszter vagy 32 bites memóriaoperandus**
- **xmmreg/mem64 – SIMD lebegőpontos regiszter vagy 64 bites memóriaoperandus**
- **xmmreg/mem128 – SIMD lebegőpontos regiszter vagy 128 bites memóriaoperandus**

17.14.1 Adatmozgató utasítások

- **MOVAPS xmmreg,xmmreg/mem128**
- **MOVAPS xmmreg/mem128,xmmreg**
  (MOVe Aligned four Packed Single-FP)
- **MOVUPS xmmreg,xmmreg/mem128**
- **MOVUPS xmmreg/mem128,xmmreg**
  (MOVe Unaligned four Packed Single-FP)

  A forrás 128 bites tartalmát a cél regiszterbe vagy memóriaterületre írják. Ha a memóriaoperandus offszetcíme nem osztható 16-tal (tehát nem illeszkedik paragrafushatárra), akkor a MOVAPS 13-as kivételt (#GP) generál. A MOVUPS nem veszi figyelembe ezt az illeszkedést, de 17-es kivételt (#AC) ettől függetlenül kiválthat, ha az adott operandus nem illeszkedik duplaszóhatárra (és

persze engedélyeztük az illeszkedés-ellenőrzés kivétel keletkezését a CR0 regiszter AM (18-as) és az EFlags regiszter AC (szintén 18-as) bitjének beállításával, valamint CPL=3).

- **MOVHPS xmmreg,mem64**
- **MOVHPS mem64,xmmreg**

  (MOVe two Packed Single-FP to/from High-order quadword of a SIMD FP register)

  Ha a forrás a memóriaoperandus, akkor annak tartalmát beírja a cél SIMD lebegőpontos regiszter felső felébe, az alsó 64 bitet békén hagyja. Egyébként a forrás SIMD regiszter felső 64 bitjét eltárolja a cél memóriaterületen.
- **MOVLPS xmmreg,mem64**
- **MOVLPS mem64,xmmreg**

  (MOVe two Packed Single-FP to/from Low-order quadword of a SIMD FP register)

  Ha a forrás a memóriaoperandus, akkor annak tartalmát beírja a cél SIMD lebegőpontos regiszter alsó felébe, a felső 64 bitet békén hagyja. Egyébként a forrás SIMD regiszter alsó 64 bitjét eltárolja a cél memóriaterületen.
- **MOVHLPS dest xmmreg,xmmreg**

  (MOVe two Packed Single-FP from High-order 64-bits of source to Lower half of destination)

  A forrás felső kvadraszavát beírja a cél alsó felébe, a felső 64 bit pedig érintetlen marad.
- **MOVLHPS dest xmmreg,xmmreg**

  (MOVe two Packed Single-FP from Lower half of source to High-order 64-bits of destination)

  A forrás alsó kvadraszavát beírja a cél felső felébe, az alsó 64 bit pedig érintetlen marad.
- **MOVMSKPS reg32,xmmreg**

  (MOVe MaSK of four Packed Single-FP)

A forrásban levő négy egyszeres valós szám legfelső bitjéből (előjelbit) képez egy négybites bináris számot, majd azt zérókiterjesztetten beírja a cél 32 bites általános regiszterbe. Ez azt eredményezi, hogy a forrás 31, 63, 95 és 127 számú bitjei sorban a cél regiszter 0, 1, 2 és 3 számú bitjeibe kerülnek, és a cél 4..31 bitjei mind nullák lesznek.

- **MOVSS xmmreg,xmmreg/mem32**
- **MOVSS xmmreg/mem32,xmmreg**

  (MOVe Scalar Single-FP)

  Ha a forrás memóriaoperandus, akkor annak tartalmát a cél alsó duplaszavába tölti, majd a többi bitet kinullázza. Ha a cél memóriaoperandus, akkor a forrás SIMD regiszter alsó 32 bitjét a célban eltárolja. Ha mindkét operandus SIMD lebegőpontos regiszter, akkor a forrás alsó 32 bitjét a cél alsó 32 bitjébe írja, a felső bitek pedig változatlanok maradnak.

17.14.2 Konverziós utasítások

- **CVTPI2PS xmmreg,mmreg/mem64**

  (ConVerT Packed signed Integer To Packed Single-FP)

  A forrásban levő két előjeles bináris egész duplaszót az MXCSR-nek megfelelő kerekítéssel egyszeres lebegőpontos valóssá konvertálja, ezeket eltárolja a cél SIMD regiszter alsó felében, a felső 64 bitet pedig békén hagyja.
- **CVTPS2PI mmreg,xmmreg/mem64**

  (ConVerT Packed Single-FP To Packed signed Integer)

  A forrás SIMD regiszterben vagy 64 bites memóriaterületen levő két egyszeres valós számot

az MXCSR-nek megfelelő kerekítéssel előjeles bináris egész duplaszóvá konvertálja, majd azokat eltárolja a cél MMX regiszterben. Ha a kerekített érték nem ábrázolható 32 biten (azaz túlcsordul), a bináris egész meghatározatlan érték (integer indefinite), azaz 80000000h lesz eltárolva.

- **CVTTPS2PI mmreg,xmmreg/mem64**
  (ConVerT Packed Single-FP To Packed signed Integer using Truncation)
  A forrás SIMD regiszterben vagy 64 bites memóriaterületen levő két egyszeres valós számot csonkolással bináris egész duplaszóvá konvertálja, majd azokat eltárolja a cél MMX regiszterben. Ha a csonkolt érték nem ábrázolható 32 biten, a bináris egész meghatározatlan érték lesz eltárolva.
- **CVTSI2SS xmmreg,reg/mem32**
  (ConVerT Scalar signed Integer To Scalar Single-FP)
  A forrásban levő előjeles bináris egész duplaszót az MXCSR-nek megfelelő kerekítéssel egyszeres lebegőpontos valóssá konvertálja, azt eltárolja a cél SIMD regiszter alsó 32 bitjében, a többi bitet pedig békén hagyja.
- **CVTSS2SI reg32,xmmreg/mem32**
  (ConVerT Scalar Single-FP To Scalar signed Integer)
  A forrás SIMD regiszterben vagy 32 bites memóriaterületen levő egyszeres valós számot az MXCSR-nek megfelelő kerekítéssel előjeles bináris egész duplaszóvá konvertálja, majd azt eltárolja a cél általános regiszterben. Ha a kerekített érték nem ábrázolható 32 biten (azaz túlcsordul), a bináris egész meghatározatlan érték lesz eltárolva.
- **CVTTSS2SI reg32,xmmreg/mem32**

(ConVerT Scalar Single-FP To Scalar signed Integer using Truncation)

A forrás SIMD regiszterben vagy 32 bites memóriaterületen levő egyszeres pontosságú valós számot csonkolással bináris egész duplaszóvá konvertálja, majd azt eltárolja a cél általános regiszterben. Ha a csonkolt érték nem ábrázolható 32 biten, a bináris egész meghatározatlan érték lesz eltárolva.

17.14.3 Lebegőpontos aritmetikai utasítások

- **ADDPS dest,source**
  (ADD Packed Single-FP)
  A forrásban levő elemeket hozzáadja a célban levőkhöz, majd az eredményeket pakoltan a célban eltárolja.
- **ADDSS dest,source**
  (ADD Scalar Single-FP)
  A forrás alsó elemét hozzáadja a cél alsó eleméhez, majd az eredményt visszaírja a célba. A célban a többi bit értéke változatlan marad.
- **SUBPS dest,source**
  (SUBtract Packed Single-FP)
  A forrásban levő elemeket kivonja a célban levőkből, majd az eredményeket pakoltan a célban eltárolja.
- **SUBSS dest,source**
  (SUBtract Scalar Single-FP)
  A forrás alsó elemét kivonja a cél alsó eleméből, majd az eredményt visszaírja a célba. A célban a többi bit értéke változatlan marad.
- **MULPS dest,source**
  (MULtiply Packed Single-FP)

A forrásban levő elemeket megszorozza a célban levőkkel, majd az eredményeket pakoltan a célban eltárolja.

- **MULSS dest,source**
  (MULtiply Scalar Single-FP)
  A forrás alsó elemét megszorozza a cél alsó elemével, majd az eredményt visszaírja a célba. A célban a többi bit értéke változatlan marad.
- **DIVPS dest,source**
  (DIVide Packed Single-FP)
  A célban levő elemeket elosztja a forrásban levőkkel, majd az eredményeket pakoltan a célban eltárolja.
- **DIVSS dest,source**
  (DIVide Scalar Single-FP)
  A cél alsó elemét elosztja a forrás alsó elemével, majd az eredményt visszaírja a célba. A célban a többi bit értéke változatlan marad.
- **SQRTPS dest,source**
  (SQuare RooT of Packed Single-FP)
  A forrásban levő elemek négyzetgyökét pakoltan eltárolja a célban.
- **SQRTSS dest,source**
  (SQuare RooT of Scalar Single-FP)
  A forrás alsó elemének négyzetgyökét a cél alsó elemében eltárolja. A célban a többi bit értéke változatlan marad.
- **RCPPS dest,source**
  (ReCiProcal approximation of Packed Single-FP)
  A forrásban levő elemek reciprokának 12 bitre pontos közelítését pakolva eltárolja a célban.
- **RCPSS dest,source**
  (ReCiProcal approximation of Scalar Single-FP)

A forrás alsó eleme reciprokának 12 bitre pontos közelítését eltárolja a cél alsó elemében. A célban a többi bit értéke változatlan marad.

- **RSQRTPS dest,source**
  (ReCiProcal SQuare RooT approximation of Packed Single-FP)
  A forrásban levő elemek reciprok négyzetgyökének 12 bitre pontos közelítését pakolva eltárolja a célban.
- **RSQRTSS dest,source**
  (ReCiProcal SQuare RooT approximation of Scalar Single-FP)
  A forrás alsó eleme reciprok négyzetgyökének 12 bitre pontos közelítését eltárolja a cél alsó elemében. A célban a többi bit értéke változatlan marad.
- **MAXPS dest,source**
  (Maximum of Packed Single-FP)
  A forrásban és célban levő elempárok nagyobbik tagjait pakoltan a célban eltárolja. Ha csak az egyik tag NaN, akkor mindig a forrásban levő értéket adja vissza változatlanul.
- **MAXSS dest,source**
  (Maximum of Scalar Single-FP)
  A forrás és a cél alsó elemei közül a nagyobbat a cél alsó elemében eltárolja. A célban a többi bit értéke változatlan marad. Ha csak az egyik tag NaN, akkor mindig a forrásban levő értéket adja vissza változatlanul.
- **MINPS dest,source**
  (Minimum of Packed Single-FP)
  A forrásban és célban levő elempárok kisebbik tagjait pakoltan a célban eltárolja. Ha csak az egyik tag NaN, akkor mindig a forrásban levő értéket adja vissza változatlanul.
- **MINSS dest,source**

(Minimum of Scalar Single-FP)

A forrás és a cél alsó elemei közül a kisebbet a cél alsó elemében eltárolja. A célban a többi bit értéke változatlan marad. Ha csak az egyik tag NaN, akkor mindig a forrásban levő értéket adja vissza változatlanul.

17.14.4 Összehasonlító utasítások

- **CMPPS dest,source,0**
- **CMPEQPS dest,source**
  (CoMPare Packed Single-FP for EQual)
- **CMPPS dest,source,1**
- **CMPLTPS dest,source**
  (CoMPare Packed Single-FP for Less Than)
- **CMPPS dest,source,2**
- **CMPLEPS dest,source**
  (CoMPare Packed Single-FP for Less or Equal)
- **CMPPS dest,source,3**
- **CMPUNORDPS dest,source**
  (CoMPare Packed Single-FP for UNORDered)
- **CMPPS dest,source,4**
- **CMPNEQPS dest,source**
  (CoMPare Packed Single-FP for Not EQual)
- **CMPPS dest,source,5**
- **CMPNLTPS dest,source**
  (CoMPare Packed Single-FP for Not Less Than)
- **CMPPS dest,source,6**
- **CMPNLEPS dest,source**
  (CoMPare Packed Single-FP for Not Less or Equal)
- **CMPPS dest,source,7**
- **CMPORDPS dest,source**
  (CoMPare Packed Single-FP for ORDered)

A háromoperandusú CMPPS utasítás a forrásban levő elemeket hasonlítja össze a célban levőkkel, és ha a harmadik operandus (közvetlen bájt) által mutatott feltétel teljesül, akkor csupa 1-es, különben csupa 0-ás bitből álló értéket tárol el pakolva a célban. A közvetlen operandus csak a 0..7 értékek valamelyike lehet. A kétoperandusú mnemonikok a CMPPS utasítást kódolják le egy adott közvetlen operandussal.

- **CMPSS dest xmmreg,xmmreg/mem32,0**
- **CMPEQSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for EQual)**
- **CMPSS dest xmmreg,xmmreg/mem32,1**
- **CMPLTSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for Less Than)**
- **CMPSS dest xmmreg,xmmreg/mem32,2**
- **CMPLESS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for Less or Equal)**
- **CMPSS dest xmmreg,xmmreg/mem32,3**
- **CMPUNORDSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for UNORDered)**
- **CMPSS dest xmmreg,xmmreg/mem32,4**
- **CMPNEQSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for Not EQual)**
- **CMPSS dest xmmreg,xmmreg/mem32,5**
- **CMPNLTSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for Not Less Than)**
- **CMPSS dest xmmreg,xmmreg/mem32,6**
- **CMPNLESS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for Not Less or Equal)**
- **CMPSS dest xmmreg,xmmreg/mem32,7**
- **CMPORDSS dest xmmreg,xmmreg/mem32 (CoMPare Scalar Single-FP for ORDered)**

A háromoperandusú CMPSS utasítás a forrás alsó elemét hasonlítja össze a cél alsó elemével, és ha a harmadik operandus (közvetlen bájt) által mutatott feltétel teljesül, akkor csupa 1-es, különben csupa 0-ás bitből álló értéket tárol el a cél alsó elemében. A közvetlen operandus csak a 0..7 értékek valamelyike lehet. A kétoperandusú mnemonikok a CMPSS utasítást kódolják le egy adott közvetlen operandussal.

- **COMISS dest xmmreg,xmmreg/mem32**
  (COMpare Scalar Single-FP and set EFlags)
  A forrás alsó elemét összehasonlítja a cél alsó elemével, majd az eredménynek megfelelően beállítja a CF, PF és ZF flag-eket. Az AF, SF és OF flag-eket minden esetben törli. Az unordered feltétel teljesülésekor érvénytelen operandus numerikus kivétel (#IA) keletkezik. Ha ezt el akarjuk kerülni, használjuk inkább az UCOMISS utasítást. Az eredményt az alábbi táblázat alapján értelmezhetjük:

Eredmény	*CF*	*PF*	*ZF*
cél>forrás	0	0	0
cél<forrás	1	0	0
cél=forrás	0	0	1
Unordered	1	1	1

- **UCOMISS dest xmmreg,xmmreg/mem32**
  (COMpare Scalar Single-FP Unordered and set EFlags)
  A forrás alsó elemét összehasonlítja a cél alsó elemével, majd az eredménynek megfelelően beállítja a CF, PF és ZF flag-eket. Az AF, SF és OF flag-eket minden esetben törli. Az unordered feltétel

teljesülésekor csak akkor keletkezik érvénytelen operandus numerikus kivétel (#IA), ha valamelyik tag SNaN volt. Az eredmény értelmezését lásd a COMISS utasítás leírásánál.

17.14.5 Logikai utasítások

- **ANDPS dest,source**
  (bitwise logical AND)
- **ANDNPS dest,source**
  (bitwise logical AND plus NOT)
- **ORPS dest,source**
  (bitwise logical OR)
- **XORPS dest,source**
  (bitwise logical eXclusive OR)

  A forrás és a cél mint 128 bites számok között végeznek valamilyen logikai műveletet bitenként, majd az eredményt a célban eltárolják. Az ANDNPS utasítás a célnak veszi az egyes komplemensét, és ezt az értéket hozza logikai ÉS kapcsolatba a forrással.

17.14.6 Keverő utasítások (Data shuffle instructions)

- **SHUFPS dest,source,imm8**
  (SHUFfle Packed Single-FP)

  Az eredmény alsó két duplaszavába a cél, míg a felső két duplaszavába a forrás meghatározott sorszámú elemeit pakolja be, majd azt a cél regiszterben eltárolja. A harmadik, bájt méretű közvetlen operandus határozza meg, mely elemeket válassza ki az utasítás: a 0..1 bitek értéke adja meg, a cél hányadik eleme kerüljön az alsó duplaszóba (0..31 bitek); a 2..3 bitek tartalmazzák

a cél azon elemének sorszámát, amely az eredmény második duplaszavába (32..63 bitek) fog kerülni. A 4..5 és a 6..7 bitek hasonló módon a forrás négy eleme közül határozzák meg azt a kettőt, amelyeket az eredmény harmadik ill. negyedik duplaszavába (64..95 és 96..127 bitek) kell írni. Természetesen a forrás és a cél lehet azonos SIMD regiszter is, és bármelyik operandusból választhatjuk ugyanazt az elemet többször is.

- **UNPCKHPS dest,source**
  (UNPaCK Packed Single-FP from High-order 64-bits of source and destination)
  A cél és a forrás felső feléből felváltva beír egy-egy elemet az eredménybe, majd azt a cél SIMD regiszterbe tárolja. Működése: az eredmény 0..31 bitjei a cél 64..95 bitjeit, 32..63 bitjei a forrás 64..95 bitjeit, 64..95 bitjei a cél 96..127 bitjeit, míg 96..127 bitjei a forrás 96..127 bitjeit fogják tartalmazni.
- **UNPCKLPS dest,source**
  (UNPaCK Packed Single-FP from Low-order 64-bits of source and destination)
  A cél és a forrás alsó feléből felváltva beír egy-egy elemet az eredménybe, majd azt a cél SIMD regiszterbe tárolja. Működése: az eredmény 0..31 bitjei a cél 0..31 bitjeit, 32..63 bitjei a forrás 0..31 bitjeit, 64..95 bitjei a cél 32..63 bitjeit, míg 96..127 bitjei a forrás 32..63 bitjeit fogják tartalmazni.

17.14.7 További MMX utasítások

- **PAVGB mmreg,mmreg/mem64**
  (AVeraGe of unsigned Bytes)

Működése megegyezik a PAVGUSB 3DNow! utasításával, tehát a forrásban és a célban levő 8 db. előjeltelen, bájt méretű elempár kerekített átlagát számítja ki, majd az eredményeket pakolva eltárolja a cél regiszterben.

- **PAVGW mmreg,mmreg/mem64**
  (AVeraGe of unsigned Bytes)
  A forrásban és a célban levő 4 db. előjeltelen, szó méretű elempár kerekített átlagát számítja ki, majd az eredményeket pakolva eltárolja a cél regiszterben.
- **PMAXUB mmreg,mmreg/mem64**
  (MAXimum of Unsigned Bytes)
  A forrásban és célban levő 8 db. előjeltelen, bájt méretű elempár nagyobbik tagjait pakolva eltárolja a cél regiszterben.
- **PMAXSW mmreg,mmreg/mem64**
  (MAXimum of Signed Words)
  A forrásban és célban levő 4 db. előjeles, szó méretű elempár nagyobbik tagjait pakolva eltárolja a cél regiszterben.
- **PMINUB mmreg,mmreg/mem64**
  (MINimum of Unsigned Bytes)
  A forrásban és célban levő 8 db. előjeltelen, bájt méretű elempár kisebbik tagjait pakolva eltárolja a cél regiszterben.
- **PMINSW mmreg,mmreg/mem64**
  (MINimum of Signed Words)
  A forrásban és célban levő 4 db. előjeles, szó méretű elempár kisebbik tagjait pakolva eltárolja a cél regiszterben.
- **PSADBW mmreg,mmreg/mem64**
  (Word Sum of Absolute Differences of unsigned Bytes)

Veszi a forrásban és a célban levő előjeltelen, bájt méretű elempárok különbségének abszolút értékét, ezeket a számokat összegzi, majd a végeredményt a cél regiszter alsó szavában eltárolja. A további biteket a célban kinullázza.

- **PMULHUW mmreg,mmreg/mem64**
  (MULtiply Unsigned Words, and store High-order words of results)
  Összeszorozza a forrásban és a célban levő előjeltelen szavakat, majd az eredmények felső szavát pakolva eltárolja a célba.
- **PMOVMSKB reg32,mmreg**
  (MOVe MaSK of Packed Bytes)
  A forrásban levő nyolc bájt legfelső bitjéből (előjelbit) képez egy nyolcbites bináris számot, majd azt zérókiterjesztetten beírja a cél 32 bites általános regiszterbe. Ez azt eredményezi, hogy a forrás 7, 15, 23, 31, 39, 47, 55 és 63 számú bitjei sorban a cél regiszter 0, 1, 2, 3, 4, 5, 6 és 7 számú bitjeibe kerülnek, és a cél 8..31 bitjei mind nullák lesznek.
- **PEXTRW reg32,mmreg,imm8**
  (EXTRact Word)
  A forrás adott sorszámú szavát a cél általános regiszterbe írja zérókiterjesztetten. A harmadik, bájt méretű közvetlen operandus alsó két bitje határozza meg a kiválasztott szó sorszámát.
- **PINSRW mmreg,reg32/mem16,imm8**
  (INSeRt Word)
  A forrás alsó szavát a cél MMX regiszter adott sorszámú szavába írja, a többi bitet viszont nem bántja. A harmadik, bájt méretű közvetlen operandus alsó két bitje határozza meg a kiválasztott szó sorszámát.
- **PSHUFW mmreg,mmreg/mem64,imm8**

(SHUFfle Packed Words)

A forrás megadott sorszámú szavait sorban bepakolja az eredménybe, amit a cél MMX regiszterben tárol el. A harmadik, bájt méretű közvetlen operandus tartalmazza a négy szó sorszámát: a 0..1 bitek az eredmény alsó szavába (0..15 bitek), a 2..3 bitek a második szóba (16..31 bitek), a 4..5 bitek a harmadik szóba (32..47 bitek), míg a 6..7 bitek a felső szóba (48..63 bitek) kerülő forrásszó sorszámát jelölik ki.

17.14.8 Cache-selést vezérlő utasítások (Cacheability control instructions)

- **MASKMOVQ mmreg,mmreg**
  (MOVe bytes of Quadword according to MASK)
  A cél MMX regiszter maszkolt értékét beírja a DS:DI vagy DS:EDI címre. A maszkolás annyit jelent, hogy a forrás MMX regiszter mindegyik bájtjának felső bitje (előjelbit) határozza meg, hogy a cél regiszter megfelelő bájtját ki kell-e írni. Ha az adott bit 1 értékű, akkor a megfelelő bájt kiírásra kerül, különben 00000000b lesz helyette eltárolva. Az utasítás kifejezetten arra a célra készült a MOVNTQ és MOVNTPS utasításhoz hasonlóan, hogy az ilyen "egyszer használatos" (non-temporal) adatok memóriába tárolásakor a cache lehetőleg ne "szennyeződjön", azaz felesleges-en ne kerüljön be a cache-be olyan adat, aminek gyors elérhetősége számunkra nem fontos.
- **MOVNTQ mem64,mmreg**
  (MOVe Quadword Non-Temporal)

A forrás MMX regiszter tartalmát a cél memóriaterületre írja, és közben a cache "szennyeződését" minimálisra csökkenti.

- **MOVNTPS mem128,xmmreg**
  (MOVe aligned Packed Single-FP Non-Temporal)
  A forrás SIMD lebegőpontos regiszter tartalmát a cél memóriaterületre írja, és közben a cache "szennyeződését" minimálisra csökkenti. A MOVAPS utasításhoz hasonlóan, ha a memória-operandus offszetcíme nem osztható 16-tal (tehát nem illeszkedik paragrafushatárra), akkor 13-as kivételt (#GP) generál.
- **PREFETCHT0 mem8**
  (PREFETCH data using the T0 hint)
- **PREFETCHT1 mem8**
  (PREFETCH data using the T1 hint)
- **PREFETCHT2 mem8**
  (PREFETCH data using the T2 hint)
- **PREFETCHNTA mem8**
  (PREFETCH data using the NTA hint)
  A megadott memóriacímet tartalmazó cache-sort (aminek mérete legalább 32 bájt) a processzorhoz "közelebb" hozzák, azaz valamelyik cache-be beolvassák. Hogy melyik cache lesz ez, azt a előreolvasás típusa határozza meg: T0 esetén bármelyik lehet, T1 esetén az elsőszintű (L1) cache, míg T2 esetén a másodszintű (L2) cache tartalmától függ, hogy mit választ a processzor. Az NTA módszer a ritkán használatos (non-temporal) adatok olvasásakor lehet hasznos. Az utasítás gépi kódú alakja 0Fh 18h *r+i, ahol "i" értéke választja ki a használni kívánt módszert: a 000b, 001b, 010b és 011b értékek sorban az NTA, T0, T1 és T2 típusú előreolvasást jelentik, és a

mnemonikok is az ennek megfelelő kódot generálják. Az ezektől eltérő "i" érték a későbbi új utasítások számára fenntartott, végrehajtásuk pedig kiszámíthatatlan következménnyel jár.

- **SFENCE (no op)**
  (Store FENCE)
  Addig várakozik, amíg az összes korábban kiadott memóriamódosító tárolási művelet be nem fejeződik. Garantálja, hogy a fogyasztó (consumer) minden adatot pontosan ugyanúgy és egyszerre kapjon meg, ahogy azokat a termelő (producer) folyamat szolgáltatja.

17.14.9 Állapot-kezelő utasítások (State management instructions)

- **LDMXCSR mem32**
  (LoaD SIMD Control/Status Register)
  A megadott címről betölti az MXCSR regiszter új tartalmát. Ha valamelyik fenntartott bitpozícióban 1 áll, 13-as kivétel (#GP) keletkezik.
- **STMXCSR mem32**
  (STore SIMD Control/Status Register)
  A megadott címre eltárolja az MXCSR regiszter tartalmát, a fenntartott bitek helyére 0-t írva.
- **FXRSTOR mem**
  (ReSTORe Floating-Point and MMX state, and SSE state)
  A megadott memóriacímen elhelyezkedő 512 bájtos területről betölti az FPU/MMX állapotot, valamint az SSE állapotot. Ezt az információt az FXSAVE utasítással tárolhatjuk el. Ha a cím nem osztható 16-tal (tehát nem illeszkedik paragrafus-határra), 13-as (#GP) vagy 17-es (#AC) kivétel

keletkezik. Hogy melyik, az implementáció-függő. Ha a CR4 regiszter OSFXSR (9-es) bitje 0 értékű, akkor az MXCSR és XMM0..XMM7 regiszterek értékét nem állítja vissza.

- **FXSAVE mem**
  (SAVE Floating-Point and MMX state, and SSE state)
  A megadott memóriacímen elhelyezkedő 512 bájt nagyságú területre eltárolja az FPU/MMX állapotot, valamint az SSE állapotot. Ha a cím nem osztható 16-tal (tehát nem illeszkedik paragrafushatárra), 13-as (#GP) vagy 17-es (#AC) kivétel keletkezik. Hogy melyik, az implementáció-függő. Az FSAVE FPU utasítással ellentétben nem inicializálja újra a koprocesszort. Ha a CR4 regiszter OSFXSR (9-es) bitje 0 értékű, akkor az MXCSR és XMM0..XMM7 regiszterek értékét nem tárolja el.

Az alábbi táblázat mutatja a terület felépítését:

Bájtpozíció	*Név*	*Tartalom*
0..1	FCW	FPU vezérlőregiszter (Control Word)
2..3	FSW	FPU státuszregiszter (Status Word)
4..5	FTW	FPU tag-regiszter (Tag Word)
6..7	FOP	FPU műveleti kód reg. (Opcode Reg.)
8..11	IP	FPU Instruction Pointer reg. offszet
12..13	CS	FPU Instruction Pointer reg. szelektor
14..15	–	–
16..19	DP	FPU Operand Pointer regiszter offszet
20..21	DS	FPU Operand Pointer reg. szelektor
22..23	–	–
24..27	MXCSR	SIMD vezérlő-/státuszregiszter
28..31	–	–
32..41	ST(0)/MM0	ST(0) FPU verem/MM0 MMX reg.
42..47	–	–

48..57	ST(1)/MM1	ST(1) FPU verem/MM1 MMX reg.
58..63	–	–
64..73	ST(2)/MM2	ST(2) FPU verem/MM2 MMX reg.
74..79	–	–
80..89	ST(3)/MM3	ST(3) FPU verem/MM3 MMX reg.
90..95	–	–
96..105	ST(4)/MM4	ST(4) FPU verem/MM4 MMX reg.
106..111	–	–
112..121	ST(5)/MM5	ST(5) FPU verem/MM5 MMX reg.
122..127	–	–
128..137	ST(6)/MM6	ST(6) FPU verem/MM6 MMX reg.
138..143	–	–
144..153	ST(7)/MM7	ST(7) FPU verem/MM7 MMX reg.
154..159	–	–
160..175	XMM0	XMM0 SIMD lebegőpontos regiszter
176..191	XMM1	XMM1 SIMD lebegőpontos regiszter
192..207	XMM2	XMM2 SIMD lebegőpontos regiszter
208..223	XMM3	XMM3 SIMD lebegőpontos regiszter
224..239	XMM4	XMM4 SIMD lebegőpontos regiszter
240..255	XMM5	XMM5 SIMD lebegőpontos regiszter
256..271	XMM6	XMM6 SIMD lebegőpontos regiszter
272..287	XMM7	XMM7 SIMD lebegőpontos regiszter
288..511	–	–

Ahol nincs feltüntetve tartalom, akkor azok a bájtok fenntartottak.

Az FPU tag-regiszter nem a megszokott formában tárolódik el. Az FSAVE utasítás által használt formátummal ellentétben csak azt tartalmazza az FTW mező, hogy az adott FPU adatregiszter érvényes-e avagy nem. Az FTW szó alsó bájtjában minden bit egy-egy fizikai FPU adatregiszterhez tartozik, a bitek sorrendje tehát TOS-tól független. Ha az adott bit értéke 1, akkor a

regiszter érvényes (valid, zero vagy special), míg 0 esetén üres (empty). A felső bájt értéke fenntartott. A műveleti kód regisztert tartalmazó szónak a felső 5 bitje nincs kihasználva (a 7-es bájt 3..7 bitjei), ezek értéke nulla.

Az utasításmutató és operandusmutató regiszterek offszetje 32 bites kódszegmens esetén duplaszó nagyságú, míg 16 bites kódszegmens használatakor csak az alsó szó érvényes, és a felső két bájt fenntartott (10, 11, 18 és 19 bájtok). Szintén erre a két regiszterre vonatkozik, hogy védett módban szelektor, míg valós módban értelemszerűen szegmens érték lesz eltárolva.

Az MXCSR regiszter fenntartott bitjei helyén 0 kerül eltárolásra.

17.15 Összefoglalás

Ebben a fejezetben összefoglaljuk a processzor által biztosított programozási környezet minden fontosabb elemét. Az itt közölt adatok az aktuális legfrissebb processzorokra (Intel Pentium III, AMD K6-2 és AMD K6-III) vonatkoznak. Ahol szükséges, a két gyártó termékei közti különbségeket is kiemeljük.

17.15.1 Általános célú regiszterek (General purpose registers)

32 bites	*16 bites*	*8 bites felső*	*8 bites alsó*
EAX	AX	AH	AL
EBX	BX	BH	BL

ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI	–	–
EDI	DI	–	–
ESP	SP	–	–
EBP	BP	–	–

17.15.2 Szegmensregiszterek (Segment selector registers)

Kódszegmens (code segment): CS
Adatszegmens (data segment): DS, ES, FS, GS
Veremszegmens (stack segment): SS

Mindegyik szegmensregiszter 16 bites. A regiszterek tartalmát azonban másképp kell értelmezni a processzor üzemmódjától függően.

Valós és virtuális-8086 módban szegmenscímet tartalmaznak ezek a regiszterek:

Bitpozíció	*Tartalom*
0..15	Szegmenscím (Segment address)

Védett módban mindegyik szegmensregiszter egy szelektorregiszterként funkcionál:

Bitpozíció	*Jelölés*	*Név*
0..1	RPL	Requested Privilege Level
2	TI	Table Indicator
3..15	–	Index

17.15.3 Utasításmutató regiszter (Instruction pointer register)

32 bites	*16 bites*
EIP	**IP**

17.15.4 FPU utasításmutató és operandusmutató regiszterek (FPU instruction pointer and operand pointer registers)

A 48 bites FPU IP és OP regiszterek tartalmát másképpen kell értelmezni a processzor üzemmódjától függően. Valós és virtuális-8086 módban 16 bites szegmenset és 16 bites offszetet tartalmaz mindkét regiszter. Védett módban a kódszegmens címméretétől függően 16 bites szelektor és 16 bites offszet, vagy pedig 16 bites szelektor és 32 bites offszet tárolódik a regiszterekben. Az IP regiszter a legutolsó, nem vezérlő FPU utasításra mutat, OP pedig a legutoljára használt memóriaoperandus címét tartalmazza.

Az IP és OP regiszterek felépítése megegyezik:

Bitpozíció	*Tartalom*
0..31	**16 vagy 32 bites offszet**
32..47	**Szegmens vagy szelektor érték**

17.15.5 FPU műveleti kód regiszter (FPU opcode register)

Ebben a regiszterben a legutolsó, nem vezérlő FPU utasítás műveleti kódja található meg. A műveleti kód első bájtjának felső 5 bitje (3..7 bitek) mindegyik FPU utasításnál állandóak (11011b), ezért ezeket nem tárolják el.

Bitpozíció	*Tartalom*
0..7	Műveleti kód 2. bájtja
8..10	Műveleti kód 1. bájtjának 0..2 bitjei

17.15.6 FPU adatregiszterek (FPU data registers)

A fizikai regiszterek neve FPR0, FPR1, …, FPR7. Azt a számot, ami a verem tetejét tartalmazó adatregiszter sorszáma, TOS-nak (Top Of Stack) nevezzük. A verem tetejét ST ill. ST(0) jelöli, míg a verem tetejéhez képesti i-edik regisztert ST(i). ST(i) értelmezésekor figyelembe kell venni, hogy a hivatkozott fizikai regiszter száma (TOS+i) mod 8 lesz. Így ha mondjuk TOS=3, ST és ST(0) az FPR3, ST(1) az FPR4, ST(4) az FPR7, ST(7) pedig az FPR2 regisztert fogja kijelölni.

Mindegyik koprocesszor adatregiszter 80 bites. A regiszterek felépítése:

Bitpozíció	*Tartalom*
0..62	Mantissza törtrész (Significand fractional part)
63	Mantissza egészrész (Significand integer part)
64..78	Eltolt exponens (Biased exponent)
79	Előjel (Sign)

17.15.7 MMX regiszterek (MMX registers)

Az MMX regiszterek neve MM0, MM1, …, MM7. Ezek a regiszterek a fizikai koprocesszor adatregiszterekre vannak ráképezve, méghozzá azok mantissza részére (0..63 bitek). Eszerint MM0 az FPR0, MM1 az FPR1, stb., MM7 az FPR7 regiszter alsó 64 bitjén helyezkedik el.

Az MMX regiszterek mérete 64 bit.

17.15.8 3DNow! regiszterek (3DNow! registers)

A 3DNow! regiszterek megegyeznek az MMX regiszterekkel, de tartalmukat a 3DNow! lebegőpontos utasítások az MMX utasításoktól eltérő módon értelmezik.

Az Intel Pentium III processzor nem támogatja a 3DNow! technológiát, s így a 3DNow! regisztereket sem.

17.15.9 SIMD lebegőpontos regiszterek (SIMD floating-point registers)

A SIMD lebegőpontos regiszterek neve XMM0, XMM1, …, XMM7. Ezek a regiszterek fizikailag függetlenek a processzor többi regiszterétől, így a koprocesszor adatregisztereitől is. Mindegyik regiszter 128 bites.

Az AMD K6-2 és K6-III processzorok nem támogatják az SSE technológiát, s így a SIMD lebegőpontos regisztereket sem.

17.15.10 Státuszregiszter (Status register)

A 32 bites EFlags regiszter alsó 16 bitje Flags néven érhető el. A regiszter bitjeinek funkcióját, a flag-ek nevét mutatja a következő táblázat:

Bitpozíció	*Jelölés*	*Név*
0	CF	Carry Flag
1	–	–
2	PF	Parity Flag
3	–	–
4	AF	Auxilliary/Adjust Flag
5	–	–
6	ZF	Zero Flag
7	SF	Sign Flag
8	TF	Trap Flag
9	IF	Interrupt Flag
10	DF	Direction Flag
11	OF	Overflow Flag
12..13	IOPL	Input/Output Privilege Level
14	NT	Nested Task
15	–	–
16	RF	Resume Flag
17	VM	Virtual-8086 Mode
18	AC	Alignment Check enable
19	VIF	Virtual Interrupt Flag
20	VIP	Virtual Interrupt Pending
21	ID	processor IDentification
22..31	–	–

17.15.11 FPU státuszregiszter (FPU status word/register)

Bitpozíció	*Jelölés*	*Név*
0	IE	Invalid operation Exception
1	DE	Denormalized operand Exception

2	ZE	division by Zero Exception
3	OE	Overflow Exception
4	UE	Underflow Exception
5	PE	Precision loss Exception
6	SF	Stack Fault
7	ES	Error Summary status
8	C0	Condition code 0
9	C1	Condition code 1
10	C2	Condition code 2
11..13	TOS	Top Of Stack pointer
14	C3	Condition code 3
15	B	FPU Busy

17.15.12 FPU tag-regiszter (FPU tag word/register)

Az FPU tag-regiszter mezőinek értéke utal a koprocesszor adatregiszterek tartalmára. Az egyes mezők a megfelelő fizikai adatregiszterhez tartoznak, így pl. Tag0 FPR0, Tag1 FPR1 stb. tartalmára utal.

Bitpozíció	*Tartalom*
0..1	Tag0
2..3	Tag1
4..5	Tag2
6..7	Tag3
8..9	Tag4
10..11	Tag5
12..13	Tag6
14..15	Tag7

A tag-ek értékét így kell értelmezni:

Tag értéke	*Tartalom*
00	**Érvényes szám (Valid)**
01	**Nulla (Zero)**
10	**Különleges (Special)**
11	**Üres (Empty)**

17.15.13 FPU vezérlőregiszter (FPU control word/register)

Bitpozíció	*Jelölés*	*Név*
0	**IM**	**Invalid operation excpt. Mask**
1	**DM**	**Denormalized operand excpt. Mask**
2	**ZM**	**division by Zero excpt. Mask**
3	**OM**	**Overflow excpt. Mask**
4	**UM**	**Underflow excpt. Mask**
5	**PM**	**Precision loss excpt. Mask**
6	**–**	**–**
7	**I**	**Interrupt enable mask**
8..9	**PC**	**Precision Control**
10..11	**RC**	**Rounding Control**
12	**IC**	**Interrupt Control**
13..15	**–**	**–**

Az I bit csak a 8087-es koprocesszoron létezik.

Az IC bitnek csak a 8087, 80187 és 80287 koprocesszorok esetén van jelentősége, de a későbbi koprocesszorok (a 80287xl-t is beleértve) és processzorok is támogatják (de nem használják) ezt a bitet a kompatibilitás miatt.

A PC és RC mezők értelmezése pedig így történik:

PC értéke	*Pontosság*
00	**Egyszeres**
01	**–**

10	Dupla
11	Kiterjesztett

RC értéke	*Kerekítés módja*
00	Legközelebbi értékhez
01	Lefelé (-Inf felé)
10	Felfelé (+Inf felé)
11	Nulla felé (csonkítás)

17.15.14 SIMD vezérlő-/státuszregiszter (SIMD floating-point control/status register)

Az MXCSR regiszter felépítése:

Bitpozíció	*Jelölés*	*Név*
0	IE	Invalid operation Exception
1	DE	Denormalized operand Exception
2	ZE	division by Zero Exception
3	OE	Overflow Exception
4	UE	Underflow Exception
5	PE	Precision loss Exception
6	–	–
7	IM	Invalid operation excpt. Mask
8	DM	Denormalized operand excpt. Mask
9	ZM	division by Zero excpt. Mask
10	OM	Overflow excpt. Mask
11	UM	Underflow excpt. Mask
12	PM	Precision loss excpt. Mask
13..14	RC	Rounding Control
15	FZ	Flush-to-Zero
16..31	–	–

Az RC mező értelmezése megegyezik az FPU vezérlőregiszterének hasonló mezőjének értelmezésével.

A fenntartott bitek értéke mindig 0.

Ez a regiszter nem létezik AMD processzorokon.

17.15.15 Vezérlőregiszterek (Control registers)

A CR0 regiszter felépítése:

Bitpozíció	*Jelölés*	*Név*
0	PE	Protection Enable
1	MP	Monitor coProcessor/Math Present
2	EM	EMulation
3	TS	Task Switched
4	ET	Extension Type
5	NE	Numeric Error handling
6..15	–	–
16	WP	Write Protect
17	–	–
18	AM	Alignment check Mask
19..28	–	–
29	NW	cache Not Write-through
30	CD	Cache Disable
31	PG	PaGing

A CR2 regiszter a laphiba lineáris címét tartalmazza:

Bitpozíció	*Tartalom*
0..31	Laphiba lin. címe (Page fault linear address)

A CR3 regiszter felépítése, ha a CR4 regiszterben PAE=0:

Bitpozíció	*Jelölés*	*Név*
0..2	–	–
3	PWT	Page-level Write-Through
4	PCD	Page-level Cache Disable
5..11	–	–
12..31	PDB	Page-Directory Base

A CR3 regiszter tartalma, ha a CR4 regiszterben PAE=1:

Bitpozíció	*Jelölés*	*Név*
0..2	–	–
3	PWT	Page-level Write-Through
4	PCD	Page-level Cache Disable
5..31	PDPTB	Page-Directory Pointer Table Base

A CR4 regiszter felépítése:

Bitpozíció	Jelölés	Név
0	VME	Virtual Mode virtual int. Extension
1	PVI	Protected mode Virtual Interrupts
2	TSD	Time-Stamp instruction Disable
3	DE	Debugging Extensions
4	PSE	Page Size Extension
5	PAE	Physical Address Extension enable
6	MCE	Machine Check exception Enable
7	PGE	Page Global Enable

8	PCE	Performance-Monitoring Counter instruction Enable
9	OSFXSR	Operating System supports FXRSTOR/FXSAVE
10	OSXMMEXCPT	Operating System supports SIMD floating-point unMasked EXCePTions
11..31	–	–

Az AMD K6-2 és K6-III processzorok a PAE, PGE, PCE, OSFXSR és OSXMMEXCPT biteket nem támogatják.

A fenntartott bitek értéke mindig 0.

17.15.16 Nyomkövető regiszterek (Debug registers)

A DR0..DR3 regiszterek felépítése:

Bitpozíció	*Tartalom*
0..31	Töréspont lin. címe (Breakpoint linear address)

A DR6 regiszter felépítése:

Bitpozíció	*Jelölés*	*Név*
0	B0	Breakpoint 0 condition detected
1	B1	Breakpoint 1 condition detected
2	B2	Breakpoint 2 condition detected
3	B3	Breakpoint 3 condition detected
4..11	–	–
12	–	–
13	BD	general Detect Breakpoint

14	BS	Single-step Breakpoint
15	BT	Task switch Breakpoint
16..31	–	–

A 4..11 és 16..31 bitek értéke mindig 1, a 12-es bit pedig mindig törölve van.

A DR7 regiszter felépítése:

Bitpozíció	*Jelölés*	*Név*
0	L0	breakpoint 0 Local enable
1	G0	breakpoint 0 Global enable
2	L1	breakpoint 1 Local enable
3	G1	breakpoint 1 Global enable
4	L2	breakpoint 2 Local enable
5	G2	breakpoint 2 Global enable
6	L3	breakpoint 3 Local enable
7	G3	breakpoint 3 Global enable
8	LE	Local Exact breakpoint enable
9	GE	Global Exact breakpoint enable
10	–	–
11..12	–	–
13	GD	General Detect enable
14..15	–	–
16..17	R/W0	breakpoint 0 Read/Write condition
18..19	LEN0	breakpoint 0 LENgth condition
20..21	R/W1	breakpoint 1 Read/Write condition
22..23	LEN1	breakpoint 1 LENgth condition
24..25	R/W2	breakpoint 2 Read/Write condition
26..27	LEN2	breakpoint 2 LENgth condition
28..29	R/W3	breakpoint 3 Read/Write condition
30..31	LEN3	breakpoint 3 LENgth condition

A 11-es, 12-es, 14-es és 15-ös bitek értéke mindig 0. A 10-es bitnek mindig 1-et kell tartalmaznia.

17.15.17 Memória-kezelő regiszterek (Memory-management registers)

A GDTR (Global Descriptor Table Register) és IDTR (Interrupt Descriptor Table Register) regiszterek felépítése megegyezik:

Bitpozíció	*Tartalom*
0..15	Határ (Limit)
16..47	Lineáris báziscím (Linear base address)

Az LDTR (Local Descriptor Table Register) és TR (Task Register) regiszterekből csak a 16 bites szelektort tartalmazó rész érhető el közvetlenül, a határt, a lineáris báziscímet és az attribútumokat tartalmazó mezők rejtettek.

17.15.18 Modell-specifikus regiszterek (Model-specific registers–MSRs)

Az Intel Pentium III processzor 79, az AMD K6-2 processzor 7, az AMD K6-III pedig 11 db. dokumentált MSR-t tartalmaz. Ezeket már felsorolni is sok lenne, ezért csak 3-at ismertetünk közülük. A gyártók dokumentációjában több-kevesebb részletességgel megtalálható mindegyik leírása.

A Time-Stamp Counter (TSC) mindhárom processzoron megtalálható, címe 10h. Felépítése nem túl összetett, a regiszter összes bitje a számláló aktuális értékét tartalmazza:

Bitpozíció	*Tartalom*
0..63	TSC aktuális értéke

Az Extended Feature Enable Register (EFER) regiszter csak AMD processzorokon létezik, címe 0C0000080h. Felépítése processzortípustól függően változik.

A K6-2 processzoron csak egyetlen bitnek van szerepe az EFER regiszterben:

Bitpozíció	*Jelölés*	*Név*
0	SCE	System Call/return Extension
1..63	–	–

A K6-III processzor három új mezőt vezetett be az EFER regiszterben:

Bitpozíció	*Jelölés*	*Név*
0	SCE	System Call/return Extension
1	DPE	Data Prefetch Enable
2..3	EWBEC	EWBE Control
4	L2D	L2 cache Disable
5..63	–	–

A SYSCALL/SYSRET Target Address Register (STAR) regiszter szintén csak AMD processzorokon használható. A regiszter a 0C0000081h címen található. Felépítése a K6-2 és K6-III processzorokon megegyezik:

Bitpozíció	*Név*
0..31	SYSCALL target ring 0 EIP address
32..47	SYSCALL CS and SS selector base
48..63	SYSRET CS and SS selector base

17.15.19 Regiszterek kezdeti értéke (Initial value of registers)

Az eddig említett regiszterek reset utáni értékét mutatja a következő táblázat:

Regiszter	*Érték*
EAX	00000000h/?
EDX	processzortól függő
EBX,ECX,ESI,EDI,ESP,EBP	00000000h
CS szelektor	0F000h
CS bázis	0FFFF0000h
CS határ	0FFFFh
CS attribútum	jelenlevő, írható/olvasható, elért
DS,ES,FS,GS,SS szelektor	0000h
DS,ES,FS,GS,SS bázis	00000000h
DS,ES,FS,GS,SS határ	0FFFFh
DS,ES,FS,GS,SS attribútum	jelenlevő, írható/olvasható, elért
EIP	0000FFF0h
FPU IP és OP	000000000000h
FPU Opcode	00000000000b
FPR0..FPR7	+0.0
MM0..MM7	0000000000000000h
XMM0..XMM7	000…00h
EFlags	00000002h
FPU Status Word	0000h
FPU Tag Word	5555h
FPU Control Word	0040h

MXCSR	**00001F80h**
CR0	**60000010h**
CR2,CR3,CR4	**00000000h**
DR0,DR1,DR2,DR3	**00000000h**
DR6	**0FFFF0FF0h**
DR7	**00000400h**
GDTR,IDTR bázis	**00000000h**
GDTR,IDTR határ	**0FFFFh**
GDTR,IDTR attribútum	**jelenlevő, írható/olvasható**
LDTR,TR szelektor	**0000h**
LDTR,TR bázis	**00000000h**
LDTR,TR határ	**0FFFFh**
LDTR,TR attribútum	**jelenlevő, írható/olvasható**
TSC	**0000000000000000h**
PMCs,Event Select Registers	**0000000000000000h**
MTRRs	**mindegyik tartomány letiltva**
EFER	**lásd megjegyzés**
STAR	**0000000000000000h**

Ha a reset során a hardver megkérte a processzort a beépített önteszt (Built-In Self-Test–*BIST*) elvégzésére, akkor EAX tartalmazza ennek eredményét. Ha EAX=00000000h, a teszt sikeres volt. Különben EAX értéke nemzéró és meghatározatlan.

A processzor típusát (család, modell, stepping) meghatározó információkat tartalmazza az EDX regiszter. Ez processzoronként és gyártónként változó tartalmú. Leírását lásd a "Processzorok detektálása" c. fejezetben. Intel Pentium III esetén értéke 0000067xh, AMD K6-2 esetén 0000058xh, míg AMD K6-III esetén 0000059xh. "x" a stepping-et jelenti.

Az EFlags felső 10 bitjének (22..31 bitek) állapota reset után definiálatlan.

Az EFER regiszter tartalma AMD K6-2 processzoron 0h, míg AMD K6-III processzoron 0000000000000002h.

Az Intel Pentium III processzoron az itt megemlített TSC, PMC, Event Select Register és MTRR MSR-eken kívül az összes többi MSR tartalma definiálatlan.

Az AMD K6-2 és K6-III processzorokon az összes MSR jól meghatározott értéket kap reseteléskor, ami általában az adott regiszter kinullázását jelenti. A két kivételt az EFER (Extended Feature Enable Register) és PSOR (Processor State Observability Register) MSR-ek jelentik, ezek a K6-2 processzoron törlődnek, a K6-III processzoron viszont más értéket kapnak.

17.15.20 Kivételek (Exceptions)

A támogatott kivételeket és azok jellemzőit sorolja fel a következő táblázat:

Szám	*Mnemo*	*Név*	*Típus*	*Hibak.*	*Valós m.*
0	#DE	Divide Error	Fault	N	I
1	#DB	Debug	Fault/Trap	N	I
2	NMI	NonMaskable Interrupt	–	–	I
3	#BP	Breakpoint	Trap	N	I
4	#OF	Overflow	Trap	N	I
5	#BR	Bound Range	Fault	N	I
6	#UD	Undefined Opcode	Fault	N	I
7	#NM	No Math Coprocessor	Fault	N	I
8	#DF	Double Fault	Abort	I (0)	I
9	–	–	–	–	–
10	#TS	Invalid TSS	Fault	I	N
11	#NP	Segment Not Present	Fault	I	N

12	#SS	Stack-Segment Fault	Fault	I	I
13	#GP	General Protection	Fault	I	I
14	#PF	Page Fault	Fault	I	N
16	#MF	Math Fault	Fault	N	I
17	#AC	Alignment Check	Fault	I (0)	N
18	#MC	Machine Check	Abort	N	I
19	#XF	SSE FP Error	Fault	N	I

Az első oszlop mutatja a kivétel sorszámát, a második a kivételt azonosító jelölést (mnemonikot), a harmadik tartalmazza a kivétel nevét, a negyedik pedig a kivétel típusát. Az ötödik oszlop mutatja, védett és virtuális módban kerül-e hibakód a verembe. Végül az utolsó oszlop azt tartalmazza, valós módban is keletkezhet-e az adott kivétel.

Az NMI megszakítás is megtalálható a fenti táblázatban, bár nem kivétel.

Igaz, hogy korábban már leírtuk, de azért újra elmondjuk, mit is jelez pontosan a kivétel típusa.

A hiba (fault) típusú kivételek olyan hibát jeleznek, amik lehetetlenné teszik a kivételt kiváltó utasítás végrehajtását, de a hiba kijavítható, és a program vagy taszk futása folytatható. A veremben levő CS:IP/EIP-másolat mindig a hibát kiváltó utasításra mutat. Védett módban, taszkváltás során keletkező hibáknál azonban megtörténhet, hogy nem a hibázó utasításra, hanem az új taszk legelső utasítására fog mutatni a másolat. Továbbá, a kivétel-kezelőbe belépve a processzor állapota a hibás utasítás végrehajtása előtti helyzetet tükrözi (kivéve néhány speciális helyzetet, amikor a korábbi állapot csak részlegesen lesz visszaállítva).

A csapda (trap) kivételek inkább tájékoztató jellegűek, a program folytatásához nem szükséges külön lépéseket tenni (de például a 3-as, töréspont kivétel esetében nincs is mit kijavítani).

A CS:IP/EIP-másolat mindig a kivételt kiváltó utasítást követő utasításra fog mutatni. A processzor állapota az adott utasítás végrehajtása utáni helyzetet tükrözi.

Az abort kivételek olyan súlyos hibákat jeleznek, amelyek kijavítása nem mindig lehetséges, sőt néha lehetetlen is. A CS:IP/EIP másolat ezért sokszor nem tudja pontosan meghatározni a hiba helyét, főleg ha egy hardverhiba miatt keletkezett a kivétel. A processzor állapota definiálatlan, a program vagy taszk futása nem folytatható.

A 80286-os és 80386-os processzorokon létező hajdani 9-es kivételt már nem támogatják az újabb processzorok.

Az AMD K6-2 és K6-III processzorok nem ismerik a 18-as és 19-es kivételeket.

17.15.21 FPU kivételek (FPU exceptions)

A következő táblázat mutatja a koprocesszor utasítás használatakor keletkező numerikus kivételeket:

Mnemonik	*Név*
#IS	Invalid operation (Stack fault)
#IA	invalid operation (Invalid Arithmetic operand)
#D	Denormalized operand
#Z	division by Zero
#O	Overflow
#U	Underflow
#P	Precision loss

Az FPU maszkolatlan numerikus kivétel esetén 2-es megszakítást (NMI) vagy 16-os kivételt (#MF) vált ki, ezt a CR0 regiszter NE (5-ös) bitje szabályozza.

17.15.22 SIMD lebegőpontos kivételek (SIMD floating-point exceptions)

Az SSE utasítások végrehajtása során keletkező numerikus kivételeket foglalja össze a következő táblázat:

Mnemonik	*Típus*	*Név*
#I	Pre	Invalid operation
#D	Pre	Denormalized operand
#Z	Pre	division by Zero
#O	Post	Overflow
#U	Post	Underflow
#P	Post	Precision loss

Az #I, #D és #Z kivételeket az utasítások végrehajtása előtt érzékeli a processzor, típusuk ezért *pre-computation*. Az #O, #U és #P kivételek ezzel szemben az utasítások végrehajtását követően keletkeznek, típusuk ezért *post-computation*.

Maszkolatlan numerikus kivétel esetén a processzor 6-os (#UD) vagy 19-es (#XF) kivétellel reagál, ezt a CR4 regiszter OSXMMEXCPT (10-es) bitje befolyásolja.

A fentebb említett numerikus kivételeken kívül számos más hiba is keletkezhet SSE utasítások végrehajtásakor. A 12-es (#SS), 13-as (#GP) és 14-es (#PF) kivételek a memória elérése közben bekövetkező hibákat jelzik. Ha az illeszkedés-ellenőrzés aktiválva van, a nem megfelelően illeszkedő operandusok 17-es (#AC) kivételhez vezetnek. 128-bites operandusok esetén, ha azok nem illeszkednek paragrafushatárra (16 bájtos határra), szintén 13-as kivétel keletkezik (kivétel a MOVUPS utasítás).

Ha a következő táblázatban mutatott feltételek nem teljesülnek, 6-os kivétel keletkezik:

Utasítás	*CPUID.XMM*	*CR0.EM*	*CR4.OSFXSR*
SFENCE	**1**	**–**	**–**
PREFETCHccc	**1**	**–**	**–**
MASKMOVQ	**1**	**0**	**–**
MOVNTQ	**1**	**0**	**–**
FXRSTOR, FXSAVE	**1**	**0**	**–**
SIMD MMX	**1**	**0**	**–**
Többi SSE	**1**	**0**	**1**

Végül, ha egy SSE utasítás végrehajtásakor a fenti feltételek fennállnak, továbbá a CR0 regiszter TS (3-as) bitje 1-es értékű, 7-es (#NM) kivétel keletkezik. Ez a feltétel az SFENCE, PREFETCHT0, PREFETCHT1, PREFETCHT2 és PREFETCHNTA utasításokra nem vonatkozik.

17.15.23 Prefixek (Prefixes)

A processzorok által ismert prefixeket sorolja fel a következő táblázat:

Gépi kód	*Mnemonik*	*Név*
66h	**–**	**Operand size**
67h	**–**	**Address size**
26h	**ES:,SEGES**	**ES: SEGment override**
2Eh	**CS:,SEGCS**	**CS: SEGment override**
36h	**SS:,SEGSS**	**SS: SEGment override**
3Eh	**DS:,SEGDS**	**DS: SEGment override**
64h	**FS:,SEGFS**	**FS: SEGment override**
65h	**GS:,SEGGS**	**GS: SEGment override**
0F0h	**LOCK**	**Bus LOCK**
0F2h	**REPNE,REPNZ**	**REPeat while Not Equal/Zero**

0F3h	REP,REPE,REPZ	REPeat (while Equal/Zero)
0Fh	–	Extended instruction prefix
0Fh 0Fh	–	3DNow! instruction prefix
0F3h	–	SSE instruction prefix

Az utolsó három utasítás "prefixek" nem számítanak valódi prefixnek, az elnevezés csak elhelyezkedésük hasonlatosságára utal. Velük bővebben az "Utasítások kódolása" c. fejezetben foglalkozunk.

Az Intel Pentium III processzoron a 0Fh 0Fh prefix érvénytelen.

Az AMD K6-2 és K6-III processzorok nem támogatják az SSE technológiát, s így az utolsó prefixet sem.

18 UTASÍTÁSOK KÓDOLÁSA

Ebben a fejezetben azt mutatjuk meg, az assembler hogyan fordítja le az egyes mnemonikokat, címzési módokat, operandusokat stb. a processzor nyelvére, azaz a gépi kódra.

Fontos megjegyezni, hogy a 80386-ostól kezdve az utasítások maximális hossza 15 bájt lehet, az ennél hosszabb kódok 6-os (#UD) kivételt váltanak ki. Ennek megdöntésére momentán az egyetlen módszer az, hogy több azonos kategóriájú prefixet alkalmazunk az adott utasítás előtt.

18.1 A gépi kódú alak felépítése

A gépi kódú utasítások több részből állnak, és ezek a részek meghatározott sorrendben következnek egymás után:

- **prefixek**
- **műveleti kód (operation code–opcode)**
- **címzési mód/operandus info paraméter (addressing mode/operand info specifier)**
- **abszolút offszet (absolute offset) vagy eltolás (displacement)**
- **közvetlen értékek**

Mindegyik gépi kódú utasítást vagy egy prefixszel, vagy egy műveleti kóddal kell kezdeni.

Prefixből több is lehet, de mindegyik prefixnek különböző csoportba kell tartoznia. Így pl. 2 szegmensfelülbíráló prefix vagy 2 címhossz prefix megadása hibát fog okozni, hatásuk pedig definiálatlan lesz.

Műveleti kódnak kötelezően szerepelnie kell, hiszen másképpen honnan tudná szegény processzor, hogy mit is kell csinálnia. Szintén triviális, hogy egyetlen utasításban csak 1 műveleti kód szerepelhet. Egyetlen kivételt a 3DNow! utasítások képzéséhez használt 0Fh 0Fh prefix képez, ami után már nem szabad műveleti kód bájtot írni.

A címzési mód/operandus info paraméter és a közvetlen értékek csak akkor következhetnek, ha azokat az adott műveleti kód megköveteli (igényli). Az abszolút offszet ill. eltolás jelenléte a használt címzési módtól függ.

Ha szükség van címzési mód/operandus info paraméterre, akkor annak bájtja(i) a műveleti kód után fog(nak) következni. Ha a címzési módhoz abszolút offszet/eltolás is kell, az mindenképpen a címzési mód/operandus info paraméter bájtja (i) után helyezkedik el. Végül az esetleges közvetlen adatbájtok zárják a sort.

Az egyes részek maximális hossza szintén korlátozva van:

- **prefixek: 5 bájt**
- **műveleti kód: 1 bájt**

- **címzési mód/operandus info: 2 bájt**
- **abszolút offszet vagy eltolás: 4 bájt**
- **közvetlen értékek: 4 bájt**

Ha ezeket összeadjuk, akkor 16-ot kapunk. Az eredmény azonban nem mond ellent az előbb említett 15 bájtos korlátnak, mivel egyszerre nem lehet mindegyik tag maximális hosszú, azaz nincs olyan gépi kódú utasítás, ami áthágná ezt a korlátot.

Bár a műveleti kód hosszának 1 bájtot írtunk, a prefixek miatt 2 és 3 bájtra is megnőhet ennek hossza.

18.2 Prefixek

A használható prefixek listáját és azok gépi kódú alakját már korábban közöltük. Ezek a prefixek 5 kategóriába (csoportba, osztályba) sorolhatók funkció és használhatóság szerint:

- **operandushossz prefix (66h)**
- **címhossz prefix (67h)**
- **szegmensfelülbíráló prefixek (26h, 2Eh, 36h, 3Eh, 64h, 65h)**
- **buszlezáró prefix (0F0h) és sztringutasítást ismétlő prefixek (0F2h, 0F3h)**
- **műveleti kód/utasítás prefixek (0Fh, 0Fh 0Fh, 0F3h)**

Operandushossz és címhossz prefixet elég sok utasítás előtt megadhatunk, gyakorlatilag minden olyan utasítás esetén, ami implicit vagy explicit regiszter- vagy memóriaoperandust használ. Az MMX és SSE utasítások esetén az operandushossz prefix fenntartott, a 3DNow! utasítások viszont figyelmen kívül hagyják ezt.

Szegmensfelülbíráló prefixet csak olyan utasítás elé írhatunk, aminek egyik operandusa memóriahivatkozás. Ha az alapértelmezett szegmenst kiválasztó prefixet kiírjuk (pl. az ADD AX,[BX] utasítás esetén a DS: prefixet), az nem számít hibának, de feleslegesen elfoglal egy bájtot.

A buszlezáró prefixet a 80386-os processzortól kezdve csak bizonyos utasításokkal használhatjuk, és azokkal is csak akkor, ha az egyik operandus memóriahivatkozás. Ha ezt nem tartjuk be, 6-os (#UD) kivétel lesz a jutalom.

A sztringutasítást ismétlő prefixeket a sztringfeldolgozó utasításokkal való használatra tervezték. Észnélküli alkalmazásuk általában 6-os kivételhez vezet, de az is előfordulhat, hogy a processzor más utasításnak fogja értelmezni a kódot. A 3DNow! utasítások figyelmen kívül hagyják őket. Az MMX és SSE utasítások esetén viszont ezek a prefixek fenntartottnak számítanak, ami az Intel dokumentációja szerint annyit jelent, hogy elképzelhető, hogy egy ilyen fenntartott kombináció a mostani (értsd: Intel Pentium III és AMD K6-III) processzorokon érvénytelen, de esetleg a jövőbeni processzorok új utasítások kódolásához fogják ezt felhasználni. Az előrefelé kompatibilitás miatt tehát csak a sztringutasítások előtt használjuk ezeket a prefixeket!

Bár azt mondtuk korábban, hogy a REPE/REPZ és REPNE/REPNZ mnemonikokat a CMPS és SCAS utasításokkal, míg a REP mnemonikot a fennmaradó INS, LODS, MOVS, OUTS és STOS utasítások előtt illik alkalmazni, azért nem ennyire szigorú a processzor. Először is a REP és a REPE/REPZ mnemonikokhoz ugyanaz a kód tartozik (0F3h), és a processzor fogja eldönteni, az adott műveleti kód esetén hogy kell a prefixet értelmezni (tehát figyelembe kell-e vennie a ZF flag állását). Másrészt pedig a REPNE/REPNZ prefix (kódja 0F2h) az utóbbi 5 utasítás esetén a REP-pel megegyező módon

viselkedik, azaz figyelmen kívül hagyja ZF tartalmát. Egyszóval, bármely prefixet megadhatjuk az említett 7 utasítás előtt.

Látható, hogy a buszlezáró prefix és a sztringutasítást ismétlő prefixek egy csoportba kerültek, így egyszerre való használatuk eleve kizárt.

Az utolsó kategória kicsit kilóg a sorból, mivel ezek inkább "álprefixek". Erre utal eltérő használatuk is.

A 0Fh bájt hajdanán még POP CS utasításként futott (a 8086-os és 8088-as processzorokon), azután egy időre illegalitásba vonult (80186-os és 80188-as procik), majd ünnepélyesen megválasztották prefixnek. Használata feltételezi, hogy a programot futtató számítógépben legalább egy 80286-os processzor van. Ha ez teljesül, még akkor sem biztos, hogy sikeresen megbirkózik a processzor a 0Fh-műveleti kód kombinációval. A prefix azt jelzi, hogy az utasítás műveleti kódját a 0Fh érték és az utána következő bájt adja meg, tehát 2 bájtos lesz az opcode. Az ilyen műveleti kódok második bájtját *kiterjesztett műveleti kódnak* (extended operation code) hívjuk.

A 0Fh 0Fh bájtsorozat még egzotikusabb. A 8086-os/8088-as processzorokon két POP CS -nek felel meg, a 80186-os/80188-as procikon pedig dupla 6-os (#UD) kivételt okoz. A 80286-ostól kezdve szinte mindenhol fenntartott kombinációnak számít, és ugyancsak 6-os kivétel a végrehajtás eredménye. Az AMD K6-2 processzortól kezdve és általában minden olyan processzoron, ami támogatja az AMD 3DNow! technológiát, ez a két bájt a 3DNow! utasítások műveleti kódjának egy darabjaként szolgál. Ebben az esetben a prefix azt jelzi, hogy az esetleges címzési mód/operandus info paraméter után egy bájt méretű közvetlen érték fog következni. Ez utóbbi bájtot *szuffixnak* (suffix) hívják. Az utasítás műveleti kódját a 0Fh 0Fh bájtok és a szuffix együtt alkotják, tehát 3 bájtos lesz a műveleti

kód. Fontos, hogy a 0Fh 0Fh "prefix" után már nem szabad semmilyen hagyományos (1 bájtos) vagy kiterjesztett műveleti kódot írni, csak szuffixot.

Az 0F3h bájt a kezdetektől fogva REP/REPE/REPZ prefixként szolgál. Azt említettük, hogy csak a sztringutasítások előtt működik ismétlő prefixként, egyéb utasítás előtt működése definiálatlan. Az Intel SSE technológiájában néhány utasítás (a pakolt-skalár utasításpárok skalár tagjai esetén) műveleti kódját ez a bájt előzi meg, tehát tekinthető akár a műveleti kód részének is.

A 0Fh és 0Fh 0Fh prefixek közül legfeljebb az egyik fordulhat elő egy adott utasításban, míg a 0F3h SSE-prefix a 0Fh kóddal együtt található meg.

A 0Fh és 0Fh 0Fh műveleti kód prefixeknek a prefixek sorában kötelezően az utolsó helyen kell állniuk. Továbbá, a 0Fh prefixet kötelezően egy kiterjesztett műveleti kódnak kell követnie.

Mivel az utasítás/műveleti kód "prefixek" a műveleti kód képzésében játszanak szerepet, leírásuk a következő, "Műveleti kód" c. fejezetben is megtalálható. Azt azért fontos megjegyezni, hogy sem az Intel, sem az AMD nem nevezi ezeket a kódokat prefixnek, mi is csak azért hívjuk ennek őket, hogy világosabbak legyenek az összefüggések.

Akárhogyan is választjuk ki a prefixeket az első 4 kategóriából, azokat tetszőleges sorrendben odaírhatjuk a műveleti kód (vagy a 0Fh ill. 0Fh 0Fh "prefixek") elé, a processzor helyesen fogja őket értelmezni.

A maximális 5 bájtos hossz elérhető, tehát van olyan utasítás, amelyik mindegyik kategóriából használ egy-egy prefixet.

18.3 Műveleti kód

Azt írtuk nemrégen, hogy a műveleti kód hossza 1 bájt. Ez azonban nem teljesen így van. Vannak ugyanis 1 bájtos, 2 bájtos és 3 bájtos műveleti kóddal rendelkező utasítások. Sőt, ha egészen korrektek akarunk lenni, akkor további 2 kategória is létezik, ezek a "másfél bájtos" és "két és fél bájtos" utasítások.

A 8086-os/8088-as és 80186-os/80188-as processzorok legelemibb utasításai egyetlen bájtból álló műveleti kódot használnak. Ebbe a csoportba a nulla-, három- és négyoperandusú utasítások, továbbá az egy- és kétoperandusúak egy része tartozik. Az egyoperandusú utasítások esetén az operandus a következő lehet: közvetlen érték (imm), relatív cím (rel address), teljes távoli pointer (sel:offs far pointer), 16 vagy 32 bites általános célú regiszter (reg16/32), szegmensregiszter (sreg). Azok a kétoperandusú utasítások esnek ebbe a csoportba, amelyek operandusai a következőek: általános célú regiszter és általános célú regiszter/memóriahivatkozás (reg és reg/mem), szegmensregiszter és általános regiszter/memóriahivatkozás (sreg és reg/mem), speciális regiszter és 32 bites általános célú regiszter (CRx/DRx/TRx és reg32), általános célú regiszter és közvetlen érték (reg és imm), valamint dupla közvetlen érték (imm és imm). Az IN és OUT utasítások, valamint az összes sztringkezelő utasítás szintén 1 bájtos műveleti kódot használ.

Az 1 bájtos és 2 bájtos kategória között átmenetet képeznek a "másfél" bájtos utasítások. Ezt úgy kell érteni, hogy az adott utasítás műveleti kódja 1 bájt, ami után mindig a címzési mód/operandus info paraméter áll. Ez utóbbinak a középső, 'Reg' jelzésű 3 bites mezője viszont nem egy regisztert határoz meg, hanem a műveleti kódnak egy darabját, egyfajta indexet képez. Ez gyakorlatilag annyit jelent, hogy egyetlen 1

bájtos műveleti kód alatt összesen 8 különböző utasítást érhetünk el. Ebbe a csoportba tartoznak az általános célú regisztert/memóriahivatkozást (reg/mem) használó egyoperandusú, a közvetlen értéket és általános célú regisztert/memóriahivatkozást (imm és reg/mem) használó kétoperandusú utasítások, továbbá a RCL, RCR, ROL, ROR, SAL, SAR, SHL és SHR utasítások.

Az előbb említett két kategória utasításai (no meg a prefixek) az egész 1 bájtos tartományt lefedik, azaz 00h-tól 0FFh-ig mindenhol van legalább 1 utasítás vagy prefix. A 80286-oson bevezetett utasítások képzéséhez ezért azt találta ki az Intel, hogy egy korábban használható utasítás kódját ezentúl az új utasítások képzéséhez használja fel. A választás a POP CS utasításra esett, mivel egyrészt ez "tiszta" 1 bájtos kódú utasítás (tehát nincs címzési mód/op. info paraméter, sem 'Reg' mező), másrészt úgyis nem túl szabályos a működése (hiszen a feltétel nélküli vezérlésátadás ezen formájában csak CS változna meg). Ezért a 80286-ostól kezdve a 0Fh bájt egyfajta "prefixként" viselkedik. Nem teljes értékű prefix, mivel nem mindegy, hol helyezkedik el, és nem írható oda egyetlen régi utasítás elé sem. Ha a processzor ezzel a bájttal találkozik, akkor a 0Fh után következő bájtot beolvassa. Ezután ez utóbbi bájtot próbálja utasításként értelmezni, de úgy, hogy a régi utasításoktól eltérő táblázatot használ az utasítás meghatározására. Ezzel azt érték el, hogy a régi utasítások és a prefixek a 00h-0Eh és 10h-0FFh tartományokban helyezkednek el, míg az új utasítások képzésére a 0Fh 00h és 0Fh 0FFh közötti bájtok szolgálnak. Összesen tehát 256 db. új utasítás kódolható le ezzel a módszerrel. Mivel a 0Fh bájt végül is kiterjeszti a műveleti kódok tartományát, *műveleti kód prefixnek* vagy *utasítás prefixnek* is nevezik. A 0Fh után álló bájt határozza meg ténylegesen az utasítást, ezért ennek neve *kiterjesztett műveleti kód* (extended operation code/opcode). A 0Fh értéket az őt

követő bájttal együtt tekintve kapjuk meg az új utasítás 2 bájtos műveleti kódját.

A "másfél bájtos" kategóriához hasonlóan a 2 bájtos (tehát 0Fh kezdetű) műveleti kódú utasítások kódolása során is megszokott dolog, hogy a tartomány jobb kihasználása érdekében a címzési mód/operandus info bájt 'Reg' mezője hozzájárul az utasítás meghatározásához. Ezért hívhatjuk ezt a csapatot "két és fél bájtos" utasításoknak is.

Mivel a kiterjesztett műveleti kódok tartománya is eléggé tele volt, az AMD vezetői úgy döntöttek, hogy az FEMMS és PREFETCH/PREFETCHW utasítások kivételével az összes többi 3DNow! utasítást egyetlen közös kiterjesztett műveleti kóddal fogják jelölni. A könnyű megjegyezhetőség kedvéért (vagy más miatt?) a választás a 0Fh 0Fh bájtkombinációra esett. Ezt a kódot az AMD K6-2 megjelenéséig egyik gyártó sem használta még fel, a célra így pont megfelelt. Ha egy 3DNow!-t támogató processzor ezzel a két bájttal találkozik, akkor szépen beolvassa az utánuk kötelezően álló címzési mód/operandus info bájtot, valamint ha szükséges, az S-I-B és abszolút offszet vagy eltolás bájtokat is. Ezek után egy bájt méretű közvetlen értéknek kell következnie. Ennek a bájtnak hasonló a szerepe, mint a 'Reg' mezőnek a "másfeles" és "két és feles" kategóriákban. A *szuffixnak* vagy *utótagnak* (suffix) nevezett bájtot ugyanis ismét műveleti kódként fogja értelmezni a processzor, persze az 1 bájtos és 2 bájtos utasításoktól eltérő táblázatot használva ehhez. Bár a 0Fh 0Fh bájtok nem alkotnak valódi prefixet, elhelyezkedésük és a 0Fh bájthoz hasonló funkciójuk miatt hívhatjuk mondjuk *3DNow! utasítás prefixnek* őket. Működése két lényeges ponton tér el a 0Fh "prefix" működésétől, mégpedig abban, hogy egyrészt kötelezően szerepelnie kell címzési mód/operandus info paraméternek, másrészt a műveleti kód 3. bájtja az utasítás kódjának végén

helyezkedik el. Mondani sem kell, hogy egyetlen más utasítás előtt sem alkalmazható prefixként ez a két bájt.

Az előbb említett megoldást az Intel is átvette a CMPPS és CMPSS SSE utasítások kódolásakor. A szuffix bájt itt az összehasonlítás feltételét adja meg.

Szintén az SSE utasításokhoz kapcsolódik, hogy számos utasítás rendhagyó módon kerül lekódolásra. Jó néhány pakolt-skalár SSE utasításpár van, melyek ugyanazt a műveletet végzik el, de az egyik utasítás pakolt lebegőpontos típusú, míg a másik skalár lebegőpontos operandusokkal dolgozik. Ilyenek pl. az aritmetikai utasítások (ADDPS-ADDSS, SUBPS-SUBSS stb.), vagy a CMPPS-CMPSS páros is. A párok mindkét tagja ugyanazt a kiterjesztett műveleti kódot használja, de a skalár utasítást egy implicit, kötelező REP/REPE/REPZ prefix (0F3h) előzi meg. Mivel a sztringutasítást ismétlő prefixeket nem használhatnánk ezekkel az utasításokkal, ebben az esetben a 0F3h bájtnak más jelentése van, nevezetesen a skalár utasítást különbözteti meg pakolt párjától. Bár ennek a valódi prefixnek már van neve, az eltérő funkció miatt *hívhatjuk SSE utasítás prefixnek* is.

Láttuk, hogy milyen módon és hány bájton tárolódik a műveleti kód a különféle utasítások esetén. Most az elsődleges műveleti kód bájt finomszerkezetét (szép szóval mikrostruktúráját:) nézzük meg. *Elsődleges műveleti kódon* (primary opcode) 1 és "másfél" bájtos esetben magát a műveleti kódot, különben pedig a kiterjesztett műveleti kód bájtot (tehát a 0Fh után álló értéket) értjük.

Sok utasítást többféle operandussal is lehet használni. Nézzük például az ADD utasítást! Az operandusok típusai lehetnek: két általános regiszter, általános regiszter és memória-hivatkozás, közvetlen érték és általános regiszter/memó-

riahivatkozás. Mindkét operandus mérete lehet 8, 16 vagy 32 bájt. Most tekintsük az ADC, SUB, SBB utasításokat. Az említett 4 utasításnak (és még másoknak is) közös jellemzője, hogy azonos feltételeket szabnak az operandusok típusát ill. méretét illetően. Hogy valami logikát vigyenek a dologba, az elsődleges műveleti kódban néhány bit szélességű mezőket vezettek be. Az azonos jelölésű mezők funkciója azonos, bárhol is helyezkedjenek el. A következetesség céljából ráadásul sok mező helye rögzített, tehát különböző utasítások műveleti kódjában ugyanazon a bitpozíción található. A példákhoz visszatérve, mind a 4 utasításra igaz, hogy az azonos fajta operandusokhoz tartozó műveleti kódjaik felépítése teljesen megegyezik. (Sőt, kicsit előreugorva, mind a 4 utasításnak van "másfél bájtos" kategóriájú műveleti kódja is, és ez a közös 80h/81h/82h/83h *r+n alakhoz vezet. Ebben az esetben tehát nemcsak a felépítés, de maga az elsődleges műveleti kód is közös.)

Persze nem minden elsődleges műveleti kód tartalmaz ilyen mezőket, ez függ az operandusok számától, típusától is. Például a CLI, HLT és PUSHF utasítások mindegyikének egyetlen műveleti kódja van, és egyikben sincsenek mezők.

A mezők hossza 1, 2, 3 és 4 bit lehet. Az alábbi táblázat felsorolja az egyes mezőket, azok szélességét, jelölését és nevét megadva:

19 PROCESSZOROK DETEKTÁLÁSA

(Extra…)

EFlags
POP CS
MOV CS,??
shift/rotate
EDX
BIOS INT 15h, AX=0C910h
CPUID
("GenuineIntel","AuthenticAMD","CyrixInstead","RiseRiseRise","CentaurHauls","UMC UMC UMC ","NexGenDriven")

20 VÉDETT MÓDÚ ASSEMBLY

(DG)

21 FELHASZNÁLT IRODALOM

21.1 Nyomtatott segédeszközök

21.1.1 Könyvek

1)Dr. Kovács Magda: 32 bites mikroprocesszorok 80386/ 80486 I./II., LSI, Budapest

21.1.2 Folyóiratok

1)CHIP Számítógép Magazin, Vogel Publishing Kft., Budapest, 1998 októberi és decemberi, 1999 márciusi, áprilisi, májusi és szeptemberi számok

21.2 Elektronikus források

21.2.1 Segédprogramok

1) Tech Help! 4.0 written by Dan Rollins, Copyright © 1985, 1990 Flambeaux Software, Inc.
2) HelpPC 2.10, Copyright © 1991 David Jurgens
3) Ralf Brown's Interrupt List Release 52, Copyright © 1989, 1996 Ralf Brown
4) Expert Help Hypertext System v1.09, Copyright © 1990, 1992 SofSolutions
5) Hacker's view (HIEW) v6.10, Copyright © 1991, 1999 SEN, Kemerovo

21.2.2 Norton Guide adatbázisok

1) The Interrupts and Ports database v1.01, Copyright © 1988 Nobody
2) Borland's Turbo Assembler v4.0 Ideal mode syntax, Copyright © 1995 Morten Elling
3) The Assembly Language database, Copyright © 1987 Peter Norton Computing, Inc.
4) The Programmers Reference v0.02b, Copyright © P.H.Rankin Hansen (Ping)
5)Intel iAPx86 instruction set, 1996

21.2.3 Szöveges formátumú dokumentumok

1) **Intel 80386 Programmer's Reference Manual 1986, Copyright © 1987 Intel Corporation**

21.2.4 PDF (Adobe Portable Document Format) kézikönyvek

1) **Intel Architecture Software Developer's Manual Volume 1, 2 and 3 (Order Number 243190, 243191 and 243192), Copyright © 1999 Intel Corporation**
2) **AMD K6®-III Processor Data Sheet (Order Number 21918), Copyright © 1999 Advanced Micro Devices, Inc.**
3) **AMD K6®-2 Processor Data Sheet (Order Number 21850d), Copyright © 1998 Advanced Micro Devices, Inc.**
4) **AMD K6® Processor Multimedia Technology (Order Number 20726c), Copyright © 1997 Advanced Micro Devices, Inc.**
5) **3DNow!™ Technology Manual (Order Number 21928d), Copyright © 1998 Advanced Micro Devices, Inc.**
6) **SYSCALL and SYSRET Instruction Specification (Order Number 21086c), Copyright © 1998 Advanced Micro Devices, Inc.**
7) **AMD Processor Recognition Application Note (Order Number 20734i), Copyright © 1998 Advanced Micro Devices, Inc.**

TÁRGYMUTATÓ