Számítógép-architektúrák I.

Oktatási segédlet

Tartalomjegyzék

[Alapismeretek 4](#_Toc95157787)

[Az első program 4](#_Toc95157788)

[Adatmozgatás 5](#_Toc95157789)

[MOV reg ← reg, MOV reg ← imm 5](#_Toc95157790)

[MOV reg ← mem, MOV mem ← reg, MOV mem ← imm 5](#_Toc95157791)

[Fixpontos aritmetika 6](#_Toc95157792)

[ADD, ADC: összeadás 6](#_Toc95157793)

[SUB, SBB: kivonás 7](#_Toc95157794)

[MUL, IMUL: szorzás 8](#_Toc95157795)

[DIV, IDIV: osztás 9](#_Toc95157796)

[Kettes komplemens kódú számokhoz: NEG, CBW, CWD 10](#_Toc95157797)

[Logikai műveletek, léptetés, forgatás 10](#_Toc95157798)

[Feltétel nélküli elágazás 12](#_Toc95157799)

[Feltételes elágazások 12](#_Toc95157800)

[Flagek szerint 12](#_Toc95157801)

[A CMP utasítást követően: 13](#_Toc95157802)

[A TEST utasítást követően: (TEST x,y: x AND y → flagek állítása) 13](#_Toc95157803)

[Ciklus-szervezés 14](#_Toc95157804)

[Veremkezelés, eljáráshívás és visszatérés 19](#_Toc95157805)

[Lebegőpontos utasítások 21](#_Toc95157806)

[Egyéb utasítások 26](#_Toc95157807)

[Táblázattal adott függvény (LUT) 26](#_Toc95157808)

[Állapotbitek írása/olvasása 26](#_Toc95157809)

[Feladatok és megoldások 28](#_Toc95157810)

[Elemi feladatok 28](#_Toc95157811)

[Adatok összege 30](#_Toc95157812)

[Negatív számok összege 30](#_Toc95157813)

[Maximumkeresés (előjel nélküli egész) 31](#_Toc95157814)

[ASCIIZ string hossza 32](#_Toc95157815)

[Az első érvényes karakter megkeresése ASCIIZ string-ben 32](#_Toc95157816)

[ASCII-7 karakter kibővítése páros paritásbittel 33](#_Toc95157817)

[ASCIIZ-HEX konverzió 33](#_Toc95157818)

[Pakolatlan BCD-HEX konverzió 34](#_Toc95157819)

[HEX-pakolatlan BCD konverzió 35](#_Toc95157820)

[Vegyes feladatok 36](#_Toc95157821)

[A Függelék: gyakorló feladatsor 39](#_Toc95157822)

[B Függelék: utasítás-csoportok 41](#_Toc95157823)

[a) Általános célú utasítások 41](#_Toc95157824)

[b) Fixpontos aritmetika 42](#_Toc95157825)

[c) BCD aritmetika 42](#_Toc95157826)

[d) Logikai műveletek 42](#_Toc95157827)

[e) Léptetés, forgatás 43](#_Toc95157828)

[f) Bit- és byte műveletek 43](#_Toc95157829)

[g) Vezérlésátadás, ciklusszervezés 43](#_Toc95157830)

[h) Sztring-műveletek 44](#_Toc95157831)

[i) B/K utasítások 45](#_Toc95157832)

[j) Az állapotregiszterre vonatkozó utasítások 45](#_Toc95157833)

[k) Egyéb utasítások 45](#_Toc95157834)

[l) FPU adatmozgatás 46](#_Toc95157835)

[m) FPU aritmetika 47](#_Toc95157836)

[n) FPU összehasonlítás 47](#_Toc95157837)

[o) FPU szinusz, koszinusz, logaritmus… 48](#_Toc95157838)

[p) Beépített konstansok 48](#_Toc95157839)

[q) Az FPU működési környezete 48](#_Toc95157840)

[C Függelék: IA32 gépi utasítás-végrehajtás vizsgálata Visual Studio-val 51](#_Toc95157841)

# Alapismeretek

Mikroszámítógép: mikroprocesszor, sínrendszer, memória, perifériák

Mikroprocesszor: CPU: címsín, adatsín, vezérlősín

* CPU: regiszterek, ALU, belső sín
* Emulátor, szimulátor
* Programozási modell
  + programterület, adatterület, veremterület, B/K (memória területre leképezett / dedikált)
  + utasítás-végrehajtás: **utasítás-mutató (EIP, IP)**, (programszámláló (PC))
  + utasításkészlet
  + regiszterkészlet
    - EAX, AX, AH, AL
    - EBX, BX, BH, BL
    - ECX, CX, CH, CL
    - EDX, DX, DH, DL
    - ESI, SI
    - EDI, DI
    - ESP, SP
    - EBP, BP
  + állapotjelzők
    - ZF (ZR)
    - CF (CY)
    - SF (PL)
    - OF (OV)
    - AF (AC)
    - PF (PE)
* Programozás assembly nyelven, \_asm{}
* Fixpontos bináris számábrázolás (bináris, hexadecimális, decimális)
* bit, B, W, DW, QW

## Az első program

|  |
| --- |
| // elso\_asm\_prg.cpp : This file contains the 'main' function. Program execution begins and ends there.  //  //#include <iostream>  #include <stdio.h>  int main()  {  int x;  char ch;  //  x = 1;  \_asm {  mov eax, x;  add eax, eax;  mov x, eax;  }  printf("x erteke: %d\n", x);  //  printf("A befejezeshez nyomjon meg egy billentyut...\n");  ch = getchar();  //std::cout << "Hello World!\n";  } |

# Adatmozgatás

## MOV reg ← reg, MOV reg ← imm

* **Általános célú regiszter** operandus nevek, méretek
* **GPR**: EAX/AX/AH/AL, EBX/BX/BH/BL, ECX/CX/CH/CL, EDX/DX/DH/DL
* **Indexregiszter**ek: ESI/SI, EDI/DI
* Lépésenkénti utasítás-végrehajtás VS **regiszter ablak**kal

|  |
| --- |
| \_asm {  mov al,0xff;  mov ch, al;  mov edx, 0xabcdef01;  mov esi, edx;  mov di, si;  nop;  } |

## MOV reg ← mem, MOV mem ← reg, MOV mem ← imm

* Memória-operandus típusok, méretek:
  + char, unsigned char
  + short int, unsigned short int
  + int, unsigned int
* VS **watch ablak**, hexedecimal/decimal
* Az **operandus címe**, az **& operátor** a watch ablakban
* Lépésenkénti utasítás-végrehajtás VS **memória-ablak**kal

|  |
| --- |
| // char ch;  printf("Adattipusok merete byte-ban: char:%d\tshort int:%d\tint: %d\n",sizeof(char),sizeof(short int),sizeof(int));  // printf("A folytatashoz nyomjon meg egy billentyut...");  // ch = getchar();  //  //--------------------------------------------------  //  /\*  MOV r/mem,imm  MOV r8/mem | r16,mem | r32,mem  MOV mem,r8 | mem,r16 | mem,r32  MOV r8,r8 | r16,r16 | r32,r32  \*/  char i8;  unsigned char u8;  short int i16;  unsigned short int u16;  int i32;  unsigned int u32;  //  \_asm{  //!!watch window hexedecimal/decimal display  mov i8,0xBF;  mov u8,0x3E;  mov i16,0xFFBF;  mov u16,0x003E;  mov i32,0xFFFFFFBF;  mov u32,0x0000003E;  //  mov cl,i8;  mov dx,u16;  mov esi,i32;  } |

# Fixpontos aritmetika

## ADD, ADC: összeadás

* Bináris összeadás
* **Állapotbitek** és a flag regiszter (**regiszter ablak**, EFL, **Flags**)
* Állapotbitek és számábrázolás, Intel/MS elnevezések:
  + Előjel nélküli egész számok: átvitelbit (**CF/CY**, carry flag), zérusjelző (**ZF/ZR**, zero flag)
  + Kettes komplemens kódú számok: átvitelbit (CF, carry flag), zérusjelző (ZF, zero flag), előjelbit (**SF/PL**, sign flag), túlcsordulás jelző bit (**OF/OV**, overflow flag)
* „Nagyméretű” számok összeadása „kisméretű” operandussal működő utasításokkal
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal

|  |  |
| --- | --- |
| /\*  ADD dest,src ;dest <- dest +src  ADD AL,imm8 ADD r/m8,imm8  ADD AX,imm16 ADD r/m16,imm16  ADD EAX,imm32 ADD r/m32,imm32  ADD r/m8,r8  ADD r/m16,r16  ADD r/m32,r32  \*/  \_asm{  mov al,0x02;  add al,u8;  mov ax,0x1000;  add ax,i16;  mov eax,0x00000001;  add eax,u32;  mov ecx,0;  add ecx,i32;  add al,cl;  add ax,cx;  add eax,ecx;  }  /\*--------------------------------------------------  ADC dest,src ;dest <- dest + src + CF  (ADD, ill. carry flag: STC CLC CMC)  ADC AL,imm8 ADC r/m8,imm8  ADC AX,imm16 ADC r/m16,imm16  ADC EAX,imm32 ADC r/m32,imm32  ADC r/m8,r8  ADC r/m16,r16  ADC r/m32,r32  !sign extension: ADD/ADC r/m16,imm8  !sign extension: ADD/ADC r/m16,imm8  !op. értelmezés: signed/unsigned => CF SF OF [ZF AF PF]  \*/  \_asm{  add ax,i16;  adc ax,cx;  stc;  adc ah,cl;  } |  |

## SUB, SBB: kivonás

* Bináris kivonás
* Állapotbitek és számábrázolás:
  + Előjel nélküli egész számok: **kölcsönbit** (**borrow flag**, IA32 esetén a CF-ben tárolva), zérusjelző (ZF, zero flag)
  + Kettes komplemens kódú számok: átvitelbit (CF, carry flag), zérusjelző (ZF, zero flag), előjelbit (SF, sign flag), túlcsordulás jelző bit (OF, overflow flag)
* „Nagyméretű”, előjel nélküli egész számok kivonása „kisméretű” operandussal működő utasításokkal
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal

|  |
| --- |
| /\*  SUB dest,src ;dest <- dest - src  SUB AL,imm8 SUB r/m8,imm8  SUB AX,imm16 SUB r/m16,imm16  SUB EAX,imm32 SUB r/m32,imm32  SUB r/m8,r8  SUB r/m16,r16  SUB r/m32,r32  --------------------------------------------------  SBB dest,src ;dest <- dest - (src + CF)  SBB AL,imm8 SBB r/m8,imm8  SBB AX,imm16 SBB r/m16,imm16  SBB EAX,imm32 SBB r/m32,imm32  SBB r/m8,r8  SBB r/m16,r16  SBB r/m32,r32  \*/  unsigned short int u16;  u16 = 0x0001;  \_asm {  mov al, 0x20;  sub al, 0x11;  //  mov al, 0;  sub al, 1;  //  mov ax, 2;  mov cx, ax;  sub ax, u16;  sbb ax, cx;  //  mov ah, 0;  mov cl, 0;  stc;  sbb ah, cl;  //  nop;  } |

## MUL, IMUL: szorzás

* Bináris szorzás
* Utasítások és számábrázolás:
  + **MUL**: előjel nélküli egész számok szorzása
  + **IMUL**: kettes komplemens kódú számok szorzása
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal

|  |
| --- |
| /\* MUL  MUL r/m8 ;AX <- AL\*r/m8  MUL r/m16 ;DX:AX <- AX\*r/m16  MUL r/m32 ;EDX:EAX <- EAX\*r/m32  Számolás: op1 N bites => MUL min, max  \*/  unsigned char u8;  unsigned short int u16;  unsigned int u32;  u8 = 0x08;  u16 = 0x0101;  u32 = 200;  \_asm {  mov al, 0x03;  mul u8;// AX <- szorzat  mov ax, 0x0004;  mul u16;// DX:AX <- szorzat  mov eax, 0x00000005;  mul u32;// EDX:EAX <- szorzat  }  /\* IMUL  IMUL r/m8 ;AX <- AL\*r/m8  IMUL r/m16 ;DX:AX <- AX\*r/m16  IMUL r/m32 ;EDX:EAX <- EAX\*r/m32  Számolás: op1 N bites => IMUL min, max  \*/  char i8;  short int i16;  int i32;  i8 = -8;  i16 = -257;  i32 = -200;  \_asm {  mov al, 0x03;  imul i8;// AX <- szorzat  mov ax, 0x0004;  imul i16;// DX:AX <- szorzat  mov eax, 0x00000005;  imul i32;// EDX:EAX <- szorzat  nop;  } |

## DIV, IDIV: osztás

* Bináris osztás
* Utasítások és számábrázolás:
  + **DIV**: előjel nélküli egész számok szorzása
  + **IDIV**: kettes komplemens kódú számok osztása
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal
* **Osztás 0-val**?

|  |
| --- |
| /\*  DIV  DIV r/m8 ;AX / r/m8 => AH | AL = maradék | hányados  DIV r/m16 ;DX:AX / r/m16 => DX:AX = maradék:hányados  DIV r/m32 ;EDX:EAX / r/m32 => EDX:EAX = maradék:hányados  Különleges: 0-val osztás, nagy szám osztása 1-gyel  \*/  unsigned char u8;  unsigned short int u16;  unsigned int u32;  u8 = 0x80;  u16 = 0x0101;  u32 = 200;  \_asm {  mov ax, 0x1001;  div u8;//AH|AL <- maradék | hányados  mov dx, 0x0020;  mov ax, 0xf211;  div u16;//DX:AX <- maradék:hányados  mov edx, 0x00000011;  mov eax, 0x1000f211;  div u32;//EDX:EAX <- maradék:hányados  }  /\*  IDIV  IDIV r/m8 ;AX / r/m8 => AH | AL = maradék | hányados  IDIV r/m16 ;DX:AX / r/m16 => DX:AX = maradék:hányados  IDIV r/m32 ;EDX:EAX / r/m32 => EDX:EAX = maradék:hányados  Különleges: 0-val osztás, nagy szám osztása 1-gyel  \*/  char i8;  short int i16;  int i32;  i8 = -108;  i16 = -257;  i32 = -200;  \_asm {  mov ax, 0x1000;  idiv i8;// AH|AL maradék | hányados  mov dx, 0x0001;  mov ax, 0xf211;  idiv i16; //DX:AX <- maradék:hányados  mov edx, 0x00000011;  mov eax, 0x1000f211;  idiv i32; //EDX:EAX <- maradék:hányados  //!  /\*mov bl,0x00;  mov ax,0x0001;  idiv bl;\*/  nop;  } |

## Kettes komplemens kódú számokhoz: NEG, CBW, CWD

* **Algoritmus** egy szám (-1)-szeresének előállítására
* Az operandus méretének növelése: **előjel-kiterjesztés**
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal

|  |
| --- |
| /\*NOT | NEG r / m8 | r / m16 | r / m32  CBW; AX <-sign - extend of AL.  CWDE; EAX <-sign - extend of AX.  \*/  unsigned char u8;  char i8;  u8 = 0xA5;  i8 = 0xA5;  \_asm {  MOV AH, u8;  NOT AH;  INC AH;  MOV AL, i8;  NEG AL;  MOV AL, i8;  CBW;//AX <- sign extended AL  CWD;//DX:AX <- sign extended AX  CWDE;//EAX <- sign extended AX  //  nop;  } |

# Logikai műveletek, léptetés, forgatás

* **Logikai függvény** algebrai alakja (minterm), **igazságtáblázat**
* **Funkcionálisan teljes rendszer**: NOT, AND, OR
* Kizáró vagy (XOR)
* Adott bit 1-be, 0-ba írása és invertálása
* Maszkolás
* Léptetés, forgatás: kilépő és belépő bitek
* Léptetés, forgatás: egy bittel/egyszerre több bittel (CX, ECX)
* Léptetés balra
* Aritmetikai és logikai léptetés jobbra
* A **paritás jelzőbit** (**PF/PE**, parity flag, e: even)
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal/állapot regiszterrel és memória-ablakkal

|  |
| --- |
| char i8;  short int i16;  int i32;  /\* AND OR XOR  AND|OR|XOR dest,src ;dest <- dest AND|OR|XOR src  AND|OR|XOR AL,imm8 AND|OR|XOR r/m8,imm8  AND|OR|XOR AX,imm16 AND|OR|XOR r/m16,imm16  AND|OR|XOR EAX,imm32 AND|OR|XOR r/m32,imm32  AND|OR|XOR r/m8,r8  AND|OR|XOR r/m16,r16  AND|OR|XOR r/m32,r32  \*/  i8 = 0xC2;  i16 = 0xA5A5;  i32 = 0x7F7F7F7F;  \_asm {  and i8, 0x0F;  or i16, 0x5A5A;  mov eax, 0xF7F7F7F7;  xor eax, i32;  }  /\* NOT  NOT r/m8  NOT r/m16  NOT r/m32  \*/  \_asm {  mov ebx, 0x55555555;  not ebx;  }  /\*  SAL op <- 2\*op; CF <- MSB  SHL op <- 2\*op; CF <- MSB  SAR op <- op / 2 (signed); CF <- LSB  SHR op <- op / 2 (unsigned); CF <- LSB  SAL|SHL|SAR|SHR r/m8,1 SAL|SHL|SAR|SHR r/m8,CL SAL|SHL|SAR|SHR r/m8,imm8  SAL|SHL|SAR|SHR r/m16,1 SAL|SHL|SAR|SHR r/m16,CL SAL|SHL|SAR|SHR r/m16,imm8  SAL|SHL|SAR|SHR r/m32,1 SAL|SHL|SAR|SHR r/m32,CL SAL|SHL|SAR|SHR r/m32,imm8  RCL|RCR|ROL|ROR: rotate; C: includes CF in rotation  RCL|RCR|ROL|ROR r/m8,1 RCL|RCR|ROL|ROR r/m8,CL RCL|RCR|ROL|ROR r/m8,imm8  RCL|RCR|ROL|ROR r/m16,1 RCL|RCR|ROL|ROR r/m16,CL RCL|RCR|ROL|ROR r/m16,imm8  RCL|RCR|ROL|ROR r/m32,1 RCL|RCR|ROL|ROR r/m32,CL RCL|RCR|ROL|ROR r/m32,imm8  \*/  \_asm {  mov al, 0x01;  shl al, 1;  mov bx, 0x8000;  sar bx, 2;  ror bx, 4;  mov edx, 0x7F000000;  mov cl, 5;  rcl edx, cl;  }  /\*  Parity Flag  XOR r,r  Adott bit 0-ba, 1-be írása, invertálása (maszkolás...)  \*/  \_asm {  mov eax, 0xC2F3D4E5;  xor eax, eax;  mov ax, 0xFFFF;  and ax, 0x1000;  or ax, 0x8000;  xor ax, 0x0001;  } |

# Feltétel nélküli elágazás

* Elágazás (branch) vagy ugrás (jump)?
* **JMP** cím
* **Folyamatábra**-részlet
* **Címke**
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal, az **EIP regiszter**

|  |
| --- |
| \_asm {  mov eax, 1;  mov eax, 2;  jmp c1;  mov eax, 3;  c1: mov eax, 4;  jmp c2;  mov eax, 5;  c2: jmp c3;  mov eax, 6;  c3: nop;  } |

# Feltételes elágazások

## Flagek szerint

* CF: JC, JNC
* SF: JS, JNS
* OF: JO, JNO
* PF: JE, JO
* ZF: JZ/JE, JNZ/JNE
* Folyamatábra-részlet programkódból
* Programkód folyamatábra alapján
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal, az **EIP regiszter**

|  |
| --- |
| //OR: OF <- 0; CF <- 0; SF, ZF, PF értéke az eredménytől függ; AF nem definiált  //AND: OF <- 0; CF <- 0; SF, ZF, PF értéke az eredménytől függ; AF nem definiált  //XOR: OF <- 0; CF <- 0; SF, ZF, PF értéke az eredménytől függ; AF nem definiált  \_asm {  L1: stc;//CF <- 1  L2: jc L4;  L3: jnc L7;  L4: mov al, 0xff;  L5: or al, al;//SF <- 1, CF <- 0  L6: js L3;  L7: mov al, 0x80;  L8: mov bl, 0x81;  L9: add al, bl;//OF <-1  L10: jo L12;  L11: xor eax, eax;  L12: nop;  } |

## A CMP utasítást követően:

* **CMP** x,y: x-y → flagek állítása
* **Előjel nélküli egész** számok esetében:
  + >: JA
  + >=: JAE
  + <: JB
  + <=: JBE
  + =: JE
  + ~=: JNE
* **Kettes komplemens kódú** számok esetében:
  + >: JG
  + >=: JGE
  + <: JL
  + <=: JLE
  + =: JE
  + ~=: JNE
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal, az **EIP regiszter**

|  |
| --- |
| //CMP x,y: x-y -> flagek  //e.n.e: JA, JAE, JB, JBE, JE, JNE  //2kk: JG, JGE, JL, JLE, JE, JNE  \_asm {  //e.ne.  mov al, 0x80;  cmp al, 0x7f;  ja L1;//elágazik, mert 0x80 > 0x7f (128 > 127)  xor al, al;  L1: cmp al, 0x7f;//AL = 0x80  jg L2;//nem ágazik el, mert 0x80 < 0x7f (-128 < 127)  xor al, al;//AL <- 0  L2: nop;  } |

## A TEST utasítást követően: (TEST x,y: x AND y → flagek állítása)

* JZ/JE, JNZ/JNE
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal, az **EIP regiszter**

|  |
| --- |
| //TEST pl.  //b7 b6 b5 b4 b3 b2 b1 b0 AND 0 1 0 0 0 0 0 0: 0 b6 0 0 0 0 0 0  //Ha b6 == 0, ZF <- 1; ha b6 == 0, ZF <- 0  unsigned char b6;  b6 = 0xff;  \_asm {  mov al, 0xcd;//0xcd 0xbd  test al, 0x40;//0xcd: 11001101 AND 01000000 => b6 = 1 => ZF = 0  jz b6\_0;//elágazik, ha b6 értéke 0  mov b6, 1;//  jmp vege;  b6\_0: mov b6, 0;  vege: nop;  } |

# Ciklus-szervezés

* **Ciklus**, **ciklusmag**
* **Ciklusszámláló**, kezdő érték, végérték
* Elöltesztelő, hátultesztelő ciklus
* Amikor a ciklus-számláló regiszter
* Az **ECX/CX** regiszter és a **LOOP** utasítás
* Lépésenkénti utasítás-végrehajtás VS regiszter-ablakkal, az **EIP regiszter**

Valamely algoritmus rendszeresen ismétlődő részét cikussal lehet megvalósíthatni.

Példáinkban számok összegét állítjuk elő ciklussal. Az alábbi folyamatábra egy lehetséges megoldást mutat (C\_SZ a ciklus-számláló, N az összeadandó számok darabszáma).

|  |
| --- |
| **START**  **C\_SZ ← N**  **IGEN**  **Összeg ← 0**  **NEM**  **STOP**  **C\_SZ = 0?**  **Összeg ← Összeg + következő szám**  **C\_SZ ← C\_SZ - 1** |

Ezt hátultesztelő ciklusnak nevezzük. A fenti megoldásban a ciklus-számláló értékét csökkentettük.

Ha a ciklus-számláló kezdeti értéke 0, akkor a vizsgálat előtt növelni kell az értékét 1-gyel, és a feltételvizsgálatnál azt kell ellenőrizni, hogy elérte-e a ciklus-számláló N értékét?

Megoldható a feladat elöltesztelő ciklussal is:

|  |
| --- |
| **START**  **C\_SZ ← N**  **IGEN**  **Összeg ← 0**  **NEM**  **STOP**  **Összeg ← Összeg + következő szám**  **C\_SZ ← C\_SZ - 1**  **C\_SZ = 0?** |

Ebben a megoldásban a ciklus-számláló értékét csökkentettük. Ha a ciklus-számláló kezdeti értéke 0, akkor a feltételvizsgálatnál azt kell ellenőrizni, hogy elérte-e a ciklus-számláló N értékét, és a vizsgálat után növelni kell az értékét 1-gyel.

Mindkét fenti példában a ciklusmagnál feltételeztük, hogy a következő szám rendelkezésre áll az összeadáshoz. Ezt egyszerűen megoldhatjuk, ha **a számok egy tömb elemei**, mert ekkor az **index** segítségével érhetők el. Gondoskodni kell az index kezdeti értékének beállításáról még a ciklus előtt, illetve a ciklusmagban arról, hogy a következő tömbelemet indexelje.

Összefoglalva a lehetőségeket:

* a ciklus lehet elöltesztelő vagy hátultesztelő
* a ciklusból való kilépéshez a ciklusváltozó végértéke lehet 0 vagy a számok darabszáma
* a tömbelemeket elérhetjük az index növelésével vagy csökkentésével.

Az alábbi folyamatábrán **hátultesztelő ciklus** esetében látható az index használata. Feltesszük még, hogy a számok a **tmb** tömbben vannak.

|  |
| --- |
| **START**  **C\_SZ ← N**  **IGEN**  **Összeg ← 0**  **NEM**  **STOP**  **C\_SZ = 0?**  **Összeg ← Összeg + tmb[index]**  **C\_SZ ← C\_SZ - 1**  **index ← 0**  **index ← index + 1** |

Ezt a változatot valósítja meg a következő programrészlet. Itt az összegzendő számok 32 bites előjel nélküli egészek. A ciklus-számláló az EDX, az index az EBX, összeg az EAX regiszterben lesz.

|  |
| --- |
| unsigned int tmb[7] = { 1,2,3,4,5,6,7 };  unsigned int osszeg = 0;  unsigned int N = 7;  \_asm {  mov edx, N;//ciklus-számláló  mov ebx, 0;//a következő tömbelem indexe  mov eax, 0;//összeg  //  CIKL: add eax, tmb[ebx];//a következő 32 bites (DW) szám hozzáadása  add ebx, 4;//a következő tömbelem indexe  //  dec edx;//a ciklus-számláló csökkentése 1-gyel  jnz CIKL;//a hozzáadás folytatása...  //  mov osszeg, eax;//a ciklus vége: az összeg tárolása  } |

Gondoljuk meg, hogyan változik a program, ha a számok 16 illetve 8 bitesek!

A következő programrészlet ugyancsak számok összegét állítja elő, de **elöltesztelő ciklus**sal és úgy, hogy **a ciklus-számláló kezdeti értéke 0**. Ebben az esetben az összegzendő számok 16 bites előjel nélküli egészek, az összeg 32 bites. Ennek az a következménye, hogy a hozzáadás előtt **a 16 bites számot 32 bitessé kell alakítani**; erre a célra a CX/ECX regisztereket használjuk.

|  |
| --- |
| unsigned short int tmb[7] = { 1,2,3,4,5,6,7 };  unsigned int osszeg = 0;  unsigned int N = 7;  \_asm {  mov edx, 0;//ciklus-számláló  mov ebx, 0;//a következő tömbelem indexe  mov eax, 0;//összeg  //  CIKL: cmp edx, N;//ciklus-számláló végértékének ellenőrzése  je CIKL\_VEGE;  inc edx;//ciklus-számláló növelése  //  xor ecx, ecx;//a következő e.n.e. szám előkészítése  mov cx, tmb[ebx];//a következő 16 bites szám beolvasása  add eax, ecx;//a következő szám hozzáadása a 32 bites összeghez  add ebx, 2;//a következő 16 bites (W) tömbelem indexe  //  jmp CIKL;//a hozzáadás folytatása...  CIKL\_VEGE: mov osszeg, eax;//a ciklus vége: az összeg tárolása  nop;  } |

Az IA32 esetén **a ciklus-szervezéshez külön utasítások is vannak**, amik feltételezik, hogy **a ciklus-számláló az ECX regiszter**:

* **LOOP**: csökkenti ECX értékét, és ha nem nulla, elágazik a címke szerinti címre
* LOOPE: csökkenti ECX értékét, és elágazik, ha ez nem nulla és ZF = 1
* LOOPNE: csökkenti ECX értékét, és elágazik, ha ez nem nulla és ZF = 0

Ide tartozik még a **JECXZ** utasítás is, ami elágazik, ha ECX 0.

Nézzük most a 8 bites előjeles (kettes komplemens kódú) számok összegzését!

|  |
| --- |
| char tmb[7] = { -1,2,-3,4,-5,6,-7 };  int osszeg = 0;  unsigned int N = 7;  \_asm {  mov ecx, N;//ciklus-számláló  mov ebx, 0;//a következő tömbelem indexe  mov edx, 0;//összeg  //  CIKL: mov al, tmb[ebx];//a következő 8 bites szám beolvasása  cbw;//B -> W  cwde;//W -> DW; itt a szám már 32 bites 2kk  add edx, eax;//a következő szám hozzáadása a 32 bites összeghez  add ebx, 1;//a következő 8 bites (B) tömbelem indexe  //  loop CIKL;//a hozzáadás folytatása...  mov osszeg, edx;//a ciklus vége: az összeg tárolása  nop;  } |

# Veremkezelés, eljáráshívás és visszatérés

* **Verem** (stack), veremszervezés: **LIFO**
* **ESP**/SP
* PUSH reg, mem, bennfoglalt konstans
* PUSHA: AX, CX, DX, BX, eredeti SP, BP, SI, DI
* PUSHAD: EAX, ECX, EDX, EBX, eredeti ESP, EBP, ESI, EDI
* PUSHF: EFLAGS alsó 16 bitjét a verembe helyezi
* PUSHFD: EFLAGS-et a verembe helyezi
* POP reg, mem
* POPA: DI, SI, BP, BX, DX, CX, AX visszatöltése a veremből
* POPAD: EDI, ESI, EBP, EBX, EDX, ECX, EAX visszatöltése a veremből
* POPF: EFLAGS alsó 16 bitjét visszatölti a veremből
* POPFD: EFLAGS-et visszatölti a veremből
* **Regiszter mentés/visszatöltés**
* **Egymásba ágyazott ciklusok** ciklus-számlálói
* **ESP-relatív címzés** **EBP**-vel, **paraméterátadás** a vermen
* CALL/RET: **eljáráshívás**, **visszatérés**

|  |
| --- |
| //PUSH POP  int x;  short y;  x = 1;  y = 2;  \_asm{  push x;  push y;  pop bx;  pop eax;  nop;  //  push 0xabcd1234;  pop edx;  nop;  }  //  \_asm{  mov al,1;  call szubr;  mov bl,2;  jmp ugr;  szubr: mov al,2;  ret;  ugr: nop;  }  //  //ESP, EBP  //ESP-relatív címzés  \_asm{  mov eax, 0x12345678;  mov ebx, 0xabcdef01;  push eax;  push ebx;  mov ebp, esp;  mov edx, [ebp];  add ebp, 4;  mov ecx, [ebp];  nop;  } |

Egymásba ágyazott ciklusok megvalósítására nézzünk most példát! Határozzuk meg az alábbi 3x5 méretű mátrix sorainak összegét tartalmazó vektort:



A megoldás: az első sor elemeinek összege 15, a második soré 40, a harmadiké 65.

A feladat egy lehetséges megoldása hátultesztelő ciklusokkal C/C++ nyelven a következő:

|  |
| --- |
| i = 0;  do{  s = 0;  j = 0;  do {  s += tmb2[i][j];  j++;  } while (j < 5);  sor\_ossz[i] = s;  i++;  } while (i < 3); |

Az \_asm{} megoldásban az egymásba ágyazott ciklusok megvalósításához használjuk a vermet! Így mind a külső, mind a belső ciklus változója lehet az ECX regiszter. Ismerni kell még a **kétdimenziós tömb** tárolását a memóriában, ami **sorfolytonosan** történik (a tömbelemek ebben a feladatban 4 byte-osak). Egy lehetséges megoldás a következő:

|  |
| --- |
| int tmb2[3][5] = { {1, 2, 3, 4, 5},{6, 7, 8, 9, 10},{11, 12, 13, 14, 15} };  int sor\_ossz[3];  int i, j, s;  \_asm {  mov esi, 0;//a tmb2 tömb első elemének indexe  mov edi, 0;//a sor\_ossz tömb első elemének indexe  //  mov ecx, 3;//külső ciklusváltozó  KULSO: push ecx;//...tárolása a veremben  mov ecx, 5;//belső ciklusváltozó  //  mov eax, 0;//részletösszeg  BELSO: add eax, tmb2[esi];//a tömbelem hozzáadása a részletösszeghez  add esi, 4;//a tmb2 tömb következő elemének indexe (a tömbelemek 4 byte-osak)  loop BELSO;  mov sor\_ossz[edi], eax;//a sorösszeg tárolása  add edi, 4;//sor\_ossz tömb következő elemének indexe  //  pop ecx;//külső ciklusváltozó  loop KULSO;  } |

A vermen történő paraméterátadásra és C függvényhívásra láthatunk példát a következő programban.

|  |
| --- |
| #include <stdio.h>  int main(void)  {  char format[] = "%s %s\n"; char \* p\_format;  char hello[] = "Hello"; char \* p\_hello;  char world[] = "world"; char \* p\_world;  p\_format = format;  p\_hello = hello;  p\_world = world;  \_asm {  mov eax, p\_world;  push eax;  mov eax, p\_hello;  push eax;  mov eax, p\_format;  push eax;  call printf;  //a verem eredeti állapotába kerüljön...  pop ebx;  pop ebx;  pop ebx;  }  printf("%s %s\n","Hello","world");  return 0;  } |

# Lebegőpontos utasítások

* Az **FPU** működési környezete (lebegőpontos regiszterkészlet/verem, TOS, vezérlő-és állapotregiszter, TAG)
* A nullacímes gép és a **programozási modell**
* **RPN**
* VS regiszter ablak, **Floating point**
* **Adatmozgatás** és **konverzió**
  + FLD fp\_mem: ST(0) ← fp\_mem
  + FLD ST(i): ST(0) ← ST(i)
  + FILD fx\_mem: ST(0) ← fx\_mem
  + FBLD bcd\_mem80: ST(0) ← bcd\_mem80
  + FST fp\_mem: fp\_mem ← ST(0)
  + FST ST(i): ST(i) ← ST(0)
  + FSTP …
  + FIST fx\_mem16, FIST fx\_mem32: ST(0) konverziója egész számra és ennek tárolása a memóriában
  + FISTP fx\_mem16, FISTP fx\_mem32, FISTP fx\_mem64int: ST(0) konverziója egész számra és ennek tárolása a memóriában majd művelet
  + FBSTP bcd\_mem80: bcd\_mem80 ← ST(0) konverzióval majd pop
* A négy **alapművelet**, néhány **matematikai függvény**
  + FADD, FADDP, FIADD
  + FSUB, FSUBP, FISUB
  + FMUL, FMULP, FIMUL
  + FDIV, FDIVP, FIDIV
  + FCHS, FABS, FSQRT, FSIN, FCOS
* Összehasonlítás, feltételes elágazás
  + FICOM fx\_mem16/32, FICOMP fx\_mem16/32: a 16 vagy 32 bites fixpontos memória-értéket duplapontossá alakítja, elvégzi az összehasonlítást ST(0)-lal, az eredményt az FPU állapotszavának bitejeiben adja meg az alábbi táblázat szerint. Nem értelmezhető a reláció például, ha valamelyik operandus NaN.
  + FCOM/FCOMP fp\_mem32/64, FCOM/FCOMP ST(i), : ST(0) és az operandus összehasonlítása, az eredmény az alábbi táblázat szerinti (P: majd pop)
  + FCOM/FCOMP/FCOMPP: ST(0) és ST(1) összehasonlítása, az eredmény az alábbi táblázat szerinti (P: majd pop; PP: majd pop kétszer)

|  |  |  |  |
| --- | --- | --- | --- |
| Reláció | C3 | C2 | C0 |
| ST(0) > operandus | 0 | 0 | 0 |
| ST(0) < operandus | 0 | 0 | 1 |
| ST(0) = operandus | 1 | 0 | 0 |
| nem értelmezhető (unordered) | 1 | 1 | 1 |

* + FTST: ST(0)-t 0.0-val hasonlítja össze, az eredmény az alábbi táblázat szerinti

|  |  |  |  |
| --- | --- | --- | --- |
| Reláció | C3 | C2 | C0 |
| ST(0) > 0.0 | 0 | 0 | 0 |
| ST(0) < 0.0 | 0 | 0 | 1 |
| ST(0) = 0.0 | 1 | 0 | 0 |
| nem értelmezhető (unordered) | 1 | 1 | 1 |

* + FSTSW AX/mem\_w: az FPU állapotszó betöltése AX-be (vagy memória-szóba)  
    AH.6 = C3, AH.2 = C2, AH.0 = C0

|  |
| --- |
|  |

* Beépített **konstansok**:
  + FLD1: ST(0) ← 1.0
  + FLDPI: ST(0) ← π
  + FLDZ: ST(0) ← +0.0
  + FLDL2T: ST(0) ← log2(10)
  + FLDL2E: ST(0) ← log2(e)
  + FLDLG2: ST(0) ← log10(2)
  + FLDLN2: ST(0) ← loge(2)

Gyakorlásképpen kiszámítjuk a kör területét a sugár ismeretében, majd meghatározzuk az elsőfokú egyenelet megoldását, végül szinusz-táblázatot készítünk.

|  |
| --- |
| #include <stdio.h>  #include <math.h> //sin  int main()  {  //kor terulete: t = r\*r\*pi;  float r, pi, tcpp, tasm, tasm1;  double pifpu;  //  r = 2.0;  pi = 3.141526;  tcpp = r\*r\*pi;  printf("Sugar: %3.5f\tTerulet: %3.5f\n", r, tcpp);    \_asm {  FLD r;  FMUL r;  FMUL pi;  FSTP tasm;  }  \_asm {  FLD r;  FMUL r;  FLDPI;  FMUL ST(0), ST(1);  FSTP tasm1;  FLDPI;  FST pifpu;  }  // Elsofoku egyenlet: ax + b = 0; x = -b/a  double a, b, x;  int flag;  flag = 0;//0: nincs megoldas 1: van megoldas (x) 2: minden valos szam megoldas  a = 2.0; b = 3.0;  //a = 0.0; b = 3.0;  //a = 0.0; b = 0.0;  if (a != 0.0) {  //x = -b/a;  //flag = 1;  \_asm {  FLD b;//ST(0) <- b  FCHS;//ST(0) <- -b  FDIV a;//ST(0) <- ST(0) / a (vagyis ST(0) <- -b/a)  FSTP x;  mov flag, 1;  }  }  else if (b == 0.0) {  flag = 2;  }  else {  flag = 0;  }  switch (flag) {  case 0:  printf("a = %f \t b = %f \t x = Nincs megoldas.\n", a, b);  break;  case 1:  printf("a = %f \t b = %f \t x = %f\n", a, b, x);  break;  case 2:  printf("a = %f \t b = %f \t x = Minden valos szam megoldas.\n", a, b);  break;  }  /\*  Szinusztablazat  \*/  float sintabl[10], fi, temp, sintablfpu[10], \*tabl;  int i;  for (i = 0; i<10; i++) {  fi = (float)i;  sintabl[i] = sin(fi);  }  tabl = sintablfpu;  \_asm {  MOV i, 0;  MOV EBX, tabl;  MOV CX, 10;  UGR: FILD i;//(float(i)  FSIN;//sin(float(i));  FSTP temp;//sintablfpu[i] <- sin(fi)  MOV EAX, temp;//emiatt float a pelda...  MOV [EBX], EAX;  ADD EBX, 4;//+4 byte (kovetkezo float elem cime)  INC i;//i++  LOOP UGR;  }  } |

Az alábbi példa a lebegőpontos számok összehasonlítását mutatja be.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main()  {  float x, y;  x = (float) 3.5;  y = (float) 3.5;  int cmp\_flag = -1;  \_asm {  fld y;  fld x;//ST(0) <- x, ST(1) <- y;  fcom;//ST(0) és ST(1) relációja (x >=< y)  fstsw ax;  and ah, 0x45;  cmp ah, 0x45;  je NEM\_ERTELM;  cmp ah, 0x40;  je EGYENLO;  cmp ah, 0x01;  je KISEBB;  cmp ah, 0x00;  je NAGYOBB;  jmp VEGE;  NEM\_ERTELM: mov cmp\_flag, 3;  jmp VEGE;  EGYENLO: mov cmp\_flag, 0;  jmp VEGE;  KISEBB: mov cmp\_flag, 1;  jmp VEGE;  NAGYOBB: mov cmp\_flag, 2;  VEGE: fnop;  }  switch (cmp\_flag) {  case 0:  printf("x = %f, y = %f: x %c y\n",x,y,'=');  break;  case 1:  printf("x = %f, y = %f: x %c y\n", x, y, '<');  break;  case 2:  printf("x = %f, y = %f: x %c y\n", x, y, '>');  break;  case 3:  printf("x = %f, y = %f: x %s y\n", x, y, "nem értelmezett a reláció");  break;  case -1:  printf("HIBA: a cmp\_flag értéke nem változott...");  break;  default:  printf("HIBA: ide nem is kerülhetne a vezérlés...");  break;  }  system("pause");  } |

# Egyéb utasítások

## Táblázattal adott függvény (LUT)

* XLAT, XLATB: AL ← DS:[(E)BX + AL]; AL előjel nélküli egész

|  |
| --- |
| char tabl[] = "ABC"; char \* p\_tabl;  p\_tabl = tabl;  \_asm {  mov ebx, p\_tabl;  mov al, 2;  xlatb;  nop;  }  unsigned char tabl[10], \*tbl;  int i;  for(i=0;i<10;i++){  tabl[i]=2\*i;  }  tbl = tabl;  \_asm{  MOV EBX,tbl;  MOV AL,5;  XLAT  } |

## Állapotbitek írása/olvasása

* CF: STC/CLC/CMC
* ZF: OR x,y: ZF ← 1 akkor és csak akkor, ha x =0 és y = 0
* Mindet: CF, ZF, AF, PF, SF, OF → LAHF/SAHF
* DF: STD/CLD → sztring-kezelő utasítások
* (IF: STI/CLI)

|  |
| --- |
| //LAHF  // b7 b6 b4 b2 b0  //AH <- EFLAGS(SF:ZF:0:AF:0:PF:1:CF)  //  //SAHF  //EFLAGS(SF:ZF:0:AF:0:PF:1:CF) <- AH  \_asm{  xor al, al;//ZF = 1  lahf;  //  and ah,ah;//ZF = 0  sahf;//ZF = 1  JZ ZF1;  mov al,0;  jmp VEGE  ZF1: mov al, 0xff;  VEGE: nop;  } |

Az alábbi program kiírja az SF ZF AF PF CF álapotbitek értékét. A kódból az is kiolvaható, hogy a LAHF utasítással megoldható például a segédátvitelbit (félbyte-os átvitelbit, AF) szerinti feltételes elágazás is, amire külön gépi utasítás nem áll rendelekezésre. (Megjegyzendő, hogy az állapotbitek más utasítással is elérhetők, amint azt a veremről szóló részben is láthattuk.)

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main()  {  //LAHF  // b7 b6 b5 b4 b3 b2 b1 b0  //AH <- EFLAGS(SF ZF 0 AF 0 PF 1 CF)  int SF, ZF, AF, PF, CF;  \_asm {  xor ax, ax;  lahf; //a kezdeti flag-ek  push ax; //mentése a veremre  //  shr ah, 1;//CF' <- CF  jc CF1;  mov CF, 0;  jmp \_PF;  CF1:mov CF, 1;  //  \_PF: shr ah, 2;//CF' <- PF  jc PF1;  mov PF,0;  jmp \_AF;  PF1:mov PF,1;  //  \_AF: shr ah, 2;//CF' <- AF  jc AF1;  mov AF, 0;  jmp \_ZF;  AF1: mov AF, 1;  //  \_ZF: shr ah, 2;//CF' <- ZF  jc ZF1;  mov ZF, 0;  jmp \_SF;  ZF1: mov ZF, 1;  //  \_SF: shr ah, 1;//CF' <- SF  jc SF1;  mov SF, 0;  jmp VEGE;  SF1: mov SF, 0;  //  VEGE: pop ax;//a kezdeti flag-ek  sahf;//visszatöltése  }  printf("S Z A P C\n");  printf("%d %d %d %d %d\n", SF, ZF, AF, PF, CF);  system("pause");  return 0;  } |

# Feladatok és megoldások

Lépésenkénti utasítás-végrehajtással ellenőrizze az eredményt! Használja ehhez a regiszter-ablakot!

## Elemi feladatok

|  |
| --- |
| \_asm {  //(1's komplemens kódban ábrázolt szám ellentettje)  mov ax, 0x5555;  mov bx, ax;  not ax;//ax = 0xaaaa  //  //(2's komplemens kódban ábrázolt szám ellentettje)  mov ax, 0x5555;//5555H  mov bx, ax;  neg ax;//ax = 0xaaab  //  //8 bites összeadás  mov al, 0x82;  mov ah, 0x12;  add ah, al;//ax = 0x9482  //  //16 bites összeadás  stc;//CF = 1 (CY = 1)  mov ax, 0x1234;  mov bx, 0x5678;  adc ax, bx;//ax = 0x68ad; bx = 0x5678  //  stc;//CF = 1  mov ax, 0x8200;  mov bx, 0x8300;  adc ax, bx;//ax = 0x0501; bx = 0x8300  //  //32 bites összeadás 16 bites utasításokkal  //AX:BX <- AX:BX + CX:DX  mov ax, 0x1000;  mov bx, 0x8000;  mov cx, 0x5000;  mov dx, 0x9000;  add bx, dx;  adc ax, cx;//ax = 0x6001 bx = 0x1000  //  //8 bites kivonás  mov ah, 0x90;  mov al, 0xa0;  sub ah, al;//ax = 0xf0a0  //  mov ah, 0x70;  mov al, 0x80;  sub ah, al;//ax = 0xf080  //  //16 bites kivonás  stc;//borrow = 1  mov ax, 0x90;  mov bx, 0x10;  sbb ax, bx;//ax = 0x007f  //  clc;//borrow = 0  mov ax, 0x90;  mov bx, 0x10;  sbb ax, bx;//ax = 0x0080  //  //32 bites kivonás 16 bites utasításokkal  //DX:AX <- DX:AX - BX:CX  mov dx, 0x6001;  mov ax, 0x1000;  mov bx, 0x5000;  mov cx, 0x9000;  sub ax, cx;  sbb dx, bx;//dx:ax = 1000:8000  //  //Adott pozíciójú bit állása  //CF <- BX 4. bitje (itt BX = b16 b15 ... b2 b1)  mov cl, 4;  mov bx, 0x5555;//010101010101 0 101  shr bx, cl;//cl-szer jobbra léptet; CF = 0 (CY = 0)  //  //ZF <- BX 0. bitjének ellentettjét (itt BX = b15 b14 ... b1 b0)  mov ax, 0x0001;//0000000000000001  mov bx, 0;  test bx, ax;//Ha (BX AND AX == 0) ZF = 1, egyébként ZF = 0  //  //Adott bit 1-be írása  //AX 3. bitjét 1-be írja (itt AX = b15 b14 ... b5 b4 b3 b2 b1 b0)  xor ax, ax;//AX <- 0  or ax, 0x0008;//0000 0000 0000 1 000  //  //Adott bit 0-ba írása  //AX 3. bitjét 0-ba írja (itt AX = b15 b14 ... b5 b4 b3 b2 b1 b0)  xor ax, ax;  not ax;//AX = 0xFFFF  and ax, 0xfff7;//1111 1111 1111 0 111  //  //Adott bit invertálása  //AX 3. bitjét invertálja (itt AX = b15 b14 ... b5 b4 b3 b2 b1 b0)  xor ax, ax;  not ax;//AX = 0xFFFF  and ax, 0xfff7;//1111 1111 1111 0 111  xor ax, ax;  not ax;  xor ax, 0x0008;//0000 0000 0000 1 000  //  //Regiszter, memória nullázása  xor cx, cx;  mov bx, 0;  //  //  //  //mov [ebx],0;//!!! regiszter indirekt címzés, itt csak szemléltetés  //xor [ebx],ax;//!!!  //  //NINCS: xor [ebx],[ebx];//xor mem, mem  //  //  //  nop;  } |

## Adatok összege

A feladatban az adatok darabszáma N, mérete byte (8 bit). Az adatok a DATA nevű tömbben vannak, méretük byte (8 bit). Az összeg a SUM nevű memóriarekeszbe kerül, mérete szó (16 bit).

|  |
| --- |
| //Ellenőrzéshez: S(n) = n\*(n+1)/2  //A feladatban az adatok darabszáma N, mérete byte (8 bit)  //Az adatok a DATA nevű tömbben vannak, méretük byte (8 bit).  //Az összeg a SUM nevű memóriarekeszbe kerül, mérete szó (16 bit).  unsigned char N = 10;  unsigned char DATA[10] = { 1,2,3,4,5,6,7,8,9,10 };  unsigned char \* chptr;  unsigned short int SUM;  chptr = DATA;  //  \_asm {  xor ax, ax;//részletösszeg  xor dx, dx;//DH = 0x00; DL: aktuális byte  mov ebx, chptr;//EBX mutató a tömb elemeire  xor ecx, ecx;//  mov cl, N;//ciklusváltozó  A1: mov dl, [ebx];//soron következő byte  add ax, dx;//16 bites lesz az összeg  inc ebx;//EBX a következő elemre mutat  loop A1;//a ciklus folytatása  mov SUM, ax;//az összegtárolása  nop;  } |

## Negatív számok összege

A feladatban az adatok darabszáma N, mérete byte (8 bit). Az adatok a DATA nevű tömbben vannak, méretük byte (8 bit). A negatív számok összege a SUM nevű memóriarekeszbe kerül, mérete szó (16 bit).

Az adatok kettes komplemens kódúak. Ellenőrizze az eredményt számítással és a program futtatásával!

|  |
| --- |
| //A feladatban az adatok darabszáma N, mérete byte (8 bit)  //Az adatok a DATA nevű tömbben vannak, méretük byte (8 bit).  //A negatív számok összege a SUM nevű memóriarekeszbe kerül, mérete szó (16 bit).  unsigned char N = 10;  char DATA[10] = { 0xF0, 0xE0, 0xD0, 0xC0, 0xB0, 0xA0,  0x90, 0x80, 0x70, 0x60 };//8 bites 2kk adatok  char \* chptr;  short int SUM;//16 bites 2kk össszeg  chptr = DATA;  //  \_asm {  xor bx, bx;//részletösszeg  mov esi, chptr;//ESI mutató a tömb elemeire  xor ecx, ecx;  mov cl, N;//ciklusváltozó  A1: mov al, [esi];//soron következő byte  or al, al;//!!! állapotbitek beállítása, itt Sign Flag (SF)  jns A2;//nemnegatív szám, nem vesz részt az összegzésben  cbw;//!!! előkészítés a 16 bites összeadáshoz: előjelkiterjesztés  add bx, ax;//16 bites lesz az összeg  A2: inc esi;//ESI a következő elemre mutat  loop A1;//a ciklus folytatása  mov SUM, bx;//az összeg tárolása  nop;  } |

## Maximumkeresés (előjel nélküli egész)

A feladatban az adatok darabszáma N, mérete byte (8 bit). Az adatok a DATA nevű tömbben vannak, méretük szó (16 bit). A legnagyobb elem a MAX nevű memóriarekeszbe kerül, mérete szó (16 bit).

a) Lépésenkénti utasításvégrehajtással ellenőrizze a program működését!

b) Módosíthatók-e úgy az adatok, hogy NEM kerül a vezérlés az xchg ax, MAX; utasításra?

|  |
| --- |
| //A feladatban az adatok darabszáma N, mérete byte (8 bit)  //Az adatok a DATA nevű tömbben vannak, méretük szó (16 bit).  //A legnagyobb elem a MAX nevű memóriarekeszbe kerül, mérete szó (16 bit).  unsigned char N = 10, CIKL;  unsigned short int DATA[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  unsigned short int \* uptr;  unsigned short int MAX;//a legnagyobb elem  uptr = DATA;  //  CIKL = N - 1;  \_asm {  mov esi, uptr;//ESI mutató a tömb elemeire  xor ecx, ecx;  mov cl, CIKL;//ciklusváltozó, mivel az első elemmel kezd  //  mov ax, [esi];//az első elem az aktuális maximum  mov MAX, ax;  A1: inc esi;  inc esi;//következő elemre mutat  mov ax, [esi];//következő elem  cmp ax, MAX;//össehasonlítás az eddigi maximummal  jb A2;//kisebb, nincs tárolás (! JB: unsigned)  xchg ax, MAX;//az új maximum tárolása  A2: loop A1;//következő elem vizsgálata  nop;  } |

## ASCIIZ string hossza

Ha a 0x00 végjelig számláljuk a karaktereket, megkapjuk a string hosszát. A feladatban a hossz a LEN nevű, byte méretű memóriarekeszbe kerül. Lépésenkénti utasítás-végrehajtással tanulmányozza a LODSB utasítás működését! Milyen változással jár, ha a JE utasítást JZ-re cseréljük?

|  |
| --- |
| char string[12] = { 'A','b','r','a','k','a','d','a','b','r','a',0x00 };  char \* cptr;  unsigned char LEN = 0;  //  cptr = string;  //  \_asm {  xor cl, cl;//aktuális hossz  mov esi, cptr;//mutató a string karaktereire  A1: lodsb;//AL <- aktuális karakter || ESI <- ESI + 1  or al, al;//AL == 0x00 (végjel) ellenőrzéshez ZF állítása  je A2;//(JZ) elértük a végjelet, készen vagyunk  inc cl;//AL nem végjel: a hossz növelése  jmp A1;//folytatás a következő karakterrel  A2: mov LEN, cl;//a hossz tárolása  nop;  } |

## Az első érvényes karakter megkeresése ASCIIZ string-ben

A feladatban az érvénytelen karakterek kódjára fennáll:

MINKOD <= kód VAGY kód > MAXKOD; MINKOD = 31, MAXKOD = 127.

Az első érvényes karakterkód a KOD, helye a POS nevű memóriarekeszekben lesz.

a) Lépésenkénti utasítás-végrehajtással ellenőrizze a program működését!

b) Rajzolja le a folyamatábrát!

c) Mi lesz CHR és POS értéke, ha mindegyik karakter érvénytelen?

d) Hogyan működik a program „üres string” (csak a végjelből áll) esetén?

|  |
| --- |
| char string[15] = { 0x01, 0x02, 0x03, 0x0a, 'T','a','n','g','a','n','y','i','k','a',0x00 };  char \* cptr;  unsigned char MINKOD = 31;  unsigned char MAXKOD = 127;  unsigned char CHR = 0;  unsigned char POS = 0;  //  cptr = string;  //  \_asm {  xor cl, cl;//pozíciószámláló  mov esi, cptr;//mutató a string karaktereire  A1: inc cl;//következő karakter pozíciója  lodsb;//AL<-karakterkód || ESI<-ESI+1  or al, al;//AL==0x00 végjel?  je A2;//végjel, kész  cmp al, MINKOD;//<= MINKOD? => érvénytelen  jbe A1;//folytatás a kvetkező karakterrel  cmp al, MAXKOD;//> MAXKOD? => érvénytelen  ja A1;//folytatás a kvetkező karakterrel  mov POS, cl;//érvényes karakter helye  mov CHR, al;//érvényes karakter kódja  A2: nop;  } |

## ASCII-7 karakter kibővítése páros paritásbittel

A feladatban bemenő adat az ASCII-t karakter kódja (INCH), kimenő adat a paritásbittel védett karakter (OUTCH).

|  |
| --- |
| char INCH, OUTCH;  INCH = 'C';  \_asm {  mov al, INCH;  and al, 0x7F;//01111111; ASCII-7  or al, al;//PF állítása  JPE A1;//páros, így MSB = 0; kész  or al, 0x80;//10000000; MSB = 1 beállítása  A1: mov OUTCH, al;//eredmény tárolása  nop;  } |

## ASCIIZ-HEX konverzió

|  |
| --- |
| // '0'..'9': 0x30 .. 0x39 => kód - 0x30  // 'A'..'F': 0x41 .. 0x46 => kód - 0x37  //ASCIIZ string -> HEX konverzió  //byte  char bytestr[3] = { 'A', '8', 0x00 };  char \* cptr;  unsigned char U8;  unsigned char FLAG;//1: sikeres, 0: sikertelen  //  cptr = bytestr;  \_asm {  mov U8, 0;//kezdeti érték  mov FLAG, 0;//kezdetben sikertelen a konverzió  xor edx, edx;//hexa szám helye  mov ebx, cptr;  KONV: mov al, [ebx];//aktuális karakter kódja  cmp al, 0x00;//végjel?  je KONV\_KESZ;//készen vagyunk  cmp al, 0x30;//'0'  jb NEMHEX;//nem hexa számjegy kódja  cmp al, 0x39;//'9'  ja A\_F;//nem decimális számjegy kódja  sub al, 0x30;//kód -> számérték; pl. 0x35 -> 0x05  jmp HELYEZ;//hexa számjegy elhelyezése  A\_F: cmp al, 0x41; 'A'  jb NEMHEX;//nem hexa számjegy kódja  cmp al, 0x46;//'F'  ja NEMHEX;//nem hexa számjegy kódja  sub al, 0x37;//kód -> számérték; pl. 0x41 -> 0x0A  HELYEZ: or dl, al;//hexa számjegy elhelyezéséhez  shl edx, 4;//a nagyobb súlyú helyek felé  inc ebx;//következő karakter címe  jmp KONV;//konverzió folytatása  NEMHEX:mov FLAG, 0;//sikertelen konverzió  jmp VEGE;  KONV\_KESZ: shr edx, 4;//!  mov U8, dl;//eredmény tárolása  mov FLAG, 1;//sikeres konverzió  VEGE: nop;  } |

a) Lépésenkénti utasítás-végrehajtással ellenőrizze a programot!

b) Hogyan kell módosítani a programot, ha a szám 16 bites?

c) Az EDX regiszter használata miatt hány jegyű hexadecimális szám konvertálása lehetséges legfeljebb ezzel a programmal?

d) Mit javasol, ha a számjegyek száma 8-nál több?

e) Rajzolja le a folyamatábrát!

f) A program alapján az ASCIIZ -> pakolt BCD konverzió kódja egyszerűen elkészíthető. Miért?

g) Írja meg a HEX -> ASCIIZ és pakolt BCD -> ASCIIZ konverziós programokat!

## Pakolatlan BCD-HEX konverzió

a) Lépésenkénti utasítás-végrehajtással ellenőrizze a programot!

b) Hogyan kell módosítani a programot több digites BCD szám esetére?

|  |
| --- |
| char BCDNUM[3] = { 0x01, 0x02, 0x09 };//pakolatlan BCD  char SULY[3] = { 1, 10, 100 };  char \* cptrnum, \*cptrsuly;  unsigned short int HEXNUM = 0;  cptrnum = BCDNUM;  cptrsuly = SULY;  \_asm {  xor bx, bx;//részletösszeg  mov esi, cptrnum;//  mov edi, cptrsuly;//  mov cx, 3;//3 digit  A1: mov al, [esi];//digit  mov dl, [edi];//súly  mul dl;//AX <- AL \* DL  add bx, ax;  inc esi;  inc edi;  loop A1;  mov HEXNUM, bx;//eredmény  nop;  } |

## HEX-pakolatlan BCD konverzió

a) Lépésenkénti utasítás-végrehajtással ellenőrizze a programot!

b) Mekkora lehet a program esetében a hexadecimális szám legfeljebb?

c) Hogyan kell módosítani a programot több digites BCD szám esetére?

|  |
| --- |
| unsigned short int HEXNUM = 0x0;  char BCDNUM[3];  char \*cptr;  cptr = BCDNUM;  \_asm {  mov cx, 3;//3 digit  mov esi, cptr;//mutató a digitek elhelyezéséhez  mov dl, 10;//a 10-zel való osztáshoz  mov ax, HEXNUM;//átalakítandó hexa szám  A1: div dl;//AX osztva r/m8: AL: hányados, AH: maradék  mov[esi], ah;//a maradék az aktuális BCD digit  xor ah, ah;//a hányadost kell tovább osztani: AX = 00 AL  inc esi;  loop A1;  nop;  } |

# Vegyes feladatok

1. Írja ki egy regiszter és egy memóriarekesz tartalmát hexadecimálisan és binárisan!
2. Írja ki a következő állapotjelzők értékét: CF, SF, OF, AF, PF, ZF!
3. Számítsa ki egy byte-okból álló tömb ellenőrző összegét, ha:
   1. az ellenőrző összeg a tömbelemek antivalenciája,
   2. az ellenőrző összeg a tömbelemek összegének (-1)-szerese (kettes komplemens kódban),
   3. mindkét módszernek hasonló előnyös tulajdonsága van, mi az?
4. Kettes komplemens kódú szavakból álló tömb esetén számlálja meg a pozitív, negatív és zérus elemeket!
5. Számítsa ki egy byte-okból álló tömb Hamming-súlyát!
6. Számítsa ki két kódszó Hamming-távolságát XOR művelet nélkül!
7. Számítsa ki két kódszó Hamming-távolságát XOR művelettel!
8. Készítse el egy byte-okból álló tömb elemeinek gyakorisági táblázatát!
9. Határozza meg egy adott karaktersorozat helyét egy másik karaktersorozatban!
10. Írjon programot bináris-lineáris konverzióra 5 bites bináris számokra, az eredmény 32 bites legyen, pl. 10111 -> 11111111111111111111111000000000,   
     00111 -> 11111110000000000000000000000000
11. Van olyan nézet, miszerint az „adatmozgatás” helyett jobb lenne „adat átmásolás”-nak hívni ezeket az utasításokat. Mi szól mellette?
12. Írjon az XCHG utasítást helyettesítő utasítás-sorozatot!
13. Miért fontos megkülönböztetnünk a „little endian” és a „big endian” byte-sorrendet? Hogyan általánosítható ez a fogalom byte-nál nagyobb méretű adatokra?
14. Ismételje át, hogyan működik a verem! Hogyan lehet két operandus értékét felcserélni (külön-külön a regiszter/regiszter, regiszter/memória, memória/memória esetben)?
15. Írjon ekvivalens utasítás-sorozatot az operandus méret növeléséhez előjel nélküli egész és kettes komplemens kódú előjele egész számok esetére! Rajzolja le a folyamatábra részletet!
16. Írjon programrészletet 32 bites számok összeadására úgy, hogy csak 8 bites összeadás utasításokat használ! Hogyan módosul a program 16 bites összeadás esetén?
17. Mikor 1 értékű a kölcsönbit (Borrow Flag)? Hol találhatjuk IA32 esetén a kölcsönbitet?
18. Írjon programrészletet 32 bites számok kivonására úgy, hogy csak 8 bites kivonás utasításokat használ! Hogyan módosul a program 16 bites kivonás esetén?
19. Az IA32 ISA más-más utasítást biztosít az előjeles (kettes komplemens kódú) és az előjel nélküli egész számok szorzására. Mi lehet ennek az oka?
20. Az IA32 ISA más-más utasítást biztosít az előjeles (kettes komplemens kódú) és az előjel nélküli egész számok osztására. Mi lehet ennek az oka?
21. Példaprogrammal vizsgálja meg, mi történik, ha 0-val osztunk?
22. Milyen célra használható a növelő/csökkentő (INC/DEC) utasítás?
23. Milyen részműveletekre vezehető vissza a NEG utasítás?
24. Milyen aritmetikai műveletet végez a processzor a CMP utasítás végrehajtása közben?
25. Tipikusan milyen utasítások követik a CMP utasítást?
26. Mi a pakolt és a pakolatlan BCD számábrázolás?
27. Mi a pszeudo-tetrád?
28. Mikor 1 értékű a félbyte-os átvitelbit (segédátvitel bit, AF/AC)?
29. Mi a 6-os korrekció?
30. Írjon programrészletet a DAS, AAS, AAD utasítások működésének szemléltetésére!
31. Egy byte megadott bitjét szeretnénk 1-be írni a többi bit megváltoztatása nélkül. Milyen logikai művelettel végezhető ez el? Hogyan alakul a megoldás, ha egyszerre több bitet szeretnénk 1-be írni hasonló feltétel mellett? Mi a teendő szó illetve duplaszó méretű operandusok esetén?
32. Egy byte megadott bitjét szeretnénk 0-ba írni a többi bit megváltoztatása nélkül. Milyen logikai művelettel végezhető ez el? Hogyan alakul a megoldás, ha egyszerre több bitet szeretnénk 0-ba írni hasonló feltétel mellett? Mi a teendő szó illetve duplaszó méretű operandusok esetén?
33. Egy byte megadott bitjét szeretnénk invertálni a többi bit megváltoztatása nélkül. Milyen logikai művelettel végezhető ez el? Hogyan alakul a megoldás, ha egyszerre több bitet szeretnénk invertálni hasonló feltétel mellett? Mi a teendő szó illetve duplaszó méretű operandusok esetén?
34. Digitális B/K esetén hogyan használható a TEST utasítás?
35. Írjon programrészletet az  logikai függvény igazságtáblázatának meghatározására!
36. Milyen aritmetikai műveletnek felel meg a SHR utasítás?
37. Milyen aritmetikai műveletnek felel meg a SAR utasítás?
38. Milyen aritmetikai műveletnek felel meg a SHL utasítás?
39. Byte operandus esetén mennyi bit vesz részt a forgatásban a ROL utasítás esetén? Írjon programrészletet a működés bemutatására! Mi a teendő szó és duplaszó esetén?
40. Byte operandus esetén mennyi bit vesz részt a forgatásban a RCL utasítás esetén? Írjon programrészletet a működés bemutatására! Mi a teendő szó és duplaszó esetén?
41. Írjon programrészletet a BT utasítás megvalósítására logikai utasításokkal!
42. Oldja meg az előző feladatot a BTS, BTR, BTC utasítások esetében!
43. Írjon programrészletet a BSF, BSR utasítások működésének szemléltetésére!
44. Gyűjtse össze, hogy a SETxx alakú utasításoknál milyen állapotbit-értékek szerepelnek!
45. Mikor 1 értékű az OF jelzőbit?
46. Mikor 1 értékű a paritásbit?
47. Mivel jár, ha LOOP-pal megvalósított hátul tesztelő ciklus esetében CX=0 a cilusváltozó kezdeti értéke?
48. Feltételes elágazás esetén hogyan számítjuk ki az elágazási célcímet? Mind a jelenleginél nagyobb, mind a kisebb célcím esetében oldja meg a feladatot!
49. Mi a különbség a RET és az IRET utasítások között?
50. Hogyan tekinthetjük meg a szegmensregiszterek (szegmens-leírók) értékét VS fejlesztői környezetben?
51. Hogyan változik az (E)SI és az (E)DI regiszterek értéke a MOVSB, CMPSB és SCASB utasítások végrehajtása után? Mitől függ ez?
52. Írjon programrészletet a REPZ és a REPNZ prefix hatásának bemutatására!
53. Mit jelent a „privilegizált utasítás” kifejezés?
54. Gyűjtsön privilegizált utasítássokat az x86 processzorok esetén!
55. Milyen feltételek mellett lehet az IN/OUT utasítások működését tanulmányozni?
56. Gyűjtse ki, hogy pontosan mely biteket érintik a LAHF, SAHF utasítások!
57. Milyen bitek találhatók a FLAGS és az EFLAGS regiszterekben?
58. Az STI/CLI/HLT utasítások privilegizáltak. Miért?
59. A CPUID utasítással megállapítottuk, hogy processzorunknak van RDRAND utasítása. Hogyan állíthatunk elő ezzel a [0;1) intervallumba eső álvéletlen számot? Milyen eloszlású lesz az eredmény?
60. Milyen feladatok megoldásához szükségesek álvéletlen számok?
61. Mi történik a NOP utasítás végrehajtása során?
62. Hogyan használható táblázattal adott függvény (look-up table) megvalósítására az XLATB utasítás? Mik a korlátok?
63. Nézzen utána, melyek azok a CPU jellemzők, amelyek a CPUID utasítással érhetők el!
64. Az RDTSC utasítás az EDX:EAX regiszterpárban adja vissza a RESET után eltelt óraciklusok számát. Ha csak a RDTSC által visszaadott alsó 32 bitet nézzük, 1 GHz órajel esetén mennyi a maximálisan eltelt idő, amit mérni lehet? Mennyi ez az érték a 64 bites mért érték esetén?
65. Hogyan tudhatjuk meg, hogy mennyi a számítógép processzorának órajele?
66. A kettes komplemens kódú számokat pontosan lehet-e lebegőpontossá alakítani?
67. A lebegőpontos számokat nem lehet mindig pontos értékű kettes komplemens kódú számokká alakítani. Milyen FP kerekítési módok állnak rendelekzésre erre a célra az FPU-ban?
68. Támogatja-e az MS VS C++ fordító az IEEE-754 szabvány szerinti BCD számokat?
69. Az FPU regisztereit „regiszter/stack” néven is megtalálhatjuk. Miért?
70. A kivonás és osztás esetében van fordított operandusú utasítás is, de az összeadás és a szorzás esetében nincs. Miért?
71. Igaz-e, hogy az FCHS (-1)-gyel történő szorzásnak felel meg?
72. Milyen kerekítési módok vannak az IEEE-754-ben?
73. Ha negatív számból vonunk négyzetgyököt, megszakítás, vagy kivétel keletkezik? Miért?
74. A lebegőpontos számábrázolás esetén a 0 számnak kétféle alakja is van. Hogyan működik ez esetben az összehasonlító utasítás?
75. Mi a lényeges eltérés az FCOM és az FICOM utasítások között?
76. Hogyan helyettesíthatő az FTST más utasításokkal?
77. Ismételje át a lebegőpontos számoknál a speciális esetek kódolásáról tanultakat!
78. Írjon programrészletet a nevezetes szögek szinuszának és koszinuszának egyidejű kiszámítására az FSINCOS utasítással!
79. Írjon programrészletet az FPTAN és az FPATAN utasítások működésének bemutatására!
80. Írjon programrészletet az FYL2XP1 utasítás működésének bemutatására!
81. Hogyan váltunk át különböző alapú logaritmusok között?
82. Milyen típusú feladatoknál használjuk a kettes, a természetes és a tízes alapú logaritmust?
83. Az FPU működési környezet képen „exponent”, „significand” olvasható. Tantárgyunkban a hagyományos elnevezéseket követtük az egyértelműség érdekében: minek felelnek meg ezek a kifejezések?
84. Az FPU állapotszóban található a TOP (top of stack pointer) bitmező. Hány bitesnek kell lennie? Igaz-e, hogy ez jelöli ki, hogy melyik regiszter az ST(0)?
85. A 16 bites vezérlőszó mely bitjeivel lehet a kerekítési módot beállítani?
86. Hogyan vizsgálhatjuk a lebegőpontos összehasonlító utasításoknál megismert feltétel kódbiteket (C3, C2, C0)?
87. Mi az FSTENV utasítás esetén a memóriatérkép?
88. Miért pontosan 80 byte szükséges az FPU regiszter tartalmak számára?
89. Hogyan működik az FNOP utasítás?
90. A Gray-kód jellemzője, hogy szomszédos kódszavai között a Hamming-távolság 1. Előjel néküli egész számok Gray-kódú alakját állítja elő a következő eljárás. Legyen az N bites előjel nélküli egész szám a következő: , ahol  az MSB, és a Gray-kódú alak . Ekkor

.

Írjon BIN → GRAY kódátalakító \_asm{} programot!

1. A fenti feladat jelöléseivel írjon GRAY → BIN kódátalakító \_asm{} programot! Útmutató: a rekurzív összefüggés ebben az esetben:

.

Hogyan adódik ez az összefüggés az előző feladat alapján?

# A Függelék: gyakorló feladatsor

1. Adja meg az  decimális szám hexadecimális alakját!
2. Adja meg az 511,125 decimális szám oktális alakját!
3. Adja meg a 32768 decimális szám 16 bites bináris alakját!
4. Mi a 0x1A hexadecimális szám 8 bites bináris alakja?
5. A 12 bites előjel nélküli egész számok esetében mi a legkisebb és a legnagyobb ábrázolható szám értéke hexadecimálisan, decimálisan és binárisan felírva?
6. A számok bináris alakjával hajtsa végre az alábbi összeadást: 0xB7 + 0xCF! Mi az átvitelbit értéke az összeadás után? Mi a zérus flag értéke az összeadás után?
7. Melyik utasítással lehet az átvitelbitet invertálni?
8. Adja meg egy vezérlőkarakter ASCII-kódját hexadecimálisan!
9. Mi a kapcsolat az angol kis- és nagybetűk ASCII-kódja között?
10. Adott a következő memória-dump részlet:  
    32FADC15 01 02 30 40  
    Mi az itt lévő 32 bites szám little endian tárolási mód esetén hexadecimális alakban?
11. Mi az ucStatus értéke az alábbi utasítások végrehajtása után? Mi lenne az ucStatus értéke a fenti utasítások végrehajtása után ucX = 0x00 értéke esetén? Mi lenne az ucStatus értéke a fenti utasítások végrehajtása után ucX = 0x80 értéke esetén?

|  |
| --- |
| unsigned char ucX;  unsigned char ucStatus;  ucX = 0xE0;  ucStatus = 0xFF;  \_asm {  mov al, ucX;  mov ah, 0x80;  cmp al, ah;  jae C1;  mov ah, 0x40;  cmp al, ah;  jae C2;  mov ucStatus, 0;  jmp VEGE;  C1: mov ah, 0xC0;  cmp al, ah;  jae C3;  mov ucStatus, 2;  jmp VEGE;  C2: mov ucStatus, 1;  jmp VEGE;  C3: mov ucStatus, 3;  VEGE: nop;  } |

1. IA32 esetén egy szó LSB-jét szeretnénk 1-be írni úgy, hogy a többi bit változatlan maradjon. Milyen logikai művelettel lehet ezt megoldani? Mi a megoldáshoz szükséges maszk?
2. IA32 esetén egy byte MSB-jét szeretnénk invertálni úgy, hogy a többi bit változatlan maradjon. Milyen logikai művelettel lehet ezt megoldani? Mi a megoldáshoz szükséges maszk?
3. Mi lesz a regiszterek értéke az alábbi utasítások végrehajtása után?

|  |  |  |
| --- | --- | --- |
| Utasítás | EAX | EBX |
| **MOV EAX, 1;** |  |  |
| **MOV EBX, 2;** |  |  |
| **XOR EAX, EBX;** |  |  |
| **XOR EBX, EAX;** |  |  |
| **XOR EAX, EBX;** |  |  |

1. Konvertálja a 0x40900000 IEEE-754 szabványú 32 bites számot decimális számmá!
2. Mi lesz y értéke az alábbi utasítások végrehajtása után? Rajzolja le a folyamatábrát a program alapján! Állítólag egy három soros assembly program ugyanazt az eredményt adja, mint az alábbi program. Mi ez a három sor?

|  |
| --- |
| unsigned char x, y;  x = 0xD1;  \_asm {  mov al, x;  shr al, 1;  jc E;  shr al, 1;  jc EN;  mov y, 0;  jmp VEGE;  EN: mov y, 2;  jmp VEGE;  E: shr al, 1;  jc EE;  mov y, 1;  jmp VEGE;  EE: mov y, 3;  VEGE: nop;  } |

# B Függelék: utasítás-csoportok

## Általános célú utasítások

|  |  |  |
| --- | --- | --- |
| MOV | Adatmozgatás:  regiszterek és memóriarekeszek között | **mov eax, x**  **mov cx, dx** |
| CMOVE/CMOVZ | Feltételes adatmozgatás: átmásolja az adatot, ha ZF = 1 | **mov x, -1;**  **xor eax, eax;//ZF = 1**  **cmovz ebx, x;**  **mov edx, 0x55555555;**  **or eax, 1;//ZF = 0**  **cmovz edx, x;** |
| CMOVNE/CMOVNZ CMOVA/CMOVNBE CMOVAE/CMOVNB CMOVB/CMOVNAE CMOVBE/CMOVNA CMOVG/CMOVNLE CMOVGE/CMOVNL CMOVL/CMOVNGE CMOVLE/CMOVNG  CMOVC  CMOVNC  CMOVO  CMOVNO  CMOVS  CMOVNS  CMOVP/CMOVPE  CMOVNP/CMOVPO | értelemszerűen a relációknál és az állapotjelzőknél megismert módon |  |
| XCHG | Csere:  regiszterek, memórierekesz tartalmának felcserélése | **xchg x, ebx** |
| BSWAP | Byte-sorrend megváltoztatása  Little endian <-> big endian | **mov eax, 0xabcdef01;**  **bswap eax;** |
| XADD | Csere és összeadás | **mov x, 0x00000010;**  **mov eax, 0x00000020;**  **xadd x, eax;** |
| CMPXCHG r/m, r | Összehasonlítás és csere  Ha AL (AX, EAX) értéke azonos r/m értékével, akkor ZF = 1 és  r/m <- r,  egyébként ZF = 1 és AL <- r/m |  |
| PUSH, POP | Operandus veremre helyezése, visszatöltés |  |
| PUSHA/PUSHAD  POPA/POPAD  PUSHF/PUSHFD | Egyszerre több regiszter illetve az állapotregiszter tartalmának veremre helyezése, visszatöltése  PUSHA: AX, CX, DX, BX, eredeti SP, BP, SI, DI  PUSHAD: EAX, ECX, EDX, EBX, eredeti ESP, EBP, ESI, EDI |  |
| CBW  CWDE | Előjel-kiterjesztés  AX <- ssssssssAL  EAX <- ssssssssssssssssAX |  |
| CWD/CDQ | Előjel-kiterjesztés  DX:AX <- s…sAX  EDX:EAX <- s…sEAX |  |
| MOVSX/ MOVZX | Adatmozgatás előjel-kiterjesztéssel illetve 0-kkal  r <- r/m |  |

## Fixpontos aritmetika

|  |  |  |
| --- | --- | --- |
| ADD/ADC | Összeadás  Összeadás állapotbittel | **add eax, x;//EAX <- EAX + x**  **adc eax, x;//EAX <- EAX + x + CF** |
| SUB/SBB | Kivonás  Kivonás kölcsönbittel | **sub eax, x;//EAX <- EAX – x**  **sbb eax, x;//EAX <- EAX – x - CF** |
| IMUL/IDIV | Szorzás/osztás (kettes komplemens kódú számok) |  |
| MUL/DIV | Szorzás/osztás (előjel nélküli egész számok) |  |
| INC/DEC | Növelés/csökkentés 1-gyel |  |
| NEG | Kettes komlemens kódú szám (-1)-szerese |  |
| CMP | (Aritmetikai) összehasonlítás |  |

## BCD aritmetika

|  |  |  |
| --- | --- | --- |
| DAA | BCD korrekció összeadás után | **xor ax, ax;**  **mov al, 0x38;//pakolt BCD**  **mov bl, 0x57;//pakolt BCD**  **add al, bl;**  **daa;//AL korrekcioja AL-be (0x0095)** |
| DAS | BCD korrekció kivonás után |  |
| AAA | ASCII korrekció összeadás után  (AL, AH:AL) | **xor ax, ax;**  **mov al, 6;**  **mov bl, 3;**  **add al, bl;//6+3=9, ervenyes tetrad**  **aaa;//nem lesz korrekcio (AL nem valtozik)**  **//**  **xor ax, ax;**  **mov al, 6;**  **mov bl, 8;**  **add al, bl;//6+8=14 (0x0E), pszeudo-tetrad**  **aaa;//6-os korrekcio: AH <-1 AL <- 4 (pakolatlan)**  **//**  **xor ax, ax;**  **mov al, 8;**  **mov bl, 9;**  **add al, bl;//8+9=17 (0x11), AF = 1**  **aaa;//6-os korrekcio: AH <- 1, AL <- 7 (pakolatlan)** |
| AAS | ASCII korrekció kivonás után |  |
| AAM | ASCII korrekció szorzás után | **xor ax, ax;**  **mov al, 9;**  **mov bl, 8;**  **mul bl;//AX <- AL \* BL (0x48 = 72D**  **aam;//AX <- korrigalt(AX) (pakolatlan)** |
| AAD | ASCII korrekció osztás előtt |  |

## Logikai műveletek

|  |  |  |
| --- | --- | --- |
| AND | Bitenkénti logikai ÉS |  |
| OR | Bitenkénti logikai VAGY |  |
| XOR | Bitenkénti logikai KIZÁRÓ VAGY |  |
| NOT | Bitenkénti logikai NEM |  |
| TEST dst, src | Logikai összehasonlítás bitenkénti ÉS művelettel (SF, ZF, PF) |  |

## Léptetés, forgatás

|  |  |  |
| --- | --- | --- |
| SAR | Aritmetikai léptetés jobbra |  |
| SHR | Logikai léptetés jobbra |  |
| SAL/SHL | Léptetés balra |  |
| ROR | Forgatás jobbra | mov al, 0x21;//0010 0001  clc;//CF = 0  ror al, 1;//AL = 1001 0000 CF = 1 |
| ROL | Forgatás balra |  |
| RCR | Forgatás jobbra CF-fel | mov al, 0x21;//0010 0001  clc;//CF = 0  rcr al, 1;//AL = 0001 0000 CF = 1 |
| RCL | Forgatás balra CF-fel |  |

## Bit- és byte műveletek

|  |  |  |
| --- | --- | --- |
| BT | A kijelölt bit CF-be másolása | clc;  bt x, 0;//…101 0 (CF = 0)  clc;  mov ebx, 0x80000000;  bt ebx, 31;//1 0…0 (CF = 1) |
| BTS | A kijelölt bit CF-be másolása, majd 1-be írása | xor ebx, ebx;//EBX = 0x00000000  stc;//CF = 1  bts ebx, 31;//CF = 0, EBX = 0x80000000 |
| BTR | A kijelölt bit CF-be másolása, majd 0-ba írása | mov ebx, 0x80000000;  mov al, 31;  btr ebx, al; |
| BTC | A kijelölt bit CF-be másolása, majd invertálása |  |
| BSF/BSR | Az első 1 értékű bit pozícióját adja vissza  BSF: LSB-től MSB felé  BSR: MSB-től LSB felé |  |
| SETE/SETZ | Az operandust 1-be írja, ha a feltétel teljesül, 0-ba ha nem teljesül | xor bl, bl;  dec bl;//BL = 0xFF  xor al, al;//ZF = 1  setz bl;//BL = 0x01 |
| SETNE/SETNZ  SETA/SETNBE  SETAE/SETNB/SETNC  SETB/SETNAE/SETC  SETBE/SETNA  SETG/SETNLE  SETGE/SETNL  SETL/SETNGE  SETLE/SETNG  SETS  SETNS  SETO  SETNO  SETPE/SETP  SETPO/SETNP |  |  |

## Vezérlésátadás, ciklusszervezés

|  |  |  |
| --- | --- | --- |
| JMP | Feltétel nélküli elágazás (ugrás) |  |
| JE/JZ | Feltételes elágazás: ugrik, ha ZF = 1 |  |
| JNE/JNZ; JA/JNBE; JAE/JNB; JB/JNAE; JBE/JNA; JG/JNLE; JGE/JNL; JL/JNGE; JLE/JNG; JC; JNC; JO; JNO; JS; JNS; JPO/JNP; JPE/JP; |  |  |
| JCXZ/JECXZ | Ugrik, ha CX (ECX) értéke 0 | Hátul tesztelő ciklus elején hasznos |
| LOOP | Ciklus; aciklusz-számláló (CSZ) az ECX |  |
| LOOPZ/LOOPE | 1. A CSZ csökkentése  2. Ugrik, ha CSZ ≠ 0 és ZF = 1 |  |
| LOOPNZ/LOOPNE | 1. A CSZ csökkentése  2. Ugrik, ha CSZ ≠ 0 és ZF = 0 |  |
| CALL | Eljárás (szubrutin) hívás |  |
| RET | Visszatérés szubrutinból |  |
| IRET | Visszatérés SW vagy HW megszakításkezelő programból |  |
| INT xx | SW interrupt  INT 3: a debugger részére (kódja 0xCC) |  |
| INTO | OF = 1 esetén indítja a hozzá tartozó megszakításkezelő programot |  |

## Sztring-műveletek

|  |  |  |
| --- | --- | --- |
| MOVSB  MOVSW, MOVSD | Byte (szó, duplaszó) másolása  DS:(E)SI: honnan  ES:(E)DI: hova | char src[10];  char \* psrc;  psrc = src;  char dst[10];  char \* pdst;  pdst = dst;  strcpy\_s(src, "abcd");  strcpy\_s(dst, "");  \_asm{  mov esi, psrc;  mov edi, pdst;  movsb;  } |
| CMPSB  CMPSW, CMPSD | Byte (szó, duplaszó) összehasonlítása és a jelzőbitek beállítása  DS:(E)SI: mit  ES:(E)DI: mivel | strcpy\_s(src, "abcd");  strcpy\_s(dst, "AbCD");  \_asm{  mov esi, psrc;  inc esi;  mov edi, pdst;  inc edi;  cmpsb;//ZF = 1 (b = b)  } |
| SCASB  SCASW, SCASD | AL-t összehasonlítja az ES:(E)DI címen lévő byte-tal, és beállítja az állapotjelzőket |  |
| LODSB  LODSW, LODSD | Az ES:(E)SI címen lévő byte-ot AL.be másolja |  |
| STOSB  STOSW, STOSD | AL-t átmásolja az ES:(E)DI címre |  |
| REP | Utasítás ismétlő prefix:  ismétli a sztringműveletet, amíg ECX nem nulla | strcpy\_s(src, "abcd");  strcpy\_s(dst, "");  \_asm{  mov ecx, 5;  mov esi, psrc;  mov edi, pdst;  rep movsb;  } |
| REPE/REPZ | Utasítás ismétlő prefix:  Az első nem egyező byte-ig ismétli a sztringműveletet |  |
| REPNE/REPNZ | Utasítás ismétlő prefix:  az első egyező byte-ig ismétli a sztringműveletet |  |

## B/K utasítások

|  |  |  |
| --- | --- | --- |
| IN AL, port8 | **Byte beolvasása bennfoglalt című portról (0x00 .. 0xFF)** |  |
| IN AL, DX | **Byte beolvasása portról, a port címe DX-ben van (0x100 –től)** |  |
| OUT port8, AL |  |  |
| OUT DX, AL |  |  |

## Az állapotregiszterre vonatkozó utasítások

|  |  |  |
| --- | --- | --- |
| STC | CF <- 1 |  |
| CLC | CF <- 0 |  |
| CMC | CF <- NOT(CF) |  |
| CLD | DF <- 0  ESI és EDI növekszik |  |
| STD | DF <- 1  ESI és EDI csökken |  |
| LAHF | Jelzőbiteket másol AH-ba |  |
| SAHF | AH bitjeit az állapotregiszter megfelelő jelzőbitjeibe írja |  |
| PUSHF  PUSHFD | A jelzőbiteket a veremre helyezi |  |
| POPF  POPFD | A veremről leemelt szó bitjeit az állapotregiszter megfelelő bitjeibe írja |  |
| STI/CLI | Maszkolható megszakításkérések engedélyzése/tiltáse |  |

## Egyéb utasítások

|  |  |  |
| --- | --- | --- |
| RDRAND r16  RDRAND r32 | Egyenletes eloszlású álvéletlenszámot állít elő  CF = 1, ha készen van  r16: 0..65535  r32: 0.. 4294967295 | unsigned short int x16, y16, z16;  \_asm{  r1: rdrand ax;  jnc r1;  mov x16, ax;  r2: rdrand ax;  jnc r2;  mov y16, ax;  r3: rdrand ax;  jnc r3;  mov z16, ax;  nop;  } |
| LEA reg, mem | Kiszámítja a memóriában lévő operandus offszetjét, és a regiszterbe írja | int x32[10] = { 0, 1, 2, 3 };  \_asm{  lea ebx, x32[1];  } |
| NOP |  |  |
| XLATB | AL-be írja a DS:[(E)BX + unsigned  AL] címen lévő memória-byte-ot |  |
| CPUID | 1. processzorjellemzőket olvas be  2. „szerializál”  (ld. RDTSC) |  |
| CRC32 | CRC előállításához egy lépés  A generátor-polinom: 0x11EDC6F41 |  |
| RDTSC | A 64 bites számláló értékét olvassa be EDX:EAX-be  Megjegyzés: ha előtte be kell fejeződnie a megelőző utasításoknak szerializálni kell (vagyis a megelőző utasítások, amik a jelzőbiteket, regisztereket, memóriarekeszeket módosítják, befejeződnek, mielőtt a következő uasítást beolvasná és végrehajtaná a processzor) | int time = 0;  float x = 1.0, y = 2.0, z = 0.;  \_asm{  cpuid;  rdtsc;  mov time, eax;  fld x;  fld y;  fdiv;  fstp z;  cpuid;//a lebegopontosakat megvarja  rdtsc;//beolvassa a szamlalo erteket  sub eax, time;  } |

## FPU adatmozgatás

|  |  |  |
| --- | --- | --- |
| FLD | FP adatot helyez a lebegőpontos veremre | float x = 1.0;  double y = 2.0;  \_asm{  fld x;  fld y;  fld st(0);  } |
| FST | A lebegőpontos verem tetején lévő FP adatot a memóriába másolja |  |
| FSTP | A lebegőpontos verem tetején lévő FP adatot leemeli a veremről és a memóriába másolja |  |
| FILD | A kettes komplemens kódú operandust 80 bites lebegőpontossá alakítja és a lebegőpontos veremre helyezi |  |
| FIST, FISTP | A lebegőpontos verem tetején lévő értéket egész számmá alakítja és a memóriába írja |  |
| FBLD, FBSTP | 18 jegyű pakolt BCD számokhoz … |  |
| FXCH | Felcseréli ST(0) és ST(1) tartalmát |  |
| FXCH ST(i) | Felcseréli ST(0) és ST(i) tartalmát |  |
| FCMOVE  FCMOVNE  FCMOVB  FCMOVBE  FCMOVNB  FCMOVNBE  FCMOVU  FCMOVNU | Feltételes adatmozgatások az EFLAGS adott bitjei szerint |  |

## FPU aritmetika

|  |  |  |
| --- | --- | --- |
| FADD, FADDP, FIADD | FP összeadás |  |
| FSUB, FSUBP, FISUB | FP kivonás |  |
| FSUBR, FSUBPR, FISUBR | FP kivonás „fordított” operandusokkal |  |
| FMUL, FMULP, FIMUL | FP szorzás |  |
| FDIV, FDIVP, FIDIV | FP osztás |  |
| FDIVR, FDIVPR, FIDIVR | FP osztás „fordított” operandusokkal |  |
| FPREM | ST(0)-ba írja az ST(0)/ST(1) osztás maradékát |  |
| FABS | ST(0) <- |ST(1)| |  |
| FSQRT | FP négyzetgyökvonás |  |
| FCHS | ST(0) előjelcseréje |  |
| FRNDINT | Az adott kerekítési mód szerint kerekíti ST(0)-t |  |
| FSCALE | ST(0) ← ST(0) ∗ 2RoundTowardZero(ST(1)) |  |
| FXTRACT | ST(0) <- normalizát mantissza  ST(1) <- az alapszám (2) kitevője  A példában:  120,1 ≈ 1,8765624761581420 x 26 |  |

## FPU összehasonlítás

|  |  |  |
| --- | --- | --- |
| FCOM, FCOMP, FCOMPP  FICOM, FICOMP | ST(0) és src FP összehasonlítása  T > src: C3, C2, C0 ← 000  ST < src: C3, C2, C0 ← 001  ST = src: C3, C2, C0 ← 100 |  |
| FCOMI | ST(0) és ST(i) FP összehasonlítása  ST(0) > ST(i): ZF, PF, CF ← 000  ST(0) < ST(i): ZF, PF, CF ← 001  ST(0) = ST(i): ZF, PF, CF ← 100 |  |
| FTST | ST(0) és 0.0 FP összehasonlítása  T > src: C3, C2, C0 ← 000  ST < src: C3, C2, C0 ← 001  ST = src: C3, C2, C0 ← 100 |  |
| FXAM | A C3C2C0 bitekkel megadja ST(0) formátumának kategóriáját:  1. Nem támogatott 0 0 0  2. NaN 0 0 1  3. Normalizált véges szám 0 1 0  4. Végtelen 0 1 1  5. Nulla 1 0 0  6. Üres 1 0 1  7. Denormalizált szám 1 1 0 |  |

## FPU szinusz, koszinusz, logaritmus…

|  |  |  |
| --- | --- | --- |
| FSIN, FCOS |  |  |
| FSINCOS | ST(0) <- COS(ST(0))  ST(1) <- SIN(ST(0)) | double hat = 6.0;  \_asm{  fldpi;  fld hat;  fdiv;//ST(0) <- \_PI/6 radian (30 fok)  fsincos;  } |
| FPTAN | Hányados tangense  ST(0) <- tg(ST(1)/ST(0)) |  |
| FPATAN | Hányados arkusz tangense  ST(0) <- arctg(ST(1)/ST(0)) |  |
| F2XM1 | ST(0) <- 2ST(0) - 1 |  |
| FYL2X | ST(0) <- ST(1) \* ld(ST(0)) | double xx = 1024.0;  double yy = 10.0;  \_asm{  fld yy;  fld xx;  fyl2x;//yy\*ld(xx)  } |
| FYL2XP1 | ST(0) <- ST(1) \* ld(ST(0) + 1) |  |

## Beépített konstansok

|  |  |  |
| --- | --- | --- |
| FLD1 |  |  |
| FLDZ | ST(0) <- +0.0 |  |
| FLDPI |  |  |
| FLDL2E | ST(0) <- ld(e) |  |
| FLDLN2 | ST(0) <- ln(2) |  |
| FLDL2T | ST(0) <- ld(10) |  |
| FLDLG2 | ST(0) <- lg(2) |  |

## Az FPU működési környezete

|  |
| --- |
| Az FPU működési környezete |
| A 16 bites FPU állapotszó |
| A 16 bites FPU vezérlőszó |
| A feltétel kód bitek és az EFLAGS regiszter |
| Az FPU regiszterek jellemzői a TAG regiszterben |

|  |  |  |
| --- | --- | --- |
| FINCSTP/FDECSTP | A lebegőpontos veremmuutatót növeli/csökkenti |  |
| FFREE | Az FPU regiszter jellemző bitjeit 11-be írja („empty”) |  |
| FINIT/FNINIT | Az FPU inicializálása |  |
| FLDCW/FSTCW | A vezérlőszó írása/olvasása |  |
| FSTENV m28  FLDENV | Az FPU működési környezetét írja a memória 28 byte-jába  (vezérlőszó, állapotszó, tag bitek, programszámláló, adat cím) | unsigned char fpuenv[28];  \_asm{  fstenv fpuenv;  fnop;  fldenv fpuenv;  } |
| FSAVE m108  FRSTOR | Az FPU működési környezetét és a regiszterek tartalmát írja a memória 108 = 28 + 80 byte-jába  (vezérlőszó, állapotszó, tag bitek, programszámláló, adat cím, regiszter tartalmak) | unsigned char fpustate[108];  for (int i = 0; i < 108; i++)  fpustate[i] = 0xaa;  \_asm{  mov ecx, 8;  cikl: fldpi;  loop cikl;  fsave fpustate;  nop;  frstor fpustate;  } |
| FNOP |  |  |
| WAIT/FWAIT | Megvárja a folyamatban lévő lebegőpontos kivételek befejeződését |  |
| FXSAVE m512  FXRSTOR | Elmenti az FPU, MMX, XMM és MXCSR regiszterek tartalmáz egy 512 byte-os memóriaterületre |  |

# C Függelék: IA32 gépi utasítás-végrehajtás vizsgálata Visual Studio-val

A tennivalókat Visual Studio 2019 fejlesztői környezettel mutatjuk be, de értelemszerűen átvihetők más környezetre is.

**1.** A Visual Studio indítása, „minta” nevű project létrehozása:

|  |  |
| --- | --- |
|  | |
|  |  |

**2.** Létrejön a **minta.cpp** állomány:

|  |
| --- |
|  |

Ide írjuk a programunkat.

**3.** A C/C++ szerint írhatjuk a változódefiníciókat, értékadásokat stb., majd következik az **\_asm{}** blokk. Ide írjuk a gépi utasításokat **mnemonik**us alakban. Hasznos a legutolsó **nop** utasítás, mert lépésenkénti utasítás-végrehajtás esetén a nop előtti utasítás végrehajtása után a nop-ra kerül a vezérlés, és még az \_asm{} blokkon belül vagyunk (a következő példában a 15. sor).

|  |
| --- |
|  |

**4.** A futtatható kódot a **Build** menüpont **Build Solution** parancsával készíthetjük el:

|  |
| --- |
|  |

Üzenet jelzi, ha nincs szintaktikai hiba, és a kódelőállítás sikeres volt:

**5.** A gépi utasítások működésének vizsgálatához **töréspont**ot kell elhelyezni. A töréspontig futtatva a programot számos új menüpont keletkezik, köztük azok is, amikre szükségünk van:

|  |
| --- |
|  |

A töréspont elhelyezéséhez a kívánt programsorra állunk a szövegkurzorral. Háromféleképp is elhelyezhetjük a töréspontot:

1. a **Debug** menü **Toggle Breakpoint** menüpontjával, vagy
2. az **F9** funkcióbillenytű lenyomásával, vagy
3. a bal oldali szürke sávra kattintva.

Egy **piros pötty** jelzi a töréspontot. Több töréspont is elhelyezhető.

Példánkban a 15. sorra helyeztünk el töréspontot. Ha a programot futtatjuk, eddig az utasításig fut, ezt már nem hajtja végre: ez lesz a soron következő végrehajtandó utasítás (**nop**).

**6.** Futtatjuk a programot a töréspontig (**Start Debugging** menüpont vagy **F5** funkcióbillentyű):

|  |
| --- |
|  |

A futás a törésponton megáll. Itt több dologra érdemes odafigyelni.

1. A töréspont piros pöttyében megjelent egy **sárga nyíl**: ez mutatja a soron következő végrehajtandó utasítást.
2. A **Debug** menü **Windows** menüpontjában több új lehetőség is megjelent. Ezek közül feladatainkhoz szükségesek a következők: **Watch, Memory,Disassembly, Registers**.
   * 1. A **Watch ablak**ban a változók neve, címe, értéke stb. jeleníthető meg.
     2. A **Memory ablak**ban a memória tartalma tekinthető meg többféle módon.
     3. A **Disassembly ablak**ban a gépi program jeleníthető meg többféleképp.
     4. A **Registers ablak**ban a processzor regiszterek tartalmát tekinthetjük meg.

Ezután a programot lépésenként is végrehajthatjuk az **F10** funkcióbillentyű nyomogatásával. Rendre az aktuális állapotot láthatjuk a kiválasztott ablakokban, így a gépi utasítás-végrehajtás követhető.

Előfordulhat, hogy debug-olás közben szemantikai hibát találunk a programunkban. Ekkor befejezzük a debug-olást (**Debug** | **Stop debugging**), kijavítjuk a forrásprogramban a hibát, újra fordítjuk a programot a Build-del, és ellenőrizzük a javított kódot a fentebb bemutatott töréspontos módszerrel.

|  |
| --- |
|  |