Assembly NASM IA-32
FPU (Floating-Point Unit) e Instruções Vectoriais

Introdução

- A norma IEEE-754 (revisão)
- Erros de representação e a sua propagação
- A Unidade de Vírgula Flutuante (FPU)
- A FPU Intel
- Instruções vectoriais

A Norma IEEE-754

- Standard de 1985 adoptado quase universalmente

- Precisão simples - 32 bits
  - ✓ Sinal 1 bit
  - ✓ Expoente 8 bits (deslocamento 127)
  - ✓ Mantissa 23 bits (com bit escondido)

- Precisão dupla - 64 bits
  - ✓ Sinal 1 bit
  - ✓ Expoente 11 bits (deslocamento 1023)
  - ✓ Mantissa 52 bits (com bit escondido)

Erros de representação de reais

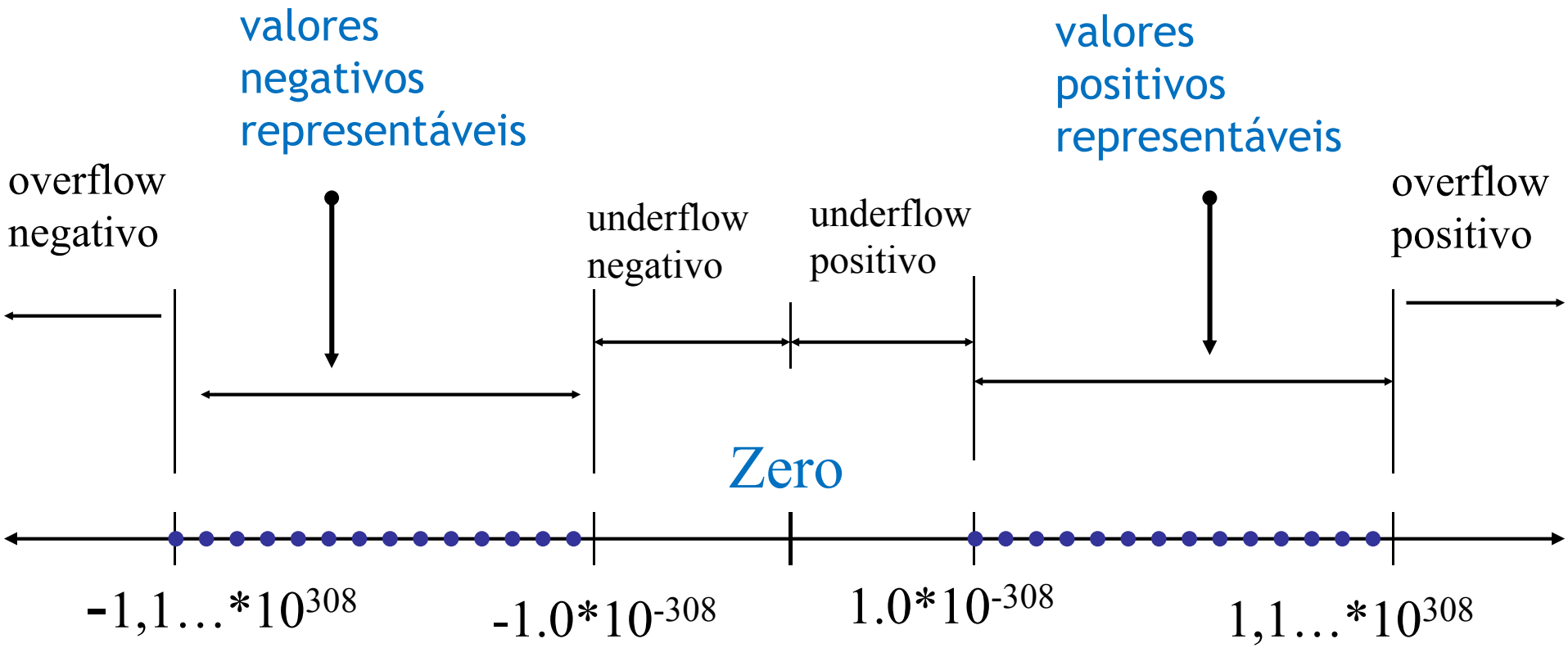
- O conjunto dos reais é "contínuo"
  - ✓ Entre quaisquer dois números há um número infinito de reais

- Com um número limitado de bits de representação só podemos representar um número finito de valores

- A representação é portanto uma aproximação
  - ✓ Quanto mais bits tivermos melhor a aproximação
  - ✓ Não se consegue representar valores demasiado grandes ou valores absolutos demasiado pequenos

Norma IEEE-754

Valores Representáveis – Precisão Dupla



Erros de Representação de Reais

- Nas regiões representáveis:
  - ✓ Procurar o número de vírgula flutuante mais aproximado do número real que se quer
  - ✓ Este arredondamento introduz um erro
  - ✓ O erro relativo (%):

$$\frac{\textit{diferença entre números (erro)}}{\textit{número pretendido}}$$

- Se aumentar o número de dígitos:
  - ✓ Da mantissa, aumenta a densidade de pontos nas regiões representáveis: *maior precisão*
  - ✓ Do expoente, aumenta o tamanho das regiões representáveis: *maior gama de valores*

Erros de Representação de Reais

- A título de exemplo consideremos uma hipotética representação a 14 bits
  - ✓ Sinal 1 bit
  - ✓ Expoente 5 bits (deslocamento 15)
  - ✓ Mantissa 8 bits

- O formato representa os números:
  - ✓ $-1,11111111_2 \times 2^{15}$ a $-1,00000000 \times 2^{-15}$
  - ✓ o zero (0)
  - ✓ $1,00000000 \times 2^{-15}$ a $1,11111111_2 \times 2^{15}$

Erros de Representação de Reais

- Não consegue representar, por exemplo, 2^{-19} ou 2^{128}

- Também não consegue representar 128,25
  - ✓ $128,25 = 1000\ 0000,01_{(2)} = 1,000000001_{(2)} \times 2^7$
  - ✓ Os 9 bits não cabem nos 8 bits da mantissa
  - ✓ Podemos aproximar com o número 128
    - ✓ $128,0 = 1,00000000_{(2)} \times 2^7$
  - ✓ O erro relativo é:

$$(128,25 - 128,0)/128,25 \approx 0,2\ \%$$

Propagação de Erros

- Os erros podem acumular-se
  - ✓ Os algoritmos devem tentar minimizá-los

- 1º exemplo: 128,25*128,25 = 16448,0625
  - ✓ Por aproximação de 128,25 a 128: 128*128 = 16384
  - ✓ Entra erro de 0,25
  - ✓ Sai erro de 64,0625 → ~0,4%

- 2º exemplo: (512+1)/8 = 64,125 = (512/8)+(1/8)
  - ✓ 513 = 10000000012 não é representável, logo não se deve fazer a soma primeiro
  - ✓ 512/8 e 1/8 são representáveis, por isso devia-se reescrever a expressão nessa forma

Operações em vírgula flutuante

$x = X * 2^{ex}$

$y = Y * 2^{ey}$

- Adição e subtracção:
  - ✓ se ex == ey:

$$x+y \rightarrow (X+Y)*2^{ex}$$

$$x-y \rightarrow (X-Y)*2^{ex}$$

  - ✓ se ex<>ey:

    há que tornar primeiro os expoentes iguais

Operações em vírgula flutuante

- Multiplicação e divisão:

  $x*y \rightarrow X*2^{ex}*Y*2^{ey} = (X*Y)*2^{ex+ey}$

  $x/y \rightarrow (X*2^{ex})/(Y*2^{ey}) = (X/Y)*2^{ex-ey}$

  (o resultado tem de ter sempre normalizado)

- Comparações:
  - ✓ Verificar sinais
    - ✓ Por exemplo: se x positivo e y negativo x>y
  - ✓ se ex > ey, então x > y
  - ✓ se ex == ey, então a comparação depende das mantissas X e Y

A Unidade de Vírgula Flutuante (FPU)

- Floating Point Unit (Unidade de Vírgula Flutuante)
- Suporta as instruções sobre as representações de vírgula flutuante (FP)
  - ✓ Soma, subtracção, multiplicação, divisão, comparação
- Pode usar os registos gerais do CPU ou dispor dos seus próprios registos
  - ✓ Os registos gerais podem não comportar as representações de FP
- A norma exige instruções de conversão entre todas as representações de FP e as representações de inteiros
  - ✓ Ao converter vírgula flutuante → inteiro, o resultado pode sair do domínio válido dos inteiros

FPU dos Processadores Intel

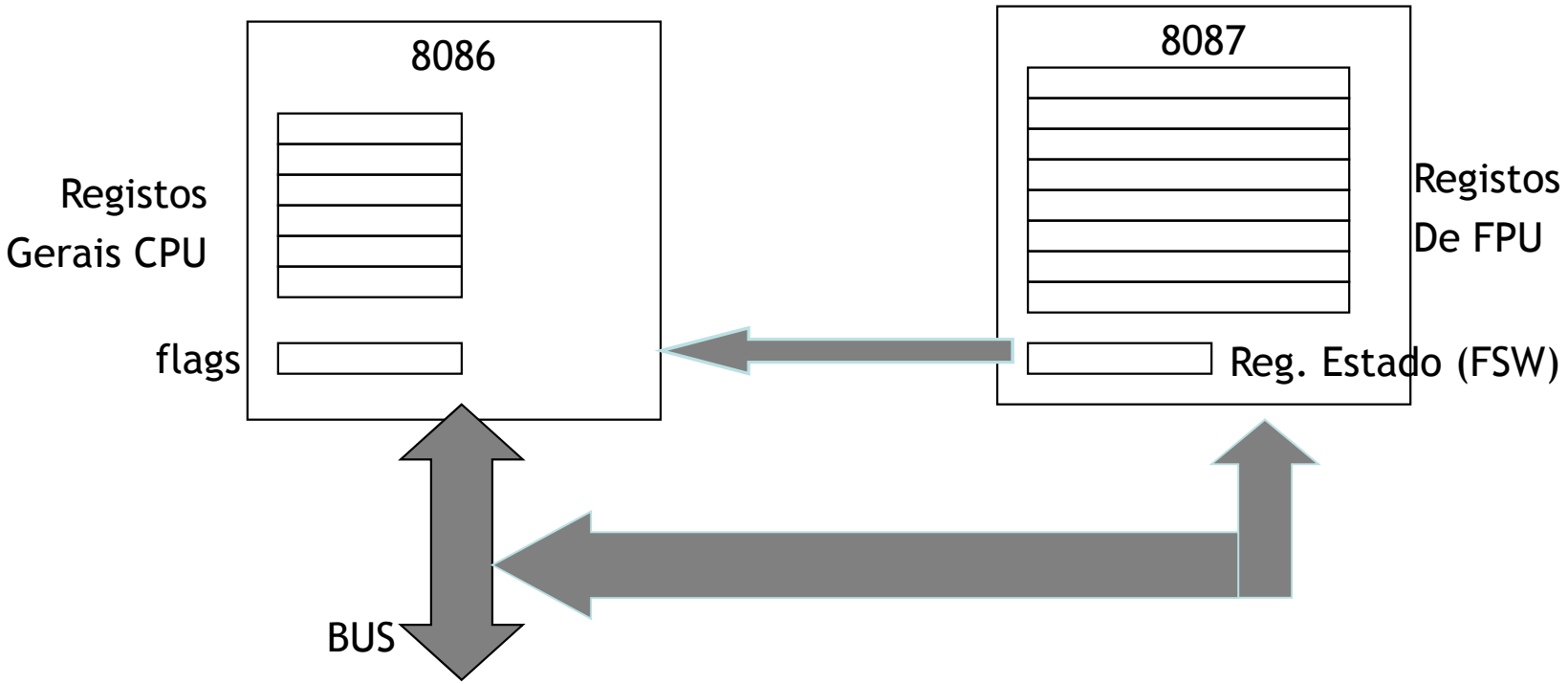
- Nos processadores Intel:
  - ✓ Unidade FPU, originalmente era um coprocessador (8087, 80287, 80387)
  - ✓ A partir do 80486 foi integrada no próprio CPU

- Reconhece os seguintes tipos de dados:
  - ✓ Inteiros com sinal (complemento para 2): 16, 32 e 64 bits
  - ✓ Vírgula flutuante (IEEE-754): 32, 64 e 80 bits

- Usa 8 registos internos de 80 bits organizados em pilha:
  - ✓ Trabalha como uma máquina de pilha

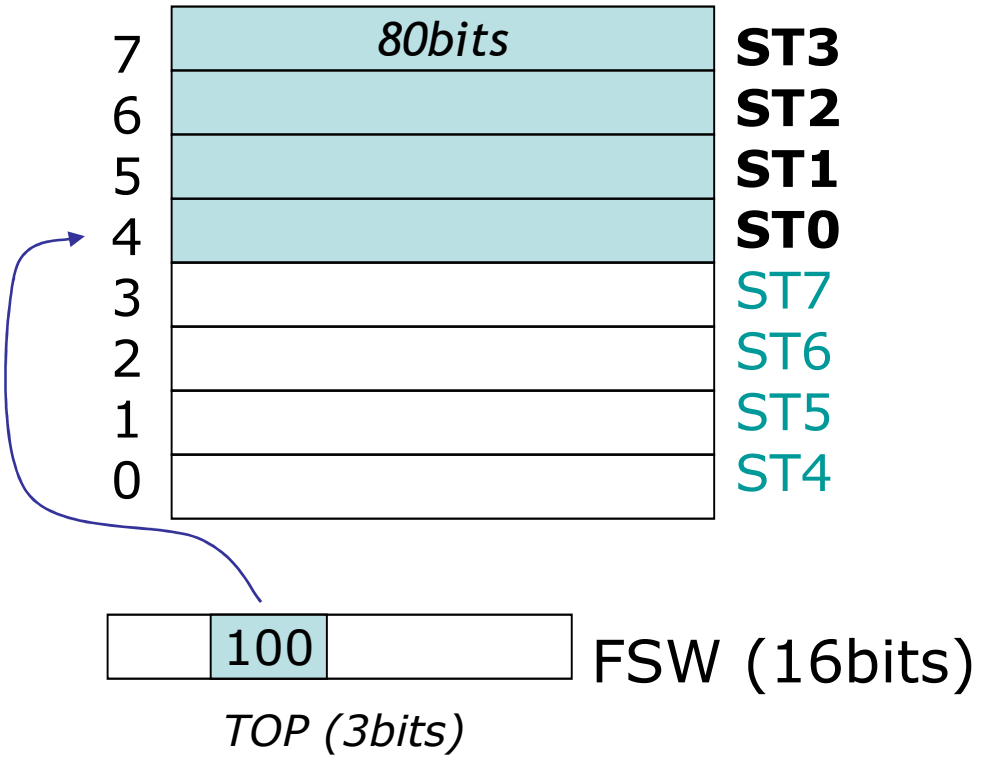
80x86 + 80x87

- A unidade FPU (ou 80x87) tem uma organização e registados diferentes do CPU 80x86 mas tem acesso à mesma memória central.
  - ✓ Não é possível a transferência directa de valores entre registos CPU e registos FPU



Pilha de Registos na FPU Intel

- Suporta operações de *Load* e *Store*:
  - ✓ *Load* = *push* para pilha de registos
  - ✓ *Store* = *pop* da pilha de registos
- O registo envolvido está implícito
  - ✓ ST0 é sempre o registo no topo da pilha
  - ✓ Depois vem ST1, ST2, …



Instruções de Transferência

- *Load* e *store*, com conversão de tipos:
  - ✓ Transferência com a memória
    - ✓ Há sempre conversão automática dos tipos de dados para/de a representação interna de 80 bits da FPU:
  - ✓ Reais IEEE:

    FLD *end* → Push para a FPU do número em vírgula flutuante em *end* (Load Floating-point)

    FSTP *end* → Pop da FPU como um número em vírgula flutuante para *end* (Store and Pop Floating-point)

    end pode referenciar valores em dword, qword ou tword
  - ✓ Inteiros c/sinal:

    FILD *end* → Push para a FPU do número inteiro em *end* (Load Integer)

    FISTP *end* → Pop do topo da pilha FPU como um número inteiro para *end* (Store and Pop Integer)

    end pode referenciar valores em word, dword ou qword

Instruções de Transferência

- Transferir para memória, sem desempilhar o valor no topo da pilha

  FST *dest* → Store Floating-point

  FIST *dest* → Store Integer

- *Load* de constantes "especiais" pré-definidas:

  FLD1 → Load constante 1

  FLDZ → Load constante 0

  FLDPI → Load constante π

  FLDL2E → Load logaritmo de e na base 2

  ...

Operações e Operandos

- Operações sobre a pilha dos 8 registadores: STn com n=0..7
  - ✓ ST0 (ou ST) é o topo da pilha

- As operações aritméticas operam normalmente sobre o topo da pilha:
  - ✓ Desempilham os operandos e depois empilham o resultado
  - ✓ ou operam sobre o topo da pilha e outra posição da pilha
  - ✓ ou operam sobre o topo da pilha e uma posição de memória

Operações Aritméticas

- Versões sobre vírgula flutuante

```
FADD end        → ST0 = ST0 + Mem[end]
FADD STn        → ST0 = ST0 + STn    com 0 ≤ n ≤ 7
FADD STn,ST0    → STn = ST0 + STn    com 0 ≤ n ≤ 7
FSUB end        → ST0 = ST0 - Mem[end]
FSUBR end       → ST0 = Mem[end] – ST0
FSUB STn        → ST0 = ST0 - STn    com 0 ≤ n ≤ 7
FSUBR STn       → ST0 = STn – ST0    com 0 ≤ n ≤ 7
```

… (**FSUB** STn,ST0 e **FSUBR** STn,ST0)

```
FMUL end        → ST0 = ST0 * Mem[end]
```

… (como no **FADD**)

```
FDIV end        → ST0 = ST0 / Mem[end]
```

… (como no **FSUB**)

Operações Aritméticas

- Versões sobre inteiros

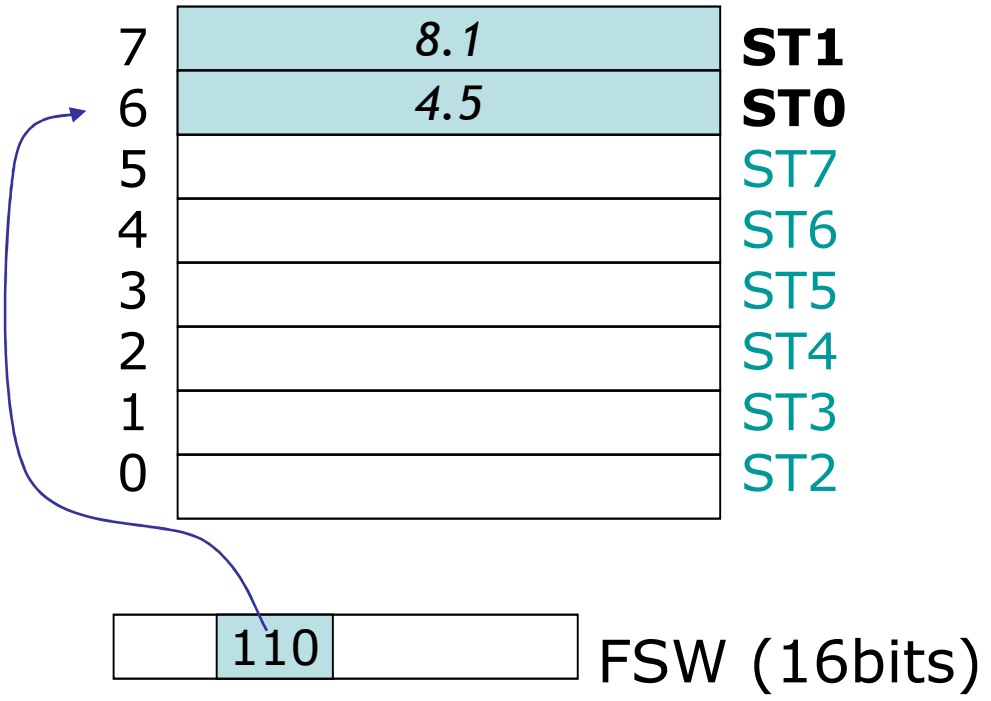
```
FIADD end      → ST0 = ST0 + (float) Mem[end]
FISUB end      → ST0 = ST0 - (float) Mem[end]
FISUBR end     → ST0 = (float) Mem[end] – ST0

FIMUL end      → ST0 = ST0 * (float) Mem[end]
FIDIV end      → ST0 = ST0 / (float) Mem[end]
FIDIVR end     → ST0 = (float) Mem[end] / ST0
```

Operações Aritméticas

- Se usarmos **FADD** ST1 não desempilhamos o operando em ST1

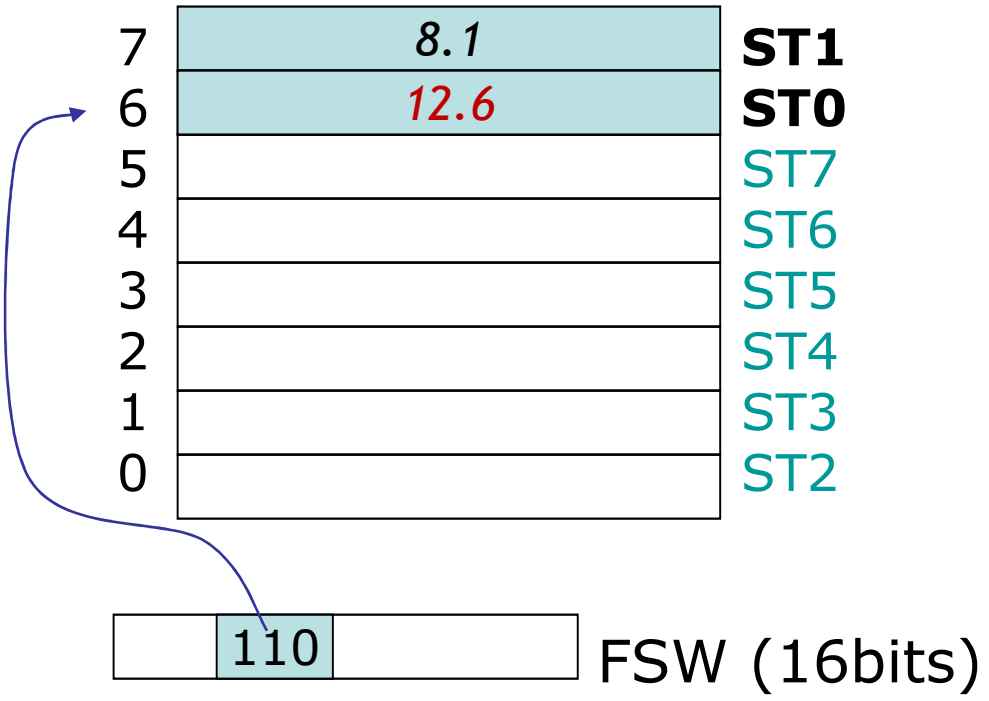
1. ST0 = ST0 + ST1



Operações Aritméticas

- Se usarmos **FADD** ST1 não desempilhamos o operando em ST0

```
1. ST0 = ST0 + ST1
```



Operações Aritméticas

- A FPU do IA-32 não tem versões das operações aritméticas binárias sem operandos
  - ✓ Por exemplo **FADD** (sem operandos)

- As instruções utilizadas para realizar a operações e desempilhar os argumentos têm o sufixo **P**

```
FADDP dest    → dest = ST0+dest    e pop ST0
FSUBP dest    → dest = ST0-dest    e pop ST0
FMULP dest    → dest = ST0*dest    e pop ST0
FDIVP dest    → dest = ST0/dest    e pop ST0
```

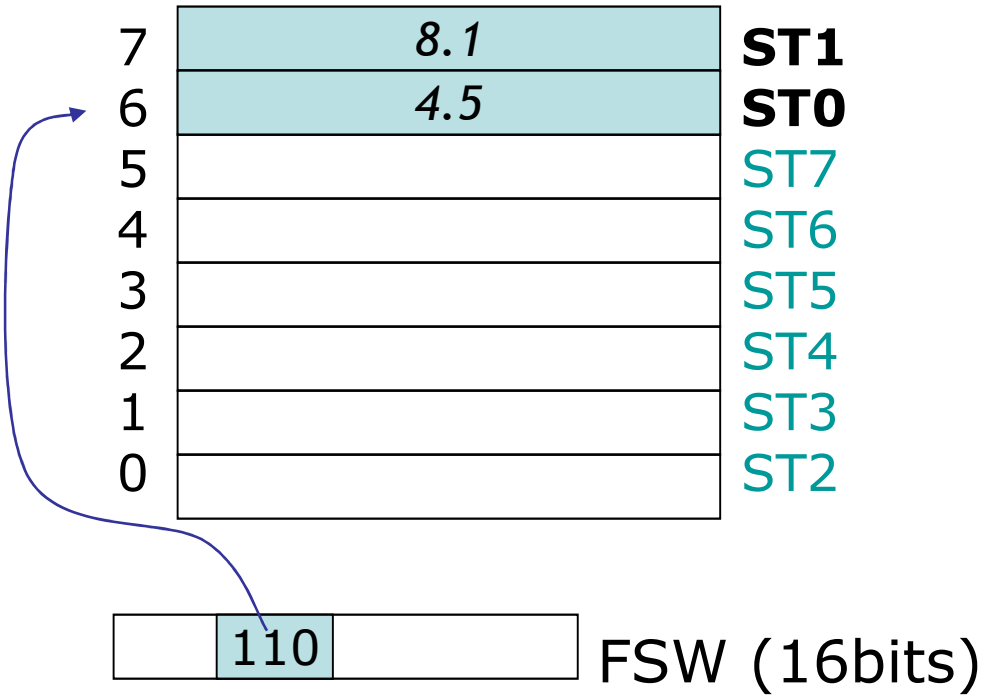
  - ✓ `dest` pode ser um endereço de memória ou um registo

Operações Aritméticas

- Portanto para desempilhar os valores no topo da pilha, somá-los usamos

```
FADDP ST1
```

1. `ST1 = ST0 + ST1`
2. `POP ST0`

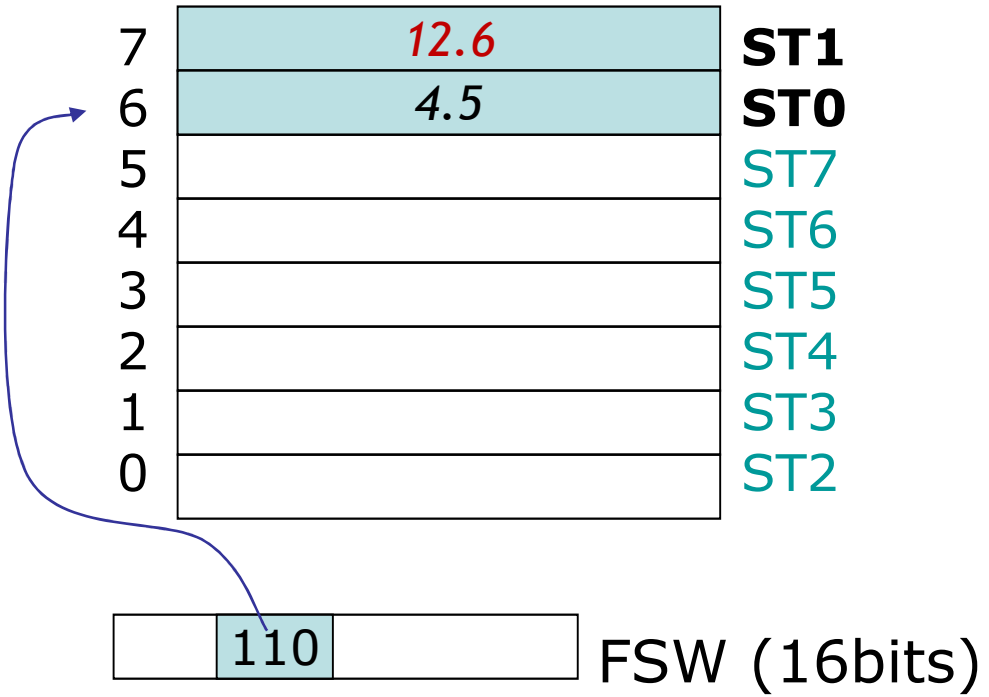


Operações Aritméticas

- Portanto para desempilhar os valores no topo da pilha, somá-los usamos

```
FADDP ST1
```

1. `ST1 = ST0 + ST1`
2. `POP ST0`

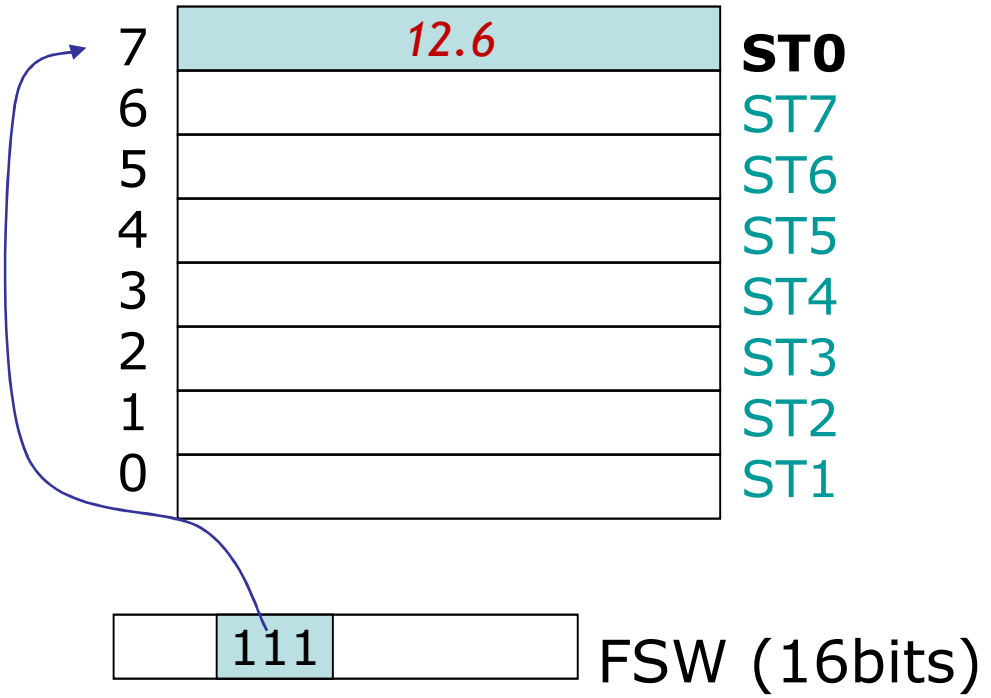


Operações Aritméticas

- Portanto para desempilhar os valores no topo da pilha, somá-los usamos

```
FADDP ST1
```

1. `ST1 = ST0 + ST1`
2. `POP ST0`



Outras Operações Aritméticas

- A FPU suporta muitas outras operações:
  - ✓ Senos, co-senos, tangentes e arcos de tangente
  - ✓ Raiz quadrada
  - ✓ Troca de sinal, valor absoluto
  - ✓ *Load* de constantes: 1,0; 0,0; π; $\log_2 e$; ln2; etc…
  - ✓ E muitas mais variantes das operações já vistas

Exemplo de Utilização da FPU

- Exemplo de avaliação de uma expressão:

  (3 + 5.1) * (4.5 + 1)

  - ✓ Usando notação pós-fixa:

    3 5.1 + 4,5 1 + *

  - ✓ Avaliando usando FPU Intel:

```
section .data
var1   dd     5.1
var2   dd     4.5
tmp    dd     0

section .bss
resultado resq 1
```

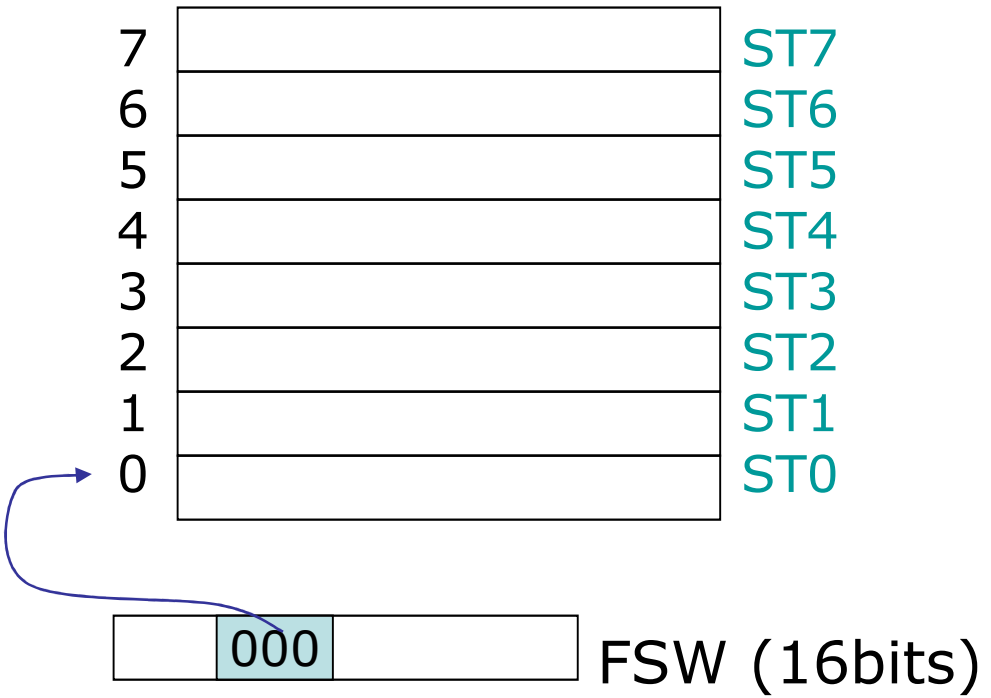
```
section .text
_start:
    mov dword [tmp], 3
    fild dword [tmp]
    fadd dword [var1]
    fld dword [var2]
    fld1
    faddp st1
    fmulp st1
    fstp qword [resultado]
    …
```

Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

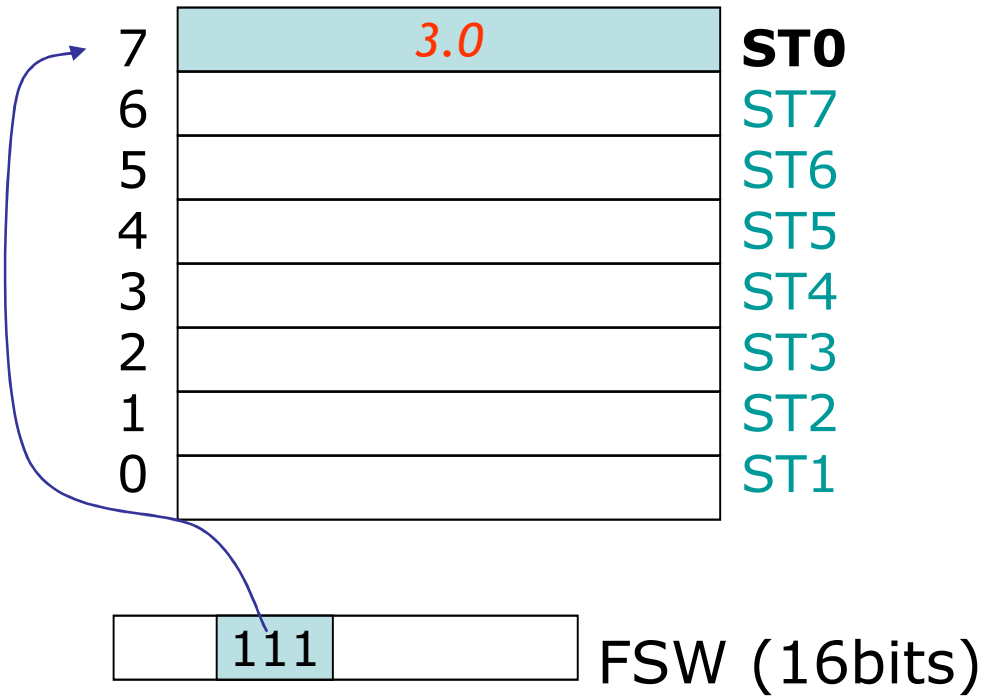


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

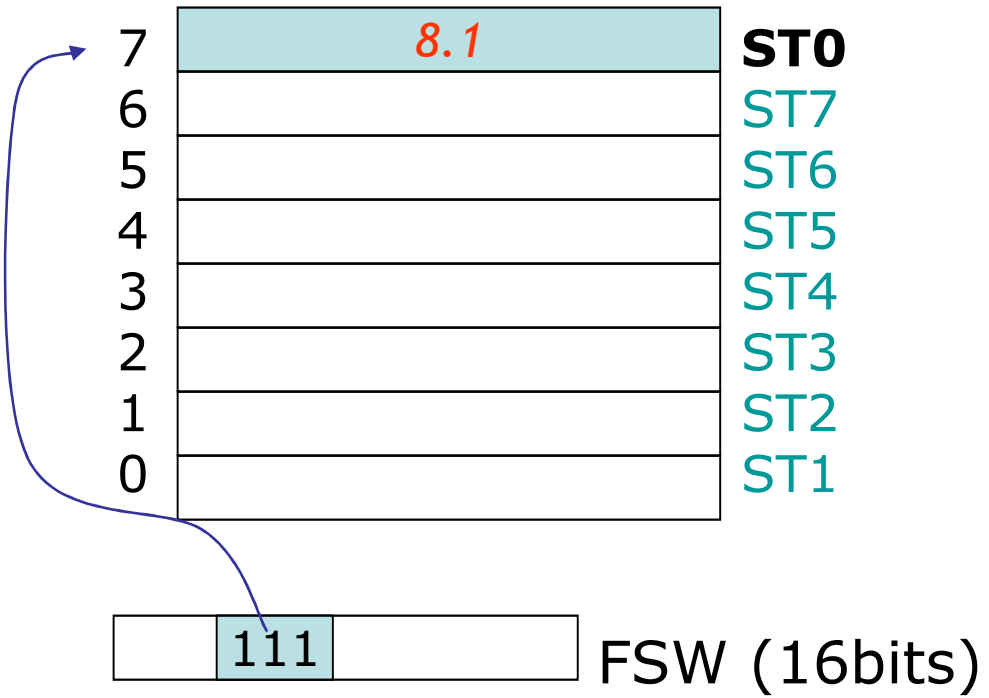


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

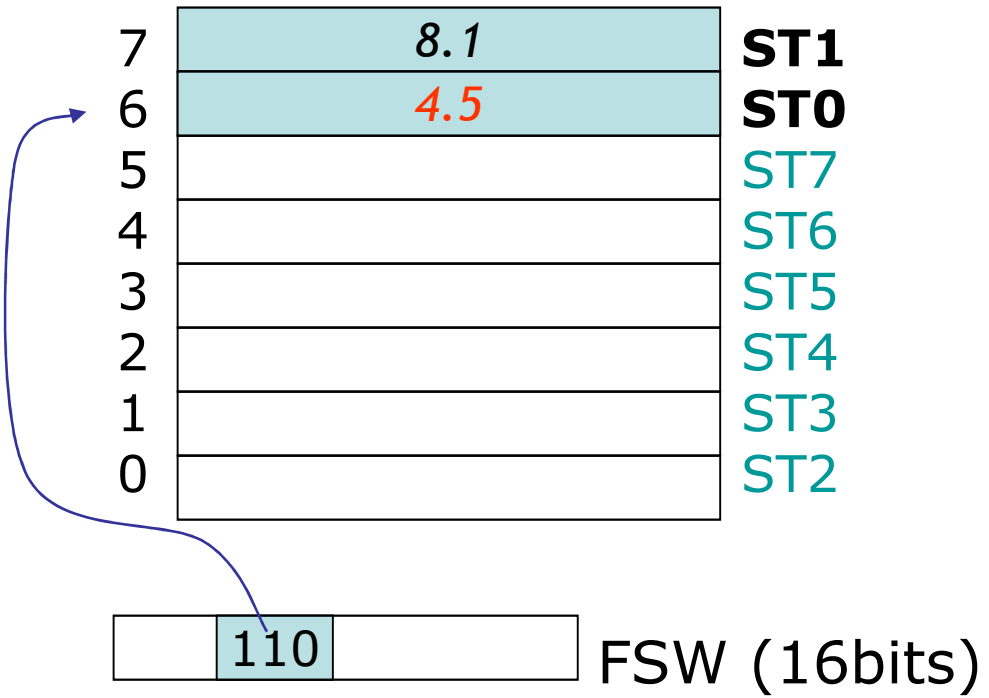


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

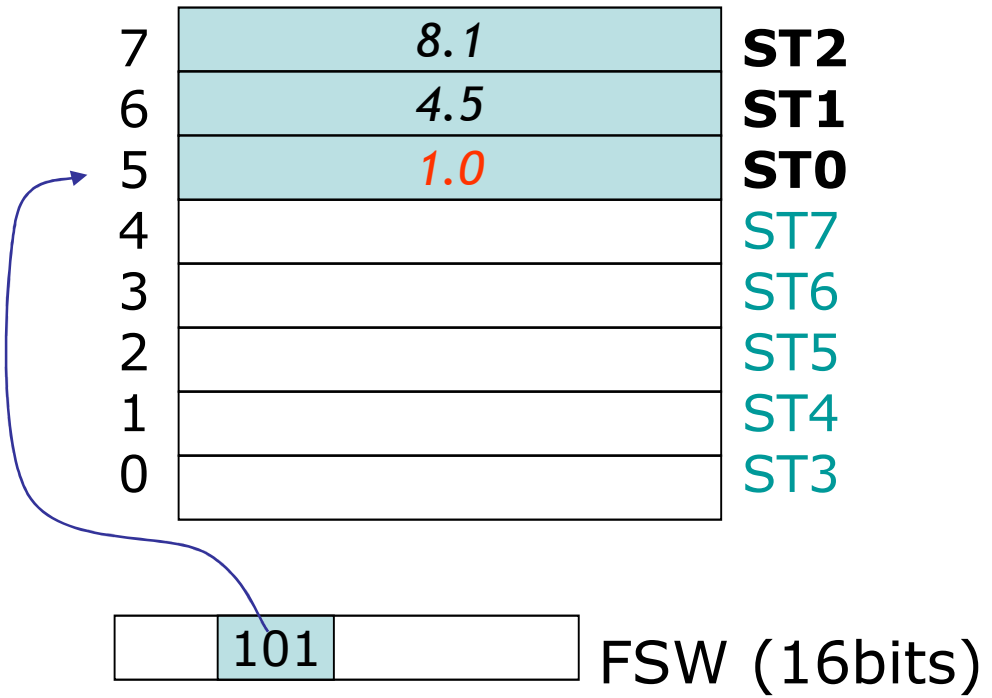


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

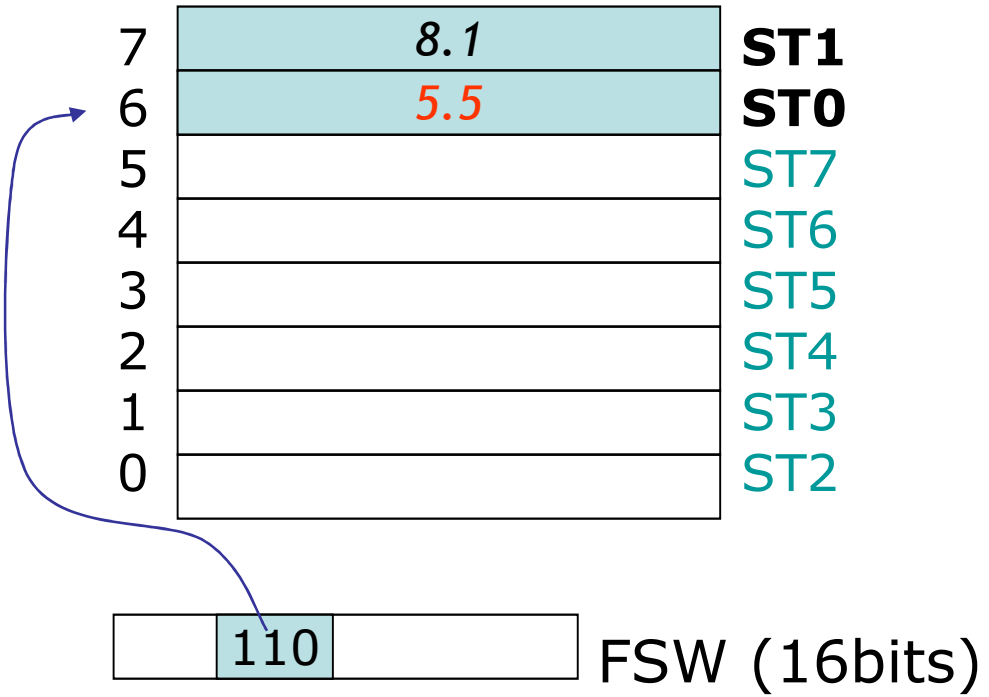


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

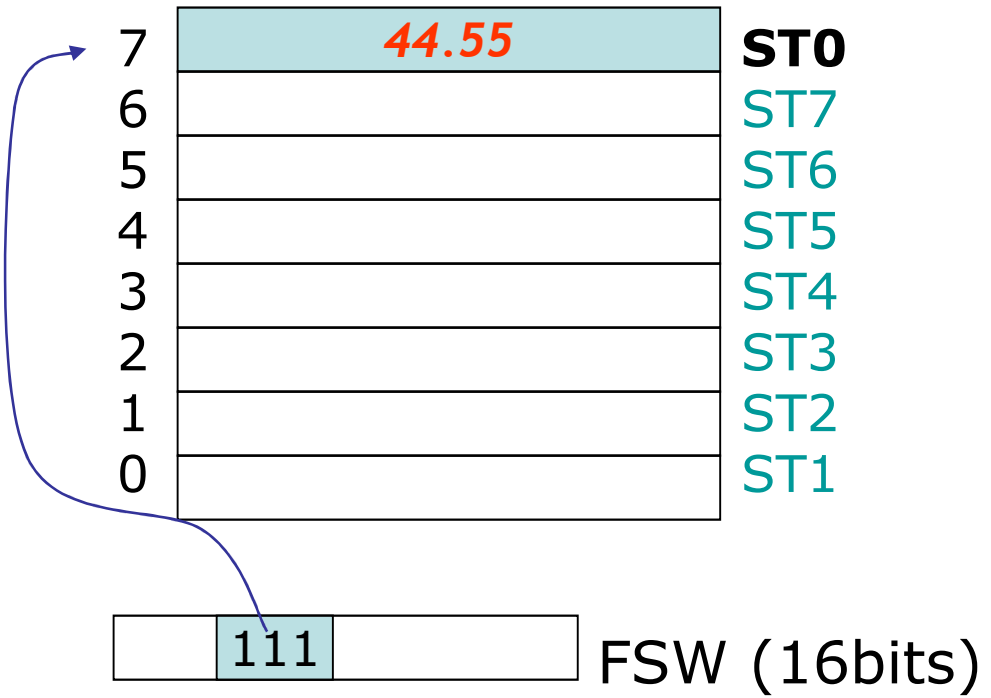


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```

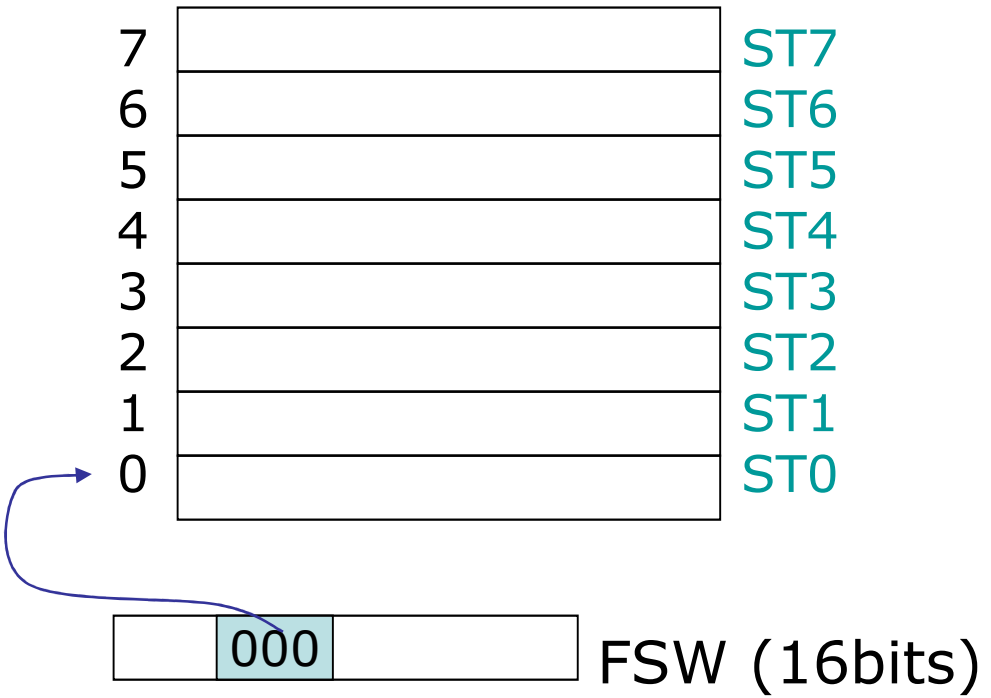


Exemplo de Utilização da FPU

```
section .data
var1        dd      5.1
var2        dd      4.5
tmp         dd      0

section .bss
resultado resq    1

section .text
_start:
   mov dword [tmp], 3
   fild dword [tmp]
   fadd dword [var1]
   fld dword [var2]
   fld1
   faddp st1
   fmulp st1
   fstp qword [resultado]
```



Outro exemplo de utilização da FPU

- Exemplo de avaliação de uma expressão:

  Número de ouro = (1 + √5)/2

  - ✓ Usando notação pós-fixa: 1 5 √ + 2 /
  - ✓ Avaliando usando FPU Intel **usando uma subrotina:**

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1
```

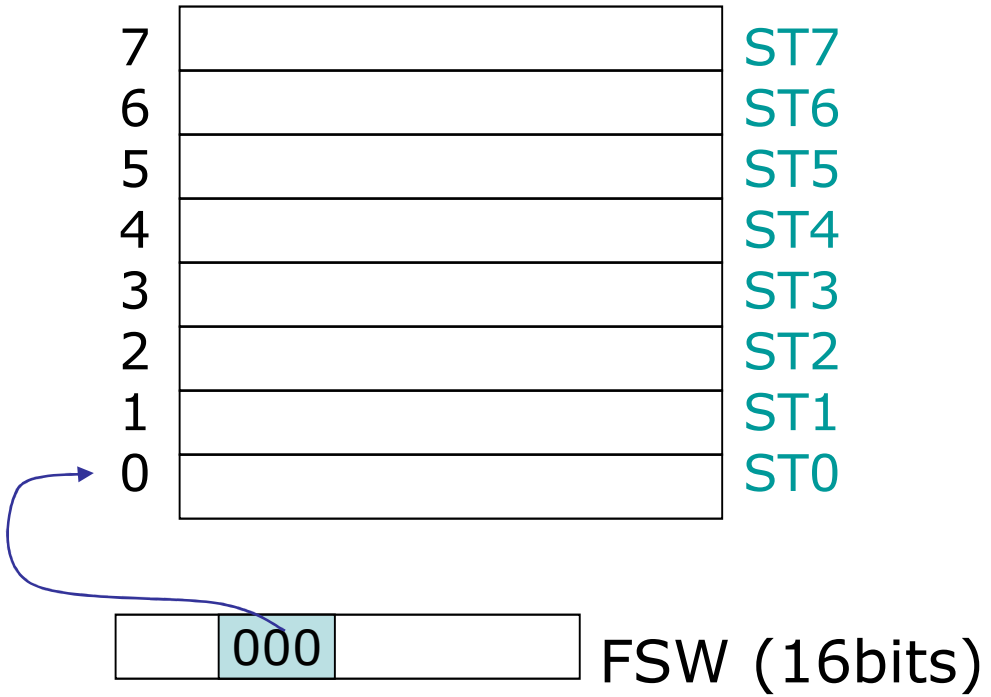
```
section .text
    …
    call aureo
    fstp qword [resultado]
    …
aureo:
    push ebp
    mov esp, ebp
    fld1
    fild dword [five]
    fsqrt
    faddp st1
    fidiv dword [two]
    pop ebp
    ret
```

Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

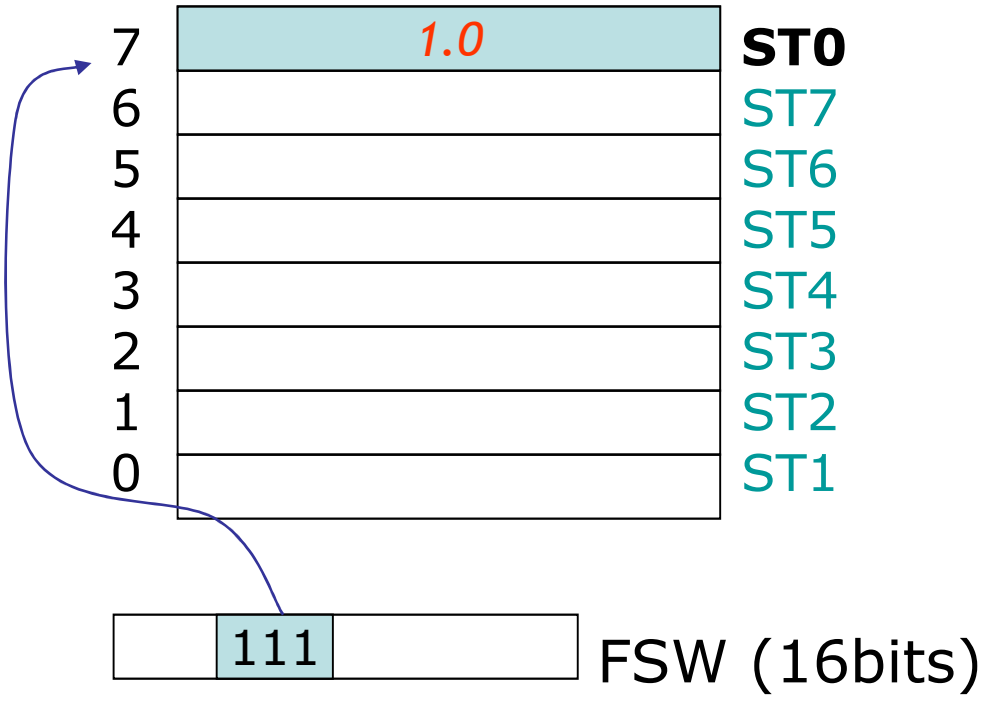


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

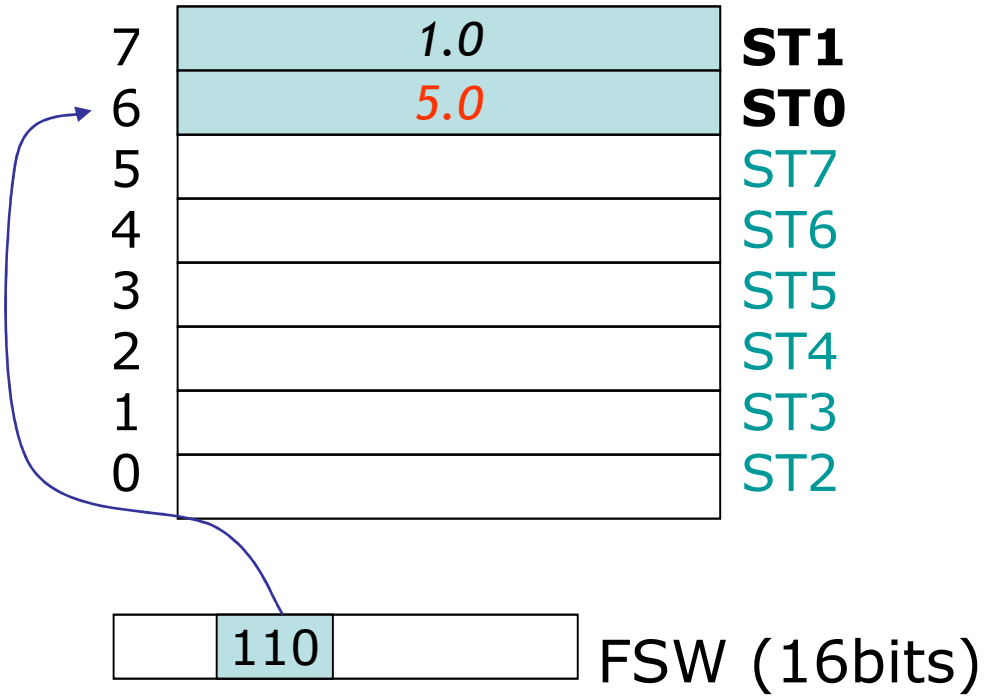


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

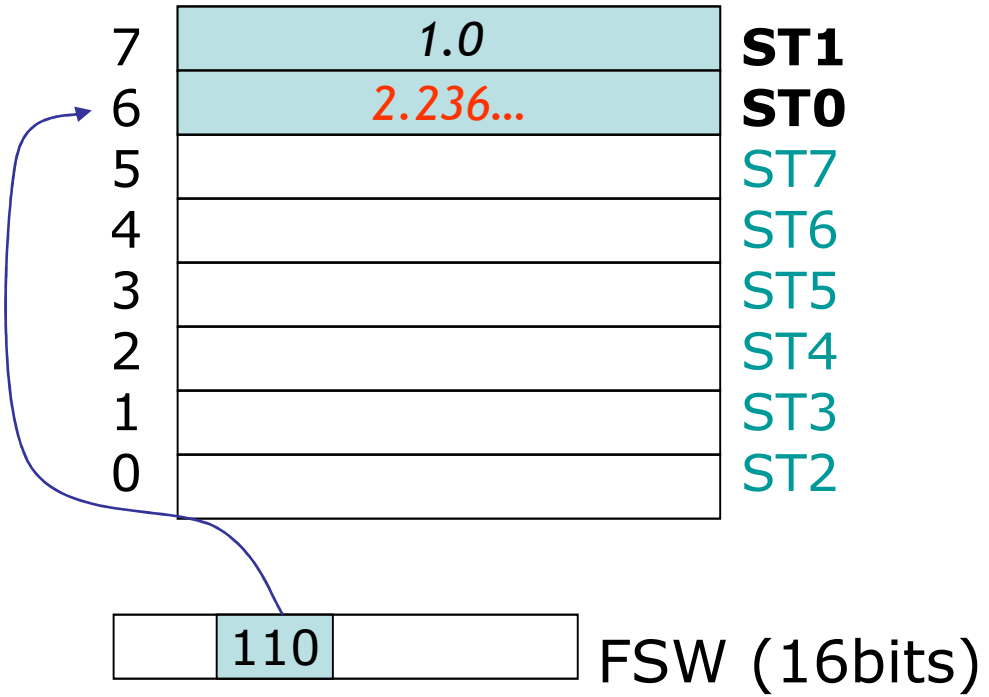


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

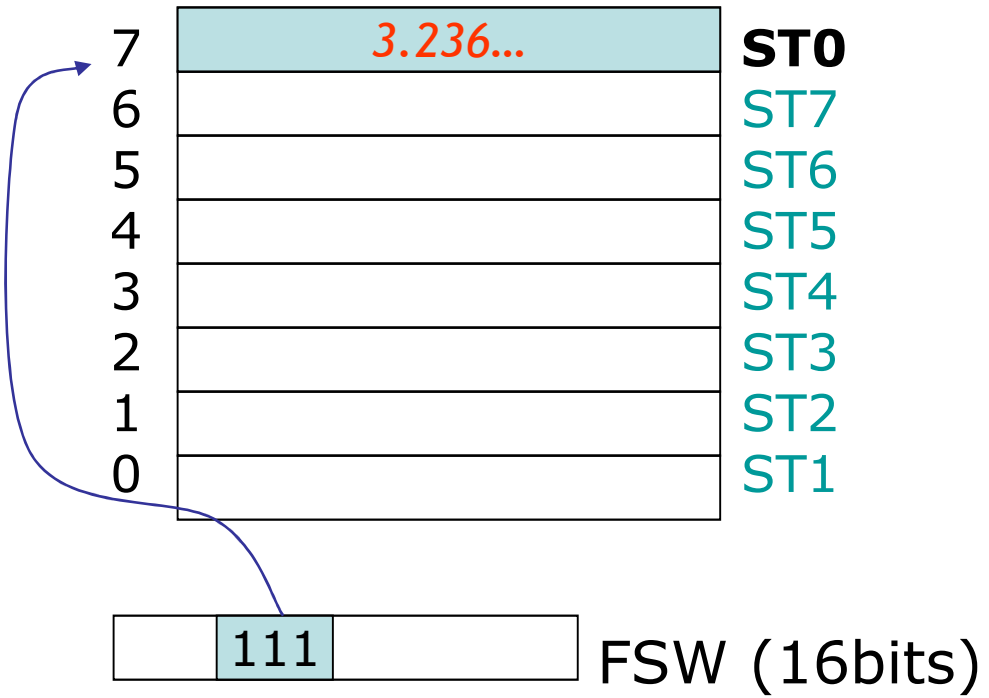


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

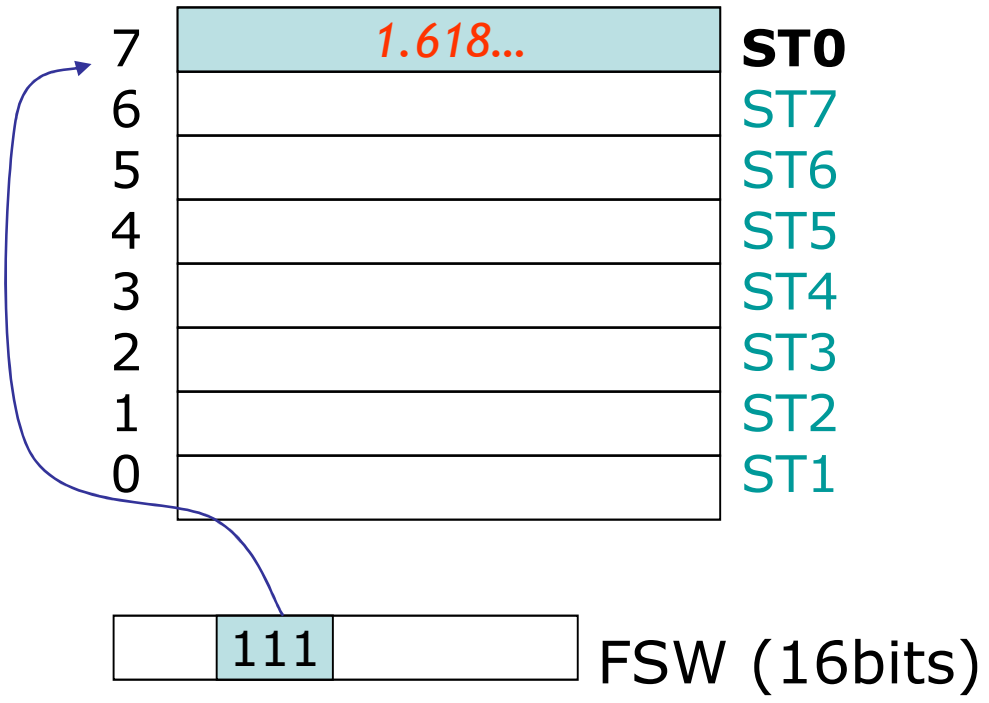


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```

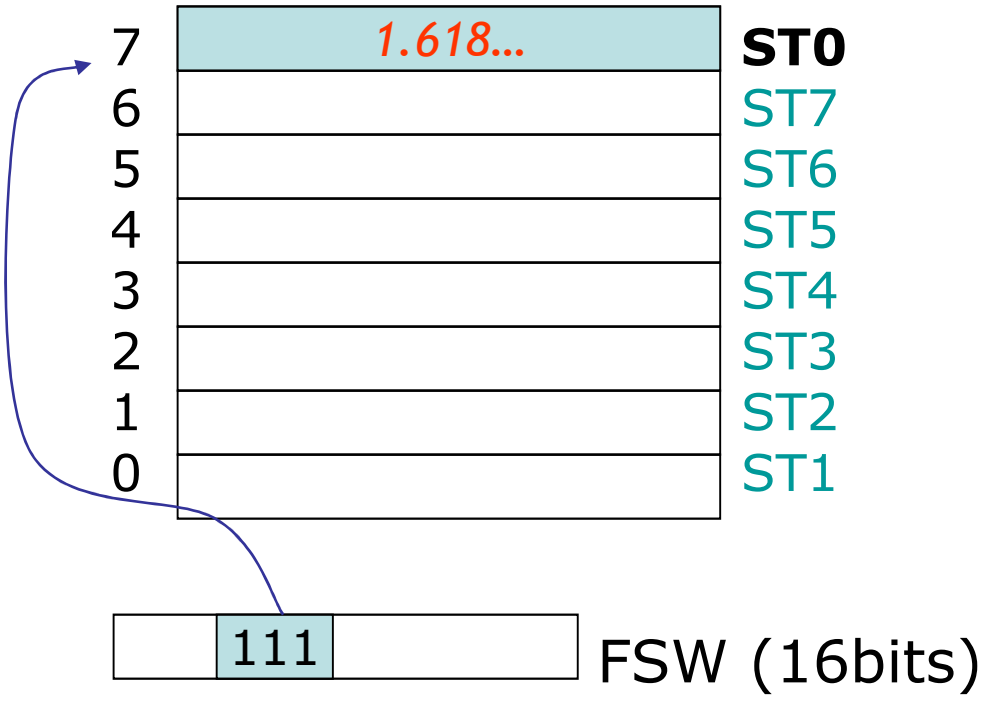


Outro exemplo de utilização da FPU

```
section .data
five   dd     5
two    dd     2

section .bss
resultado resq 1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```



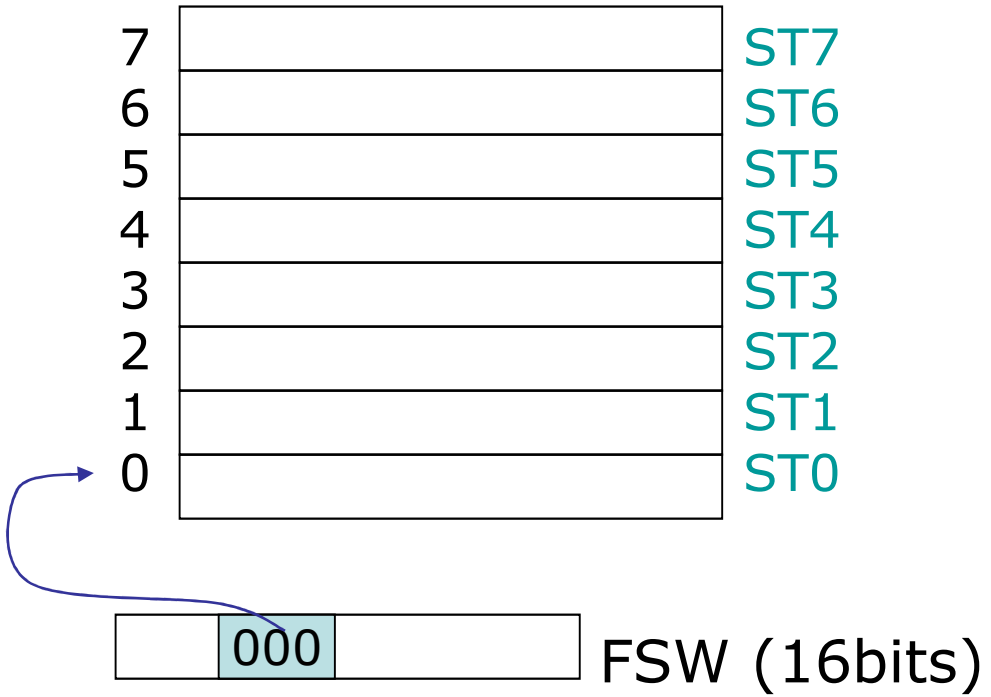
Resultado em st0 não eax

Outro exemplo de utilização da FPU

```
section .data
fivedd      5
two         dd      2

section .bss
resultado resq      1

section .text
   …
   call aureo
   fstp qword [resultado]
   …
aureo:
   push ebp
   mov ebp, esp
   fld1
   fild dword [five]
   fsqrt
   faddp st1
   fidiv dword [two]
   pop ebp
   ret
```



Resultados das Operações

- *Overflow* – resultado demasiado grande (positivo ou negativo)
- *Underflow* – resultado demasiado perto de zero, podendo resultar em zero por falta de precisão
- *Precisão* – resultado arredondado por falta de precisão
- *Overflow* e *Underflow* na FPU pode provocar a terminação do programa (*crash*)

«Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.» -- Null et al.

Registo de Estado/Flags

- FSW (16bits) tem campos de bits para:
  - ✓ TOP (3 bits): indicam qual o reg. no topo da pilha (ST0)
  - ✓ Condition codes: indicam o resultado da última comparação
  - ✓ Bits que indicam a ocorrência de erros:
    - ✓ FPE: erro de precisão
    - ✓ FUE: erro de underflow
    - ✓ FOE: erro de overflow
    - ✓ FZE: erro de divisão por zero
    - ✓ FDE: erro de normalização
    - ✓ FIE: erro devido a operando inválido
    - ✓ STF: indica stack overflow ou stack underflow

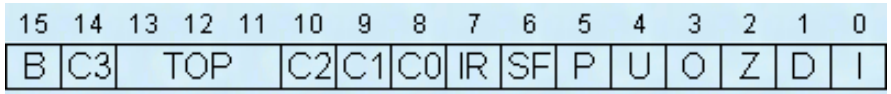


Fig.1.3 Status Word Fields

Comparação

- Colocam o resultado da comparação nos bits C3,C2,C0 do registo de estado FSW

C3	C2	C0	Indicação do resultado
0	0	0	ST0 > operando
0	0	1	ST0 < operando
1	0	0	ST0 = operando
1	1	1	Comparação inválida

- Para que se possam usar as instruções de salto condicional existentes é necessário transpor estas alterações para o registo de estado da ALU - EFLAGS
- Os processadores a partir do Pentium Pro contêm a instrução **FCOMI** que faz essa transposição
  - ✓ Antes era preciso fazer a cópia explicitamente

Comparação

- Os bits C3, C2 e C0 são copiados para as flags ZF, PF e CF: C3 → ZF ; C2 → PF ; C0 → CF
- Estas flags são as avaliadas pelos saltos condicionais sobre inteiros sem sinal
- Portanto é essa familia de saltos que se deve utilizar:
  - ✓ Saltar se ST0 > operando: **JA**
  - ✓ Saltar se ST0 < operando: **JB**
  - ✓ Saltar se ST0 = operando: **JE**
  - ✓ Saltar se ST0 e operando não são comparáveis: **JP**
- Exemplo: saltar se ST0 > X, sendo X um valor a 64 bits

```
fcomi qword [X]
ja maior
```

Propriedades e Comparação

- Devido aos arredondamentos e perca dos bits menos significativos, não podemos garantir que as operações na FPU são comutativas ou distributivas:

  (a + b) + c = a + (b + c) ?

  a*(b + c) = ab + ac ?

  ✓ Relembre o exemplo ("nossa" representação 14 bits):

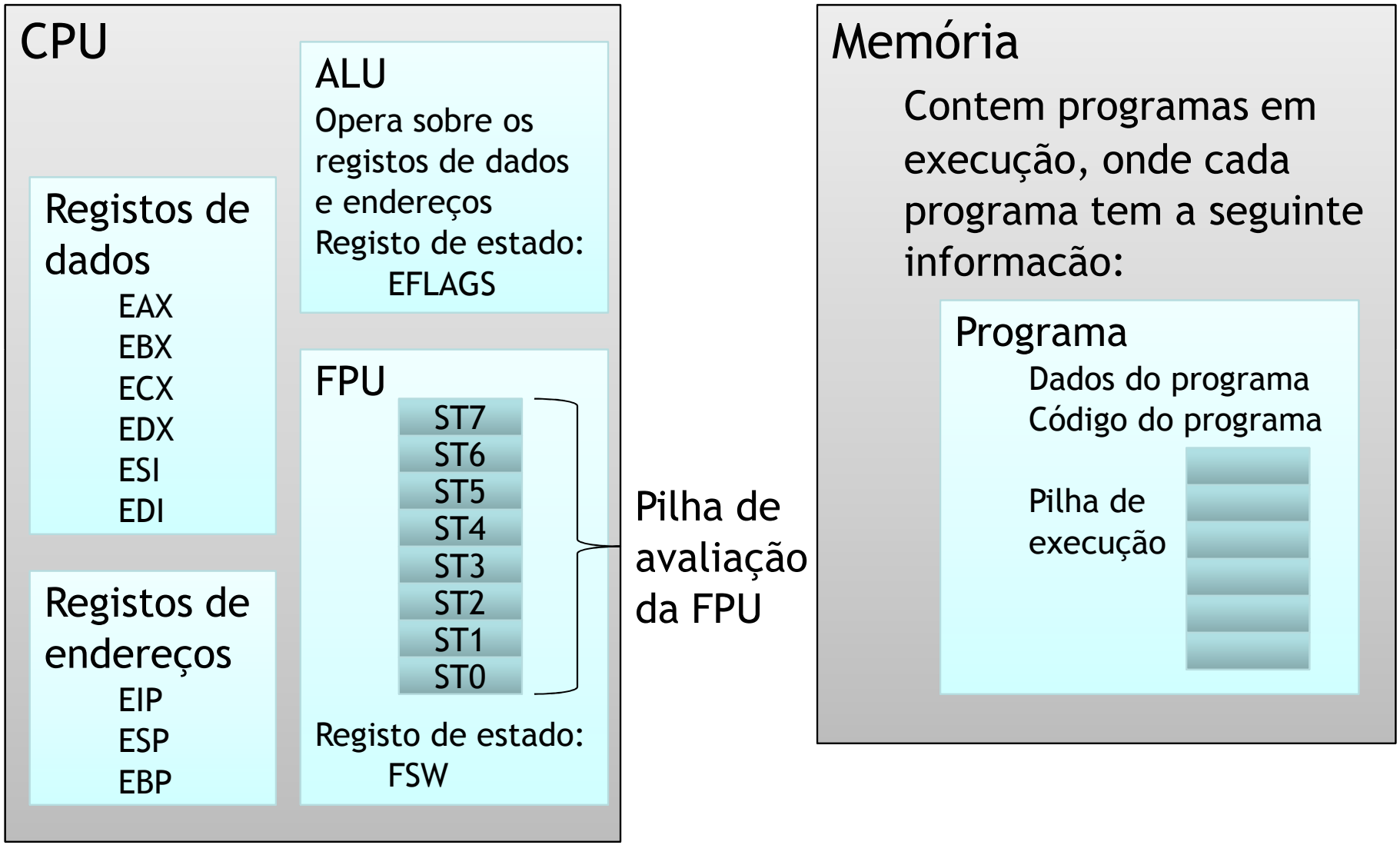
  (512+1)/8 = 64,125 = (512/8)+(1/8)

  mas $513 = 1000000001_2$ não é representável, logo não se deve fazer a soma primeiro;

- Também não podemos comparar directamente a igualdade entre dois números FP. Devemos considerar o erro admitido e verificar se a diferença entre os números está dentro desse erro:

  abs(x-y) < epsilon

Arquitecturas Intel IA-32 com FPU



SIMD - Instruções Vectoriais

- Tradicional - SISD (Single Instruction Single Data)
  - ✓ Um fluxo de instruções opera sobre um fluxo de dados
    - ✓ Operações sobre escalares
  - ✓ Exemplo: R3 ← R1 + R2
    - ✓ Uma soma de dois escalares para obter um terceiro
- Vectorial - SIMD (Single Instruction Multiple Data)
  - ✓ Um fluxo de instruções opera sobre vários fluxo de dados
    - ✓ Operações sobre grupos de escalares
  - ✓ Exemplo: V3 ← V1 + V2
    - ✓ Uma soma de todos os elementos no vector V1 com os de V2 para obter um terceiro vector (V3)
    - ✓ Equivale a:

```
for (i=0; i<N; i++)
    V3[i] = V1[i] + V2[i];
```

Instruções Vectoriais nas Arq. Intel

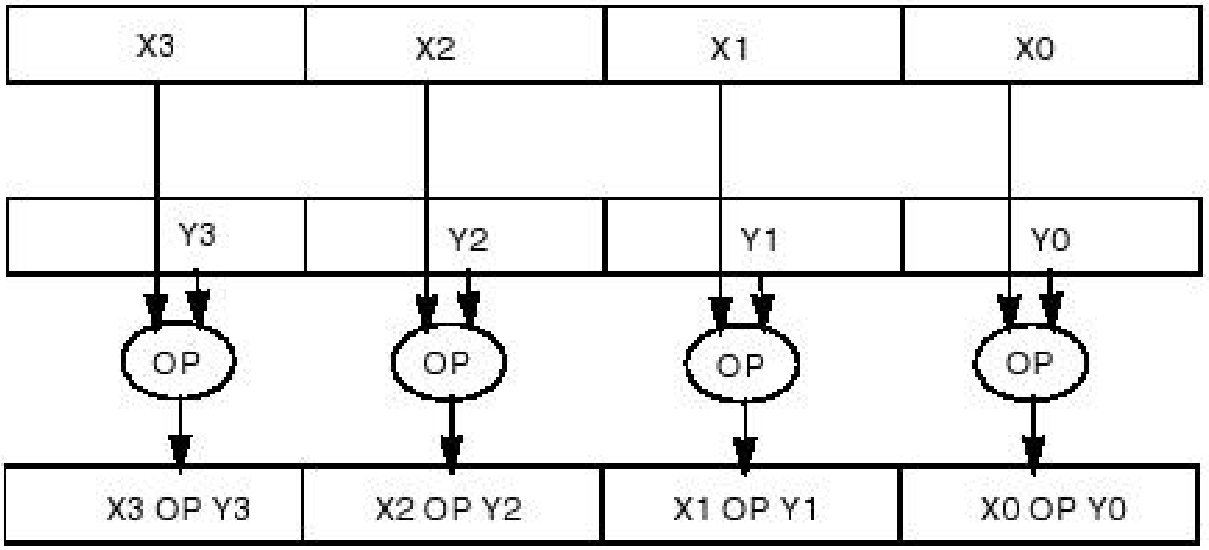
- Instruções vectoriais são incorporadas nas arquitecturas tradicionais

  - ✓ Intel IA-32 tem as extensões:
    - ✓ MMX (MultiMedia eXtension) → Pentium
    - ✓ SSE (Streaming SIMD Extensions) → Pentium III
    - ✓ SSE2, SSE3, SSE4 → Pentium IV
    - ✓ AVX (em desenvolvimento)

  - ✓ AMD, além das extensões da Intel:
    - ✓ 3DNow!, SSE5

Intel SSE

- 8 registos de 128bits (XMM0 – XMM7)
  - ✓ Podem ser interpretados como 2 registos de 64 bits, 4 de 32, 8 de 16 ou 16 de 8
- Suporta load/store destes grupos de valores
- Suporta operações aritméticas (inteiros ou reais) e lógicas sobre estes registos:



Comparação SISD/SIMD

```
  mov ecx, [N]
aloop:
  mov eax, [V1+ecx]
  add eax, [V2+ecx]
  mov [V3+ecx], eax
  dec ecx
  jnz aloop
```

```
  movups xmm0, [V1]
  movups xmm1, [V2]
  addps  xmm0, xmm1
  movups [V3], xmm0
```

- Menos instruções → menos fetchs
- Podem ser reduzidos (ou mesmo eliminados) os ciclos e existem menos dependências de dados

Vantagens/Desvantagens

- Vantagens
  - ✓ Menos instruções no programa
  - ✓ Menos fetch/decode
  - ✓ Paralelismo sobre grupos de valores
  - ✓ Menos acessos à memória
  - ✓ Menos problemas no aproveitamento do pipelining
    - ✓ A estudar mais à frente

- Desvantagens
  - ✓ Unidades de execução paralelas e dedicadas
  - ✓ Que código é realmente "vectorizável"?

Código Vectorizável

- Muitos algoritmos dependem de vectores ou matrizes
  - ✓ Podem facilmente ser implementados usando instruções vectoriais

- Muitos algoritmos dependem da aplicação das mesmas operações a grupos de valores
  - ✓ Agrupamos estes valores em vectores