

Linguagens de Montagem

Ponto-Flutuante Aula 11

Edmar André Bellorini

Ano letivo 2021

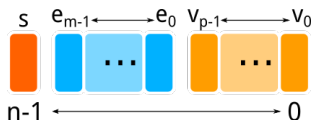
Introdução

Introdução

■ Ponto-Flutuante (*Floating-Point* - FP)

- Formato binário para representação de números reais
- Notação científica

$$\pm v * B^{\pm e}$$



onde:

- v : significando (termo *mantissa* é depreciado)
- B : base numérica, sendo 2 em FP
- e : expoente
- $\pm: 0$ positivo; 1 negativo

Introdução - Padrões IEEE-754

■ IEEE-754 de precisão simples

■ float



■ $3,1415_d \approx 0\ 10000000\ 10010010000111001010110_{ieee32}$

■ IEEE-754 de precisão dupla

■ double



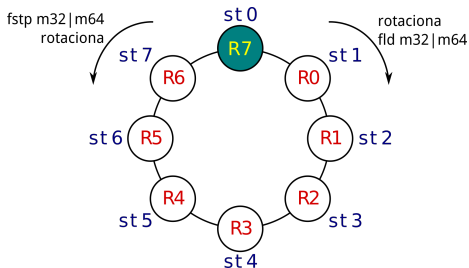
■ $-3,1415_d \approx 1\ 100000000000\ 1001001000011100101011000000100... \\ ...000110001001001101111_{ieee64}$

do you wanna some fun? [click here!](#) or [here!](#) another one?!

Hardware x86_64 de 1ª Geração

■ FPU: início da década de 1980

- Projeto com ênfase na notação polonesa reversa (RPN)
- Pilha de Registradores FPU (st_i)



- Instruções unárias, com operando destino implícito

`FADD [T0] st[1-7]/mem32/mem64/mem80`

- Concorrência de recursos com MMX (final de 1990)
Quer saber mais sobre FPU?

Hardware x86_64 de 2ª Geração

- SSE: final da década de 1990
 - Streaming SIMD Extensions
Single Instruction, Multiple Data - by Flynn
 - Evolução do MMX
 - Várias interações, sendo SSE4 a mais atual, seguido da AVX
 - Projeto com ênfase em registradores *thanks to the engineering God!*
 - Instruções são binárias, isto é, operandos fonte e destino
 - Instruções Escalares e Vetoriais para Inteiros e FP
 - 16 Registradores de propósito específico (Escalar/Vetorial)
xmm0 até *xmm15*
de 128 bits cada um
 - É o padrão para operações em FP na arquitetura x86_64

Hardware x86_64 de 3ª Geração

- AVX: início da década de 2010
 - Advanced Vector Extensions
 - Evolução do SSE
 - Adiciona instruções ternárias com destino, fonte, fonte
 - Adiciona instruções vetoriais com mais inteiros (AVX2)
 - Aumenta o tamanho de cada registrador para 256bits (ou 512bits no AVX512)
 - Registradores *ymm0* até *ymm15*
 - Registradores SSE *xmm* são registradores AVX *ymm*
 - Semelhante ao que acontece com Regs de Propósito Geral

Observação sobre esta aula

- Apresentação dos conceitos relacionados com FP
- Operações Escalares
 - Instruções de Conversão
 - Instruções de Transferência
 - Instruções Aritméticas
 - Instruções de Comparação
- Conceitos sobre SSE vetorial **não** serão discutidos
 - Na próxima aula (possivelmente)
- Suporte à SSE e/ou AVX em Linux/Terminal

```
lshw -c cpu
```

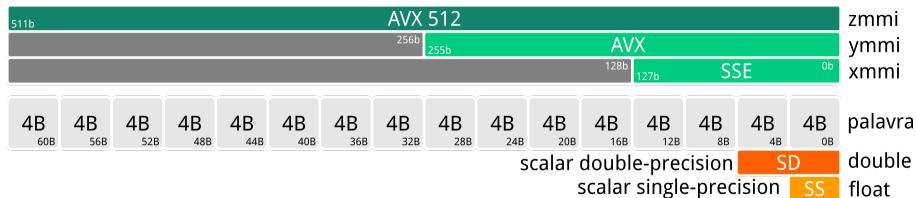
em capacidades, procure por SSE e/ou AVX

Registradores

Registradores de Propósito Específicos (FP)

- `xmm0` até `xmm15`
- Operações escalares em FP
- Instrução determina o tamanho da palavra
 - Palavra com Precisão Simples - 32 bits
modificador `s` (single)
 - Palavra com Precisão Dupla - 64 bits
modificador `d` (double)
- Passagem de parâmetros
`xmm0` até `xmm7`
Lembram do `xor rax, rax` para `printf`?
- Temporários (caller-saved)
`xmm8` até `xmm15`
- Retorno FP de função em `xmm0`

Registadores de Propósito Específicos (FP)



Instruções de Conversão

Conversão de Inteiro para FP

CVTSI2SS

- Convert Signed Integer to Scalar Single-Precision

```
CVTSI2SS xmmi, r32|r64|mem32|mem64
```

Converte um inteiro sinalizado (32 ou 64 bits) para FP (32 bits)

CVTSI2SD

- Convert Signed Integer to Scalar Double-Precision

```
CVTSI2SD xmmi, r32|r64|mem32|mem64
```

Converte um inteiro sinalizado (32 ou 64 bits) para FP (64 bits)

Conversão de FP para Inteiro

CVTSS2SI

- Convert Scalar Single-Precision to Signed Integer

```
CVTSS2SI r32|r64, xmmi|mem32
```

Arredonda um FP (32 bits) para Inteiro sinalizado de 32 ou 64 bits

CVTSD2SI

- Convert Scalar Double-Precision to Signed Integer

```
CVTSD2SI r32|r64, xmmi|mem64
```

Arredonda um FP (64 bits) para Inteiro sinalizado de 32 ou 64 bits

Conversão entre FP Single e Double

CVTSS2SD

- Convert Scalar Single-Precision to Scalar Double-Precision

```
CVTSS2SD xmmi, xmmj|mem32
```

Converte um FP (32 bits) para FP (64 bits)

CVTSD2SS

- Convert Scalar Double-Precision to Scalar Single-Precision

```
CVTSD2SS xmmi, xmmj|mem64
```

Converte um PF (64 bits) para FP (32 bits)

Instruções de Transferência

Transferência de FP

MOVSS

- Move Scalar Single-Precision

```
MOVSS xmmi, xmmj|mem32  
MOVSS mem32, xmmi
```

Move um FP (32 bits) entre xmmi e mem32

MOVSD

- Move Scalar Double-Precision

```
MOVSD xmmi, xmmj|mem64  
MOVSD mem64, xmmi
```

Move um FP (64 bits) entre xmmi e mem64

Exemplo a11e01.asm: Conversão entre INT e FP

```
47 l1:
48 ; eax = printf("Inteiro %d convertido para float = %f\n",
49 mov rax, 1 ; existe um FP a ser impresso e já está em xmm0
50 mov rdi, pfstr2
51 mov esi, [inteiro1]
52 ; notas sobre printf e FP
53 ; printf destrói xmmi após chamada, opcional: salvar
54 MOVSS [float1], xmm0 ; salva FP em float1
55 ; considera %f como double, necessário converter SS
56 CVTSS2SD xmm0, xmm0 ; float 2 double 4 printf
57 call printf
```

Exemplo a11e01.asm - GDB

```
b 11
b 12
b 13
b fim
r          ; executar
Digite um Inteiro: <42>; entrada do usuario
p (int) inteiro1          ; inteiro em memoria
p $xmm0                   ; lista de float/double de xmm0
c
p (float) float1          ; float xmm0 em memoria
c
Inteiro 42 convertido para double = 42.000000
p $xmm0                   ; valor foi destruido por printf
...
```

Instruções Aritméticas

Adição Escalar - ADDSS_{size}

ADDSS

■ ADD Scalar Single-Precision

```
ADDSS xmmi, xmmj|mem32
```

$$xmmi = xmmi + xmmj|mem32$$

```
VADDSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = xmmj + xmmk|mem32$$

ADDSD

■ ADD Scalar Double-Precision

```
ADDSD xmmi, xmmj|mem64
```

```
VADDSD xmmi, xmmj, xmmk|mem64
```

Subtração Escalar - SUBS_{size}

SUBSS

■ SUB Scalar Single-Precision

```
SUBSS xmmi, xmmj|mem32
```

$$xmmi = xmmi - xmmj|mem32$$

```
VSUBSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = xmmj - xmmk|mem32$$

SUBSD

■ SUB Scalar Double-Precision

```
SUBSD xmmi, xmmj|mem64
```

```
VSUBSD xmmi, xmmj, xmmk|mem64
```

Multiplicação Escalar - MULS_{size}

MULSS

■ MUL Scalar Single-Precision

```
MULSS xmmi, xmmj|mem32
```

$$xmmi = xmmi * xmmj|mem32$$

```
VMULSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = xmmj * xmmk|mem32$$

MULSD

■ MUL Scalar Double-Precision

```
MULSD xmmi, xmmj|mem64
```

```
VMULSD xmmi, xmmj, xmmk|mem64
```

Divisão Escalar - DIVS_{size}

DIVSS

■ DIV Scalar Single-Precision

```
DIVSS xmmi, xmmj|mem32
```

$$xmmi = xmmi / xmmj | mem32$$

```
VDIVSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = xmmj / xmmk | mem32$$

DIVSD

■ DIV Scalar Double-Precision

```
DIVSD xmmi, xmmj|mem64
```

```
VDIVSD xmmi, xmmj, xmmk|mem64
```


Raiz Quadrada Escalar - SQRTS_{size}

SQRTSS

■ SQR T Scalar Single-Precision

```
SQRTSS xmmi, xmmj|mem32
```

$$xmmi = \sqrt{xmmj|mem32}$$

SQRTSD

■ SQR T Scalar Double-Precision

```
SQRTSD xmmi, xmmj|mem64
```

$$xmmi = \sqrt{xmmj|mem64}$$

Exemplo a11e02.asm: distância entre 2 pontos 3D

```
66      ; xmm0 = (x1 - x2)^2
67      subsd xmm0, xmm3      ; (x1 - x2)
68      mulsd xmm0, xmm0      ; ^2
69
70      ; xmm1 = (y1 - y2)^2
71      subsd xmm1, xmm4      ; (y1 - y2)
72      mulsd xmm1, xmm1      ; ^2
73
74      ; xmm2 = (z1 - z2)^2
75      subsd xmm2, xmm5      ; (z1 - z2)
76      mulsd xmm2, xmm2      ; ^2
77
78      ; adição das 3 diferenças ao quadrado
79      addsd xmm0, xmm1
80      addsd xmm0, xmm2
81
82      ; raiz quadrada em xmm0
83      sqrtsd xmm0, xmm0
```

Instruções Lógicas

AND vetorial* - ANDP_{size}

ANDPS

- Bitwise AND of Packed Single-Precision

```
ANDPS xmmi, xmmj|mem128
```

$$xmmi = xmmi \wedge (xmmj|mem128)$$

```
VANDPS xmmi, xmmj, xmmk|mem128
```

$$xmmi = xmmj \wedge (xmmk|mem128)$$

ANDPD

- Bitwise AND of Packed Double-Precision

```
ANDPD xmmi, xmmj|mem128
```

```
VANDPD xmmi, xmmj, xmmk|mem128
```

OR vetorial* - ORP_{size}

ORPS

- Bitwise OR of Packed Single-Precision

```
ORPS xmmi, xmmj|mem128
```

$$xmmi = xmmi \vee (xmmj|mem128)$$

```
VORPS xmmi, xmmj, xmmk|mem128
```

$$xmmi = xmmj \vee (xmmk|mem128)$$

ORPD

- Bitwise OR of Packed Double-Precision

```
ORPD xmmi, xmmj|mem128
```

```
VORPD xmmi, xmmj, xmmk|mem128
```

XOR vetorial* - XORP_{size}

XORPS

- Bitwise XOR of Packed Single-Precision

```
XORPS xmmi, xmmj|mem128
```

$$xmmi = xmmi \oplus xmmj|mem128)$$

```
VXORPS xmmi, xmmj, xmmk|mem128
```

$$xmmi = xmmj \oplus xmmk|mem128)$$

XORPD

- Bitwise XOR of Packed Double-Precision

```
XORPD xmmi, xmmj|mem128
```

```
VXORPD xmmi, xmmj, xmmk|mem128
```

Instruções Comparativas

Máximo Escalar - MAXS_{size}

MAXSS

- Maximum Scalar Single-Precision

```
MAXSS xmmi, xmmj|mem32
```

$$xmmi = \text{MAX}(xmmi, xmmj|mem32)$$

```
VMAXSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = \text{MAX}(xmmj, xmmk|mem32)$$

MAXSD

- Maximum Scalar Double-Precision

```
MAXSD xmmi, xmmj|mem64
```

```
VMAXSD xmmi, xmmj, xmmk|mem64
```


Mínimo Escalar - MINS_{size}

MINSS

- Minimum Scalar Single-Precision

```
MINSS xmmi, xmmj|mem32
```

$$xmmi = MIN(xmmi, xmmj|mem32)$$

```
VMINSS xmmi, xmmj, xmmk|mem32
```

$$xmmi = MIN(xmmj, xmmk|mem32)$$

MINSD

- Minimum Scalar Double-Precision

```
MINSD xmmi, xmmj|mem64
```

```
VMINSD xmmi, xmmj, xmmk|mem64
```

Exemplo a11e03.asm: float abs(float f)

```
75      ; absf?  
76      ; xmm2 = (float) 0  
77      ; xmm1 = xmm2 - xmm0  
78      ; max(xmm0, (0-xmm1))  
79      XORPS  xmm2, xmm2      ; xmm2 = (float) 0  
80      VSUBSS xmm1, xmm2, xmm0 ; xmm1 = xmm2 - xmm0  
81      MAXSS  xmm0, xmm1      ; max(xmm0, (0-xmm1))
```

Comparar Escalar - COMIS_{size}

COMISS

- Compare Scalar Ordered Single-Precision

```
COMISS xmmi, xmmj|mem32
```

Compara xmmi com xmmj|m32 e ajusta EFLAGS

Veja tabela no próximo slide

COMISD

- Compare Scalar Ordered Double-Precision

```
COMISD xmmi, xmmj|mem64
```

Compara xmmi com xmmj|m64 e ajusta EFLAGS

Comparar Escalar - COMIS_{size}

Tabela de J_{cc} e ordem de utilização:

Ordem	CMP Op ₁ , Op ₂	ZP PF CF	Jcc	ZF PF CF	Descrição
1	UNORDERED	1 1 1	JP	X 1 X	Op _i é NAN
2	EQUAL	1 0 0	JE	1 X X	Op ₂ ==Op ₁
3	GREATER	0 0 0	JA	0 X 0	Op ₂ >Op ₁
4	LESS	0 0 1	JB	X X 1	Op ₂ <Op ₁

Exemplo a11e04.asm: Comparação entre FPs

```
46      movss xmm0, [scfF1]
47      movss xmm1, [scfF2]
48
49      lif1:
50      COMISS xmm0, xmm1
51      JP lerro      ; para teste, trocar para NJP
52      JE liguais    ; xmm1 == xmm0?
53      JA lmaior     ; xmm1 < xmm0?
54      JB lmenor     ; xmm1 > xmm0?
```

Atividades

AT1101

Função dis3Dlm

- Cálculo de distância entre 2 pontos
 - Pontos com 3 coordenadas
- Assinatura da Função dis3Dlm:

```
double dis3Dlm(int p1[], int p2[]){...}
```

- Dicas:
 - Adaptação do exemplo a11e02.asm

AT1102

Função potência para FP Single-Precision

- Criar um código que leia um `float` (base) e um `inteiro` (exp)
- Execute a função:

```
float powlm(float base, int exp){...}
```

- Dicas:
 - Diversos exemplos desta aula contém leitura de FP e INT
 - Parâmetros via registrador `xmmi` e REGs
 - Retorno em `xmm0`
 - `printf` só imprime `double`