Számítógép architektúrák

- **Számítógépek felépítése**
- **Digitális adatábrázolás**
- **Digitális logikai szint**
- **Mikroarchitektúra szint**
- **Gépi utasítás szint**
- **Operációs rendszer szint**
- **Assembly nyelvi szint**
- **Probléma orientált (magas szintű) nyelvi szint**
- **Perifériák**

Operandus megadás

- **Közvetlen operandus** (immediate operand): Az operandus megadása az utasításban:

```
MOV(37,EAX);
```

- **Regiszter címzés** (register addressing): Assemblyben általában a nevükkel hivatkozunk a regiszterekre. (Gépi kódban viszont mindig számokkal --- éppúgy, mint a direkt címzésnél.)
- **Direkt címzés** (direct addressing): A memóriacím megadása az operandusban. Az utasítás mindig ugyanazt a címet használja. Az operandus értéke változhat, de a címe nem (fordításkor ismert kell legyen!).

```
MOV([$AB56_1200],EAX);ADD(x,EAX);
```

- **Regiszter-indirekt címzés** (register indirect addresing): A címrészen valamelyik regisztert adjuk meg, de a megadott regiszter nem az operandust tartalmazza, hanem azt a **memóriacímet**, amely az operandust tartalmazza (**mutató - pointer**).

  Rövidebb és a cím változtatható.

```
MOV(&x,EBX); ...; MOV([EBX],EAX);
```

Önmódosító program (Neumann): Ma már kerülendő (cache problémák!), pl. regiszter-indirekt címzéssel kikerülhetjük.

Pl.:, a 100 szóból álló **A** tömb elemeinek összeadása (egy elem 4 bájt), ~ **5.18. ábra.** (átírva Pentiumra)

```
    MOV  (0,EAX);// gyűjtsük az eredményt EAX-ben,
                 //  kezdetben ez legyen 0.
    MOV  (&A,EBX);// az A tömb címe
    MOV  (&A+400,ECX);// a tömb utáni első cím
c:  ADD  ([EBX],EAX);//  regiszter-indirekt címzés a
                 // tömb aktuális elemének elérésére
    ADD  (4,EBX);// EBX tartalmát növeljük 4-gyel
    CMP  (EBX,ECX);// végeztünk?
    JB   c;//  ugrás a C címkéhez, ha nem
    ...      // kész az összegzés
```

Indexelt címzés (indexed addressing): Egy eltolási érték (offset) és egy (index) regiszter tartalmának összege lesz az operandus címe, **5.19-20. ábra**. (átírva Pentiumra)

```
    MOV  (0,EAX);// gyűjtsük az eredményt EAX-ben,
                 //  kezdetben ez legyen 0.
    MOV  (0,ECX);// az index kezdő értéke
    MOV  (100,EBX);// a tömb mérete
c:  ADD  (A[ECX*4],EAX);//  indexelt címzés a
                 // tömb aktuális elemének elérésére
    ADD  (1,ECX);// ECX tartalmát növeljük 1-gyel
    CMP  (ECX,EBX);// végeztünk?
    JB   c;//   ugrás a C címkéhez, ha nem
    ...      // kész az összegzés
```

- **Bázisindex címzés** (based-indexed addressing): Egy eltolási érték (offset) és két (egy bázis és egy index) regiszter tartalmának összege lesz az operandus címe. Ha **EBX** ***A*** címét tartalmazza, akkor

```
C:      ADD(A[ECX*4],EAX);//
```

helyett a

```
C:      ADD([EBX+ECX*4],EAX);//
```

utasítás is írható.

- **Verem címzés** (stack addressing): Az operandus a verem tetején van. Nem kell operandust megadni az utasításban. (Pl. az **IJVM IADD** utasítása.)

Az Intel 8086/8088 társzervezése

A memória byte szervezésű.
Egy byte 8 bitből áll. word, double word.

Byte sorrend: Little Endian (LSBfirst).

A negatív számok 2-es komplemens kódban.

Szegmens, szegmens cím
a szegmensen belüli „relatív” cím, logikai cím, virtuális cím, OFFSET, displacement, eltolás, Effective Address (EA)
fizikai cím (Address)

Az Intel 8086/8088 üzemmódjai

valós (real) védett (protected)

szegmens cím

szegmens regiszter

↓

szegmens regiszter page tábla elem

tartalma * 16 ↓

szegmens kezdőcíme

fizikai cím =

szegmens kezdőcíme + szegmensen belüli cím

Az Intel 80x86 CPU regiszterei

- Általános célú regiszterek
- Speciális regiszterek
  - alkalmazások számára hozzáférhető
    - STATUS/Flag regiszter (SR, Flags, F, EF)
    - (Utasítás számláló (IP, EIP))
  - operációs rendszer (ill. a vezérlő) számára fenntartott
- szegmens regiszterek
  - CS: Kód
  - SS: Verem
  - DS: Adat
  - ES, FS, GS: Extra adat

Az Intel 80x86 CPU általános regiszterei

- 8 bites regiszeterek
  - **AL, AH, BL, BH, CL, CH, DL, DH**
- 16 bites regiszterek
  - **AX, BX, CX, DX, SI, DI, BP, SP**
- 32 bites regiszterek
  - **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**
- Átfedés van közöttük:
  - **AX=EAX%65536=256*AH+AL , BX, CX, DX**
  - **SI=ESI%65536 , DI, BP, SP**
- „jelentés” a név mögött:
  - AX=Accumulator, BX=Base register,
    CX=Counter register, DX=Data register,
    SI=Source Index, DI=Destination Index
    BP=Base_Pointer, SP=Stack_Pointer (EZEKET NE HASZNÁLD!)
  - AL=Accumulator Low, AH=Accumulator High, stb...
  - EAX=Extended Accumulator, stb...

A flag regiszter

Egyedi „zászlók” - logikai változók – gyűjteménye

- Állapot jelzők
  - bit-0: CF - Carry – átvitel / előjel nélküli túlcsordulás
  - bit-2: PF - Parity – párosság
  - bit-4: AF - Auxiliary carry – közbenső átvitel
  - bit-6: ZF - Zero – zéró
  - bit-7: SF - Sign – előjel
  - bit-B: OF - Overflow – Előjeles Túlcsordulás
- Vezérlő bitek
  - bit-8: TF - Trace – nyomkövető
  - bit-9: IF - Interrupt enable – Megszakítás engedélyezés
  - bit-A: DF - Direction – Irány jelző

----ODIT.SZ-A-P-C

Címzésmódok (operandusok megadási módjai)

```
1: konstans, azonnali    MOV(54,AL); ADD($54,AL);
2: regiszter             MOV(54,AL); SUB(BH,AL);
   – Memória címzések:
3: direkt, közvetlen     MOV(AX,[$AF4D]); MOV(valt,AX);
4: regiszter indirekt    MOV([EAX],EAX); ADD(34,[EBX]);
5: bázis relatív         MOV([EBX+23],BL); MUL(valt[ESI]);
5: indexelt              MOV([$AD34+ECX*2],AX);MUL(v[esi*2])
7: bázisrelatív indexelt MUL(v[EBX+ECX*2])

HLA:  MOV($54,AL)
MASM: MOV AL,54H
```

Direkt memória címzés: a címrészen az operandus logikai címe (eltolás, displacement)

```
MOV     AX, SZO     ; AX új tartalma SZO tartalma
MOV     AL, KAR     ; AL új tartalma KAR tartalma
```

Valahol a **DS** által mutatott szegmensben:

```
SZO     DW      1375H
KAR     DB      3FH
```

(DS:SZO) illetve **(DS:KAR)**

```
MOV AX, KAR                 ; szintaktikailag hibás
MOV AL, SZO                 ; szintaktikailag hibás
MOV AX, WORD PTR KAR ; helyes, de …
MOV AL, BYTE PTR SZO    ; helyes, de …
MOV((type byte SZO),AL);//HLA esetén (típus kényszerítés)
```

Program terület címzés (16 ill. 32 bites módban)

Automatikus szegmens regiszter: **CS**

A végrehajtandó utasítás címe: **(CS:IP)** (CS:EIP)

Egy utasítás végrehajtásának elején:

IP = IP + az utasítás hossza. (**EIP = EIP**+hossz)

- **IP relatív címzés:**
  IP = IP + az előjeles közvetlen operandus
  (**EIP = EIP +** az előjeles közvetlen operandus)
- **Direkt utasítás címzés:** Az operandus annak az utasításnak a címe, ahova a vezérlést átadni kívánjuk.

  Közeli **(NEAR):** **EIP** ← a 16 (32) bites operandus

  Távoli **(FAR):** **(CS:EIP)** ← a 32 (48) bites operandus.

  CALL VALAMI ; az eljárás típusától függően
  ; **NEAR** vagy **FAR**

- **Indirekt utasítás címzés:** Bármilyen adat címzési móddal megadott szóban vagy dupla szóban tárolt címre történő vezérlés átadás. Pl.:

```
JMP AX      ; ugrás az AX-ben tárolt címre
JMP [BX] ; ugrás a (DS:BX) által címzett
; szóban tárolt címre.
JMP FAR [BX] ; ugrás a (DS:BX) által
; címzett dupla szóban tárolt címre.
```

JMP(EAX);

JMP([EBX]);

Linux/Windows (HLA) : Nincs Távoli ugrás/eljárás

Az utasítások szerkezete

prefixum	*operációs kód*	*címzési mód*	*Operandus(ok)*
0 - 2 byte	1 byte	0 - 2 byte	0 - 8 byte

- ***Prefixum*:**
  - utasítás (ismétlés / LOCK),
  - explicit szegmens megadás: **MOV AX, CS:S ;** S nem a DS-ben, hanem CS-ben van**,**
  - Cím méret módosítás (16/32 bit)
  - Operandus méret módosítás (16/32 bit)
- ***Operációs kód*:** szimbolikus alakját mnemonic-nak nevezzük
- ***Címzési mód*:** hogyan kell az operandust értelmezni
  - ***Mod-r/m byte***
  - ***SIB byte***
- ***Operandus*:** mivel kell a műveletet elvégezni
  - Memória cím / eltolás
  - Azonnali operandus – konstans

A gépi kód (ADD utasítás néhány formája)

- %000000_d_w %mm_reg_r/m [SIB] [DISP vagy disp]

  %000000 -> ADD ; d -> melyik a forrás ;
  w -> operandus méret (16bit: 1-2; 32bit 1-4)

  mm -> mod (0-1-2: memória, 3 regiszter)

  regiszter:

  ```
   8 bit:  AL, CL, DL, BL, AH, CH, DH, CH
  16 bit:  AX, CX, DX, BX, SP, BP, SI, DI
  32 bit: EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI
  ```

  r/m, ha memória: DISP+

16bit: **BX+SI, BX+DI, BP+SI, BP+DI, SI, DI, BP, BX**

Kivétel: Ha mm=0 és r/m=5 ==> (BP helyett csak) DISP

32bit: **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**

mod<11 és r/m=4 --> **SIB byte**: %sk_ind_bas

mod=00 és r/m=5 --> 32-bites eltolas (DISP)

- %100000_s_w %mm_000_r/m [SIB] [DISP vagy disp] data

  s --> adat előjelkiterjesztése (1byte-ról „méretre”)

- %0000010_w data (data)

  ADD(kons,AL); ADD(kons,AX); ADD(kons,EAX);

Adatmozgató utasítások (Minden flag változatlan)

- **MOV(forrás,cél); //Nincs (m,m), (sr,sr), (k,sr)**
- **MOVSX(forrás,cél_reg);MOVZX(forrás,cél_reg)**
- **XCHG(m/r,m/r); // azonos méret, NINCS m,m**
- **XLAT; // MOV([EBX+AL],AL)**
- **LEA(mem,cél_reg32);// "=" MOV(&mem,cél_reg32)**
- **LAHF; SAHF; // Load, Save az AH szempontjából ..**
- **CBW; CWD; CDQ; //movsx(al,ax);movsx(ax,dx:ax)**
  //movsx(eax,edx:eax)

VEREM műveletek

- **PUSH(forrás);//mem, reg, 16/32 bit**
  - **PUSHW(forrás); // mem16, reg16, konst; ESP-=2**
  - **PUSHD(forrás); // mem32, reg32, konst; ESP-=4**
- **POP(forrás); //mem, reg, 16/32 bit**
- **PUSHF; PUSHFD;**
- **POPF; POPFD; // Flagek mégis változhatnak!!**
- **PUSHA; PUSHAD;**
- **POPA; POPAD;**

Aritmetikai utasítások

- **ADD(forrás,cél);SUB(forrás,cél);//cél =forrás**
- **ADC(forrás,cél);SBB(forrás,cél);//cél =forrás+CF**
- **INC(m/r); DEC(m/r);// CF nem változik**
- **NEG(m/r);// cél= 0-cél**
- **CMP(cél,forrás);//mint a SUB(forrás,cél), de nem tárol**

Feltételes ugrások CMP után

- **J... címke;**
  - **pozitív mennyiségek hasonlítása után**
    JA, JB, JE, JAE, JBE, Above
    JNA, JNB, JNE, JNAE, JNBE Below
  - **előjeles mennyiségek hasonlítása után**
    JG, JL, JE, JGE, JLE, Greater
    JNG, JNL, JNE, JNGE, JNLE Less
- **INTMUL(forrás,cél);intmul(kons,forrás,cél);**
- **MUL(forrás);IMUL(forrás);**
- **DIV(osztó);IDIV(osztó);//kivétel**

Az ADD utasítás

```
MOV(-83,AL);ADD(121,AL);
```

```
173   %1010_1101   -83
121  +%0111_1001  +121
 38   %0010_0110   +38
```

```
CF=1  (38<>173+121=38+1*256)
PF=0  (3db 1-es bit)
AF=1  (13+9=6+1*16)
ZF=0  (38<>0)
SF=0  (+38>=0)
OF=0  ((-83)+(+121)=(+38))
```

```
MOV(173,AL);ADD(AL,AL);
```

```
173   %1010_1101   -83
173  +%1010_1101   -83
 90   %0101_1010   +90
```

```
CF=1  (90<>173+173=90+1*256)
PF=1  (4db 1-es bit)
AF=1  (13+13=10+1*16)
ZF=0  (90<>0)
SF=0  (+90>=0)
OF=1  ((-83)+(-83)<>(+90))
```

Az ADD utasítás

```
MOV(73,AL);ADD(121,AL);
```

```
 73   %0100_1001    +73
121  +%0111_1001   +121
194   %1100_0010    -62
```

```
CF=0  (194=73+121=194+0*256)
PF=0  (3db 1-es bit)
AF=1  (9+9=2+1*16)
ZF=0  (192<>0)
SF=1  (-62<0)
OF=1  ((+73)+(+121)<>(-62))
```

```
MOV(41,AL);ADD(133,AL);
```

```
 41   %0010_1001    +41
133  +%1000_0101   -123
174   %1010_1110    -82
```

```
CF=0  (174=41+133=174+0*256)
PF=0  (5db 1-es bit)
AF=0  (9+4=14+0*16)
ZF=0  (174<>0)
SF=1  (-82<0)
OF=0  ((+41)+(-123)=(-82))
```

„Program szervezés” -- elágazások

```
MOV(73,AL);
ADD(121,AL);
If(@s)then
  neg(al);
Endif;
Cmp(valt,EAX);
If(@AE)then
  add(7,eax);
Endif;
Cmp(EAX,765);
If(@ge)then
   sub(3,EAX);
Else
   add(3,eax);
Endif;
```

```
  MOV(73,AL);
  ADD(121,AL);
  JNS kihagy1;
  Neg(al);
kihagy1:
  Cmp(valt,EAX);
  JNAE kihagy2;
  add(7,eax);
kihagy2:
  Cmp(EAX,765);
  Jl hamis;
  sub(3,EAX);
  jmp vege;
hamis: add(3,eax);
vege:
```

MUL és IMUL utasítások

- **MUL(op8);// AX =op8 * AL**
- **MUL(op16);// DX:AX =op16* AX**
- **MUL(op32);// EDX:EAX=op32*EAX**

- **Az operandus nem lehet konstans (mem/reg)**

- **MUL – előjel nélküli tényezők, szorzat**
- **IMUL – előjeles tényezők, szorzat**

- **OF=CF=(AX<>AL) ill. DX:AX<>AX ….**
- **Többi állapotjelző határozatlan!!!**

DIV és IDIV utasítások

- **DIV(op8);// AL= AX/op8 AH=AX%op8**
- **DIV(op16);// AX= (DX:AX) / op16**
  DX= (DX:AX) % op16
- **DIV(op32);// EAX=(EDX:EAX) / op32 // hányados**
  EDX=(EDX:EAX) % op32 // maradék

{EAX=7;EDX=0;EBX=3} ==> DIV(EBX) ==> {EAX=2;EDX=1}

- **Az operandus nem lehet konstans (mem/reg)**

- **DIV – előjel nélküli operandusok**
- **IDIV – előjeles operandusok**

{EAX=-7;EDX=-1;EBX=-4} ==> DIV(EBX) ==> {EAX=1;EDX=-3}

- **Minden állapotjelző határozatlan!!!**
- **Ha a hányados nem ábrázolható 8/16/32 biten, akkor „eltérülést” (megszakítást) okoz az utasítás végrehajtása.**

Logikai utasítások

- OR(forrás,cél);
- XOR(forrás,cél);
- AND(forrás, cél);
- TEST(forrás,cél);// TEST(AL,AL); TEST(8,AL)

- PF, ZF, SF: értelemszerűen
- CF=OF=0
- AF: határozatlan

- NOT(cél); // Minden flag változatlan

Forgatások, léptetések

XXX(számláló,cél-operandus);

- **Léptetések: SHL, SHR, SAL, SAR**
- **Forgatások: ROL, ROR, RCL, RCR**
- **Számláló: konstans vagy a CL regiszter**

Az állapotjelzők:

- **Ha a számláló egyenlő nullával, akkor minden állapotjelző (és a cél-operandus is) változatlan marad.**

Egyébként:

- **PF, ZF, SF, AF:**
  - **Léptetéseknél: értelemszerűen, ill. AF határozatlan**
  - **Forgatásoknál: változatlanok**
- **CF: Az utoljára „kilépő/átforduló” bit.**
- **OF:**
  - **Ha a számláló értéke 1, akkor OF jelzi, hogy megváltozott-e az előjelbit értéke a művelet hatására.**
  - **Ha a számláló nagyobb egynél, akkor OF határozatlan értékű lesz.**

Forgatások, léptetések

```
MOV(%1011_0111,AL);//   AL=%1011_0111

SHL(1,AL);          //   AL=%0110_1110  CF=1  OF=1
SHR(1,AL);          //   AL=%0011_0111  CF=0  OF=0
SHR(1,AL);          //   AL=%0001_1011  CF=1  OF=0

ROR(1,AL);          //   AL=%1000_1101  CF=1  OF=1
ROR(1,AL);          //   AL=%1100_0110  CF=1  OF=0
ROR(1,AL);          //   AL=%0110_0011  CF=0  OF=1
RCR(1,AL);          //   AL=%0011_0001  CF=1  OF=0
RCR(1,AL);          //   AL=%1001_1000  CF=1  OF=1

SAR(1,AL);          //   AL=%1100_1100  CF=0  OF=0
SAR(1,AL);          //   AL=%1110_0110  CF=0  OF=0
SHR(1,AL);          //   AL=%0111_0011  CF=0  OF=1
SAR(1,AL);          //   AL=%0011_1001  CF=1  OF=0
SAR(1,AL);          //   AL=%0001_1100  CF=1  OF=0
```

Forgatások, léptetések

```
MOV(%1011_0111_1110_0101,AX);
MOV(%1100_0010_0001_0010,BX);
                  // AX=%1011_0111_1110_0101
SHRD(3,BX,AX);    // AX=%0101_0110_1111_1100  CF=1
SHLD(3,BX,AX);    // AX=%1011_0111_1110_0110  CF=0
```

Közvetlen bitelérések

- **Flag regiszter bitjei**
  - **CLC, CLD, CLI, STC, STD, STI, CMC**
- **BT(hányadik, miben); // Bit Test CF-et állítja**
- **BTC, BTR, BTS; // BT and Complement, Reset, Set**
- **miben: mem/reg16/reg32**
- **hányadik reg/konst: 0--15/31**
  - **regiszter méret**
  - **Max. 255 – ha hányadik konstans**
  - **limit nélkül, ha hányadik reg és miben mem**
  - **ha reg, akkor azonos méretű mibennel**

- **BSF(miben,hova);// legalacsonyabb helyiértékű 1-es bit sorszáma mibenben --> hova / ha ZF=0**
- **BSR(miben,hova);// legmagasabb helyiértékű 1-es bit sorszáma mibenben --> hova**
  - **miben és hova azonos méretű (16/32 bit)**
  - **hova: regiszter**

Sztringkezelő utasítások

Adatmozgató utasítások

- `LODSB; LODSW; LODSD; // DS:ESI // direction flag`
- `STOSB; STOSW; STOSD; // ES:EDI // AL, AX, EAX`
- `MOVSB; MOVSW; MOVSD; // ESI, EDI`

hasonlító utasítások

- `SCASB; SCASW; SCASD; // CMP(Acc,[EDI])//Acc: AL, AX, EAX`
- `CMPSB; CMPSW; SMPSD; // CMP([ESI],[EDI]) // [ESI]-[EDI]`

Ismétlő prefixek `// ECX`

- `REP.MOZGATO`
- `REPE.HASONLITO`
- `REPNE.HASONLITO`

HLA sztringjei

- str.strRec, melynek mezői:
  - max-hossz; (uns32) // rekord címe - 8
  - hossz; (uns32) // rekord címe - 4
  - maga a sztring + lezáró 0 + üres // Ennek a mezőnek a címe, a rekord címe
- str.strRec.length, str.strRec.MaxStrLen
- STATIC str_var:string;// valójában mutatót hoz létre
- stralloc(max_hossz); mov(eax,str_var); .... ; strfree(str_var);
- stdin.gets(str_var); std_in.a_gets(); mov(eax,str_var2);
- stdout.put(str_var, nl, (type string EAX), nl);

A HLA (High Level Assembly) „Hello World!” programja

```
Program HelloWorld;
#include(”stdlib.hhf”)
begin HelloWorld;
stdout.put(”Hello World!”,nl);
end HelloWorld;

Program hw; // Sok-sok megjegyzés
#incule(”stdlib.hhf”)
begin hw;
stdout.put  // „szabad” szintaxis
(
   ”Hello”  // stringek automatikus összefűzése
     ” World!” nl  // nl == new line
);
end hw;
// Fordítás, futtatás: hla hw; hw ill. hla hw && hw
```

Alapvető könyvtári függvények, adat deklarációk

```
program plST;  // Azonosítók: [_a-zA-Z][_a-zA-Z0-9]*
#include(”stdlib.hhf);
static a:int32:=-7;
       b:int32;
begin plST;
stdout.put(”a értéke: ”,a,nl); // Írás a „képernyőre”
stdout.put(”írj be egy számot:”);
stdin.get(b);              // Olvasás a „billenytűzetről”
stdout.put(”A beírt szám: ”,b,nl);
end plST;

//„static”   : adat deklarációs rész megkezdése
//„a”, „b:” : Változók nevének megadása („neutrális”)
//„:int32”    : típus - 32 bites előjeles egész szám
//„:=-7”     : kezdőérték megadása  -- nem kötelező
```

Általános memória használat

```
.....: storage szekció   ---
.....: static szekció    ----(@nostorage)
.....: read-only szekció --- (align(4)  )
.....: rendszer konstansok
.....: program kód       --- byte 3,7,9;(pseudo opcode)
16  MB: halom (default méret, azaz a fordítás parancs-
          sorában megadható opciókkal módosítható)
16  MB: verem (default méret) (VAR szekció)
128 KB: Operációs rendszer

Alignment:1-2-4-8-16, „nagy objektumok”: 8-16
align(x); // A következő adat úgy lesz elhelyezve, hogy
  // címe osztható legyen x-el
```

Programszerkezeti elemek

- **„Feltételek” formája:**
  - **a<ebx, al<=6, <>, !=, =, ==, >=, <, >**
  - **@z, @ae,**
  - **{!}boolean_var,**
  - **reg {not} in low..Hi**
- **Elágazások**
  - **if(feltétel) utasítások {{elseif(feltétel) utasítások} else utasítások} endif**
- **Ciklusok**
  - **while(felt) do ut... endwhile**
  - **for(ut;felt;utasítás) do ut... endfor**
  - **repeat ut... until(felt)**
  - **forever ut.... endfor**
    - **break; breakif(felt);**
    - **continue; continueif(felt);**
  - **begin nev; ut...; end nev;**
    - **exit nev; exitif(felt) nev;**

Eljárások, paraméter átadás

- **Procedure elj_nev(formális_paraméter_lista);@opciók;**
  lokális deklarációk
  begin elj_nev;
  utasítások
  end elj_nev;
- **Eljárás Hívása: elj_nev(aktuális_paraméter_lista);**
- **Paraméter átadás**
  - **Módja (formális paraméter elé írva)**
    - **VAL, VAR, RESULT, VALRES, ...**
  - **Helye (formális paraméter után lehet pl.: in EAX)**
    - **verem, regiszter**
- **Regiszterek megőrése**
- **lokális deklarációk**
  - **láthatóság**
  - **élettartam**
- **exit elj_nev;exitif(felt) elj_nev;**

Rekurzív és re-entrant eljárások

Egy eljárás **rekurzív**, ha önmagát hívja közvetlenül, vagy más eljárásokon keresztül.

Egy eljárás **re-entrant**, ha többszöri belépést tesz lehetővé, ami azt jelenti, hogy az eljárás még nem fejeződött be, amikor újra felhívható. A rekurzív eljárással szemben a különbség az, hogy a rekurzív eljárásban „programozott”, hogy mikor történik az eljárás újra hívása, re-entrant eljárás esetén az esetleges újra hívás ideje a véletlentől függ. Ez utóbbi esetben azt, hogy a munkaterületek ne keveredjenek össze, az biztosítja, hogy újabb belépés csak másik processzusból képzelhető el, és minden processzus saját vermet használ.

Rekurzív és re-entrant eljárások

Ha egy eljárásunk készítésekor betartjuk, hogy az eljárás a paramétereit a vermen keresztül kapja, kilépéskor visszaállítja a belépéskori regiszter tartalmakat – az esetleg eredményt tartalmazó regiszterek kivételével –, továbbá a fenti módon kialakított munkaterületet használ lokális változói elhelyezésére, akkor az eljárásunk rekurzív is lehet, és a többszöri belépést is lehetővé teszi (re-entrant).

CPU Állapot (regiszterek) megőrzése

```
Program nem_mukodik;
#include("stdlib.hhf")
procedure tiz_szokoz;
begin tiz_szokoz;
mov(10,ecx);
repeat stdout.put(' ');dec(ecx);until(@z);
end tiz_szokoz;

begin nem_mukodik;
mov(20,ecx);
repeat tiz_szokoz();stdout.put("*",nl);dec(ecx);
until(@z);
end nem_mukodik;
// Javítás:
// valaki (hívó/hívott) menti ECX-et
// PUSH(ECX); . . . POP(ECX);
```

Lokális változók

```
program demo;#include(„stdlib.hhf”)
static i:uns32:=10;j:uns32:=20;

Procedure elso;var i:int32;j:uns32;begin eslo;
mov(10,j);for(mov(0,i);i<10;inc(i))do
stdout.put(„i,j=”,i,” „,j,nl);dec(j);endfor;
end elso;

procedure masodik;var i:int32;begin masodik;
mov(10,j);for(mov(0,i);i<10;inc(i))do
stdout.put(„i,j=”,i,” „,j,nl);dec(j);endfor;
end masodik;

begin demo;
elso(); masodik();
stdout.put(„i=”,i,”   j=”,j,nl);
end demo;
```

Paraméter átadás

- **VAL: Procedure demo_val(N:uns32);// VALUE**
  - **demo_val(10);//konstans**
  - **demo_val(eax);//32 bites regiszter**
  - **demo_val(uns32_valtozo);// nem fog**
  - **demo_val(dword_valtozo);// megváltozni**
- **VAR: Procedure demo_var(VAR N:uns32);// VARIABLE**
  - **demo_var(uns32_valtozo);// "meg fog"**
  - **demo_var(dword_valtozo);// változni**
    - **közvetlenül a címét kapjuk meg az eljárásban**
- **RESULT: Procedure demo_result(RESULT N:uns32);**
  - **demo_var(uns32_valtozo);// "meg fog"**
  - **demo_var(dword_valtozo);// változni**
    - **lokális másolatot kapunk, kezdőérték nélkül**
- **VALRES: Procedure demo_valres(VALRES N:uns32);**
  - **demo_var(uns32_valtozo);// "meg fog"**
  - **demo_var(dword_valtozo);// változni**
    - **lokális másolatot kapunk, kezdőértékkel**

Függvények, visszatérési érték

- FÜGGVÉNYEK: Olyan eljárások, melyek „fő feladata”, hogy egy konkrét „függvényértéket” meghatározzanak -kiszámoljanak-, és azt „visszadják” a hívónak.
- Kompozit utasítások
- A **„@returns”** opció
- **procedure betu_e(c:char);@returns(„eax”); ....**
  - **mov(betu_e(al),ebx);**
    - **betu_e(al);mov(eax,ebx);**
  - **if(betu_e(chr)) then ... endif;**
    - **betu_e(chr); if(eax) then ... endif;**

Eljárások opciói

- **@forward; @noframe; @nodisplay; @noalignstack; @external; @use reg32; @cdecl; @stdcall; @pascal**
- **Paraméterátadás helye:**
  - **regiszter**
  - **verem**
  - **kód**
- **Aktivációs Rekord**
- **CALL és RET utasítások**
  - **CALL elj_nev**
  - **CALL(reg/mem)**
- **@external;#include();// „nagy” programok,**
- **@external**
  - **csak a globális szinten**
  - **csak eljárás, static, readonly, storage**
  - **static c:char;@external(„var_c”);**
- **Lokális változók „igazítása”**
  - **var [4:1]//**
  - **var {[max{:min}]}{:=start}**

```
procedure AddandZero( var p1_ref: uns32; var p2_ref:uns32;
  p3:uns32 );@nodisplay;@noframe;
var p1: uns32; p2: uns32;
begin AddandZero;
push( ebp ); mov(esp,ebp); sub( _vars_, esp );
  AND($FFFF_FFFC,ESP);
// Note: _vars_ is "8" in this example.
push(eax); mov(p1_ref, eax); mov([eax], eax); mov(eax, p1);
  pop(eax);
// Actual procedure body begins here:
mov( p3, eax ); add( p1, eax ); Mov( eax, p2 ); mov( 0, p1 );
// Clean up code associated with the procedure's return:
push( eax ); push( ebx );
mov( p1_ref, ebx ); mov( p1, eax ); mov( eax, [ebx] );
mov( p2_ref, ebx ); mov( p2, eax ); mov( eax, [ebx] );
pop( ebx ); pop( eax );
mov(ebp,esp);pop(ebp); ret( 12 );
end AddandZero;//-----------------------
Procedure AaZ( VALRES p1:uns32; RESULT p2:uns32;
  p3:uns32);@nodisplay;
Begin AaZ;
Mov(p3,EAX); Add(p1,EAX); Mov(EAX,p2); Mov(0,p1);
End AaZ;
```

```
procedure uhoh( valres i:int32; valres j:int32 ); @nodisplay;
begin uhoh;
mov( 4, i ); mov( i, eax ); add( j, eax );
stdout.put( "i+j=", (type int32 eax), nl );
end uhoh;
...
var k: int32;
...
mov( 5, k ); uhoh( k, k ); // Mennyi lesz k értéke a hívás után?
... // Mi lenne („i+j”, ill. „k”), ha valres helyett VAR-t használnánk?
procedure DisplayAndClear( val i:int32 ); @nodisplay; @noframe;
begin DisplayAndClear;
push( ebp ); // NOFRAME, so we have to do this manually.
mov( esp, ebp ); and($ffff_fffc,esp);
stdout.put( "I = ", i, nl );
mov( 0, i );
mov(ebp,esp);pop( ebp );
ret(); // Note that we don't clean up the parameters.
end DisplayAndClear;
...
push( m );
call DisplayAndClear;
pop( m );
stdout.put( "m = ", m, nl );
```

Display:

```
procedure dummy(a:int64;b:int64);
var aa:int16;//[ebp-10];
      procedure dm(a:int8;b:int16);
      var x:int8;//[ebp-13]
          y:int16;//[ebp-15]
      begin dm;//push(ebp);push([ebp-4]);push([ebp-8]);
               //lea([esp+8],ebp);push(ebp);sub(4,esp);
               //and($FFFF_FFFC,esp);
      mov(y,ax);
      add(x,al);
      end dm; //mov(ebp,esp);pop(ebp);ret(8);
var bb:int8;//[epb-11]
begin dummy;//push(ebp);push([ebp-4]);lea([esp+4],ebp);
             //push(ebp);sub(4,esp);and($FFFF_FFFC,esp);
mov(aa,ax);
add(bb,al);
end dummy;//mov(ebp,esp);pop(ebp);ret(16);
```

„Pszeudo-utasítások”

A fordítónak – esetünkben az assemblernek-- szóló utasításokat nevezzük pszeudo-utasításoknak, ill assembler direktíváknak.

Ide tartoznak:

- A már megismert opciók -- pl:
  - @nodisplay módosítja az eljárás standard belépési kódját
  - @align(x) módosítja a változók címét
- Feltételes fordítás direktívái
- Macro definiálás/hívás direktívái
- „Pszeudo-változók” használata
- Egyéb, a fordítás során használható eszközök

Feltételes fordítás

A fordító programok általában – így az assembler is – feltételes fordítási lehetőséget biztosít. Ez azt jelenti, hogy a program bizonyos részeit csak abban az esetben fordítja le, ha – a fordítóprogram számára ellenőrizhető – feltétel igaz illetve hamis.

```
#if( constant_boolean_expression )
<< Statements to compile if the >>
<< expression above is true.    >>
#elseif( constant_boolean_expression )
<< Statements to compile if the >>
<< expression immediately above >>
<< is true and the first expres->>
<< sion above is false.         >>
#else
<< Statements to compile if both    >>
<< the expressions above are false. >>
#endif
```

Makró és blokk ismétlés

Makró definíció:

```
#macro identifier ( optional_parameter_list ) ;
statements
#endmacro
```

Pl.

```
#macro MyMacro;
  ?i = i + 1;// Módosítja az i nevű pszedo-változó értékét
#endmacro
#macro MacroWParms( a, b, c );
  ?a = b + c;
#endmacro
```

Makró hívás: (Deklarálás és láthatóság kapcsolata.)

```
procedure SomeProc;
begin SomeProc;
#macro mov0eax;
mov( 0, eax )
#endmacro
 ...
mov0eax;  // legal here
 ...
end SomeProc;

…

mov0eax; // undefined symbol here.
```

Dupla szavas összeadás:

```
#macro m_osszead(a) ;
  add((type uns32 a),EAX);
  adc((type uns32 a[4]),EDX);
#endmacro
procedure e_osszead(a:uns64);
begin e_osszead;
   add((type uns32 a),EAX);
   adc((type uns32 a[4]),EDX);
end e_osszead;
```

Ha a programban valahol dupla szavas összeadást kell végezzünk, akkor hívnunk kell az eljárást illetve a makrót:

MACRO: m_osszead(a); -->

```
add((type uns32 a),EAX);
adc((type uns32 a[4]),EDX);
```

ELJÁRÁS: e_osszead(a); -->

```
PUSH((type uns32 a[4]));PUSH((type uns32 a));
CALL e_osszead;

// ill. az eljárásban
PUSH(EBP);MOV(ESP,EBP);AND($FFFF_FFFC,ESP);
add((type uns32 a),EAX);
adc((type uns32 a[4]),EDX);
MOV(EBP,ESP);POP(EBP);RET(8);
```

Látható, hogy eljárás esetén legalább kettővel több utasítást kell végrehajtanunk, mint makró esetében (**`CALL`** és **`RET`**).

Még nagyobb különbséget tapasztalunk, ha regiszeter **`(pl. EDX:EAX)`** helyett paraméterként kívánjuk megadni az összeadandókat.

A fenti példában rövid volt az eljárás törzs, és ehhez képest viszonylag hosszú volt a paraméter átadás és átvétel. Ilyenkor célszerű a makró alkalmazása.

De ha a program sok helyéről kell meghívnunk egy hosszabb végrehajtandó programrészt, akkor általában célszerűbb eljárást alkalmazni.

Pszeudo változók, ill. „text”, mint macro:

```
val i:=0; // olyan, mint CONST, csak
  változhat

#macro SetI( v );
   ?i := v;
#endmacro

SetI( 2 );
```

The above macro and invocation is roughly equivalent to the following:

```
const
v : text := "2";


Add(i,EAX); // --> Add(0,EAX);
?i := v;
Add(i,EAX); // --> Add(2,EAX);
```

- #include(string_expression);
- #system("dir")
- #print(comma, separated, list, of, constant, expressions, ...)
- #error("Error, unexpected value. Value = " + #string(theValue))
- #while(constant_boolean_expression)
  << Statements to execute as long >>
  << as the expression is true. >>
  #endwhile
- #for(loop_control_var := Start_expr to end_expr
  << Statements to execute as long as the loop control variable's >
  << value is less than or equal to the ending expression. >>
  #endfor
- #for(loop_control_var := Start_expr downto end_expr ...
- #for(loop_control_var in composite_expr) …