Software Básico

Instruções de Ponto Flutuante

INSTITUTO DE INFORMÁTICA - UFG

Reconhecimento

- Material produzido por:
  - Noemi Rodriguez – PUC-Rio
  - Ana Lúcia de Moura – PUC-Rio
- Adaptação
  - Bruno Silvestre – UFG

Ponto Flutuante

- Arquiteturas 8086, 286 e 386
  - Não havia ponto flutuante integrado no processador
  - FPU era um co-processador (“x87”)
  - Pilha de 8 registradores de 80 bits (formato não padrão IEEE)
  - Dificuldade para geração de código
- Arquitetura 486 e Pentium I e II
  - FPU integrada com o processador
  - Mas ainda seguindo o modelo anterior

Ponto Flutuante

- Pentium II introduziu a extensão MMX
  - Código de ponto flutuante baseado não mais em pilha, mas em um conjunto de registradores
    - 8 registradores de 64bits
    - MM0 a MM7
  - Pode processar um conjunto de números em paralelo

Ponto Flutuante

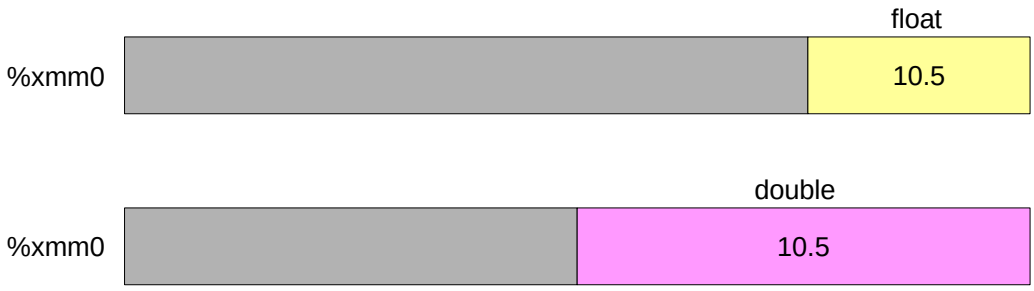
- Extensão SSE (Pentium III)
  - Melhora o MMX (novas instruções)
  - Em 32 bits: 8 registradores de 128bits
    - XMM0 a XMM7
  - **Em 64 bits:** 16 registradores de 128bits
    - XMM0 a XMM15
    - Em Assembly: %xmm0, %xmm1, …, %xmm15
  - Suporta apenas single-precision floating (float)

Ponto Flutuante

- SSE2 (Pentium 4)
  - Suporta a double-precision floating (double)
  - Extensão importante!

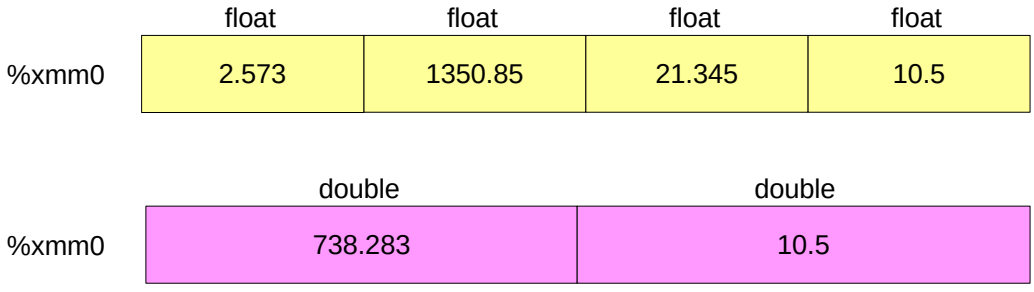
Registradores Packed vs Scalar

- Scalar:
  - Um único valor de ponto flutuante (*float* ou *double*) na parte baixa do registrador



Registradores Packed vs Scalar

- Packed:
  - Registrador armazena um “vetor” de elementos



Registradores Packed vs Scalar

- Packed:
  - Registrador armazena um “vetor” de elementos

“add_float_packed” %xmm0, %xmm1, %xmm3

%xmm0	4.0	3.0	2.0	1.0
	+	+	+	+
%xmm1	1.0	2.5	1.0	2.5
	=	=	=	=
%xmm3	5.0	5.5	3.0	3.5

Chamadas de Procedimentos

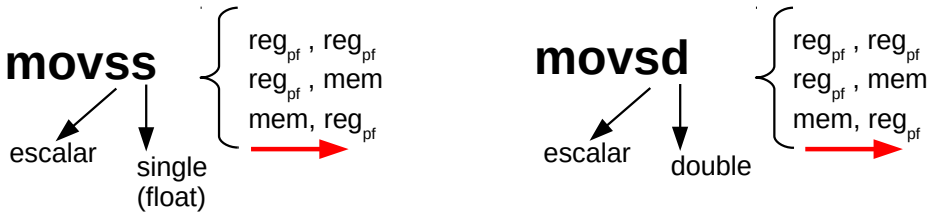
- Até 8 argumentos de ponto flutuante passados em registradores (%xmm0 a %xmm7)
  - Inteiros ainda são passados em %rdi, %rsi, …
- Todos os registradores %xmm são “*caller-saved*”
  - Podem ser sobrescritos pela função chamada
- Valor de retorno de ponto flutuante em %xmm0

```
double foo (float f, int i, double d) {...}
```

%xmm0 %xmm0 %edi %xmm1

Instruções de Ponto Flutuante

Movimentação de Dados



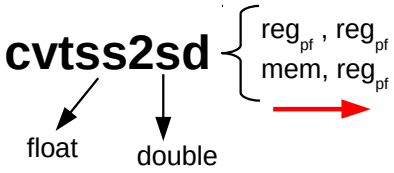

```
double foo(double a, double b){
  return b;
}
```

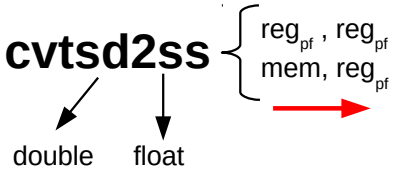
```
foo:
  movsd %xmm1, %xmm0
  ret
```

```
float boo (float *a) {
  return *a;
}
```

```
boo:
  movss (%rdi), %xmm0
  ret
```

Conversões de Dados





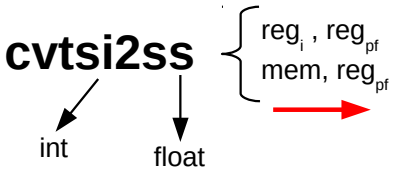

```
double foo (float f) {
  return (double) f;
}
```

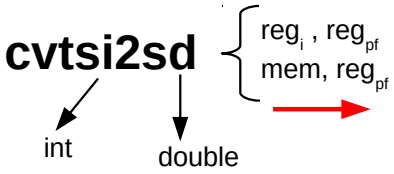
```
foo:
  cvtss2sd %xmm0,%xmm0
  ret
```

```
float boo (double *a) {
  return (float)*a;
}
```

```
boo:
  cvtsd2ss (%rdi),%xmm0
  ret
```

Conversões de Dados





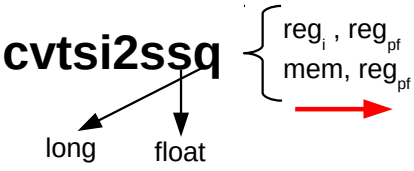

```
double foo (int i) {
  return (double) i;
}
```

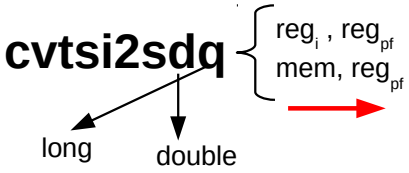
```
foo:
  cvtsi2sd %edi,%xmm0
  ret
```

```
float boo (int *a) {
  return (float)*a;
}
```

```
boo:
  cvtsi2ss (%rdi),%xmm0
  ret
```

Conversões de Dados





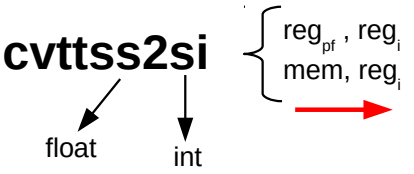

```
double foo (long l) {
  return (double) l;
}
```

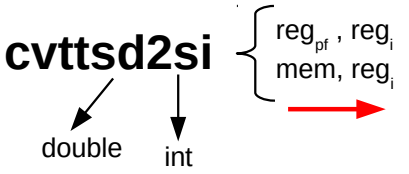
```
foo:
  cvtsi2sdq %rdi,%xmm0
  ret
```

```
float boo (long *a) {
  return (float)*a;
}
```

```
boo:
  cvtsi2ssq (%rdi),%xmm0
  ret
```

Conversões com Truncamento





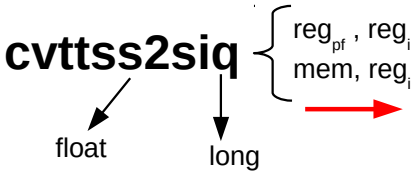

```
int foo (double d) {
  return (int) d;
}
```

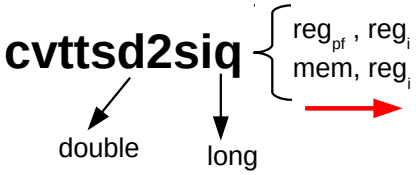
```
foo:
  cvttsd2si %xmm0, %eax
  ret
```

```
int boo (float *a) {
  return (int)*a;
}
```

```
boo:
  cvttss2si (%rdi), %eax
  ret
```

Conversões com Truncamento






```
long foo (double d) {
  return (long) d;
}
```

```
foo:
  cvttsd2siq %xmm0, %rax
  ret
```

```
long boo (float *a) {
  return (long)*a;
}
```

```
boo:
  cvttss2siq (%rdi),%rax
  ret
```

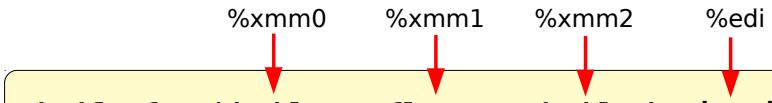
Operações Aritméticas

Float	Double
addss	addsd
subss	subsd
mulss	mulsd
divss	divsd
maxss	maxsd
minss	minsd
sqrtss	sqrtsd

reg_{pf} , reg_{pf}

mem, reg_{pf}

Exemplo

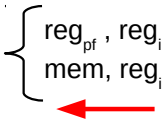



```
double foo (double a, float x, double b, int i) {
  return a*x - b/i;
}
```

```
foo:
  cvtss2sd %xmm1,%xmm1      /* (double) x        */
  mulsd %xmm1, %xmm0        /* %xmm0 = a * x     */
  cvtsi2sd %edi, %xmm1      /* (double) i        */
  divsd %xmm1, %xmm2        /* %xmm2 = b / i     */
  subsd %xmm2, %xmm0        /* %xmm0 = a*x - b/i */
  ret
```

Comparação de Valores

ucomiss
ucomisd



- Testar resultado com as condições de comparações sem sinal (ja, jae, jb, jbe)

```
int foo (double a, double b)
{
  if (a > b) return 1;
  return 0;
}
```

```
foo:
  movl $0, %eax
  ucomisd %xmm1, %xmm0
  jbe fim
  movl $1, %eax
fim:
  ret
```

Variável Global

- Temos as diretivas “.float” e “.double”

Linguagem C

```
float  x = 10.0;
double y = 25.4;

int main() {
  ...
}
```

Linguagem Assembly

```
.data
x: .float  10.0
y: .double 25.4

.text
globl main
main:
   ...
```

Valores Constantes

- As instruções de ponto flutuante não aceitam imediatos
- Temos que criar variáveis (ou constantes) e usá-las para inicializar outras variáveis ou registradores

```
void foo() {
  float x = 12.5;
  if (x > 5.0) {
    ...
  }
}
```

Valores Constantes

```
void foo() {
  float x = 12.5;
  if (x > 5.0) {
    ...
  }
}
```

```
.data
tmp01: .float 12.5
tmp02: .float  5.0

.text
foo:
   movss   tmp01, %xmm0
   movss   tmp02, %xmm1
   ucomiss %xmm1, %xmm0
   jbe     endif
   ...
```

Valores Constantes

```
void foo() {
  float x = 12.5;
  if (x > 5.0) {
    ...
  }
}
```

```
.section .rodata
tmp01: .float 12.5
tmp02: .float  5.0

.text
foo:
   movss   tmp01, %xmm0
   movss   tmp02, %xmm1
   ucomiss %xmm1, %xmm0
   jbe     endif
   ...
```

Melhor é declarar como constantes, ou seja, declarar na seção de “Read-Only Data” conhecida como seção “.rodata”.

Preservando Valores

```
double aux(double x);
double foo (double a, double b) {
    return a + aux(b);
}
```

```
foo:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16,  %rsp        /* espaço na pilha para 'a' */
  movsd %xmm0, -8(%rbp)    /* guarda 'a'               */
  movsd %xmm1, %xmm0       /* parametro 'b' para boo   */
  call  aux                /* retorno em %xmm0         */
  addsd -8(%rbp), %xmm0    /* soma 'a'                 */
  leave
  ret
```