

Sistemas Hardware-Software

Aula 05 – Condicionais

Engenharia

Fabio Lubacheski

Maciel C. Vidal

Igor Montagner

Fábio Ayres



Aula passada !

Questão referente a aula passada !

Qual a seguinte instrução na arquitetura x86-64 está correta para calcular: $\text{\%rax} = 9 * \text{\%rdi}$?

- A. `leaq (, %rdi, 9), %rax`
- B. `movq (, %rdi, 9), %rax`
- C. `leaq (%rdi, %rdi, 8), %rax`
- D. `movq (%rdi, %rdi, 8), %rax`
- E. Não faço ideia ...

Funções e seus argumentos

Argumentos inteiros ou ponteiros são passados nos registradores (**nesta ordem**):

1. %rdi

2. %rsi

3. %rdx

4. %rcx

5. %r8

6. %r9

Valores de retorno inteiros ou ponteiros são colocados no registrador **%rax**.



Aula de hoje !

Instruções de comparação **condicional**

Instruções de comparação realizam uma **operação aritmética** com o propósito de **guiar a execução** condicional de um programa.

Instrução	Significado
cmp A, B	Compara B com A , ou seja, calcula B - A
test A, B	Calcula A & B , ou seja, executa AND bit a bit

cmp e **test** **não modificam** o registrador de destino.

Em vez disso, ambas as instruções modificam uma série de valores **de um bit** conhecidos como **sinalizadores de código de condição** (**FLAG de estado**).

Instruções de comparação: **cmp**

- Instrução **cmp A, B**
 - Compara valores **B** com **A**, funciona como **B-A** sem gravar resultado no destino

```
mov $5, %eax # A
```

```
mov $3, %edx # B
```

```
cmp %eax, %edx # (B-A) => 3 - 5 => -2 < 0, ou seja, B < A
```

Flag setado	Significado
CF=1 (Carry Flag)	Se $B < A$, resultado negativo $(B - A) < 0$
ZF=1 (Zero Flag)	Se $B == A$, resultado $(B - A) == 0$

Não vamos realizar uma discussão aprofundada sobre **flags de estado**, nessa aula, maiores informação acessem o livro referência:

Computer Systems: A Programmer's Perspective (capítulo 3)

Instruções de comparação: **test**

- 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**

```
add $1, %edx # só pra zerar SF  
mov $0,%rdi  
test %rdi,%rdi
```

Flag Setado	Significado
ZF=1	Se $A \& B == 0$, se o número é igual a zero
SF=1	Se $A \& B < 0$, se o número é negativo

Estado do processador e Flags de estado

Informação sobre o programa sendo executado:

- Dados temporários (%rax, ...)
 - Topo da pilha (%rsp)
 - Posição da instrução atual (%rip, ...)
 - resultado testes recentes (**CF, ZF, SF, OF**)
- Registrador de 1 bit**

Registradores

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

%rip Instruction pointer



Flags de estado

Usando FLAGS - Instrução **set***

Preenchem o **byte mais baixo** do destino com **0x00** ou **0x01**, dependendo de combinações dos FLAGS de estado. Não alteram os 7 bytes restantes

Instrução	Descrição
sete	Equal /Zero
setne	Not Equal / Not Zero
sets	(signed) Negativo
setns	(signed) Não-negativo
setl	(signed) Less than
setle	(signed) Less than or Equal
setge	(signed) Greater than or Equal
setg	(signed) Greater than
setb	(unsigned) Below
seta	(unsigned) Above

Exemplos

```
int func(int a ) {  
    return a == 5 ;  
}
```

func:

```
    cmpl    $5, %edi  
    sete    %al  
    movzbl  %al, %eax  
    ret
```

```
int func(int a ) {  
    return a == 0 ;  
}
```

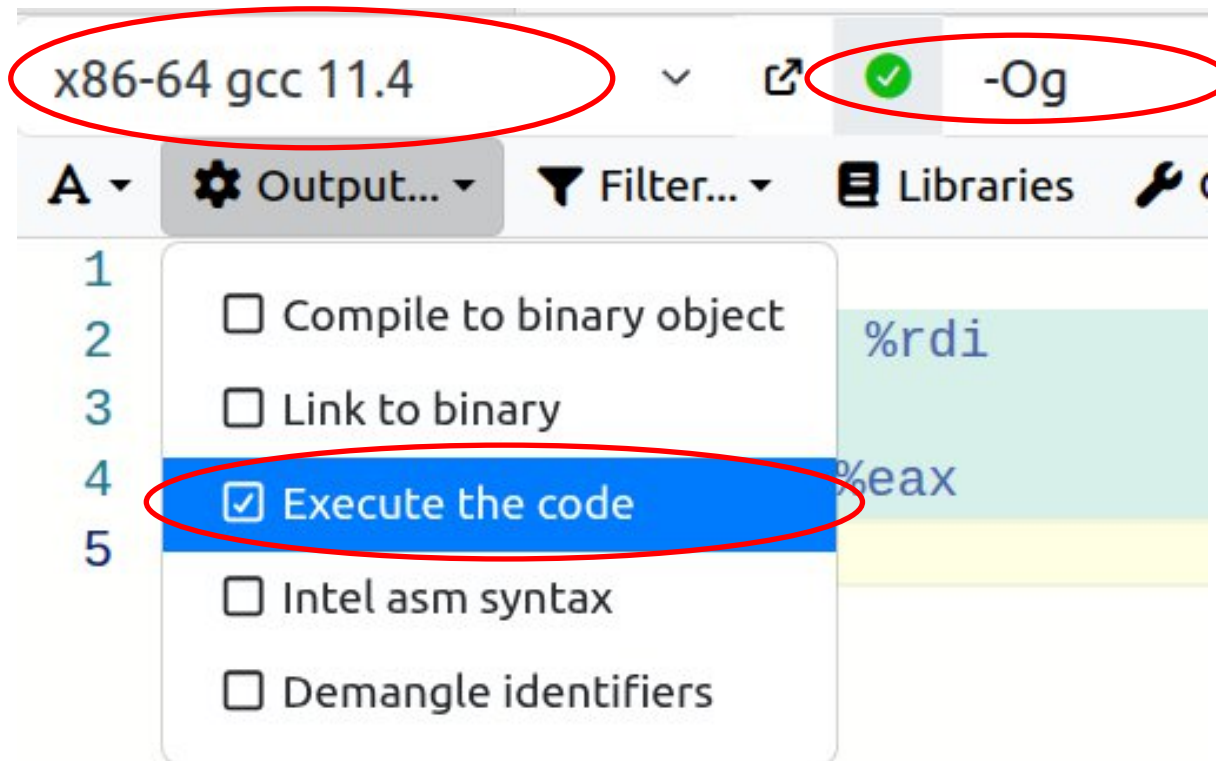
func:

```
    testl   %edi, %edi  
    sete    %al  
    movzbl  %al, %eax  
    ret
```

Tradução de função **assembly** => **C**

Para ajudar na **tradução reversa de programas em Assembly para C**, podemos usar a ferramenta **Compiler Explorer**.

Para acessar Compiler Explorer: <https://godbolt.org/> e configure conforme abaixo:



Atividade prática

Expressões booleanas

1. Identificar expressões booleanas a partir de código assembly
2. Reconstruir expressões booleanas em C a partir de sequências de instruções **cmp/test** e **set** (até exercício 6)

Desvios (ou saltos) condicionais

Instrução	Descrição
jmp	Incondicional
je	Equal /Zero
jne	Not Equal / Not Zero
js	(signed) Negativo
jns	(signed) Não-negativo
jl	(signed) Less than
jle	(signed) Less than or Equal
jge	(signed) Greater than or Equal
jg	(signed) Greater than
jb	(unsigned) Below
ja	(unsigned) Above

Resumo de instruções para condição

		cmp a,b	test a,b
je	"Equal"	$b == a$	$b \& a == 0$
jne	"Not equal"	$b != a$	$b \& a != 0$
js	"Sign" (negative)	$b - a < 0$	$b \& a < 0$
jns	(non-negative)	$b - a \geq 0$	$b \& a \geq 0$
jg	"Greater"	$b > a$	$b \& a > 0$
jge	"Greater or equal"	$b \geq a$	$b \& a \geq 0$
jl	"Less"	$b < a$	$b \& a < 0$
jle	"Less or equal"	$b \leq a$	$b \& a \leq 0$
ja	"Above" (unsigned >)	$b > a$	$b \& a > 0U$
jb	"Below" (unsigned <)	$b < a$	$b \& a < 0U$

cmp 5,b

je: $b == 5$

jne: $b != 5$

jg: $b > 5$

jl: $b < 5$

test a, a

je: $a == 0$

jne: $a != 0$

jg: $a > 0$

jl: $a < 0$

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 (x < 3) {  
    return 1;  
}  
return 2;
```

```
cmpq $3, %rdi  
jge T2  
T1: # x < 3:  
    movq $1, %rax  
    ret  
T2: # !(x < 3):  
    movq $2, %rax  
    ret
```

Como fazer saltos na linguagem C

Definimos um *label* (rótulo) 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**

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

Código **Assembly**

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

Código **gotoC**

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

Código na **linguagem C**

```
if (a > 4) {
    (bloco1);
}
. . .
```

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:  
. . .
```

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;  
}
```

000000000000000000 <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

Condicionais: if e if/else

1. Identificar as expressões booleanas testadas em instruções de pulo condicional
2. Reconstruir o fluxo de controle de um programa em C a partir de sua versão compilada

Insper

www.insper.edu.br