Assembly
Matematika Assembly-ben

Iványi Péter

BCD aritmetika

- Speciális kódolás
  - Minden számjegy egy nibble-ben (4 bit) tárolunk

```
Dec          0    1    2    3    4
BCD       0000 0001 0010 0011 0100

Dec          5    6    7    8    9
BCD       0101 0110 0111 1000 1001
```

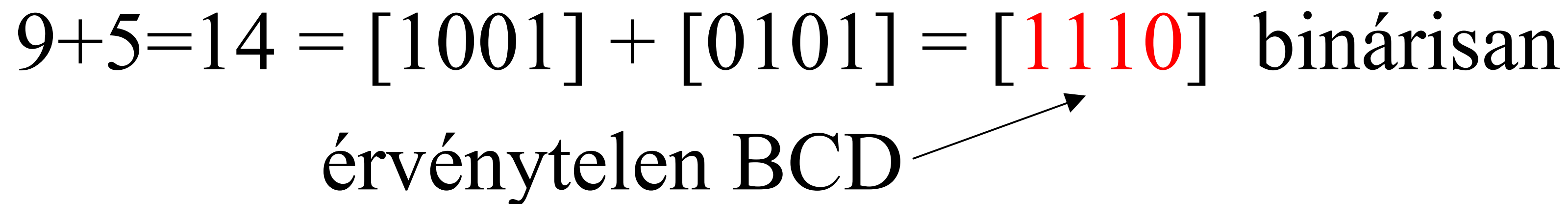
BCD aritmetika

- Decimális 127

```
0001  0010  0111
```

BCD aritmetika

- Összeadás
  - Összeadás bináris formában
  - Konverzió BCD formába
    - (Ha egy nibble 9-nél nagyobb akkor adjunk hozzá 6-ot)



6 1 4

BCD aritmetika

- Elektronikus rendszerekben használják, ahol
  - Számokat kell megjeleníteni és
  - Nincs mikroprocesszor

Matematikai koprocesszor

- A CPU integer (egész) műveleteket tud végezni
- Lehet emulálni a valós számokkal való számítást, de lassú
- FPU
  - matematikai processzor sokkal gyorsabban tud számolni
  - Verem alapú műveletek

Verem, kitérő

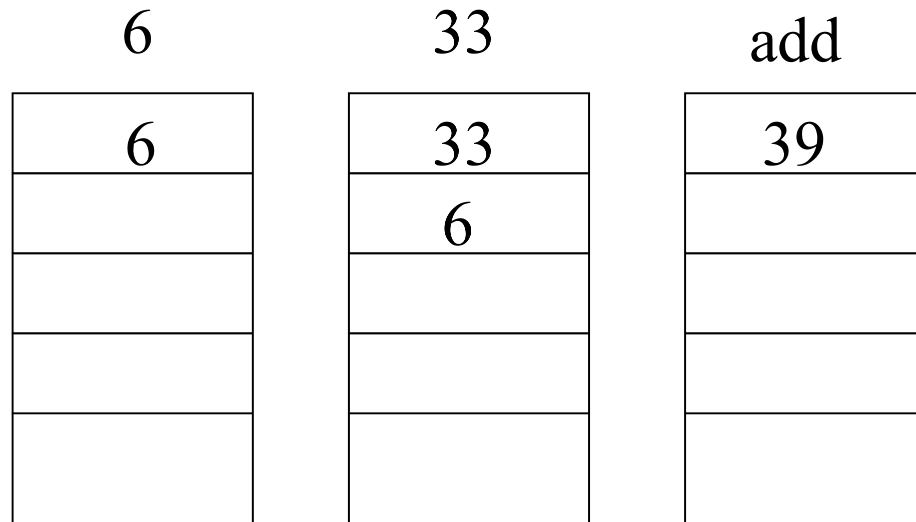
- Az utoljára bevitt adatot lehet először kivenni
- LIFO – Last In First Out
- Stack –nek is szokták nevezni
- Sok helyen használják
  - Operációs rendszerek
  - Függvények hívása

Postscript

- Programozási nyelv, Adobe Systems Inc.
  - PDF elődje
- Stack alapú
- Szótárakat (dictionary) használ
- Interpreter alapú – Postscript interpreter értelmezi a programot

Postscript műveletek

- Post-fix jelölés Matematikai jelöles
  - 6 33 add 6+33
- Az operandus PUSH-t jelent
- A művelet kiveszi a stack-ből az értékeket, majd az eredményt visszateszi a stack-re



$$6 + 3 / 8$$

Post-fix jelöléssel:

`3   8   div   6   add`

3	8	div	6	add
3	8	0.375	6	6.375
	3		0.375	

6 + 3 / 8

Post-fix jelöléssel:

6 3 8 div add

6	3	8	div	add
6	3	8	0.375	6.375
	6	3	6	
		6		

8 – 7 * 3

Két módszerrel:

```
8 7 3 mul sub
```

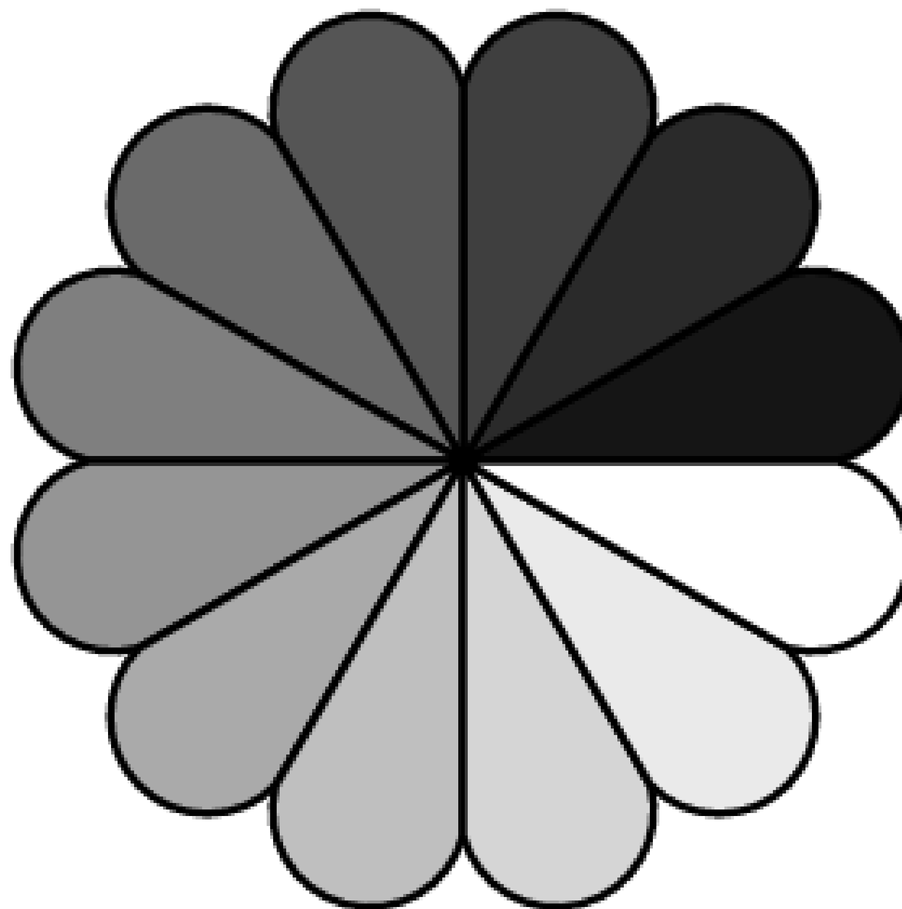
```
7 3 mul 8 exch sub
```

7 3 mul	8	exch	sub
21	8	21	-13
	21	8	

Példa program

```
%!PS-Adobe-2.0
/inch {72 mul} def
/wedge { newpath 0 0 moveto 1 0 translate 15 rotate
0 15 sin translate 0 0 15 sin -90 90 arc closepath } def
gsave
  4.25 inch 4.25 inch translate
  1.75 inch 1.75 inch scale
  0.02 setlinewidth
  1 1 12
  { 12 div setgray
    gsave wedge gsave fill grestore 0 setgray stroke grestore
    30 rotate
  } for
grestore
showpage
```

Példa program eredménye



FPU

- 8 regiszter
  - ST(0)-ST(7)
  - Mindegyik 80 bit (10 byte) hosszú
    - AX 16 bites
    - EAX 32 bites
- 3 regiszter, 16 bites
  - Status, Control, Tag

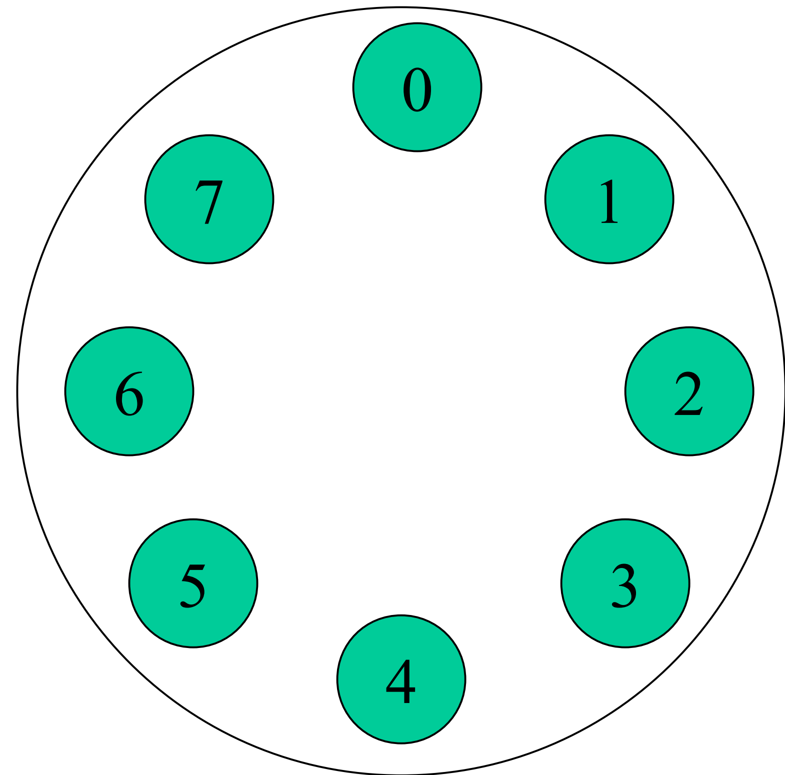
FPU regiszterek

Úgy képzeljük el, mint egy pisztoly tárat

Betöltő műveletnél óramutató járásával egyezően elfordul és a „12 óránál” levő regiszterbe betölti

Így az inicializálás után először a 7-es regiszterbe töltünk adatot

Ha a 7-es nem üres, a következő töltésnél a 6-osba töltünk.



FPU regiszterek

- 1. Szabály: Az FPU regiszternek üresnek kell lennie, hogy adatot tölthessünk be
- Adatot kivenni egy regiszterből, csak matematikai művelettel lehet
  - A „12 óránál” levő regiszterből veszi ki az elemet, majd
  - a „tárat” óramutató járásával egyező módon elfordítja
- A regiszterek száma soha nem változik

FPU regiszterek

- Feltölteni csak a verem tetejére lehet: ST(0)
- Ha egy újabb értéket töltünk fel
  - A régi érték már ST(1)-be kerül
  - Az új érték az ST(0)-ba kerül
- 2. Szabály: A programozónak kell számon tartania, hogy az értékek mely regiszterekben vannak

Control regiszterek

- PC:
  - 00: 24 bites szám (32 bit)
  - 01: nem használt
  - 10: 53 bites szám (64 bit)
  - 11: 64 bites szám (alap esetben) (80 bit)

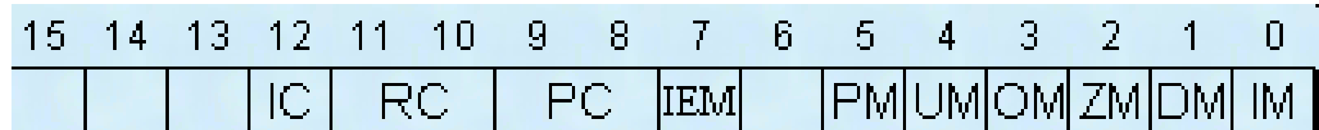


Fig.1.2 Control Word Fields

Control regiszterek

- 0. Bit: Érvénytelen művelet megszakítás
- 2. Bit: Osztás zérussal megszakítás
- 3. Bit: Túlcsordulás megszakítás
- 4. Bit: Alulcsordulás megszakítás

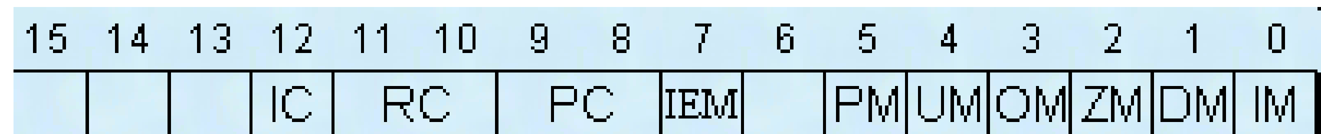


Fig.1.2 Control Word Fields

Státusz regiszter

- B: dolgozik-e az FPU
- TOP: melyik regiszter van felül (12 óránál)
- C3, C2-C0: összehasonlítás utáni állapot
- SF: verem hiba
  - Nem üres helyre akarunk tölteni
  - Üres helyről akarunk kivenni

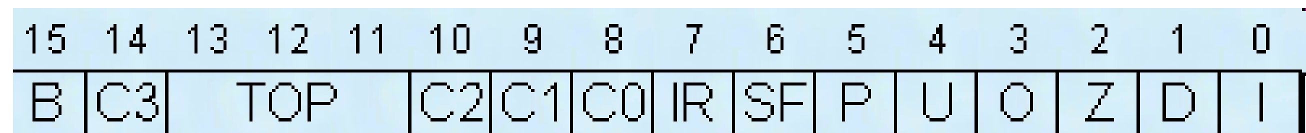


Fig.1.3 Status Word Fields

Tag regiszter

- A 80 bites regiszterek tartalmáról tárol információt

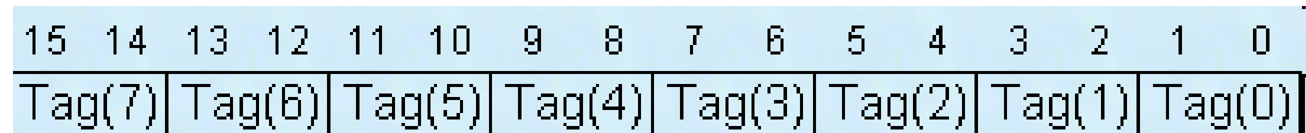


Fig.1.4 Tag Word Fields

Egész számok

- Minden egész szám előjeles
- Kettes komplemenst használja
- CPU regisztereket nem lehet közvetlenül használni

```
PUSH AX
FIXXX  word [ESP]        ; művelet egész számokkal
POP AX
```

Lebegőpontos számok

- Folytonos ← → Diszkrét matematika
- Számok bináris ábrázolása
  - **IEEE 754**
- **`float`**: 32 biten van ábrázolva
- Ez azt jelenti, hogy 2^{32} valós számot lehet pontosan reprezentálni
- Ezzel szemben végtelen sok valós szám van
- Ábrázolható tartomány:

  ±1.40129846432481707e-45

  ±3.40282346638528860e+38

Lebegőpontos számok

- 32 bit:
  - s : előjel bit (1 bit)
  - e : exponenciális kitevő (8 bit)
  - m : mantissza (23 bit)

$$(-1)^s \cdot m \cdot 2^{(e-127)}$$

seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

31 0

Lebegőpontos számok

- *m* mantissza: (0-22 bit)
  - $2*10^{-1} = 0.2*10^{0} = 0.02*10^{1}$ ugyanaz ezért
  - normalizálva van az érték, mint bináris tört
- Bináris törtek

  0.1101 = 1/2 + 1/4 + 1/16 = 13/16 = 0.825
- Nem minden szám reprezentálható:

  0.1 = 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + ...
  - vagyis az első 23 bitet használjuk csak, a maradékot „eldobjuk”

  0.000110011001100110011...

Lebegőpontos számok

- Mantissza a tizedes ponttól jobbra levő rész
- Automatikusan feltételezünk egy 1-est a tizedes pont előtt
  - `1.mmmmm...`
- De így hogyan reprezentálhatjuk a zérust:
  - Ha minden bit zérus
- De akkor hogyan reprezentáljuk 1.0 –et, hiszen a tizedes pont előtti 1-et automatikusan feltételezzük

Lebegőpontos számok

- Megoldás: az exponenciális biteket 127-el módosítjuk
- *e* kitevő: (30-23 bit)
  - 5 esetén: 127 + 5 = 132 binárisan 10000101
  - -5 esetén 127 – 5 = 122 binárisan 01111010

Lebegőpontos számok

Egy példa a lebegőpontos szám ábrázolásra:

```
0.085:

bits:    31   30-23              22-0
binary:   0 1111011 01011100001010001111011
decimal: 0   123             3019899
```

2^{e-127} (1 + m / 2^{23}) =

2^{-4}(1 + 3019899/8388608) =

11408507/134217728 =

0.085000000894069671630859375

Lebegőpontos számok

- Speciálisan reprezentált számok:
  - Minusz végtelen (-inf):
    - ha az összes exponenciális bit 1
    - előjel bit 1
  - Plusz végtelen (+inf):
    - ha az összes exponenciális bit 1
    - előjel bit 0
  - NaN : Not a Number
    - ha az összes exponenciális bit 1
    - valamelyik mantissza bit 1

Lebegőpontos számok

```
binary:    0 1111111 00000000000000000000000
decimális: 1

binary:    0 1111110 00000000000000000000000
decimális: 0.5
```

Pontosság és teljesség

- Két különböző fogalom
- Pontosság:
  – Az érték mennyire van közel a valódi értékhez
- Teljesség:
  – Mennyi információ van az adott értékről

Egész számok

- Pontosak
  - Ha van egy kettes számom és ahhoz egyet hozzáadok, akkor biztos hogy hármat fogok kapni
  - Bármilyen műveletet végzünk és az értelmezési tartományba esik a válasz, akkor mindig pontos értéket kapunk
- Ugyanakkor nem teljesek, abban az értelemben, hogy nem képesek például a tört részeket reprezentálni

Lebegőpontos számok

- Fordított helyzet
- Teljesek:
  - „Önkényesen” soha nem hagynak el információt a számról
  - “Elvileg” minden számot tudnak reprezentálni ha elég bit áll rendelkezésre
- De nem pontosak
  - Kerekítési hiba (Roundoff error)
  - Kioltó hiba (Cancelation error)
  - …

Kerekítési hiba

```
#include <stdio.h>
int main()
{
  double x1 = 0.3;
  double x2 = 0.1 + 0.1 + 0.1;
  double x3 = 0.5;
  double x4 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

  printf("%.20f\n", x2);
  if(x1 == x2) printf("egyenlo\n");
  else printf("nem egyenlo\n");
  printf("%.20f\n", x4);
  if(x3 == x4) printf("egyenlo\n");
  else printf("nem egyenlo\n");
  return(0);
}
```

Kerekítési hiba

A futtatás eredménye:

```
$ num3.exe
0.30000000000000004441
nem egyenlo
0.50000000000000000000
egyenlo
```

Lebegőpontos számok

- Kernighan és Plauger:
  - „A lebegőpontos számok olyanok mint egy kupac homok. Amikor elmozdítunk egy kicsit, el is veszítünk egy kicsit és csak piszok marad a kezünkben.”

Kioltó hiba

```
#include <stdio.h>

int main()
{
  double x1 = 10.000000000000004;
  double x2 = 10.000000000000000;
  double y1 = 10.00000000000004;
  double y2 = 10.00000000000000;
  double z = (y1 - y2) / (x1 - x2);
  printf("%f\n", z);
  return(0);
}
```

Kioltó hiba

- A várt eredmény:

`0.000000000000004 / 0.00000000000004 = 10.0`

- A kapott eredmény

`11.5`

Stabilitás

- Egy matematikai probléma jól kondicionált ha a bemeneti paraméterek kis változására az eredmény is mértékben változik.
- Egy algoritmus numerikusan stabil ha bemeneti paraméterek kis változására az eredmény is kis mértékben változik.
- A művészet és tudomány az, hogy numerikusan stabil algoritmusokat találjunk jól kondicionált problémák megoldására.

Stabilitás

- A pontosság függ a probléma kondicionáltságától és az algoritmus stabilitásától.
- Pontatlanságot okozhat, ha:
  - Stabil algoritmust alkalmazunk rosszul kondicionált problémára; vagy
  - Instabil algoritmust alkalmazunk jól kondicionált problémára

Instabilitás

- Probléma: az `f(x) = exp(x)` függvény
  - Jól kondicionált probléma
- Algoritmus: a Taylor sorozat első négy elemét használjuk
  - `g(x) = 1 + x +` x^2 `/ 2 +` x^3 `/ 3`

  - `f(1) = 2.718282`
  - `g(1) = 2.666667`

Instabilitás

- Ha `x < 0` akkor az algoritmus instabil!!!
- De ha az e^{-x} függvény Taylor sorát vesszük az már stabil lesz.

Rosszul kondicionáltság

$$x_n = (R+1)\,x_{n-1} - R\,(x_{n-1})^2$$

- Vegyük a fenti egyenletet
- Kezdő érték: $x_0 = 0.5$
- R=3
- 100 iterációt futtatunk
- A műveleteket többféleképpen csoportosítjuk

Rosszul kondicionáltság

n	(R+1)x-R(xx)	(R+1)x-(Rx)x	((R+1)-(Rx))x	x + R(x-xx)	pontos
50	0.0675670955	0.0637469566	0.0599878799	0.0615028942	0.0622361944
100	0.0000671271	0.1194574394	1.2564956763	1.0428230334	0.7428865400

- Négy különböző értéket kaptunk!!!
- Az összeadás, szorzás, kivonás stabil, de
- A probléma rosszul kondicionált.
- Ha R > 2.57 az egyenlet kaotikus!

Hibák az életben

Ariane 5

- European Space Agency
- 10 év, 7 billió dollárba került a fejlesztés
- 1996 június 4-én felrobbant
- A rakéta és terhének összértéke: 500 millió dollár
- Ok: Szoftver, numerikus hiba

Ariane 5

- Az irányító rendszer 36.7 másodperccel a fellövés után a rakéta oldal irányú sebességét reprezentáló számot próbálta konvertálni, egy 64 bites számot 16 bites formátumra
- A rendszert leállítja magát, mert érvénytelen adatot kap.
- A másodleges rendszer is leáll, hiszen ugyanaz a szoftver.
- Az irányító rendszer így hibás utasítást „ad”, mintha egy nem létező fordulatot kellene kompenzálni.
- A rakéta hirtelen irányt váltott (bár nem volt rá szükség)
- Olyan erők ébredtek melyre az önmegsemmisítés bekapcsolt 39 másodperccel a fellövés után

Patriot rakéta

- 1991 február 25, Öböl háború
- Patriot nem tudta eltalálni az iraki Scud rakétát
- 28 katona halt meg és 100 sérült meg
- Ok: Szoftver, numerikus hiba

Patriot rakéta

- Az egy tized másodpercekben mért időt a rendszer 1/10 –el szorozta meg hogy másodpercekben kapja meg az időt
- Az adatot 24 biten reprezentálta
- 1/10 –et nem lehet pontosan reprezentálni binárisan, így a 24. bit utáni rész levágódik. Ez egy kerekítési hiba.
- Sokszor elvégezve a szorzást a hiba növekszik:
  - 100 órás üzem esetén az eltérés: 0.34 másodperc

Patriot rakéta

- Scud sebessége: 1.676 m/s
- Így több mint fél kilómétert tesz meg a Scud 0.34 másodperc alatt

Vissza az FPU-hoz

FPU

- Minden műveletnek az egyik argumentuma a stack-en kell lennie
- Alap esetben:

  ST = ST(1) művelet ST
- FADD = FADD ST(1), ST
- Bármelyik regiszter használható
  - FADD ST,ST(n)
  - FADD ST(n),ST
- Az egész számokkal végzendő művelet esetén az F betű után egy I betű van
  - FIADD egész

Utasítások

- FWAIT
  - A CPU addig vár amíg az FPU dolgozik
  - Például, amikor az FPU egy értéket tárol, amit a CPU-nak kell használnia

```
FISTP memória
FWAIT
MOV EAX, memória
```

Utasítások

- FINIT, FNINIT
  - Az FPU-t alapállapotba helyezi,
  - minden regisztert töröl
  - FINIT előtt kiad a rendszer egy FWAIT utasítást

Utasítások

- FSTSW cél
- FNSTSW cél
  - A Status regisztert a cél helyen tárolja

```
stword DB ?

FSTSW stword       ; memóriába másolunk
FWAIT              ; várunk míg befejezte
TEST stword, 5     ; zérus osztás volt?
JNZ error
```

Utasítások

- FSTCW cél
- FNSTCW cél
  - A Control regisztert a cél helyen tárolja
- FLDCW forrás
  - A Control regisztert feltölti a forrás helyről
  - Az FPU beállításainak módosítása

Utasítások

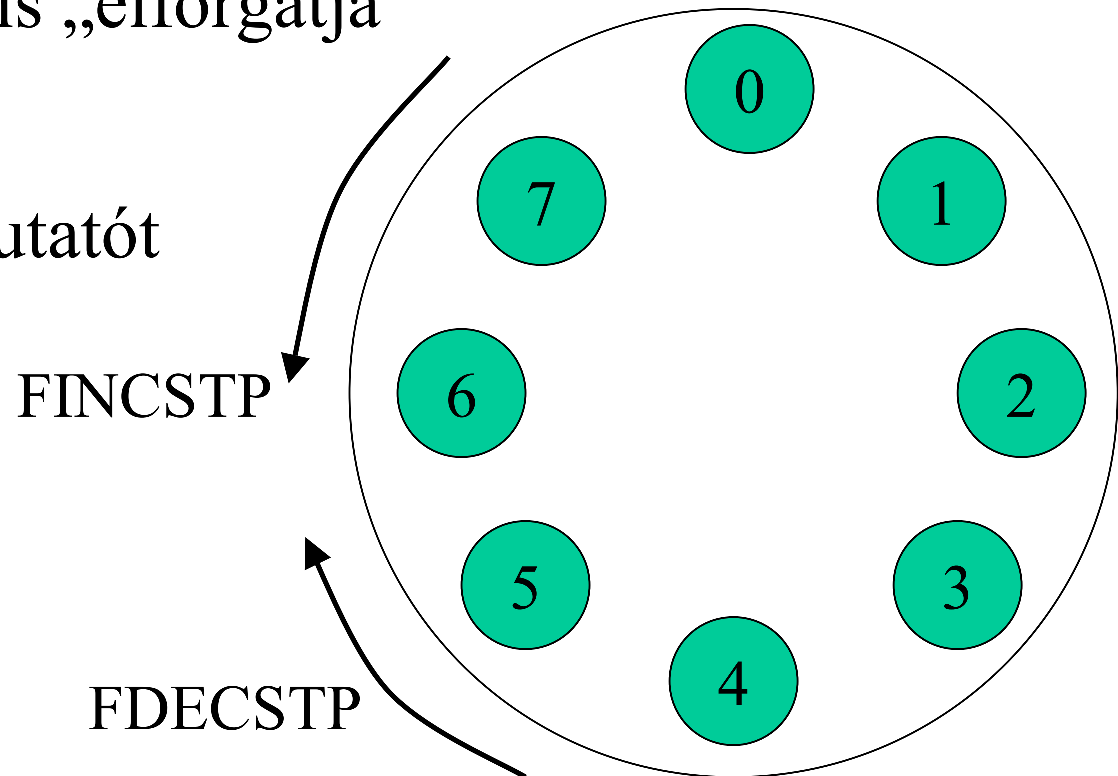
- FCLEX, FNCLEX
  - Törli a kivétel biteket
- FSAVE, FNSAVE cél
  - Az FPU teljes állapotát elmenti a cél helyen
- FRSTOR
  - Az FPU állapotát visszaállítja
- FFREE st(n)
  - Az adott regisztert szabadnak jelöli

+0	WORD	**Control Word**
+2	WORD	(unused)
+4	WORD	**Status Word**
+6	WORD	(unused)
+8	WORD	**Tag Word**
+10	WORD	(unused)
+12	DWORD	**Instruction Pointer**
+16	WORD	**Code Segment**
+18	WORD	(unused)
+20	DWORD	**Operand Address**
+24	WORD	**Data Segment**
+26	WORD	(unused)
+28	TBYTE	**ST(0)**
+38	TBYTE	**ST(1)**
+48	TBYTE	**ST(2)**
+58	TBYTE	**ST(3)**
+68	TBYTE	**ST(4)**
+78	TBYTE	**ST(5)**
+88	TBYTE	**ST(6)**
+98	TBYTE	**ST(7)**

Fig.3.1 FPU Save State Format

Utasítások

- FDECSTP
  - Csökkenti a verem mutatót
  - ST(7) regiszterből ST(0) regiszter lesz lényegében
  - A többi regisztert is „elforgatja”
- FINCSTP
  - Növeli a verem mutatót



Utasítások, valós számok

- Adatmozgatás
- FLD src
  - Jobbra forgatja a regisztereket (csökkenti) és
  - az src-ból az ST(0)-ba másolja az értéket
- FLD ST(3)
  - ST(3) értékét az új ST(0)-ba másolja
  - ST(4) és ST(0) értéke ugyanaz lesz
- FLD ST
  - ST(1) és ST(0) értéke ugyanaz lesz

Utasítások, valós számok

- FLD szám
  - A szám kerül az ST(0) regiszterbe
- FLD qword [mem]
  - A memóriából másolja az adatot az ST(0)-ba
  - Ha kell konvertálja 80-es számmá

Utasítások, valós számok

- FST dest
  - Tárolja az ST(0) értékét a cél területen
- FSTP dest
  - Tárolja az ST(0) értékét a cél területen és az ST(0)-át leemeli a veremről (felszabadítja)
- FXCH
  - Az ST(0) és ST(1) értékének felcserélése vagy
  - Egy másik regiszter és ST(0) felcserélése
    - FXCH ST(3)

Utasítások, valós számok

- FLDZ
  - Jobbra forgatja a regisztereket (csökkenti) és
  - az ST(0) regiszterbe zérus kerül
- FLD1
  - Jobbra forgatja a regisztereket (csökkenti) és
  - az ST(0) regiszterbe 1 kerül
- FLDPI
  - Jobbra forgatja a regisztereket (csökkenti) és
  - az ST(0) regiszterbe π értéke kerül

Utasítások, egész számok

- FILD
- FIST
- FISTP

Utasítások, valós számok

FABS

Abszolút érték számítása

FADD

Két valós szám összeadása

FCHS

Előjel váltás

FDIV

Két valós szám osztása

FMUL

Két valós szám szorzata

FSUB

Két valós szám különbsége

FSQRT

Négyzetgyök

Utasítások, egész számok

FIADD

Két egész szám összeadása

FIDIV

Két egész szám osztása

FIMUL

Két egész szám szorzata

FSUB

Két egész szám különbsége

Utasítások

Argumentumok radiánban

FCOS

Koszinusz

FSIN

Szinusz

FPTAN

Tangens

FCOM

Valósz számok összehasonlítása

FICOM

Egész számok összehasonlítása

FTST

Verem tetejét zérussal hasonlítjuk össze

FNOP

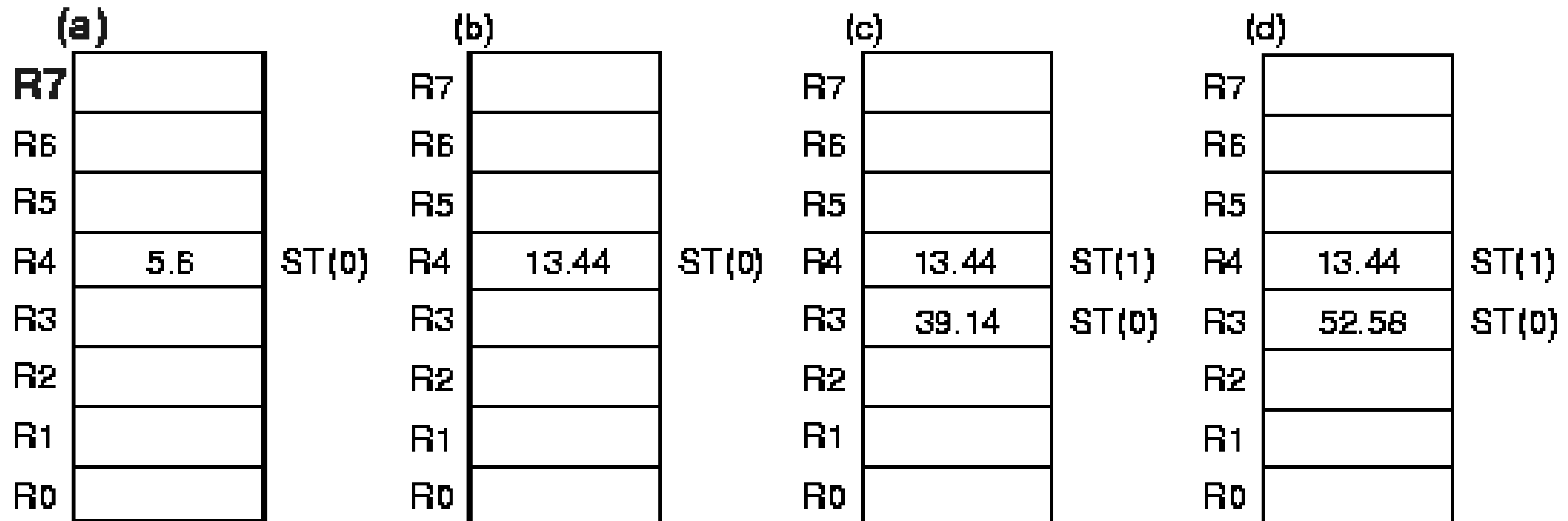
mint NOP, nem csinál semmit

Egyszerű példa

Computation
Dot Product = (5.6 x 2.4) + (3.8 x 10.3)

Code:

```
FLD  value1 ;(a) value1=5.6
FMUL value2 ;(b) value2=2.4
FLD  value3 ;  value3=3.8
FMUL value4 ;(c)value4=10.3
FADD ST(1)  ;(d)
```



Kör területe

```
rad  dd   4.0
area dd   3.0

..start:
  mov    ax, cs
  mov    ds, ax
  finit
  fldpi
  fld    dword [rad]
  fmul   st0
  fmul   st1
  fstp   dword [area]
  ...
```

$z=sqrt(x^2+y^2)$

```
X       dd      4.0
Y       dd      3.0
Z       dd      0

..start:
   mov     ax, cs
   mov     ds, ax
   finit
   fld     dword [X]
   fld     st0
   fmulp   st1, st0
   fld     dword [Y]
   fld     st0
   fmulp   st1, st0
   faddp   st1, st0
   fsqrt
   fstp    dword [Z]

   ; eredmény Z-ben
```

C kód

```
sinhp = 0.14;
i = 0x2ffffff;

while(i-- > 0)
{
  sinhp = sin(sinhp) + cos(sinhp);
}
result = sinhp;
```

Assembly kód

```
double _014 = 0.14;
double result = -1;
__asm
{
    mov ecx, 0x2ffffff        ; 50331647
    fld [_014]
  redo:
    fsincos
    faddp st(1), st(0)
    dec ecx
    jnz redo
    fst result
}
```

C és asembly kód

Running ASM code...

Time Elapsed: 4086 ms, result: 1.258728

Running C code...

Time Elapsed: 8122 ms, result: 1.258728