ASSEMBLY vizsgakérdések

1. Mi alapján lehet meghatározni a programozás során használt regiszterek számát és azok feladatait?

Általános célú regiszterek:
AX (accumulator): bizonyos aritmetikai utasítások használják forrás és/vagy cél tárolására
BX (base): címzéshez
CX (counter): számlálóként használja néhány ismétlő utasítás (pl. LOOP)
DX (data): I/O utasítások használják, illetve néhány aritmetikai utasítás számára különös jelentőséggel bír

Ezek felső és alsó 8 bitje közvetlenül elérhető (pl. AH, AL, DH, DL… stb.)

Egyéb:
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
BP (base pointer): címzésre használhatjuk, de általában veremnél szoktuk
IP (instruction pointer): a következő végrehajtandó utasítás címét tartalmazza, közvetlenül nem elérhető, de írható/olvasható a vezérlésátadó utasításokkal
SR (Status Register), vagy FLAGS: a processzor aktuális állapotát, az előző művelet eredményét mutató, illetve a processzor működését befolyásoló biteket, ún. flag-eket tartalmaz, bizonyos utasítások manipulálják
CS (code segment): a végrehajtandó program kódját tartalmazó szegmens címe, nem érhető el közvetlenül, de bizonyos utasítások manipulálhatják
DS (data segment): az alapértelmezett, elsődleges adatterület szegmensének címe
ES (extra segment): másodlagos adatszegmens címe
SS (stack segment): a verem szegmensének címe

Milyen módon lehet Assemblyben lebegőpontos értékeket használni? Milyen forrásnyelvi támogatásokat ismer?
Adatdefiniáló direktívák segítségével tárolhatunk lebegőpontos értékeket.
Direktíva: a fordítónak szóló, a fordítás körülményeit meghatározó utasítás (NEM mnemonik)
Adattároló direktívák:

- DB (define byte): 1 bájt
- DW (define word): 2 bájt
- DD (define double-word): 4 bájt
- DQ (define quadro word): 8 bájt
- DT (define tenbyte): 10 bájt

Lebegőpontos számok tárolására legalább double word-öt kell használnunk, ugyanis 32 bit kell nekik. Tehát, a

```
szam   db      1.5
```

helytelen, míg a

```
szam   dd      1.5
```

már helyes.

A 8086 nem ismeri a lebegőpontos számokat, így szükség van a 8087-es koprocesszorra, ha kezelni akarjuk őket. Ez a koprocesszor 8 regiszterrel rendelkezik, ezek csak veremként kezelhetők, közvetlenül nem érhetők el.
Külön utasítások jöttek létre a lebegőpontos értékekkel való műveletekhez (ezeket mind a koprocesszor végzi el):

- FADD
- FMUL
- FDIV
- stb..

2. Egy utasítás hossza (bájtszáma) milyen paraméterektől függ a különböző típusú utasításoknál?

Az x86 utasítások mérete 1-től 15 bájtig terjedhet.
Az utasítások a következőképpen épülnek fel:

- prefixek (max. 5 bájt)
- műveleti kód (max. 1 bájt)
- címzési mód/operandus info paraméter (max. 2 bájt)
- absolute offset (eltolás) (max. 4 bájt)
- közvetlen értékek (max. 4 bájt)

Egyszerre nem lehet mind maximális hosszúságú, tehát a 15 bájt korlátot nem tudjuk megszegni.

Előjel nélküli és előjeles műveletek esetén milyen különbségek vannak a feltételes ugrásoknál?

Feltételes ugrásoknál FLAG-ek státusza alapján dől el, hogy ugrunk-e vagy sem.
A feltételes ugrásokat 2 csoportra oszthatjuk:

- Előjeles: JG, JL, stb… (greater - less)
- Előjel nélküli: JA, JB (above - below)

Az előjeles esetben szükségünk van a Sign Flag-re -> befolyásolja, hogy ugrunk-e, vagy sem (előjeles összehasonlítás).
Előjel nélküli esetben nincs rá szükség, viszont a Carry Flag-re igen. (előjel nélküli összehasonlítás).
A feladattól függ, hogy melyiket érdemes használni. Például, ha IDIV/IMUL-t (előjeles osztás/szorzás) használunk, és utána egyből feltételesen akarunk ugrani, akkor az előjeles ugró utasításokat érdemes használni, mivel előjeles műveletek esetén más flag-ek állnak be.

3. Az utasítás hossza (bájtszáma) milyen esetekben és milyen módon függ az utasítás paramétereitől?

Az utasítások hossza függ a paraméterek számától, a paraméterek hosszától, illetve típusától.
A MOV utasítás esetén például, ha AX-be mozgatunk 0-t, akkor az utasítás hosszabb lesz, mintha AL-el tennénk ezt, mivel ilyenkor a 0-t 2 bájtos értékként kezeli (mivel AX is 2 bájtos).
DEC/INC utasítások esetében, ha 1 bájtos az operandus (pl. AL, AH, DL, DH, stb…), akkor hosszabb lesz az utasítás a 2 bájtos operandussal rendelkező változathoz képest, mivel külön bájtban jelzi, hogy az operandus 1 bájtos.

Bizonyos több paraméteres utasításoknál (pl. OR, ADD, stb…), ha a második operandus konstans, akkor több helyet fog foglalni az utasítás, mintha mind a két paraméter regiszter lenne (hiszen el kell tárolni a konstanst).
Címzés esetében, ha offszetet használunk, akkor függ ennek a méretétől is. Például, a MOV AL, [BX + 1] rövidebb lesz, mint a MOV AL, [BX + 1024], mivel ilyenkor az offszet (1024) nem fér el 1 bájtban.

Milyen módokon változik (változhat) meg az IP regiszter értéke? Mutasson példákat az értékek konkrét utasításokhoz kapcsolódó változtatására!
Az IP regiszter értékét az ugró utasítások megváltoztathatják. Ha ilyenek nem szerepelnek a programunkban, akkor IP folyamatosan nő (nem mindig egyesével, illetve 100-on kezdődik).

Ugrások hatására: Amennyiben a cimke az ugrás helyétôl egy elôjeles byte-nyi távolságra van (-128 ill. +127 byte), ebben az esetben a jmp utasítás relatív cimzést használ. Ez azt jelenti, hogy az ugrás távolsága elôjelesen hozzáadódik az IP regiszter tartalmához, ezáltal megvalósul a vezérlésátadás.

Abban az esetben ha az ugrás helye távolabb van, mint amit egy elôjeles byte-on ábrázolni tudunk, abszolút címzéssel hajtódik végre a jmp. Ez pedig azt jelenti, hogy a cimkéhez tartozó offset cím kerül az IP regiszterbe, azaz símán felülíródik az IP regiszter tartalma. A címkéhez tartozó offset cím a szimbólumtábla alapján érhetô el.

4. Milyen részekre tagolható egy Assembly parancssor? Melyik résznek mi a szerepe a kialakítandó kódban?
<címke>: <utasítás> <paraméter(ek)> ; megjegyzés
<címke>: <mnemonik> <operandusok> ; megjegyzés

A megjegyzés a program visszafejtését segíti (mikor mit csinálunk). Címkét ugrás vagy más hivatkozások esetén használunk, a paraméterek (és operandusok) száma pedig az utasítástól függ. Kis és nagybetű nem számít.
Milyen adatmozgató utasításokat ismer? Mutasson példákat az adatok forrás és cél helyeinek ismertetésével!
MOV - adatok mozgatása
XCHG - adatok cseréje
PUSH - adat betétele a verembe
PUSHF - Flag regiszter betétele a verembe
POP - adat kivétele a veremből
POPF - Flag 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 - FLAG regiszter alsó bájtjának betöltése AH-ba

SAHF - AH betöltése a FLAG regiszter alsó bájtjába
CLC - CF flag törlése
CMC - CF flag invertálása
STC - CF flag beállítása
CLD - DF flag törlése
STD - DF flag beállítása
CLI - IF flag törlése
STI - IF flag beállítása

Példák

```
MOV AX, BX   ; regiszterből regiszterbe, AX-be BX értékét tesszük
MOV AX, [BX] ; a BX által mutatott értéket tesszük AX-be
MOV [BX], AX ; a BX által mutatott címre tesszük AX értékét
MOV DX, offset címke          ; a címke offset-jét (DS szegmenshez képest) tesszük DX-be
PUSH CX      ; CX értékét betesszük a verembe (SP értéke változik)
POP CX       ; a verem tetején lévő értéket kivesszük és eltároljuk CX-ben (SP értéke változik)
```

5. Miben különböznek a direktívá(ka)t tartalmazó sorok az Assembly forrásprogramokban a parancssortól?

Direktíva: a fordítónak szóló, a fordítás körülményeit meghatározó utasítás (NEM mnemonik)
Parancssor: processzornak szóló utasítás

Mivel a direktívák is byte-ok lesznek, ezért lehet velük is “működő” programot írni.

Mire használjuk a léptető és forgató utasításokat? Mutassa be melyik milyen környezetben és mire használható!

SHL (balra léptetés): 2-vel való szorzás (2^n, ha n-szer léptetünk balra)
SHR (jobbra léptetés): 2-vel való osztás (2^n, ha n-szer léptetünk jobbra)

A forgató utasítások hasonlóan működnek, mint a léptetők, viszont a kilépő bitek a másik oldalon belépnek. Ezek a ROL/ROR/RCR/RCL utasítások (az utóbbi 2 CF-en keresztül forgat, azaz a kilépő bit mindig CF-be kerül, majd következő forgatásra “beforog”).

A forgató utasítások hasznosak például, ha egy regiszter alsó és felső bájtját meg szeretnénk cserélni (ilyenkor 8-cal elforgatunk balra, vagy jobbra, de nem CF-en keresztül)

CF-en keresztüli forgatással és feltételes ugrással egy regiszter tetszőleges bitjének az értékét lekérdezhetjük.
Például (ah n. bitjét szeretnénk balról):

```
XOR bx, bx   ; bx-et használjuk az ellenőrzésre (bx = 0)
MOV cl, n    ; a lépésszám 1, vagy CL lehet
RCL ah, cl   ; “elforgatjuk” ah-t CF-en keresztül CL lépésszámmal
ADC bx, 0    ; hozzáadunk 0 + CF-et
RCR ah, cl   ; visszaforgatjuk ah-ba az eredeti értéket (ha szükségünk van rá)
TEST bx, 1   ; ha a bit 1
```

```
JZ cimke        ; akkor ugrunk
…               ; ez itt akkor fut le, ha a bit nem 1 volt
cimke:
…
```

6. Milyen direktívákat használ(hat)unk egy Assembly program forrássoraiban? Melyik milyen hatással bír?

- modul/forrásfájl lezárása (END)
- szegmens definiálása (SEGMENT...ENDS)
- szegmenscsoport definiálása (GROUP)
- szegmens hozzárendelése egy szegmensregiszterhez (ASSUME)
- értékadás a $ szimbólumnak (ORG): hatására úgy teszünk, mintha az operandusként megadott címen lennénk (az assemblernek ez azt jelenti, hogy minden szegmensbeli offszetcímhez adjon hozzá ennyit) <- például ORG 100h, ilyenkor olyan, mintha a 256. byte-on lennénk, a .COM programok esetében ez kell, mivel a 100h-s cím előtt a PSP található
- 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): az aktuális címen megadott mennyiségű bájtot (1, 2, 4, 6, 8, 10) lefoglalunk és egy címkét adunk neki
- konstans/helyettesítő szimbólum létrehozása (=, EQU): hatására az EQU bal oldalán lévő szimbólum összes előfordulását a jobb oldalon lévő kifejezés értékével fogja helyettesíteni az assembler
- 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, .8086, .186, .286, .386, stb.)

Hogyan használható az INT utasítás? Mi indokolja a használatát a CALL FAR használatával szemben?

Az INT utasítás egy bájt méretű közvetlen adatot kér operandusként. Ez az adat a kért szoftver-megszakítást azonosítja.
Ezek az adatok a 0000h szegmens első 1024 bájtján találhatóak, ezt hívjuk a megszakítás-vektor táblának. Ennek mindegyik bejegyzése egy 4 bájtos távoli pointer (szegmens:offszet típusú memóriacím), nevük megszakítás-vektor.
Ilyen alapon használható lenne a CALL FAR utasítás is úgy, hogy a pontos címeket használjuk. Viszont, a két utasítás (INT és CALL FAR) működésben különböznek egymástól:

- INT k: PUSHF, PUSH CS, PUSH IP, majd IF és TF flageket törli (ezzel biztosítva, hogy a megszakítás lefolyását nem fogja semmi sem megakadályozni), végül [CS:IP] <- [0000h:k*4]
  ha a megszakításkezelő végzett, akkor IRET (POP IP, POP CS, POPF)
- CALL FAR: PUSH IP, PUSH CS, majd [CS:IP] <- új érték
  ha végeztünk, akkor RETF (POP CS, POP IP)

CALL FAR esetén megszakítható a végrehajtás, mivel az IF és TF flageket nem törli. Ezen kívül tudnunk kell a megszakításkezelő pontos címét. Ráadásul, az INT esetén extra információval szolgálunk a programozónak: biztosan megszakítást akarunk hívni (CALL FAR esetén ez nem egyértelmű, mivel másra is használható).
RÁADÁSUL A MEGSZAKÍTÁSOK ÁTIRÁNYÍTHATÓK!

7. Milyen lépéseken keresztül jutunk el a feladat megfogalmazásától a működő programig Assembly nyelven történő programozáskor?

1. A feladatot leképezzük programozási problémára, és kiválasztjuk az eszközöket
2. A feladat részekre bontása
3. Használandó regiszterek és változók kiválasztása
4. A különböző részeknek Assembly programrészleteket feleltetünk meg
5. Program megírása
6. Lefordítjuk (mondjuk Turbo Assembler-rel), ezzel létrejön egy .OBJ kiterjesztésű fájl
7. Linkeljük (mondjuk Turbo Linker-rel), így az opcióktól függően .EXE, vagy .COM kiterjesztésű, futtatható program áll elő
   Ha több forrásfájlunk van, akkor külön mindegyiket lefordítjuk, majd a linkerrel összekapcsoljuk őket.
8. Futtatás, majd hibakeresés

Milyen módon lehet az XT struktúrájában a különböző perifériákkal adatot cserélni? Milyen utasítástípusok vonatkoznak az egyes csoportokra?
A 80X86 processzor kialakítása a memória eléréséhez hasonlóan biztosítja az I/O műveletek végrehajtását. Itt ugyan már nincs szegmens regiszter szerinti címzés, de 2^16 (azaz 0-tól 0FFFFH-ig) I/O cím kialakítására van lehetőség.

A legegyszerűbb a direkt címzésű IN (Input) és OUT (Output) utasítás. Az IN utasítás adatforrása és az OUT utasítás célja is az akkumulátor regiszter. Ha az I/O művelet 8 bites, akkor az AL regiszter, ha 16 bites, akkor az AX regiszter az, ahol az adat helyet kap.

IN: olvasás a paraméterként megadott portról
Formája: IN AL/AX, 8 bites konstans
IN AL/AX, DX

Használhatunk AL-t, vagy AX-et. Ebbe lesz beolvasva egy bájt/szó a második paraméterként megadott portról. A második paraméter 8 bites konstans, vagy a DX regiszter lehet (a 256...65535 portokat csak a DX segítségével tudjuk elérni)
OUT: írás portra.

OUT: írás portra
Forma: OUT 8 bites konstans, AL/AX

OUT DX, AL/AX
Az AL vagy AX regiszter tartalmát kiküldi a konstanssal vagy a DX regiszterrel jelzett portra.

Lehetőségünk van még szoftvermegszakításokkal is adatokat cserélni a perifériákkal (INT utasítás).
10h: képernyővel kapcsolatos szolgáltatások
13h: lemezműveletek
14h: soros ki-/bemenet
16h: billentyűzet
21h: DOS szolgáltatások
25h: közvetlen lemezolvasás
26h: közvetlen lemezírás
27h: rezidenssé tétel
33h: egérkezelés

triviális példák a megszakítások használatára (karakter bekérése, szöveg kiírása, stb...)

8. Milyen összefüggés van a gépi kódú utasítások és az Assembly program utasításai között?
Bármely gépi kódú program leírható utasításra pontosan Assembly-vel. Minden egyes Assembly utasításnak van megfelelője a gépi kódban, tehát a leképezés 1:1. Mondhatjuk azt is, hogy az Assembly nyelv gyakorlatilag a gépi kód egy (ember által) olvashatóbb változata.

Milyen címzési módokkal érhetünk el egy (konkrét) memóriaelemet? Egy utasításban egyszerre hány bájt „mozdulhat” meg?
Címzési módok:

Típus	**Eltolás**	**Forma**	**Alapértelmezett szegmensregiszter**
Közvetlen címzés	konkrét eltolási érték (numerikus, vagy szimbólum neve)	[konstans]	DS
Regiszter-indirekt	az eltolási címet az illető regiszter adja	[BX], [SI], [DI]	DS
Kombinált	regiszter és konstans összege	[BX+konstans] [SI+konstans] [DI+konstans]	DS
Regiszter-indirekt	regiszterek összege	[BX+SI], [BX+DI]	DS
Kombinált	regiszterek összege + konstans	[BX+SI+konstans] [BX+DI+konstans]	DS
Kombinált	BP + konstans	[BP+konstans]	SS

Regiszter-indirekt	regiszterek összege	[BP+SI], [BP+DI]	SS
Kombinált	regiszterek összege + konstans	[BP+SI+konstans] [BP+DI+konstans]	SS

A 8086-os processzor esetén egyszerre 2 bájt mozdulhat meg, mert akkora az adatbusz (a címbusz viszont 20 bites), amin folyik az adatáramlás. Ezen kívül a regiszterek is ekkorák.

9. Egy gépi kódú programot DisAssemblerrel visszafejtve előállítható-e a forrásprogram? Válaszát indokolja is!

A forrásprogramban vannak kommentek, címkék (névvel ellátva), illetve direktívák. Ezeket nem tudjuk visszafejteni (mivel a program bájtok sorozata, a kommenteket pedig a fordító figyelmen kívül hagyja), ezért nem.

Ismertesse a 8086-os processzor belső szerkezeti felépítését az Assembly nyelv használatának szempontjait figyelembe véve!

A 8086-os processzor 20 bites címbusszal rendelkezik, tehát a memóriacímek 20 bitesek lehetnek maximum (ez 1MB, azaz 1024x1024 byte méretű memória megcímzéséhez elegendő). A processzor csak a szegmentált címzést ismeri. A szegmensek méretét 64KB-ban szabták meg, mivel így az offszet 16 bites lesz, ez pedig belefér bármelyik regiszterbe.

A 8086 14 db. 16 bites regiszterrel rendelkezik:

- 4 általános célú regiszter (AX, BX, CX, DX): ezek alsó és felső bájtja külön neveken el is érhető (pl. AL, BL, AH, BH, stb.)
- 2 indexregiszter (SI és DI)
- 3 mutatóregiszter (SP, BP, IP)
- 1 státuszregiszter (SR, vagy FLAGS)
- 4 szegmensregiszter (CS, DS, ES, SS)

A szó méretét 2 bájtban állapították meg az Intel-nél, ezért az általános célú regiszterek szavasak (és ebből kifolyólag az alsó és felső feleik bájtosak). Ezen kívül az adatbusz is 2 bájtos, tehát egyidejűleg egyszerre csak 2 bájtot tudunk “megmozgatni”

A 8086-os processzor esetén a verem mérete maximálisan 64KB. Ez azért van, mert a verem csak egy szegmensnyi területet foglalhat el (azok pedig 64KB-osak). A verem szegmensét az SS regiszter tárolja, a verem tetejére pedig az SP mutat. Mivel lefelé bővülő veremről beszélünk, ezért új adat betételekor SP értéke csökken (mert az adatok egyre alacsonyabb memóriacímen helyezkednek el).

A 8086 16 biten ábrázolja a portszámokat, így összesen 65536 db. portot érhetünk el a különféle I/O utasításokkal.

A FLAGS regiszter az előző művelet eredményével kapcsolatos információkat tartalmaz (feltéve, hogy az utasítás hozzányúlt), ezeket fel lehet használni pl. feltételes ugrásokra.

10. Milyen módon érhető el a 8086-os processzor 20 bites memória címzése az általa használt 16 bites regiszterekkel?

20 bites címek képzésére egy regiszter nem elég (mivel azok 16 bitesek), ezért regiszterpárokat kell alkalmaznunk. Ilyenkor az egyik regiszter a cím szegmens részét, a másik regiszter pedig a cím offset részét fogja tartalmazni.

Így a cím két 16 bites összetevőre osztható fel. Ezek után a szegmens részt megszorozzuk 16-tal (azaz 4-szer balra léptetjük). Az így kapott - immár 20 bites - értékhez hozzáadjuk az offsetcímet.
Tehát: Fizikai cím = SZEGMENS*16 + OFFSET
Ebből látható, hogy előfordul olyan eset, amikor két szegmens “átfedi” egymást. Például a 0040h:0002h és a 0030h:0102h címekből ugyanaz a fizikai cím fog előállni (00402h).

Példa 20 bites cím képzésére:
1234h:5678h a címünk
A szegmens részt eltoljuk balra 4-gyel (16-tal szorzunk):
12340h (mivel 16-os számrendszerben vagyunk, ezért itt 1-gyel tolunk el)
Hozzáadjuk az offset-et:
12340h + 5678h = 1 79B8h lesz a 20 bites címünk.

Milyen eszközöket használhatunk az Assembly nyelven a 8086-os processzornál adattömbök kezelésére?
Egy dimenziós tömbök (vektorok) létrehozásához használhatjuk a define direktívát. Attól függően, hogy mekkora (hány bájtos) elemeket szeretnénk tárolni, a DB, DW, DD, stb… változatokra lesz szükségünk.
Mivel a tömböket folytonosan tároljuk, ezért elég a kezdőcímet tudni (a define-ból ezt tudjuk), illetve az elemszámot.
Példa egy bájtos numerikus értékek tárolására:

```
elem_szam    EQU    5
tomb    db       5, 100, 12, 34, 55
```

Ha nem akarjuk előre rögzíteni az elemszámot, akkor akár rábízhatjuk a fordítóra is:

```
tomb    db       5, 100, 12, 34, 55
elem_szam    EQU    $ - tomb
```

Ezek után címzéssel elérhetjük az elemeket:

```
MOV    AL, [tomb]                    ; az első elemet tesszük AL-be
       AL, [tomb + 1]                ; a másodikat
       AL, [tomb + elem_szam - 1]    ; az utolsót
… stb.
```

Példa tömb elemeinek összegzésére:

```
...
       XOR BX, BX            ; a bázisregisztereket felhasználhatjuk “indexelésre”
       XOR DX, DX            ; dx-ben tároljuk az eredményt
       MOV CX, elem_szam ; ennyiszer fusson le a ciklus
ciklus:
       MOV AL, [tomb + BX]
       CBW   ; kiterjesztjük AL-t 2 bájtossá előjelesen
       ADD DX, AX
       INC BX
       LOOP ciklus
```

...

11. Sorolja fel a8086-os processzorban használható flageket! Melyik minek a jelzésére használható? Van-e több flagnek együttes jelentése?

- Carry Flag (CF): 1, ha volt aritmetikai átvitel az eredmény legfelső bitjén (előjel nélküli aritmetikai túlcsordulás), 0, ha nem
- Parity Flag (PF): 1, ha az eredmény legalsó bájtja páros számú 1-es bitet tartalmaz, különben 0.
- Auxillary Flag (AF): 1, ha volt átvitel az eredmény 3-as és 4-es bitje (tehát az alsó és felső nibble) között.
- Zero Flag (ZF): 1, ha az eredmény zérus lett.
- Sign Flag (SF): az eredmény legfelső bitjének (előjelbit) tükörképe.
- Trap Flag (TF), ha ez 1, akkor a lépésenkénti végrehajtás engedélyezve van (pl. debug-hoz)
- Interrupt Flag (IF): ha ez 1, akkor a maszkolható hardvermegszakítások engedélyezettek.
- Direction Flag (DF): ha ez 0, akkor a sztringutasítások növelik SI-t és/vagy DI-t, különben csökkentésük történik.
- Overflow Flag (OF): 1, ha előjeles aritmetikai túlcsordulás történt

Összefüggések (cmp utasítás használata után):

- ha SF = OF vagy ZF = 1, akkor akkor a bal oldali operandus nagyobb, vagy egyenlő mint a jobb (előjelesen)
- ha SF = OF és ZF = 0, akkor a bal oldali operandus nagyobb (előjelesen)
- ha CF = 0 vagy ZF = 1, akkor a bal oldali operandus nagyobb, vagy egyenlő (előjel nélkül)
- ha CF = 0 és ZF = 0, akkor a bal oldali operandus nagyobb (előjel nélkül)
- ha SF != OF, akkor a bal oldali operandus kisebb (előjelesen)
- ha SF != OF vagy ZF = 1, akkor a bal oldali operandus kisebb, vagy egyenlő (előjelesen)
- (ha CF = 1, akkor a bal oldali operandus kisebb, előjel nélkül)
- ha CF = 1 vagy ZF = 1, akkor a bal oldali operandus kisebb, vagy egyenlő (előjel nélkül)

Milyen aritmetikai műveletek végezhetők el a 8086-os processzorban? Ismertesse példákkal!

- ADD op1, op2 ; op1 := op1 + op2
- ADC op1, op2 ; op1 := op1 + op2 + C (carry)
- INC op ; op := op + 1, C-re nincs hatással
- SUB op1, op2 ; op1 := op1 - op2
- CMP op1, op2 ; elvégzi op1 - op2-őt, de nem tárolja el, a flag-ek pedig beállnak
- SBB op1, op2 ; op1 := op1 - op2 - C (carry)
- DEC op ; op := op1 - 1, C változatlan
- NEG op ; az operandus kettes komplemensét képzi

- MUL op ; előjel nélküli szorzás, op nem lehet közvetlen operandus...
  ; … AX := AX * op, ha op 8 bites,
  ; ... DX:AX := AX * op, ha op 16 bites
- IMUL op ; előjeles szorzás
- CBW ; AL kiterjesztése AX-be (8-ból 16 bites lesz az érték)
- CWD ; AX kiterjesztése DX:AX-be (16 bit -> 32 bit)
- DIV op ; előjel nélküli osztás, op nem lehet közvetlen operandus…
  ; Ha op 8 bites: AH-ba a maradék, AL-be a hányados kerül
  ; (ilyenkor AX-et osztjuk op-val)
  ; Ha op 16 bites: AX-be a hányados, DX-be a maradék
  ; (ilyenkor DX:AX-et osztjuk op-val)
- IDIV op ; előjeles osztás

12. Hogyan tudunk memóriarezidens programokat írni DOS operációs rendszernél? Hogyan jut „szóhoz” a memóriarezidens program miután „kilépett”?

Memóriarezidens program: olyan program, amihez tartozó memóriaterületet, vagy annak egy részét nem szabadítja fel a DOS a program befejeződésekor. A program kódja ilyenkor a memóriában marad.

Az ilyen programokat megszakítások, vagy hardveresemények élesztheti fel. Szokás ezeket TSR-nek (Terminate and Stay Resident) is hívni.

A programunkat rezidenssé tehetjük a 27h-s megszakítás hívásával a program végén. Ez a “terminate but stay resident” hívás, viszont így a programunk nem adhat vissza kilépési kódot. MS-DOS 2.0-tól kezdve viszont lehetőségünk van a 21h-s megszakítást használni a rezidenssé tételhez, és ez engedi a kilépési kódot is. Viszont ehhez AH-ba a 31h-s kódot, AL-be pedig a kilépési kódot kell elhelyeznünk, valamint DX-ben meg kell adnunk a rezidenssé teendő terület méretét 16 bájtos egységekben (paragrafusokban) a PSP szegmensétől számítva. A memóriarezidens program “kilépés” után megszakításokkal juthat szóhoz.

Ezek a programok általában két részből állnak:

- Inicializációs rész: a kilépés előtt egy vagy több megszakítást átirányít a memóriában maradó programrészre (a rezidens részre!)
  Megszakítások átirányítására a DOS kínál lehetőséget: ha AH-ba 25h-t teszünk és hívjuk a 21h-s megszakítást, akkor az AL-ben megadott számú megszakítás vektorát a DS:DX által leírt címre állítja be.
  Ha pedig AH-ba 35h-t teszünk, akkor az AL számú megszakítás-vektor értékét ES:BX-ben adja vissza. Bármilyen megszakítást átirányíthatunk.
  Az inicializációs rész futásának idejére le kell tiltani a megszakításokat (CLI), majd a végén engedélyezni (STI).
- Rezidens rész: ez a rész kapja meg a vezérlést az átirányított megszakítások hívásakor. Befejeztekor a vezérlés visszatér oda, ahol a megszakítást hívták, de ehhez az IRET utasításra van szükség.

Milyen logikai műveletek állnak rendelkezésünkre a 8086-os processzorban? Melyik műveletet milyen célra használjuk? Mutasson példákat!

- AND op1, op2 ; bitenkénti logikai és művelet, az eredmény op1-ben tárolódik
  Az és művelettel maszkolhatunk (tetszőleges biteket nullázhatunk). A maszkolás

egyik gyakori használati esete az, mikor karakterből numerikus értéket szeretnénk kinyerni (0-9 karakterek esetén), például olyankor, mikor számokat kérünk be a felhasználótól.

- OR op1, op2 ; bitenkénti logikai vagy művelet, az eredmény op1-ben tárolódik
  A vagy művelettel egy regiszter tetszőleges bitjeit 1-re állíthatjuk.
- XOR op1, op2 ; bitenkénti kizáró-vagy művelet, az eredmény op1-ben tárolódik
  A kizáró-vagy kézenfekvő megoldás egy regiszter nullázására. Ha egy regisztert önmagával “kizáró-vagyolunk”, akkor nullázhatjuk.
- NOT op ; op bitjeit megfordítja
  A bitek megfordítása egyes komplemenst eredményez, ezért ha NOT-ot alkalmazunk egy bitsorozaton, majd hozzáadunk 1-et, akkor kettes komplemenst kaphatunk (negatív számok reprezentációja).
- TEST op1, op2 ; bitenkénti logikai és művelet, az eredményt eldobja, a FLAG-ek beállnak
  Hasznos akkor, ha meg akarjuk vizsgálni azt, hogy egy numerikus értékünk páros-e (1-gyel végzünk és műveletet). Ilyenkor, ha nem 0 az eredmény, akkor ugorhatunk (jnz).

triviális példák kóddal ide

13. Mutassa be az XT megszakítási rendszerét! Milyen módon lehet megszakítás-kezelő programot írni DOS operációs rendszerben?

Megszakítás: az aktuális folyamatot a CPU felfüggeszti, új tevékenység elvégzése, majd visszatérés és a program folytatása. Az XT-k 8 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 férhet hozzá.
Beszélhetünk hardveres, szoftveres, illetve külső és belső megszakításokról.
Megszakítások fajtái prioritások szerint:

1. Reset (nyomógomb)
   Az utasítás végrehajtása közben is félbeszakítja a processzor működését és a CPU alaphelyzetbe áll.
2. NMI (Non Maskable Interrupt)
   Nem maszkolható (nem tiltható) megszakítás: utasítás befejezése, regiszterek mentése és ugrás. Általában kritikus eseményeknél használják (pl. áramkimaradásnál RAM mentése háttértárra). Az IBM PC-ken RAM, I/O paritáshiba, vagy a koprocesszor hatására keletkezik ilyen.
3. IRQ (Interrupt Request)
   Megszakítás-kérés: hardver megszakítás, külső eszköz kérése. Csak akkor érvényesül, ha a megszakítás jelzőbit engedélyezi (IF=1, de ez nem vonatkozik az NMI-re)
4. INT
   Szoftver megszakítás: Assembly utasítás (INT 0..255), amellyel a program tetszőleges helyén ráugorhatunk a megszakítást kiszolgáló rutinra (amit a

hardveregység hív IRQ-val, vagy más által megírt program, pl. a ROM-BIOS-ban). Ezek a program futása során szinkron keletkeznek (hiszen egy gépi kódú utasítás váltja ki őket), és nem maszkolhatók.

A megszakítások kezeléséhez a CPU egy speciális rutint (programrészt) hajt végre a megszakítás detektálása után. Ezt a rutint hívjuk megszakítás-kezelőnek (interrupt-handler). Megszakítás-kezelő program megvalósításához memóriarezidens programot kell készíteni (lásd: 12-es első kérdés). Azokat a megszakításokat, amelyeket kezelni szeretnénk, átirányítjuk ennek a programnak a rezidens rutinjára, majd ha végeztünk a kezeléssel, vissztérünk. Fontos megjegyezni, hogy megszakítás-kezelő programoknál, ha bizonyos regisztereket felhasználunk, akkor azokat kötelesek vagyunk először menteni a STACK-re, a FLAG-ekkel együtt, majd ha végeztünk, akkor visszatérés előtt azokat újra kiolvasni. Ezzel biztosítjuk, hogy nem veszítünk el fontos adatokat.
Kicsit “naívabb” megoldás, ha először MINDENT beteszünk a verembe (PUSHA), majd mindent kiolvasunk. Ez abból a szempontból előnyös, hogy ha változtatunk a feladaton, akkor is jól fog működni.

Az előjel nélküli egészek kezelését ez előjeles egészek kezelésétől mi különbözteti meg az Assembly használata során? Mutasson példákat is!
Mivel a számokat binárisan ábrázoljuk, ezért előjel nélkül n db. biten 2^n különböző számot tudunk ábrázolni (0-tól 2^n - 1-ig). Előjelesen pedig -2^(n-1), illetve +2^(n-1)-1-ig.
Bizonyos műveletek/utasítások esetében máshogy kell eljárni az előjeles és az előjel nélküli számok tekintetében. Ilyen például a szorzás/osztás, vagy a jobbra shiftelés.
Ezekből megkülönböztethetünk előjeles, illetve előjej nélküli változatokat.
Szorzáshoz a MUL (előjel nélküli), illetve az IMUL (előjeles) utasítást használhatjuk.
Formájuk: MUL/IMUL op
Ha az operandus 8 bites, akkor AL-t szorozzuk op-val, és az eredmény AX-be kerül.
Ha 16 bites, akkor AX-et szorozzuk op-val, és az eredmény DX:AX-be kerül.
Az előjel a szorzó és a szorzandó előjelbitjeinek logikai kizáró vaggyal történő összekapcsolásával áll elő.
Osztáshoz pedig a DIV (előjel nélküli), és az IDIV (előjeles) szükséges.
Formájuk: DIV/IDIV op
Ha az operandus 8 bites, akkor AX-et osztjuk op-val, a hányados AL-be, a maradék pedig AH-ba fog kerülni.
Ha az operandus 16 bites, akkor DX:AX-et osztjuk op-val, a hányados AX-be, a maradék pedig DX-ben foglal helyet.
Az előjel eldöntése ugyanúgy történik, mint a szorzásnál. A maradék pedig megkapja az osztandó előjelét.
Mikor az előjel nélküli jobbra shiftelést használjuk, akkor az előjelet elveszíthetjuk, hiszen 0 jön be a bal oldalon. Ha az előjelbitünk 1 volt, akkor ezt meg kell tartani, erre lett kitalálva az előjeles jobbra shiftelés (SAR). Balra shiftelésnél is 2 ilyen utasításunk van, viszont azok ugyanazok az utasítások 2 különböző névvel. Ott egyébként sem áll fenn ez a probléma.

14. Milyen módon lehet használni a STACK-et a 8086-os processzorban? A használati elemeket példákkal ismertesse?

Adatokat, indexet, értékeket lehet tárolni benne. Maximális mérete 64kB. A verem szegmensét az SS regiszter tárolja, míg a verem tetejére az SP mutat.
Push és pop utasítások értelmezhetőek rajta.
Push : Csak 16 bites értékeket tud a stackre tenni (pl:AX,BX). Működése során elmozdítja az SP értékét 16 bittel, 2-vel lecsökkentve az értékét. Majd a stack tetejére teszi a 16 bites értéket. Így az SP azt mutatja meg, hogy hová tettük be (SS:SP) a legutolsó értéket.
Pop: Ugyan ez csak kiveszi az értéket az SS:SP helyről, majd megnöveli az SP értékét kettővel.
Pushf és **Popf** utasítással a FLAGS regisztert lehet a stackre tenni.
8 bites pop és push nincs. Az SP pointer értékének az a jó, ha páros, mert ha páratlan értékre teszünk be egy 16 bites értéket az lassítja a procit, illetve 8086-osnál ez nem is lehetséges.

Példa: Ha az AX=1212H, BX=3434H, CX=5656H, SS értéke nem számít, az SP=0100H.

1) PUSH AX
2) PUSH BX
3) POP AX
4) PUSH CX
5) POP BX
6) POP CX

0. esemény: SP=> SS:0100H:?? (kérdőjel= nem tudjuk milyen érték van ott)
1. esemény után: SS:0100H:??
SP=>SS:00FEH:1212H
2. esemény után: SS:0100H:??
SS:00FEH:1212H
SP=>SS:00FCH:3434H
3. esemény után: SS:0100H:??
SP=>SS:00FEH:1212H
SS:00FCH:3434H AX=3434H, BX=3434H, CX=5656H
4. esemény után: SS:0100H:??
SS:00FEH:1212H
SP=>SS:00FCH:56556H AX=3434H, BX=3434H, CX=5656H
5. esemény után: SS:0100H:??
SP=>SS:00FEH:1212H
SS:00FCH:5656H AX=3434H, BX=5656H, CX=5656H
6. esemény után: SP=>SS:0100H:??
SS:00FEH:1212H
SS:00FCH:5656H AX=3434H, BX=5656H, CX=1212H

A különböző típusú számok ábrázolásánál milyen értéktartományok állnak rendelkezésünkre? Hogyan léphetünk egyikből át a másikba?
1 bájtnyi helyen előjelesen [-128,+127] között lehet értékeket ábrázolni, míg előjel nélkül a [0-255] intervallum között. Azaz, ha n biten tárolunk számokat, akkor előjel nélkül 0 és 2^n-1 között, míg előjelesen -2^(n-1) és +2^(n-1)-1 között lehet.

Áttérés egyik típusból a másikba a következőképpen lehet:

- Előjel nélküliből előjelesbe 8 biten:
    - 0-127 között nem kell átalakítás
    - 127 fölött SEHOGY
- Előjel nélküli 8 bitesből előjel nélküli wordbe:
    - Elé teszünk 8 db 0-át :D
- 16 bites előjel nélküliből 8 bites előjelnélkülibe:
    - 255-nél nagyobb esetén SEHOGY
    - 255-nél kisebb esetén lecsapjuk az első 8 bitet
- Előjeles 8 bitesből előjeles 16 bitesbe
    - [-128,+127] között a normál 8 bites első bitjével töltjük fel a felső 8 bitet
        - pl: +1 esetén: 00000000|00000001
        - -1 esetén: 11111111 | 11111111
        - +127: 0000000 | 01111111
        - -127: 11111111 | 10000001
        - -128: 11111111 | 10000000

Az assembly beépített lehetőségeket is ajánl:

- CBW (convert byte to word): Az AL-ben lévő előjeles számot az AX-be teszi előjelesen.
- CWD (convert word to double word): Az AX-ben lévő előjeles 16 bites számot 32 bitessé átalakítja és a DX:AX regiszetekbe teszi.