Oktatási segédanyag

Assembly programozás

mérnökinformatikus BSc tantárgyhoz

Dr. Varga Imre

Informatikai Kar
Debreceni Egyetem

2024

Utolsó módosítás: 2024.09.02

1. Bevezetés

Az assembly a programozás egy hardverközeli formája. A magas-szintű programozási nyelvek (pl. C, Java, Python) alatt helyezkedik el, de sokkal olvashatóbb, mint a bináris gépi kód. Az assembly függ az adott **utasításkészlet architektúrá**tól (ISA: Instruction-Set Architecture). Ez a jegyzet csak az **x86-64** architektúrára korlátozódik, amely szinte minden Intel és AMD processzor alapja. Ennek az assemblynek két nyelvjárása létezik: az Intel és az AT&T szintaxis. A szerző most csak az **Intel szintaxis**ra fókuszál. Az assembly programok megértéséhez hardveres ismeretekre is szükség van. Az x86-64 ISA ismerete nélkülözhetetlen (de itt csak részben kerül bemutatásra).

Egy assembly forráskód Linux operációs rendszer alatt egy "`.s`" kiterjesztésű egyszerű szöveges fájl. Az egyik utasítást el kell látni egy main címkével, ez lesz a futtatás belépési pontja, itt kezdődik majd a végrehajtás. A `gcc` paranccsal lehet lefordítani az assembly kódot, hogy egy futtatható állományt kaphassunk. Például a `test.s` assembly forráskód lefordítható egy `run` nevű futtatható állományra az alábbi parancs terminálbeli kiadásával:

```
gcc test.s -o run
```

Néhány esetben egy `-no-pie` kapcsolóra is szükség lehet fordításnál, ahhoz, hogy egyszerű pozíciófüggetlen futtatható kódot kapjunk. Sikeres fordítás esetén a

```
./run
```

terminál paranccsal futtatható a program. (A fenti példában a forrás és futtatható állományok ugyanabban a mappában vannak.) A program (azaz a `main` függvény) visszatérési értéke egy környezeti változóba kerül, amit a futás befejezése után közvetlenül kiadott

```
echo $?
```

parancs segítségével tekinthetünk meg. Linux alatt ez a visszatérési érték unsigned char típusú adatként értelmezendő, azaz csak a visszatérési érték utolsó bájtja fontos nem-negatív egész értékként (0-255). A megszokott konvenciók szerint a `0` érték jelzi a sikeres programfutást (többnyire).

2. Regiszterek és a memória

Regiszter készlet

A CPU-ban minden adat és utasítás regiszterekben van tárolva. Ezek kis méretű szekvenciális digitális hálózatok a processzorban, amelyek csak néhány bájt adatot tudnak tárolni, de nagyon gyorsan hozzáférhető módon. A programozó számos különböző méretű **általános célú fixpontos regiszter**t használhat.

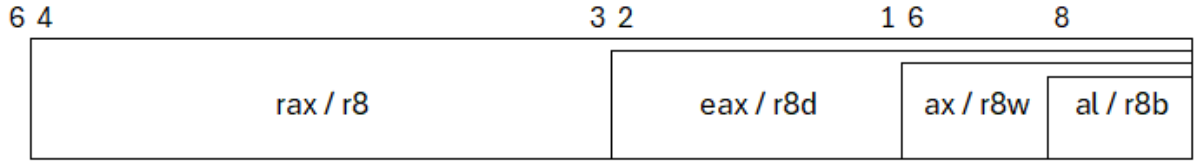
64-bites (quad word, négyszeres szóhosszúságú) regiszterek:
`rax, rbx, rcx, rdx, rbp, rsp, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, r15`

32-bites (double word, dupla szóhosszúságú) regiszterek:
`eax, ebx, ecx, edx, ebp, esp, edi, esi, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d`

16-bites (word, szóhosszúságú) regiszterek:
`ax, bx, cx, dx, bp, sp, di, si, r8w, r9w, r10w, r11w, r12w, r13w, r14w, r15w`

8-bites (byte) regiszterek:
`al, ah, bl, bh, cl, ch, dl, dh, bpl, spl, dil, sil, r8b, r9b, r10b, r11b, r12b, r13b, r14b, r15b`

A kisebb regiszterek mindig a nagyobb regiszterek részei (egyfajta alias-ok a teljes regiszter töredékeire vonatkozóan) ahogy ezt az alábbi ábra is illusztrálja.



Például az `rbx` regiszter kis helyiértékű felének a neve `ebx`, aminek az alsó fele a `bx`, ami pedig a `bh` (felső) és a `bl` (alsó) bájtból áll.

Ha egy 32-bites értéket egy 64-bites regiszterben tárolunk, akkor magasabb helyiértékű 32 bit csupa `0` bit lesz, minden más esetben, ha egy regiszter egy kisebb részlete felülíródik, akkor a magasabb helyiértékű rész tartalma változatlan marad.

Az összes fent említett regiszter fixpontos adatábrázolást használ, szóval csak egész jellegű értékeket (`signed/unsigned int/short/long/char`) tárolhatunk bennük. A lebegőpontos értékek (például valós számok) tárolására más regiszterek szolgálnak (lásd az x87 architektúrát vagy az MMX vektor regisztereket, amelyek nem kerülnek bemutatásra ebben az anyagban).

Az általános célú regisztereken kívül van néhány speciális célt (is) szolgáló regiszter. A `rip` (instruction pointer) regiszter játssza a **program számláló** szerepét, vagyis ez tárolja mindig (a RAM kódszegmens nevű részében tárolt) következőként végrehajtandó gépi kódú utasítás 64-bites memóriacímét. Ez nélkülözhetetlen a normál vezérléshez (control flow). A normál soros végrehajtás során a regiszter értéke növekszik az aktuális utastás méretével.

Az **állapot regiszter** neve ebben az architektúrában `rflags`. Ennek az egyes bitjei különböző processzor állapotok/beállítások meglétét vagy hiányát jelzik. Négy bitnek ezek közül nagy jelentősége van még egyszerű kódok esetén is, ezek a következők:

`CF` (carry flag, átvitel bit): értéke 1, ha a legutóbbi aritmetikai vagy bitenkénti művelet során az eredmény több biten ábrázolható, mint amennyi rendelkezésre áll, különben (azaz, ha az operandusok nem tartalmaznak kevesebb bitet, mint az eredmény) az értéke 0.

`OF` (overflow flag, túlcsordulás bit): értéke 1, ha a aritmetikai művelet eredménye helytelen előjeles fixpontos értelmezés esetén, különben 0-t tárol. Egyfajta váratlan előjelváltás előfordulását jelzi.

`ZF` (zero flag, nulla bit): értéke 1 (igaz), ha a legutóbbi művelet eredménye 0, különben a 0 bitet tárolja.

`SF` (sign flag, előjel bit): értéke 1, ha a legutóbbi művelet eredménye előjeles fixpontos számábrázolást alkalmazva negatív érték. Gyakorlatilag ez az eredmény legnagyobb helyiértékű bit másolata. (Előjel nélküli számábrázolás esetén értékét nem szokás figyelembe venni.)

Ezek a bitek az összehasonlítás művelet során is változnak, szóval nélkülözhetetlenek a feltétel kiértékelés során.

Vannak még úgynevezett szegmens szelektor regiszterek is az x86-64 architektúrában, amelyek manapság már csak az x86 ISA kompatibilitás miatt szükségesek. Itt van a regiszterek listája: `CS` (kód szegmens), `SS` (verem szegmens), `DS` (adat szegmens), `ES`, `FS`, `GS`.

Memória

A változók a RAM-ban vannak eltárolva. Az elhelyezkedésüket **memória cím**ekkel (azaz egyfajta memória cella azonosítókkal) adhatjuk meg. Minden egyes bájt a memóriában saját címmel rendelkezik, de az ezen belüli bitek nem címezhetőek. A címek egy 64-bites előjel nélküli egész számként képzelhetőek el. Amikor a processzor egy adatot ír vagy olvas a memóriában, akkor nem csak a terület kezdő memóriacímét kell megadni, hanem az elérni kívánt adatnak a méretét is (`byte`: 8 bit, `word`: 16 bit, `double word`: 32 bit, `quad word`: 64 bit).

Az egynél több bájtot tartalmazó adatok a memóriában **little-endian bájt sorrendben** vannak tárolva, azaz a lekisebb helyi értékű bájt helyezkedik el a legkisebb memóriacímen a legnagyobb helyi értékű bájt pedig a változó memóriaterületének az utolsó bájtján. (Tehát fordított bájtsorrendet alkalmazunk, de a bitek sorrendje nem változik az egyes bájtokon belül. Például a tízes számrendszerbeli 27 970 800 érték (ami kettes számrendszerbeli számként így néz ki: 1 10101010 11001100 11110000) a következő 32 bites bitsorozatként lesz eltárolva: 11110000 11001100 10101010 00000001.

A RAM egy része speciálisan van kezelve. Ez az úgynevezett **verem**, amely LIFO működést valósít meg speciális assembly utasítások segítségével (bár a tartalom továbbra is közvetlenül is címezhető). A verem legfontosabb része a teteje. Az `rsp` regiszter tartalmazza a verem tetejének a címét (azaz a legutóbb verembe helyezett érték kezdőcímét). Az `rsp` értéke automatikusan csökken a **push** művelet során betett érték méretével, illetve automatikusan csökken a **pop** művelet során kivett érték méretével. Tehát a veremtartalom a kisebb memóriacímek felé terjeszkedik.

3. Assembly utasítások általában

Az assembly forráskód minden sora maximum egy utasítást tartalmaz. Minden utasítás 4 komponensből állhat: címke, mnemonic, oprandus(ok), megjegyzés.

Címke

Ez egyfajta azonosítója az adott utasításnak. Rendszerint a programozónak hivatkoznia kell egy utasításban egy másikra, ezt az utasítás egyedi címkéje segítségével teheti meg. A címke egy opcionális utasítás komponens, szóval néha kihagyható, de ha szükséges, akkor az utasítás sor elején kell elhelyezni.

Formailag a címke egy alfanumerikus karaktereket (az Angol ABC betűit és számjegy karaktereket valamint aláhúzás jelet (‘_’) vagy pont karaktert (‘.’) tartalmazó karaktersorozat, amelyet kettősponttal (‘:’) fejezünk be. Az assembly nyelv nem case-sensitive, azaz a kis és nagybetűk nem számítanak különbözőnek. A ponttal kezdődő címkék lokális neveket/azonosítókat jelentenek, amelyek csak az adott forrásfájlon belül érvényesek. A címke egy olvasható szöveg a programozó számára, de a háttérben a rendszer számára vagy egy utasítás memóriacímét jelenti (ami a vezérlésátadásnál lehet hasznos) vagy egy statikusan foglalt változó memóriacímét jelenti. Egynél több címke is hivatkozhat ugyanarra a memória címre.

Mnemonic

Ez az assembly utasítás/művelet neve, amely többnyire egyfajta rövidítés, ami egy konkrét műveletre hivatkozik, például az aritmetikai jobbra tolás (shift) művelet mnemonic-ja a ‘`sar`’. Minden utasítássornak tartalmaznia kell egy mnemonic-ot. Az x86-64 ISA mnemonic-ok százait biztosítja, de ez a jegyzet csak a leggyakrabban alkalmazottakat fedi le (lásd 4. Fejezet). A pont (‘`.`’) karakterrel kezdődő mnemonic-ok compiler direktívák, azaz a fordítás menetét befolyásoló utasítások.

Operandus

Ezek olyan értékek, amelyeken az adott műveletet végre kell hajtani. A mnemonic és az operandusok között szóköz vagy tabulátor karakter szükséges. Egy utasításnak lehet 0, 1, 2 vagy 3 operandusa. Az x86-64 architektúrában a három operandus nagyon ritka. A nulla operandus azt jelenti, hogy a művelet végrehajtható önmagában bármiféle további adat nélkül. Néha az operandus explicit megadása nem szükséges (habár maga az operandus szükséges), mert ilyen esetben a szükséges érték egy előre meghatározott helyen kell legyen. Ha egynél több operandus van, akkor azokat vesszővel (‘`,`’) kell elválasztani. Az Intel szintaxis szerint az első operandus a **cél operandus**, ahová az eredmény eltárolásra kerül felülírva az eredeti operandus értéket.

Az operandus lehet konstans, regiszter vagy memória tartalom. Számkonstansok (literálok) megadhatóak tízes számrendszerben (pl. `196`) vagy a “`0x`” előtag használatával tizenhatos számrendszerben is (pl. `0xC4`). Mind pozitív, mind negatív értékek is használhatóak assembly kódban. (8 bites operandus esetén az `11111111` bitsorozat jelenthet `-1` vagy `255` értéket is.) A művelet maga fogja meghatározni, hogy melyik értelmezést (előjeles vagy előjel nélküli) kell alkalmazni a számolás során. A cél operandus természetesen nem lehet konstans csak regiszter vagy memóriatartalom.

Csak az egyik operandus lehet memória tartalom technikai korlátozások miatt. Ha az operandus egy memória tartalom, akkor annak a **memóriacím**ét és a **reprezentációs hossz**át is meg kell adni. Az érték kezdőcímétszögletes zárójelek (“`[ ]`”) közzé kell írni az alkalmazott **címzési mód**nak megfelelően, míg a hossz (azaz a kezdőcímen kezdődő folytonos memóriaterületen lévő bájtok száma) a szögletes zárójel előtt kerül megadásra az alábbi 4 kifejezés egyikével: `BYTE PTR`, `WORD PTR`, `DWORD PTR`, `QWORD PTR` (rendre 1, 2, 4 és 8 bájt). A címe egy elég komplex formában adható meg, amely a következő általános alakban adható meg: `[SS: B + I * SF + C]`, ahol `SS` egy szegmens szelektor regiszter, `B` egy alap címet tartalmazó regiszter, `I` játssza az index regiszter szerepét (azaz megadja hány

egységnyi távolságra az alap címtől), `SF` egy skála faktor (adategység méret) és `C` egy konstans eltolási érték. A legtöbb komponense ennek a címszámításnak kihagyható. Ha `I` nincs jelen, akkor az `SF` sem szerepelhet. Az `SF` lehet konstans 1, 2, 4, vagy 8. A `C` megadható egy konstansként vagy egy memóriafoglaló utasítás címkéje segítségével is. A címszámolás megértéséhez egy példa kerül bemutatásra. A programozó egy egész típusú értékeket tartalmazó tömb 5. elemére szeretne hivatkozni. Magas szintű programozási nyelvek (például a C) esetén ez így írható: `MyArray[4]`. Tegyük fel, hogy a tömb indexe előzetesen el lett tárolva az `rcx` regiszterben és a tömb eleje 48 bájttal kisebb címen van, mint az `rbp` regiszterben tárolt cím, valamint, hogy az int típus 4 bájton van reprezentálva. Egy assembly utasításban az operandust ekkor `DWORD PTR [rbp+rcx*4-48]` formában adhatjuk meg.

Egyik vagy mindkét operandus megadható regiszterként. A legtöbb esetben egy utasítás (regiszter vagy memória) operandusai azonos méretűek kell legyenek. Ha a **forrás operandus** (azaz Intel szintaxis esetén a második operandus) egy konstans, akkor ennek tárolási hosszát a céloperandus mérete határozza meg.

Megjegyzés

Kettőskereszt (‘#’) karaktertől az adott sor végéig tartó forráskód tartalmat az assembler (assembly fordító) figyelmen kívül hagyja, tehát a programozó egy így sor végi megjegyzéseket /kommenteket használhat. Ezek nagyon hasznosak assemblyben, mert a kód nem annyira olvasmányos, mint egy magas szintű programozási nyelv esetén, sokkal könnyebb elveszni a részletekben.

4. Gyakran használt utasítások részletesen

A leggyakrabban használt fordítási direktívák és végrehajtható utasítások rövid leírásai megtalálhatóak lentebb. Sokkal részletesebb leírásokért tanulmányozd az alábbi dokumentumokat [1, 2].

Direktívák

`.2byte`

Lefoglal 2 bájtot (azaz egy inicializált `static short int` változót). A lefoglalt terület memóriacíme az utasítás elé írt címke segítségével hivatkozható. Egyetlen operandusa van, egy konstans, amely a lefoglalt területen kerül eltárolására, inicializálva a változót.

`.4byte`

Lefoglal 4 bájtot (azaz egy inicializált `static int` változót). A lefoglalt terület memóriacíme az utasítás elé írt címke segítségével hivatkozható. Egyetlen operandusa van, egy konstans, amely a lefoglalt területen kerül eltárolására, inicializálva a változót.

`.bss`

Az inicializálatlan adatszegmens kezdetét jelzi. Nincs operandusa.

`.byte`

Lefoglal 1 bájtot (azaz egy inicializált `static char` változót). A lefoglalt terület memóriacíme az utasítás elé írt címke segítségével hivatkozható. Egyetlen operandusa van, egy konstans, amely a lefoglalt területen kerül eltárolására, inicializálva a változót.

`.comm`

Inicializálatlan memóriaterületet foglal az adat szegmensben. Két operandusa van: Az első a lefoglalt memóriaterület címét jelentő nevesített konstansként viselkedik, míg a második a lefoglalandó terület mérete bájt egységben. Inicializálatlan statikus skalár vagy tömb változók létrehozása.

`.data`

Az adatszegmens kezdetét jelzi. A statikus memóriafoglalások ide történnek. Nincs operandusa.

`.globl`

Egy vagy több (később címkeként használt) szimbolikus név követi, amelyeket globális nevekként lehet használni. Például a `main` egy globális név kell legyen, mivel bárhonnan el kell tudni érni a végrehajtás belépési pontját. Ha több név is fel van sorolva, akkor vessző (‘`,`’) karakterrel kell elválasztani őket. Ez különösen akkor válik fontossá, ha a program több forráskódból áll.

`.intel_syntax`

Többnyire a forráskód első sora, amikor a programozó Intel szintaxist akar használni a kód hátralévő részében. Egy operandusa van: `noprefix`.

`.section`

Ezzel kezdődik a csak olvasható adatszegmens. Egy operandusa van: `.rodata`. Például a sztring konstansok tárolódnak itt.

`.string`

Sztring konstansok tárolása a memóriában. A karakterek számára lefoglalt memóriaterület kezdőcíme a sor elejére írt címkével hivatkozható. Egy operandusa van: egy macskakörmök közé tett karaktersorozat, mint például ez: `"Hello World!\n"`. A lefoglalt bájtok számát a sztring hossza határozza meg beleszámolva a sztringzáró (‘`\0`’) karaktert is (azaz a fenti példában 14 bájt). Ez a helyfoglalás a RAM csak olvasható adatszegmens részébe kell kerüljön, tehát a `.section .rodata` utasítás után.

`.text`

A kódszegmensbe kerülő sorok kezdetét jelzi (ahol a végrehajtható utasítások vannak). Kihagyható, ha a kódban nincs megadva más szegmens (pl. `.data`, `.bss`).

`.zero`

Nulla kezdőértékkel rendelkező statikus memóriafoglalás skalár és tömb változók számára. A lefoglalt memóriaterület kezdőcíme a programsor elején lévő címke segítségével hivatkozható. Egyelten operandusa van, a lefoglalni kívánt terület mérete bájtban megadva. A terület minden bájtjának összes bitje 0 lesz.

Végrehajtható utasítások

`add`

Az operandusok összeadása. Az összeg felülírja a cél operandust (azaz az első operandust Intel szintaxis esetén). Az operandusok és az eredmény értelmezhető előjeles és előjel nélküli fixpontos értékként is. Az `SF`, `ZF`, `CF` és `OF` állapot regiszter bitek változhatnak az eredménynek megfelelően.

`and`

Bitenkénti ÉS művelet az operandusok között. Az operandusok minden egyes bitpozíciója szimultán módon kerül felhasználásra a logikai ÉS művelet igazságtáblájának megfelelően. Az eredmény bitsorozat felülírja a cél operandust. Az `OF` és `CF` bitek törlődnek, az `SF` és `ZF` az eredménynek megfelelően állítódnak be.

`call`

Alprogram hívás. Egy operandusa van (rendszerint egy szimbolikus név), amely egy alprogram első utasításának memóriacímére hivatkozik. Az utasításszámláló (`rip`) aktuális értéke elmentődik a verem tetejére, a veremmutató (`rsp`) értéke csökken 8-cal, majd az `rip` felülíródik az operandust jelentő memóriacímmel. Ennek hatására tehát a következőként végrehajtandó utasítás az alprogram első utasítása lesz.

`cbw`

Bájt (8-bites érték) konverziója szó hosszúságúvá (16-bites) előjeltartó kiterjesztéssel. Nincs explicit operandus. Az `ah` regiszter felülíródik az `al` regiszter legnagyobb helyiértékű bitjének a másolataival. Ez egyfajta `char` → `short` konverzió.

`cdq`

Dupla szó (32-bites érték) konverziója négyszeres szóhosszúságúvá (gyakorlatilag két 32-bites érték kombinációjává) előjeltartó kiterjesztés segítségével. Nincs explicit operandus. Az `edx` regiszter felülíródik az `eax` regiszter legnagyobb helyiértékű bitjének a másolataival, így tehát az eredmény legnagyobb helyiértékű 4 bájtja az `edx` regiszterbe kerül a legkisebb helyiértékű 4 bájt pedig az `eax` regiszterben marad. Ez egyfajta `int` → `long` konverzió.

`cdqe`

Dupla szó (32-bites érték) konverziója négyszeres szóhosszúságúvá (64-bites) előjeltartó kiterjesztés segítségével. Nincs explicit operandus. Az `rax` regiszter legnagyobb helyiértékű 4 bájtja felülíródik az `eax` regiszter legnagyobb helyiértékű bitjének a másolataival. Ez egyfajta `int` → `long` konverzió.

`cmp`

A két operandus értékének összehasonlítása, azaz ellenőrzésre kerül, hogy az értékek megegyeznek-e vagy melyik operandus a nagyobb mindkét felé értelmezés (előjeles és előjel nélküli fixpontos) esetén külön-külön. Gyakorlatilag az első operandus értékéből kivonódik a második operandus értéke, de a különbség eldobásra kerül). Egyik operandus értéke sem változik meg. Az összehasonlítás/kivonás alapján az `SF`, `ZF`, `CF` és `OF` állapot regiszter bitek frissülnek, amit az alábbi táblázat foglal össze.

cmp Op1, Op2		ZF	CF	SF	OF
unsigned	Op1 < Op2	0	1		
signed				1	0
Op1 == Op2		1			
unsigned	Op1 > Op2	0	0		
signed				0	0

Minden feltételes vezérlésátadó utasítás (`j`$_{cc}$) ezeket az állapot regiszter biteket használja (amelyeket tehát többnyire a `cmp` utasítás állít be).

`cwd`

Szó (16-bites érték) konverziója dupla szó hosszúságúvá (32-bites) előjeltartó kiterjesztéssel. Nincs explicit operandus. Az `eax` regiszter legnagyobb helyiértékű 2 bájtja felülíródik az `ax` regiszter legnagyobb helyiértékű bitjének a másolataival. Ez egyfajta `short` → `int` konverzió.

`dec`

Dekrementáció. Egyetlen operandusa van, amelynek az értéke eggyel csökken a végrehajtás során. Az `SF`, `ZF` és `OF` állapot regiszter bitek az eredménynek megfelelően frissülnek.

`div`

Előjel nélküli egész osztás. Egy (!) operandusa van, az osztandó nincs explicit módon megadva. Ha az osztó/operandus 32 bites érték, akkor az osztandó az `edx:eax` regiszterkombináció (azaz egyetlen 64-bites érték, amelynek a legnagyobb helyiértékű 4 bájtja az `edx` regiszterben, az alsó 4 bájtja pedig az `eax` regiszterben található). Az egész osztás elvégzése után a 32-bites hányados (nulla felé csonkolt egész szám) felülírja az `eax` regiszter tartalmát és a 32-bites maradék felülírja az `edx` regisztert. Előjel nélküli értelmezést használ, szóval az értékek nem lehetnek negatívak. Az `SF`, `ZF`, `CF` és `OF` állapot regiszter bitek nem változnak.

`idiv`

Előjeles egész osztás. Egy (!) operandusa van, az osztandó nincs explicit módon megadva. Ha az osztó/operandus 32 bites érték, akkor az osztandó az `edx:eax` regiszterkombináció (azaz egyetlen 64-bites érték, amelynek a legnagyobb helyiértékű 4 bájtja az `edx` regiszterben, az alsó 4 bájtja pedig az `eax` regiszterben található). Az egész osztás elvégzése után a 32-bites hányados (nulla felé csonkolt egész szám) felülírja az `eax` regiszter tartalmát és a 32-bites maradék felülírja az `edx` regisztert. Előjeles értelmezést használ, szóval az értékek lehetnek negatívak. Az `SF`, `ZF`, `CF` és `OF` állapot regiszter bitek nem változnak.

`imul`

Előjeles szorzás. Most csak a 2 operandusú forma kerül elmagyarázásra (habár létezik 1 és 3 operandusú alak is). A szorzat felülírja a cél operandust. A `CF` és `OF` bitek beállításra kerülnek, ha az (esetlegesen túl nagy) eredményt csonkolni kellett ahhoz, hogy beférjen a cél operandusba.

`inc`

Inkrementálás. Egyetlen operandusa van, amelynek az értéke eggyel növekszik a végrehajtás során. Az `SF`, `ZF` és `OF` állapot regiszter bitek az eredménynek megfelelően frissülnek.

`ja`

Ugrás, ha felette van. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `CF` értéke 0 és a `ZF` is 0, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a

következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjel nélküli értelemben nagyobb, mint a második. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnbe` utasítás is.)

`jae`

Ugrás, ha felette van vagy egyenlő. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `CF` értéke 0, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjel nélküli értelemben nagyobb, mint a második vagy a két érték egyenlő. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnb` utasítás is.)

`jb`

Ugrás, ha alatta van. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `CF` értéke 1, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjel nélküli értelemben kisebb, mint a második. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnae` utasítás is.)

`jbe`

Ugrás, ha alatta van vagy ha egyenlő. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `CF` értéke 1 vagy a `ZF` is 1, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjel nélküli értelemben kisebb, mint a második vagy a két érték egyelő. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jna` utasítás is.)

`jc`

Ugrás, ha az átvitel bit be van állítva. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket. Ha a `CF` értéke 1, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy

másik részére ugrik, vagy csak átkerül a következő utasításra. Az állapotregiszter bitek nem változnak a végrehajtás során. (A `jz`, `jo` és `js` feltételes ugró utasítások hasonlóan viselkednek, csak rendre a `ZF`, `OF` és `SF` biteket ellenőrzik.)

`je`

Ugrás, ha egyenlőek. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `ZF` értéke 1, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandus értéke egyenlő második operandus értékével. Az állapotregiszter bitek nem változnak a végrehajtás során.

`jg`

Ugrás, ha nagyobb. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `ZF` értéke 0 és az `OF` ugyanazt a bitet tartalmazza, mint a `SF`, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjeles értelemben nagyobb, mint a második. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnle` utasítás is.)

`jge`

Ugrás, ha nagyobb vagy egyenlő. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha az `OF` ugyanazt a bitet tartalmazza, mint a `SF`, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjeles értelemben nagyobb, mint a második vagy a két érték egyenlő. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnl` utasítás is.)

`jl`

Ugrás, ha kisebb. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha az `OF` nem ugyanazt a bitet tartalmazza, mint a `SF`, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter

tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjeles értelemben kisebb, mint a második. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jnge` utasítás is.)

`jle`

Ugrás, ha kisebb vagy egyenlő. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `ZF` értéke 1 és az `OF` nem ugyanazt a bitet tartalmazza, mint a `SF`, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandusa előjeles értelemben kisebb, mint a második vagy a két érték egyenlő. Az állapotregiszter bitek nem változnak a végrehajtás során. (Hasonlóan viselkedik a `jng` utasítás is.)

`jmp`

Feltétel nélküli ugrás. Átadja a vezérlést a program egy másik pontjára, azaz a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel (ami a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje).

`jnc`

Ugrás, ha az átvitel bit nincs beállítva. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket. Ha a `CF` értéke 0, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. Az állapotregiszter bitek nem változnak a végrehajtás során. (A `jnz`, `jno` és `jns` feltételes ugró utasítások hasonlóan viselkednek, csak rendre a `ZF`, `OF` és `SF` biteket ellenőrzik.)

`jne`

Ugrás, ha nem egyenlőek. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Ez a feltételes utasítás ellenőrzi az állapot regiszter biteket (amelyeket korábban feltehetően egy `cmp` utasítás állított be). Ha a `ZF` értéke 0, akkor a programszámláló (`rip`) felülírásra kerül az operandusként megadott címmel, különben a `rip` értéke növekszik ennek az utasításnak a méretével. Tehát a végrehajtás vagy a program egy másik részére ugrik, vagy csak átkerül a következő utasításra. A feltétel megegyezik azzal az állapot regiszter tartalommal, amelyet a `cmp` utasítás eredményez, ha az első operandus értéke nem egyenlő második operandus értékével. Az állapotregiszter bitek nem változnak a végrehajtás során.

`lea`

Töltsd be a címet utasítás, amely meghatározza a második operandusban (valamely címzési módban) megadott memóriacímet és eltárolja az első operandusként megadott regiszterbe. Formailag a második operandus úgy néz ki, mint egy memória tartalom megadás,

de a tárolási hossz megadására szolgáló kifejezés hiányzik, csak a szögletes zárójelek közti rész van jelen. Az állapotregiszter bitek nem változnak.

`leave`

Az alprogramok végén alkalmazott utasítás. Nincs operandusa. Egyenértékű az alábbi utasítások egymás utáni megadásával: `mov rbp, rsp` és `pop rbp`.

`loop`

Iterációszervező utasítás, ahol az `rcx` regiszter ciklusszámlálóként működik. Egy operandusa van, amely egy memóriacím (a gyakorlatban egy szimbolikus név, amely valahol a programban egy másik utasítás címkéje). Minden egyes alkalommal, amikor a `loop` utasítás végrehajtódik az `rcx` regiszter értéke eggyel csökken, majd megvizsgálásra kerül, hogy elérte-e már a nullát. Ha az `rcx` értéke nem 0, akkor az operandusként megadott cím történő vezérlésátadást hajt végre, azaz az `rip` regiszter értéke felülíródik az adott címmel (ami a ciklus elején lévő utasítás címe). Ha viszont az `rcx` értéke 0, akkor a végrehajtás a `loop` utáni következő utasítással folytatódik, azaz a ciklus befejeződik.

`mov`

Adatmozgató utasítás, amely Intel szintaxis esetén átmásolja a második operandus (azaz a forrás) értékét az első operandusba (azaz a célba) felülírva annak korábbi értékét. Az állapotregiszter bitek nem változnak.

`movsx`

Adatmozgatás előjeltartó kiterjesztéssel. Két operandusa van, de ezek mérete nem egyezik meg, konkrétan a cél operandus több bitet tartalmaz, mint a forrás operandus. Az utasítás a második operandus értékét átmásolja az első operandus legkisebb helyiértékű bitpozícióiba, a magasabb helyiértékeket pedig feltölti a forrás operandus előjelbitjeinek másolataival. Tulajdonképpen ugyanaz az előjeles érték kerül a cél operandusba, mint ami a forrásban volt, csak több biten ábrázolva. Az állapotregiszter bitek nem változnak.

`movzx`

Adatmozgatás zéró kiterjesztéssel. Két operandusa van, de ezek mérete nem egyezik meg, konkrétan a cél operandus több bitet tartalmaz, mint a forrás operandus. Az utasítás a második operandus értékét átmásolja az első operandus legkisebb helyiértékű bitpozícióiba, a magasabb helyiértékeket pedig feltölti 0 bitekkel. Tulajdonképpen ugyanaz az előjel nélküli érték kerül a cél operandusba, mint ami a forrásban volt, csak több biten ábrázolva. Az állapotregiszter bitek nem változnak.

`mul`

Előjel nélküli szorzás. Csak egy operandusa van (!). Ha az operandus 32 bites érték, akkor az `eax` regiszter értéke szorzódik meg az explicit operandus értékével és a keletkező 64 bites szorzat az `edx:eax` regiszterkombinációba kerül (azaz a szorzat magas helyiértékű 32 bitje az `edx` regiszterbe, míg a legkisebb helyiértékű 32 bit külön az `eax` regiszterbe felülírva az operandust). A `CF` és `OF` bitek 0 értéket tárolnak, ha az `edx` regiszter csupa 0 biteket tartalmaz (azaz a szorzat elfér 32 biten), különben a ezek az állapotregiszter bitek 1-re állítódnak.

`neg`

Az egyetlen operandus előjelváltása. Az operandus értéke felülíródik a saját kettes komplemensével, azaz a hatása hasonló a mínusz eggyel való szorzáshoz.

`nop`

„Nincs művelet” utasítás. Nem történik semmilyen hasznos műveletvégzés, de okoz némi késlekedést. Néha ezt a forráskódban történő „helyfenntartásra” használják. Nincs hatása az általános célú regiszterek, valamint az állapotregiszter tartalmára, természetesen a programszámláló változik végrehajtás közben.

`not`

Bitenkénti dagadás (azaz egyes komplemens képzés). Egyetlen operandussal rendelkezik, amelynek a bitjei ellentétes értékűre változnak a végrehajtás során.

`or`

Bitenkénti VAGY művelet az operandusok között. Az operandusok minden egyes bitpozíciója szimultán módon kerül felhasználásra a logikai VAGY művelet igazságtáblájának megfelelően. Az eredmény bitsorozat felülírja a cél operandust. Az `OF` és `CF` bitek törlődnek, az `SF` és `ZF` az eredménynek megfelelően állítódnak be.

`pop`

Érték kivétele a verem tetejéről. Az `rsp` regiszterben tárol memóriacím felhasználásával a verem tetején lévő 64-bites érték átmásolódik a RAM-ból a cél operandusba és azt követően a veremmutató regiszter értéke 8-cal növekszik. (A memória tartalma közben nem változik, azaz nincs fizikai törlés.)

`push`

Érték mentése a verem tetejére. Először a veremmutató regiszter (`rsp`) értéke csökken 8-cal, majd az egy kapott érték, mint memóriacím használódik fel egy memória írási művelet során, ahol az egyetlen operandusban megadott 64 bites érték átmásolásra kerül az `rsp` által megadott címre. Az operandus tartalma nem változik.

`ret`

Visszatérés alprogramból. Nincs operandusa (azaz a visszatérési érték nem egy operandus). A veremmutató regiszter (`rsp`) által címzett memóriaterületről 64 bit bemásolásra kerül a programszámlálóba (`rip`) majd az `rsp` értéke növekszik 8-cal.

`rol`

Balra forgatás. Két operandusa van. Hasonlóan viselkedik, mint egy `shl` utasítás, csak a legnagyobb helyiértékekről balra kitolt bitek a célregiszterben „megüresedő” legkisebb helyiértékekre kerülnek jobbról. A második operandus a mozgatás mértékét adja meg bitpozíció egységben.

`ror`

Jobbra forgatás. Két operandusa van. Hasonlóan viselkedik, mint egy `shr` utasítás, csak a legkisebb helyiértékekről jobbra kitolt bitek a célregiszterben „megüresedő” legnagyobb helyiértékekre kerülnek balról. A második operandus a mozgatás mértékét adja meg bitpozíció egységben.

`sal`

Aritmetikai balra tolás (shift-elés). Az első operandus minden egyes bitje balra tolódik a második operandusban megadott számú bitpozícióval. A jobb oldalon „megüresedő” legkisebb helyiértékű bitek 0-k lesznek. A korábban legnagyobb helyiértékeken tárolt bitek elvesznek az

utolsót (legkisebb helyiérékűt) kivéve, amely a CF állapotregiszter bitbe kerül. Ez az utasítás ugyanúgy viselkedik, mint az `shl` utasítás.

`sar`

Aritmetikai jobbra tolás, más néven előjeltartó shift-elés. Az első operandus minden egyes bitje jobbra tolódik a második operandusban megadott számú bitpozícióval. A bal oldalon „megüresedő" legnagyobb helyiértékű bitek az első operandus eredeti értékében szereplő előjelbit másolataival töltődnek fel. A korábban legkisebb helyiértékeken tárolt bitek elvesznek a legnagyobb helyiérékűt kivéve, amely a CF állapotregiszter bitbe kerül. Ez az utasítás másként viselkedik, mint az `shr` utasítás. Kettő hatványaival történű előjeles osztásként is felfogható.

`seta, setb, setbe, setc, sete, setea, setg, setge, setl, setle, setnc, setne, setnz, setz`

Egy bájt beállítás feltételtől függően. Ennek az utasítás csoportnak (amelyre gyakran $\texttt{set}_{cc}$-ként hivatkozunk) csak egyetlen operandusa van. Az operandus egy feltétel alapján vagy csupa nulla bitre állítódik (`00000000`, azaz egész típusú 0, ami egyfajta *hamis* értékként is felfogható) vagy a `00000001` értéket kapja meg (ami az egyész típusú 1 érték, mint egyfajta *igaz* érték). Maga a konkrét feltétel a mnemonic-ban van megadva hasonlóan a feltételes vezérlésátadó ($\texttt{j}_{cc}$) utasításokhoz. A döntés az `SF`, `ZF`, `CF`, és `OF` állapot regiszter bitek alapján születik meg (amelyeket feltehetően egy korábbi `cmp` utasítás állított be). Az állapot regiszter bitjei nem változnak a végrehajtás során.

`shl`

Logikai balra tolás (shift-elés). Az első operandus minden egyes bitje balra tolódik a második operandusban megadott számú bitpozícióval. A jobb oldalon „megüresedő" legkisebb helyiértékű bitek 0-k lesznek. A korábban legnagyobb helyiértékeken tárolt bitek elvesznek az utolsót (legkisebb helyiérékűt) kivéve, amely a CF állapotregiszter bitbe kerül. Ez az utasítás ugyanúgy viselkedik, mint az `sal` utasítás.

`shr`

Logikai jobbra tolás (shift-elés). Az első operandus minden egyes bitje jobbra tolódik a második operandusban megadott számú bitpozícióval. A bal oldalon „megüresedő" legnagyobb helyiértékű bitek 0-k lesznek. A korábban legkisebb helyiértékeken tárolt bitek elvesznek az utolsót (legnagyobb helyiérékűt) kivéve, amely a CF állapotregiszter bitbe kerül. Ez az utasítás másként viselkedik, mint az `sar` utasítás.

`sub`

Az operandusok különbsége. Az első operandus értékéből kivonásra kerül a második operandus értéke és a különbség felülírja a cél operandust. Az operandusok és az eredmény értelmezhető előjeles és előjel nélküli fixpontos értékként is. Az `SF`, `ZF`, `CF` és `OF` állapot regiszter bitek változhatnak az eredménynek megfelelően.

`test`

Logikai összehasonlítás. Egy bitenkénti ÉS művelet kerül végrehajtásra az operandusok felhasználásával és az eredménynek megfelelően az `SF` és a `ZF` állapotbitek beállítódnak. Magának az ÉS-elésnek az eredménye eldobásra kerül, azaz az operandusok nem változnak meg. (Nullától eltérő értékek egyenlősége vizsgálható ezáltal.)

`xchg`

Az operandusok felcserélése. Két operandusa van és mindkettő megváltozik a végrehajtás során. Az első operandus értéke eltárolódik a másodikban és ezzel párhuzamosan a második operandus értéke bekerül az elsőbe.

`xor`

Bitenkénti kizáró vagy művelet. Az operandusok minden egyes bitpozíciója szimultán módon kerül felhasználásra a logikai XOR (kizáró vagy) művelet igazságtáblájának megfelelően. Az eredmény bitsorozat felülírja a cél operandust. Az `OF` és `CF` bitek törlődnek, az `SF` és `ZF` az eredménynek megfelelően állítódnak be.

5. Alprogram hívási konvenciók

A programozóknak alkalmazniuk kell néhány konvenciót azért, hogy olyan kódot írjanak, amely kompatibilis más programozók által írt kódokkal. Most bemutatásra kerül, hogy a megegyezés szerint hogyan kell alprogramokat (azaz függvényeket és eljárásokat) meghívni, paramétereket átadni, visszatérési értéket átadni a hívónak, lokális (dinamikus élettartamkezelésű) változókat kezelni, illetve rekurziót megvalósítani.

Először a paraméterek értékeit kell előkészíteni átadásra (természetesen a paraméter kiértékelés után). Ha a paraméterek száma nem több, mint 6 akkor mindegyik regisztereken keresztül kerül átadásra. Az első fixpontos ábrázolású paraméter értékét mindig az `rdi` regiszterbe kell tölteni, a másodikat az `rsi`-be, a harmadikat az `rdx`-be, a negyediket az `rcx`-be az ötödiket az `r8`-ba a hatodikat pedig az `r9` regiszterbe. Ha vannak további fixpontos paraméterek, azokat a vermen keresztül kell a hívottnak átadni. Ha az alprogramnak vannak valós szám paraméterei (is), akkor azokat lebegőpontos regisztereken keresztül adhatóak át. Az első lebegő pontos paraméter értékét az `xmm0` vektor regiszter kisebb helyiértékű bitjeibe kerülnek (egyedüli értékként), a második paraméter értéke az `xmm1` regiszter „aljára” és így tovább. Azoknál az alprogramoknál, ahol a paraméterlista változó a lebegőpontos paraméterek számát el kell menteni az `eax` regiszterbe hívás előtt. Ezután jöhet a vezérlés átadása a hívott programegységnek a `call` utasítás segítségével. Ez a verembe menti a programszámláló (`rip`) regiszter aktuális értékét (ami gyakorlatilag a visszatérési cím, azaz visszatérés után a hívás utáni következő utasítással kell folytatni a végrehajtást). Ennek hatására természetesen a veremmutató regiszter (`rsp`) értéke 8-cal csökken, majd a `rip` regisztert felül kell írni a `call` utasítás operandusaként megadott címmel (ami gyakorlatilag a hívott fél első utasítása elé helyezett címke). Az alprogramok elején a bázispointer regiszter (`rbp`) értékét be kell tenni a verembe (`rsp` ismét csökken 8-cal), aztán a `rsp` aktuális értékét át kell másolni az `rbp`-be végül az `rsp` értékét csökkenteni kell annyival, ahány bájtra szükségünk van a lokális változók számára. Ezt követően, ha vannak paraméterek, akkor ezek értékét a „köztes tárolókból” (a megadott regiszterekből és a veremből) át kell másolni a lokális változókba. Amennyiben az alprogram össze további szükséges utasítása (függvény/eljárás törzs) lefutott meg kell kezdeni a visszatérés folyamatát. Amennyiben egy függvényről beszélünk a visszatérési értéket be kell másolni az `eax` regiszterbe (lebegőpontos visszatérési érték esetén az `xmm0`-ba). Ezt követően helyre kell állítani a vermet és a regisztereket a `mov rsp, rbp` majd a `pop rbp` utasítás (vagy ezek helyett a `leave` utasítás) használatával. Ezt követi csak a vezérlés visszaadása a hívónak a `ret` utasítás által, amely kiveszi az éppen a verem tetején lévő visszatérési címet (rsp nő 8-cal) és beteszi az rip regiszterbe, így tehát a következő végrehajtandó utasítás már a

hívó egyik utasítása lesz. Ha a vermet is használni kellett paraméterátadásra, akkor a hívó eltávolítja a betett paramétereket és ezzel az alprogramhívás befejeződött.

6. Példa

Lentebb található két példa kód. Az egyik egy C nyelven írt program, a másik egy Intel szintaxist használó x86-64 assembly kód (Linux operációs rendszerre). Ezek azonos jelentésűek, összehasonlításuk segíthet megérteni, hogy mit kell írni egy assembly kódba, ha egy olyan tevékenységet szeretnénk elvégezni, ami magasszintű programozási nyelven az adott formában van megadva. Tanulmányozd a kódokat nagy figyelemmel!

A program két függvényt tartalmaz. A `square` függvénynek egy paramétere van (ami egy lokális változó) és a paraméter négyzetével tér vissza. A main programegységben van egy ciklus, ami egy tömböt inicializál kis egész számok négyzetével és ezután kiírja az utolsót a képernyőre. Az assembly kódot sokkal optimalizáltabb módon is meg lehetett volna adni, de így pontosan/teljesen egyenértékű a C kóddal.

```
/* C program: ciklus, tomb, fuggveny, stb. */
#include<stdio.h>
int square(int Num){
  int N2;
  N2 = Num*Num;
  return N2;
}
int main(){
  int i = 0, S[10];
  while(i<10){
    S[i] = square(i);
    i++;
    }
  printf("Utolso: %d\n",S[9]);
  return 0;
}
```

```
# x86-64 assembly program: ciklus, tomb, fuggveny, stb.
        .intel_syntax noprefix
        .globl    square
square: push      rbp
        mov       rbp, rsp
        sub       rsp, 4              # hely a lokalis valtozoknak
        mov       DWORD PTR [rbp-20], edi  # Paraméter a RAM-ba
        mov       eax, DWORD PTR [rbp-20]
        imul      eax, eax                 # Negyzetre emeles
        mov       DWORD PTR [rbp-4], eax   # Valtozoba mentes
        mov       eax, DWORD PTR [rbp-4]   # Visszatérési érték
        mov       rsp, rbp
        pop       rbp
        ret
        .globl    main
main:   push      rbp
        mov       rbp, rsp
        sub       rsp, 64             # Hely a lokalis valtozoknak
```

```
        mov     DWORD PTR [rbp-44], 0
        jmp     .L4
.L5:    mov     edi, DWORD PTR [rbp-44]  # Parameter atadasa
        call    square                   # Hivas
        movsx   rdx, DWORD PTR [rbp-44]
        mov     DWORD PTR [rbp+rdx*4-40], eax
        add     DWORD PTR [rbp-44], 1    # Az i inkrementacioja
.L4:    cmp     DWORD PTR [rbp-44], 9    # Feltetel kiertekeles
        jle     .L5                      # Tovabb vagy vissza
        mov     esi, DWORD PTR [rbp-4]   # Elso parameter
        lea     rdi, [.LC0]              # masodik parameter
        mov     eax, 0                   # Nincs float parameter
        call    printf                   # Konyvtari fgv hivas
        mov     eax, 0                   # EXIT_SUCCESS
.L7:    mov     rsp, rbp
        pop     rbp
        ret
        .section .rodata
.LC0:   .string  "Last: %d\n"            # Formatum sztring
```

Irodalom

1. *x86 and amd64 instruction reference* https://www.felixcloutier.com/x86/ (2023).
2. *Intel® 64 and IA-32 Architectures Software Developer's Manual* https://irh.inf.unideb.hu/~vargai/download/assembly/Intel.pdf (Intel, 2021).
3. Ray Seyfarth: *Introduction to 64 bit assembly programming for Linux and OS X*, (Amazon, 2013).
4. R. E. Bryant, D. R. O'Hallaron: *Computer Systems - A programmer's perspective* (Pearson, 2016).
5. Joseph Cavanagh: *X86 Assembly Language and C Fundamentals*, (CRC Press, 2013).
6. Richard Blum: *Professional Assemby Language*, (Wiley, 2005).