

Sistemas Hardware-Software

Aula 04 – Funções

Engenharia

Fabio Lubacheski

Maciel C. Vidal

Igor Montagner

Fábio Ayres



Aula passada !

movq : Combinações de operandos

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

Não é permitido fazer transferência direta memória-memória com uma única instrução

Modos simples de endereçamento

Normal (R) $\text{Mem}[\text{Reg}[R]]$

- Registrador R especifica o endereço de memória

`movq (%rcx), %rax`

Deslocamento (Displacement) $D(R)$ $\text{Mem}[\text{Reg}[R]+D]$

- Registrador R especifica início da região de memória
- Constante de deslocamento D especifica offset

`movq 8(%rbp), %rdx`

E os tamanhos?

O tamanho do dado é especificado na instrução! MOV não converte tipos!

Usamos um sufixo com o tamanho do tipo:

Q = **quad** word (8 bytes)

L = **long** word (4 bytes)

W = **word** (2 bytes)

B = **byte** (1 bytes)

Também podemos ver o tamanho dos registradores usados!

Modo de endereçamento completo

Forma geral: $D(Rb, Ri, S)$

Representa o valor $Mem[Reg[Rb] + S * Reg[Ri] + D]$

Ou seja:

- O registrador **Rb** tem o endereço base
 - Pode ser qualquer registrador inteiro
- O registrador **Ri** tem um inteiro que servirá de índice
 - Qualquer registrador inteiro menos **%rsp**
- A constante **S** serve de multiplicador do índice
 - Só pode ser 1, 2, 4 ou 8
- A constante **D** é o offset



Aula de hoje !

lea

“Prima” da instrução **mov**

- Mas ao invés de pegar dados da memória, **apenas calcula o endereço** de memória desejado
 - Daí vem o nome: *Load Effective Address*

Funcionamento: **lea** *Mem*, *Dst*

- **Mem**: operando de endereçamento da forma D(Rb, Ri, S)
 - Exemplo: **\$0x4(%rax, %rbx, 4)**
- **Dst**: registrador destino
 - Exemplo: **%rsi**

Efeito final: calcula o endereço especificado pelo operando **Mem**, e armazena em **Dst**

lea versus mov

Exemplo:

```
lea $0x4(%rax, %rbx, 8), %rsi
```

Resulta em

$$R[\%rsi] = 4 + R[\%rax] + 8 \times R[\%rbx]$$

Compare com:

```
mov $0x4(%rax, %rbx, 8), %rsi
```

que resulta em

$$R[\%rsi] = M[4 + R[\%rax] + 8 \times R[\%rbx]]$$

(Ou seja, enquanto o **lea** só calcula o endereço, o **mov** vai lá buscar na memória)

Usos da instrução **lea**

lea: equivale em C a **p = &v[i]**

mov: equivale em C a **p = v[i]**

A instrução **lea** também é muito usada para fazer cálculos matemáticos simples, por exemplo:

```
long m12(long x) {  
    return x*12;  
}
```

```
leaq (%rdi,%rdi,2), %rax    // t <- x + x*2  
salq $2, %rax              // return t << 2
```

Vantagem: **lea** é muito rápida, faz contas com dois registradores e armazena em um terceiro!

Operações aritméticas simples

<i>Instrução</i>	<i>Cálculo</i>	
<code>addq S, D</code>	<code>D = D + S</code>	
<code>subq S, D</code>	<code>D = D - S</code>	
<code>imulq S, D</code>	<code>D = D * S</code>	
<code>salq S, D</code>	<code>D = D << S</code>	# Tanto arit. como lógico, o mesmo # que <code>shlq</code>
<code>sarq S, D</code>	<code>D = D >> S</code>	# Aritmético: o sinal é mantido # (signed)
<code>shrq S, D</code>	<code>D = D >> S</code>	# Lógico: o bit mais a esq é zerado # (unsigned)
<code>xorq S, D</code>	<code>D = D ^ S</code>	
<code>andq S, D</code>	<code>D = D & S</code>	
<code>orq S, D</code>	<code>D = D S</code>	

Para saber mais acesse:

<https://www.felixcloutier.com/x86/>

Operações aritméticas simples

- Instrução determina signed vs unsigned
- **mul reg** – multiplicação sem sinal de **reg** por %RAX
resultado armazenado em %RDX:%RAX
- **imul reg** – multiplicação com sinal de **reg** por %RAX
resultado armazenado em %RDX:%RAX
- Vale para divisão também!

Operações aritméticas simples

- Instruções de um operando operandos:

<i>Instrução</i>	<i>Cálculo</i>	
incq D	$D = D + 1$	# Incremento.
decq D	$D = D - 1$	# Decremento.
negq D	$D = -D$	# Negativo.
notq D	$D = \sim D$	# Operador “not” bit-a-bit.

- Ver livro para mais instruções da bibliografia básica para saber mais.

Para referência completa:

<https://software.intel.com/en-us/articles/intel-sdm>

(somente 4684 páginas!)

Aqui tem um resumo que ajuda também:

<https://web.stanford.edu/class/cs107/guide/x86-64.html>

Atividade prática

Funções: argumentos, retorno e chamada

1. Identificar os tipos de argumentos recebidos por uma função
2. Identificar o tipo do valor de retorno de uma função
3. Identificar quais argumentos são passados ao realizar a chamada de uma função.

Insper

www.insper.edu.br