

Sistemas Hardware-Software

Aula 6 – Programação em nível de máquina (III)

2019 – Engenharia

Igor Montagner, Fábio Ayres igorsm1@insper.edu.br

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**

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!

Operações aritméticas simples

- Instruções de dois operandos:

Instrução Cálculo

addq	S, D	D = D + S	
subq	S, D	D = D - S	
imulq	S, D	D = D * S	
salq	S, D	D = D << S	# Tanto arit. como lógico.
sarq	S, D	D = D >> S	# Aritmético.
shrq	S, D	D = D >> S	# Lógico.
xorq	S, D	D = D ^ S	
andq	S, D	D = D & S	
orq	S, D	D = D S	

Não há distinção entre signed e unsigned. (Porque?)

Operações aritméticas simples

- Instruções de um operando operandos:

Instrução Cálculo

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

Para referência completa:

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

(somente 4684 páginas!)

Estado do processador

■ Informação sobre o programa sendo executado:

- Dados temporários (%rax, ...)
- Topo da pilha (%rsp)
- Posição da instrução atual (%rip, ...)
- **Flags de estado dos testes recentes (CF, ZF, SF, OF)**

Registradores

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

%rip Instruction pointer

CF

ZF

SF

OF

Códigos de condição

Códigos de condição

São como registradores de um bit só, que são preenchidos de acordo com o status de uma operação realizada.

<i>Sigla</i>	<i>Nome</i>	<i>Significado</i>
CF	Carry Flag	unsigned overflow
SF	Sign Flag	resultado negativo
OF	Overflow	Flag overflow complemento-de-2
ZF	Zero Flag	resultado zero

Códigos de condição

Os códigos de condição são “efeitos colaterais” de operações aritméticas.

Considere a instrução **add S, D**, que calcula $T = S + D$ e armazena o resultado **T** de volta em **D**:

Flag set?	Significado
CF	$S + D$ deu carry-out. Equivale a overflow de unsigned.
ZF	$T == 0$
SF	$T < 0$ (interpretando T como signed, claro).
OF	$S + D$ deu overflow de complemento-de-2, ou seja, $(S > 0 \ \&\& \ D > 0 \ \&\& \ T < 0) \ \ (S < 0 \ \&\& \ D < 0 \ \&\& \ T \geq 0)$

Nota: a instrução **lea** não gera códigos de condição.

Instruções de comparação

Permitem preencher os códigos de condição sem modificar os registradores:

- Instrução **cmp A, B**
 - Compara valores A e B
 - Funciona como **sub A, B** sem gravar resultado no destino

Flag set?	Significado
CF	Carry-out em $B - A$
ZF	$B == A$
SF	$(B - A) < 0$ (quando interpretado como signed)
OF	Overflow de complemento-de-2: $(A > 0 \ \&\& \ B < 0 \ \&\& \ (B - A) < 0) \ $ $(A < 0 \ \&\& \ B > 0 \ \&\& \ (B - A) > 0)$

Instruções de comparação

- Instrução **test A, B**
 - Testa o resultado de **A & B**
 - Funciona como **and A, B** sem gravar resultado no destino
 - Útil para checar um dos valores, usando o outro como máscara
 - Normalmente usado com A e B sendo o mesmo registrador, ou seja: **test %rdi, %rdi**

Flag set?	Significado
ZF	$A \& B == 0$
SF	$A \& B < 0$ (quando interpretado como signed)

Acessando os códigos de condição

Instruções **set**

- Preenchem o byte mais baixo do destino com 0x00 ou 0x01, dependendo de combinações de códigos de condição
- Não alteram os 7 bytes restantes

Acessando os códigos de condição

Instrução	Condição	Descrição
sete	ZF	E qual /Zero
setne	~ZF	N ot E qual / Not Zero
sets	SF	(signed) N egativo
setns	~SF	(signed) N ão-negativo
setl	(SF^OF)	(signed) L ess than
setle	(SF^OF) ZF	(signed) L ess than or E qual
setge	~(SF^OF)	(signed) G reater than or E qual
setg	~(SF^OF) & ~ZF	(signed) G reater than
setb	CF	(unsigned) B elow
seta	~CF & ~ZF	(unsigned) A bove

Atividade prática

Faremos a parte 1 do handout de hoje.

Duração: 30 minutos

Desvios (ou saltos) condicionais

Permitem saltar para outra parte do código dependendo dos códigos de condição. **Finalmente vamos ter `if` !!!**

Equivalem ao código C:

```
if (condição) {  
    goto label;  
}
```

Exemplo:

```
cmp    $0xa,%rdi    # Compara %rdi:10  
jge     400573        # Se >, pula para 400573
```

Desvios (ou saltos) condicionais

Instrução	Condição	Descrição
jmp	1	Incondicional
je	ZF	E qual /Zero
jne	~ZF	N ot E qual / Not Zero
js	SF	(signed) N egativo
jns	~SF	(signed) N ão-negativo
j1	(SF^OF)	(signed) L ess than
jle	(SF^OF) ZF	(signed) L ess than or E qual
jge	~(SF^OF)	(signed) G reater than or E qual
jg	~(SF^OF) & ~ZF	(signed) G reater than
jb	CF	(unsigned) B elow
ja	~CF & ~ZF	(unsigned) A bove

O comando **goto**

Definimos um *label* usando a sintaxe nome :

goto desvia o fluxo para a linha de código abaixo do label

```
int main(int argc, char **argv) {  
    goto pula_para_ca;  
    printf("Este printf não aparece!\n");  
pula_para_ca:  
    printf("Print2!\n");  
}
```

goto só funciona dentro de uma mesma função

O par de comandos **if-goto**

O par de comandos if-goto é equivalente às instruções
cmp/test seguidas de um jump condicional

```
cmp 0x4, %rdi
jle label
(bloco 1)
label:
...
```

```
if (a <= 4) {
    goto label;
}
(bloco1)
label:
. . .
```

O par de comandos **if-goto**

O par de comandos if-goto é equivalente às instruções
cmp/test seguidas de um jump condicional

```
cmp 0x4, %rdi
jle label
(bloco 1)
label:
...
```

```
if (a <= 4) {
    goto label;
}
(bloco1)
label:
. . .
```

Vamos chamar código **C** que use somente if-goto de **gotoC**!

Padrões de geração de código

Compiladores transformam o código **C** de diversas maneiras durante geração de código.

C

```
if (cond) {  
    (bloco1)  
}  
. . .
```

gotoC

```
if (!cond)  
    goto depois;  
  
(bloco1)  
  
depois:  
. . .
```

Padrões de geração de código

Compiladores transformam o código **C** de diversas maneiras durante geração de código.

C

```
if (cond) {  
    (bloco1)  
} else {  
    (bloco2)  
}  
. . .
```

gotoC

```
if (!cond)  
    goto else;  
  
(bloco1)  
goto fim;  
  
else:  
(bloco2)  
  
fim:  
. . .
```

Padrões de geração de código

Compiladores transformam o código **C** de diversas maneiras durante geração de código.

C

```
if (cond) {  
    (bloco1)  
} else {  
    (bloco2)  
}  
. . .
```

gotoC

```
if (!cond)  
    goto else;  
  
(bloco1)  
goto fim;  
  
else:  
(bloco2)  
  
fim:  
. . .
```

Código C com goto

Para entender o código assembly, devemos traduzir código C normal em código C com **goto**

```
long foo(long x, long y) {  
    long result;  
    if (x > y) {  
        result = x - y;  
    }  
    else {  
        result = y - x;  
    }  
    return result + 1;  
}
```

```
long foo_j(long x, long y) {  
    long result;  
  
    int ntest = x <= y;  
    if (ntest) goto Else;  
    result = x - y;  
    goto Done;  
  
Else:  
    result = y - x;  
  
Done:  
    result = result + 1;  
    return result;  
}
```

Código C com goto

```
long foo_j(long x, long y) {  
    long result;  
  
    int ntest = x <= y;  
    if (ntest) goto Else;  
  
    result = x - y;  
    goto Done;  
  
Else:  
    result = y - x;  
  
Done:  
    result = result + 1;  
    return result;  
}
```

00000000000000000000 <foo>:

0:	48 39 f7	cmp	%rsi,%rdi
3:	7e 08	jle	d <foo+0xd>
5:	48 29 f7	sub	%rsi,%rdi
8:	48 89 fe	mov	%rdi,%rsi
b:	eb 03	jmp	10 <foo+0x10>
d:	48 29 fe	sub	%rdi,%rsi
10:	48 8d 46 01	lea	0x1(%rsi),%rax
14:	c3	retq	

Atividade prática

Faremos a parte 2 do handout de hoje.

Duração: 30 minutos

Insper

www.insper.edu.br