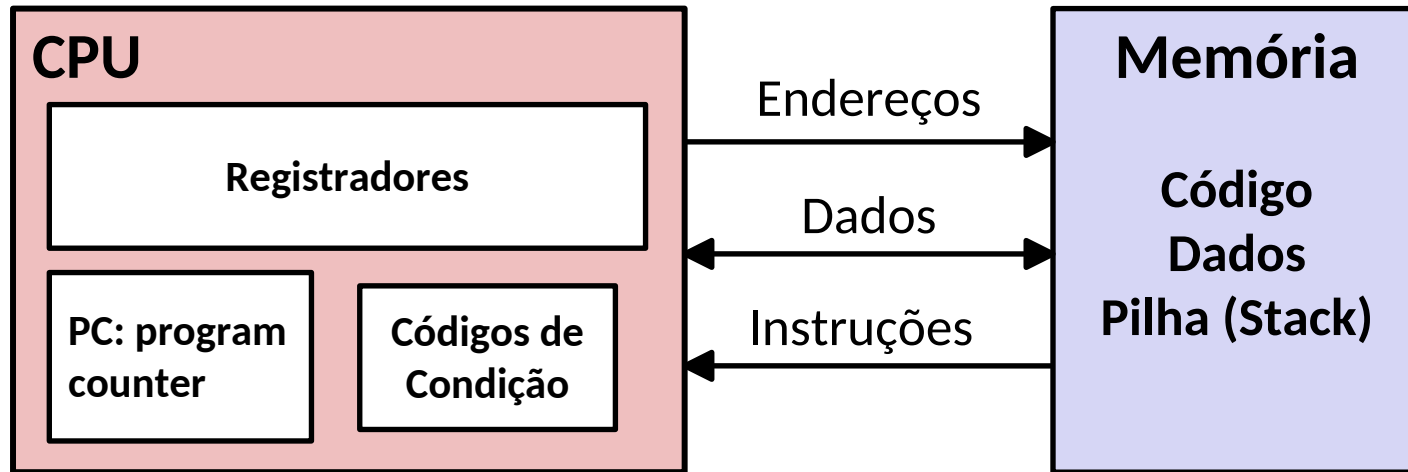


# Sistemas Hardware-Software

Aula 04 – Funções

Maciel C. Vidal  
Igor Montagner

# A visão do programador



## PC: Program counter

%rip: Endereço da próxima instrução

## Registradores

Dados de uso muito frequente

## Códigos de condição

Informação sobre o resultado das operações aritméticas ou lógicas mais recentes

## Memória

Um vetor de bytes

Armazena código e dados

Armazena estado atual do programa (pilha)

# Registradores inteiros 64/32 bits

<b>%rax</b>	<b>%eax</b>
<b>%rbx</b>	<b>%ebx</b>
<b>%rcx</b>	<b>%ecx</b>
<b>%rdx</b>	<b>%edx</b>
<b>%rsi</b>	<b>%esi</b>
<b>%rdi</b>	<b>%edi</b>
<b>%rsp</b>	<b>%esp</b>
<b>%rbp</b>	<b>%ebp</b>

<b>%r8</b>	<b>%r8d</b>
<b>%r9</b>	<b>%r9d</b>
<b>%r10</b>	<b>%r10d</b>
<b>%r11</b>	<b>%r11d</b>
<b>%r12</b>	<b>%r12d</b>
<b>%r13</b>	<b>%r13d</b>
<b>%r14</b>	<b>%r14d</b>
<b>%r15</b>	<b>%r15d</b>

- Podem se referir aos 8 bytes (%rax), 4 bytes mais baixos (%eax), 2 bytes mais baixos (%ax), byte mais baixo (%al) e segundo byte mais baixo (%ah)

# Registradores inteiros 32/16/8 bits

Significado original (obsoleto)

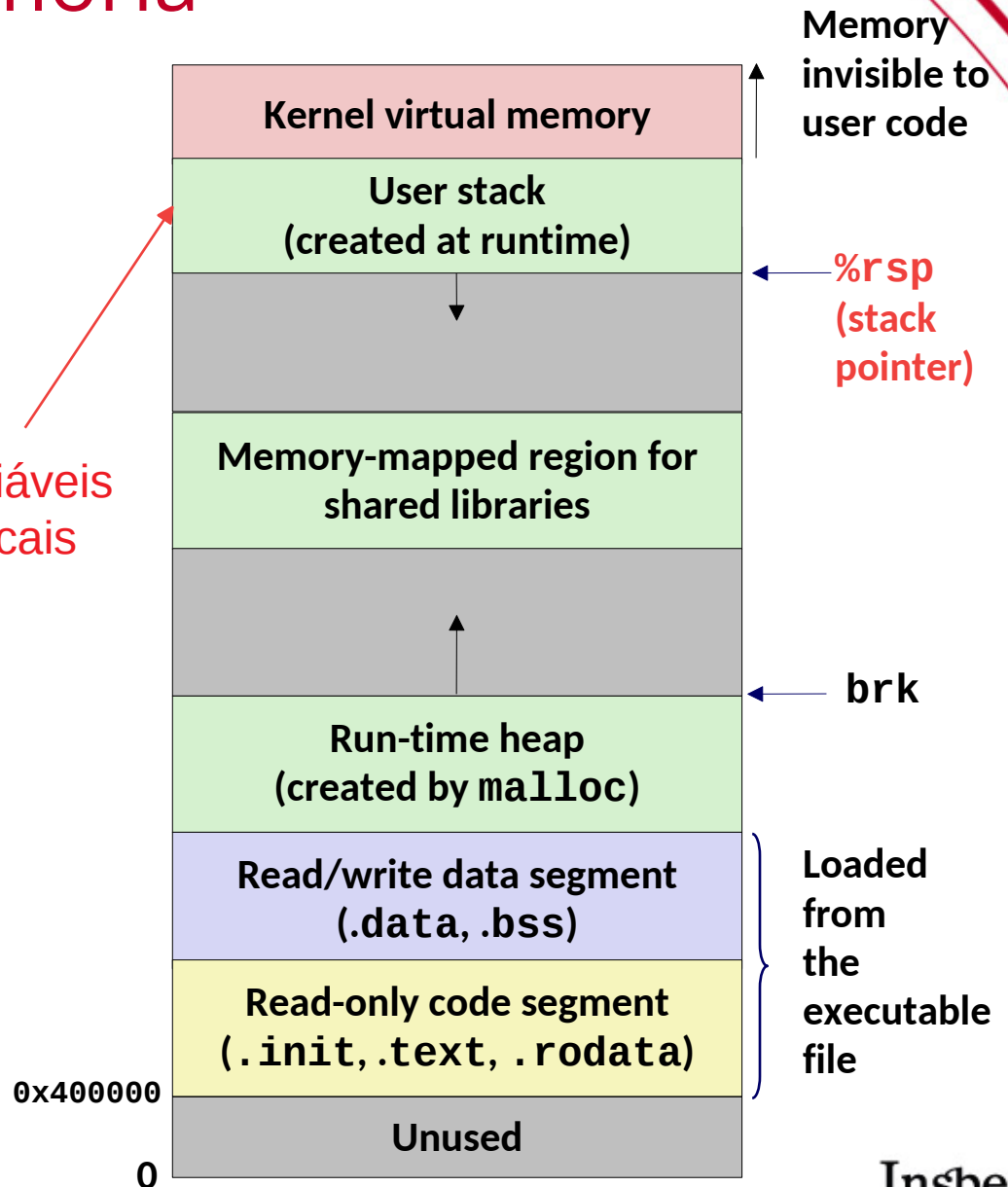


# Executável na memória

## Executable Object File

ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)

Variáveis  
locais



# Movendo Dados

**movq Source, Dest**

Tipos de operandos:

- **Imediato (Immediate):** Constantes inteiras
  - Exemplo: **\$0x400**, **\$-533**
  - Não esqueça do prefixo '\$'
  - Codificado com 1, 2, ou 4 bytes
- **Registrador:** Um dos 16 registradores inteiros
  - Exemplo: **%rax**, **%r13**
- **Memória:** 8 bytes (por causa do sufixo 'q') consecutivos de memória, no endereço dado pelo registrador
  - Exemplo mais simples: **(%rax)**
  - Vários outros modos de endereçamento

# movq : Combinações de operandos

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	?
		Mem	movq \$-147, (%rax)	?
	Reg	Reg	movq %rax, %rdx	?
		Mem	movq %rax, (%rdx)	?
	Mem	Reg	movq (%rax), %rdx	?

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

# 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*



# Alguns modos simples de endereçamento

Normal (R)      Mem[Reg[R]]

- Registrador R especifica o endereço de memória

**movq (%rcx),%rax**

Deslocamento (Displacement)      D(R)      Mem[Reg[R]+D]

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

**movq 8(%rbp),%rdx**

# 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

# 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!

# Tradução de função: Exemplo

Vamos traduzir a seguinte função:

```
(gdb) disas minhafunc
Dump of assembler code for function minhafunc:
   0x0000000000000129 <+0>:      endbr64
   0x000000000000012d <+4>:      mov     %edi,%eax
   0x000000000000012f <+6>:      add     (%rsi),%eax
   0x0000000000000131 <+8>:      mov     %eax,(%rsi)
   0x0000000000000133 <+10>:     ret
End of assembler dump.
```

# Tradução de função: Exemplo

Vamos traduzir a seguinte função:

```
(gdb) disas minhafunc
Dump of assembler code for function minhafunc:
   0x0000000000000129 <+0>:      endbr64
   0x000000000000012d <+4>:      mov     %edi,%eax
   0x000000000000012f <+6>:      add     (%rsi),%eax
   0x0000000000000131 <+8>:      mov     %eax,(%rsi)
   0x0000000000000133 <+10>:     ret
End of assembler dump.
```

Vamos olhar também a função main:

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000000134 <+0>:      endbr64
   0x0000000000000138 <+4>:      sub     $0x10,%rsp
   0x000000000000013c <+8>:      lea     0xc(%rsp),%rsi
   0x0000000000000141 <+13>:     mov     $0x5,%edi
   0x0000000000000146 <+18>:     call    0x1129 <minhafunc>
   0x000000000000014b <+23>:     mov     0xc(%rsp),%eax
   0x000000000000014f <+27>:     add     $0x10,%rsp
   0x0000000000000153 <+31>:     ret
End of assembler dump.
```

# 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.



# Operações aritméticas simples

- Instruções de dois operandos:

<i>Instrução</i>		<i>Cálculo</i>	
<b>addq</b>	<b>S, D</b>	<b>D = D + S</b>	
<b>subq</b>	<b>S, D</b>	<b>D = D - S</b>	
<b>imulq</b>	<b>S, D</b>	<b>D = D * S</b>	
<b>salq</b>	<b>S, D</b>	<b>D = D &lt;&lt; S</b>	<b># Tanto arit. como lógico.</b>
<b>sarq</b>	<b>S, D</b>	<b>D = D &gt;&gt; S</b>	<b># Aritmético.</b>
<b>shrq</b>	<b>S, D</b>	<b>D = D &gt;&gt; S</b>	<b># Lógico.</b>
<b>xorq</b>	<b>S, D</b>	<b>D = D ^ S</b>	
<b>andq</b>	<b>S, D</b>	<b>D = D &amp; S</b>	
<b>orq</b>	<b>S, D</b>	<b>D = D   S</b>	

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

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

- Ver livro para mais instruções

Para referência completa:

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

(somente 4684 páginas!)

# Insper

[www.insper.edu.br](http://www.insper.edu.br)