

Sistemas Hardware-Software

Aula 04 – Funções

Engenharia

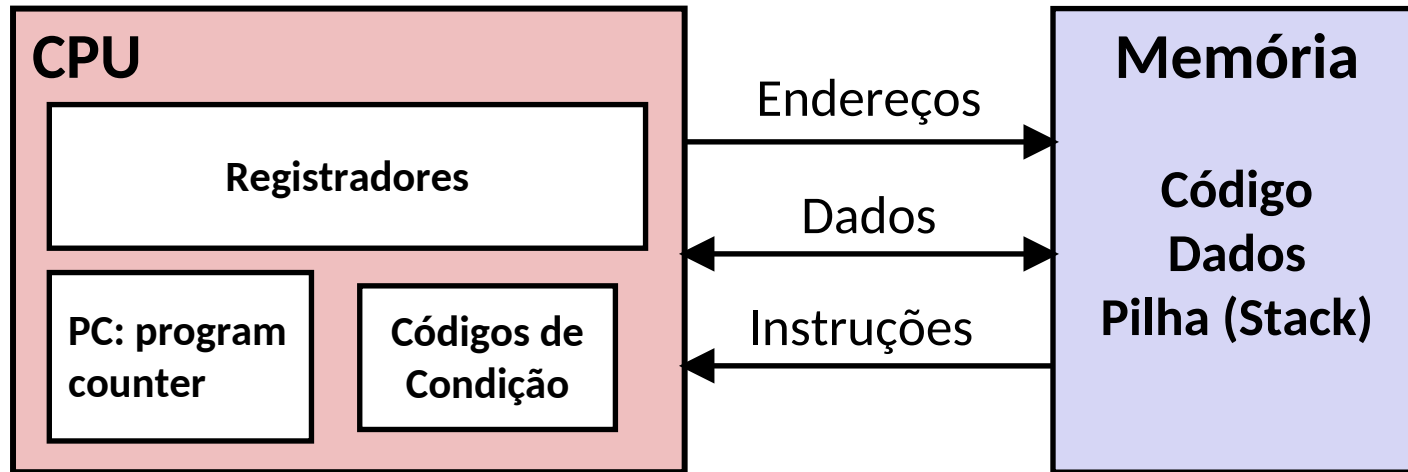
Fabio Lubacheski

Maciel C. Vidal

Igor Montagner

Fábio Ayres

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)

Registadores inteiros 64/32 bits

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

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

- 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

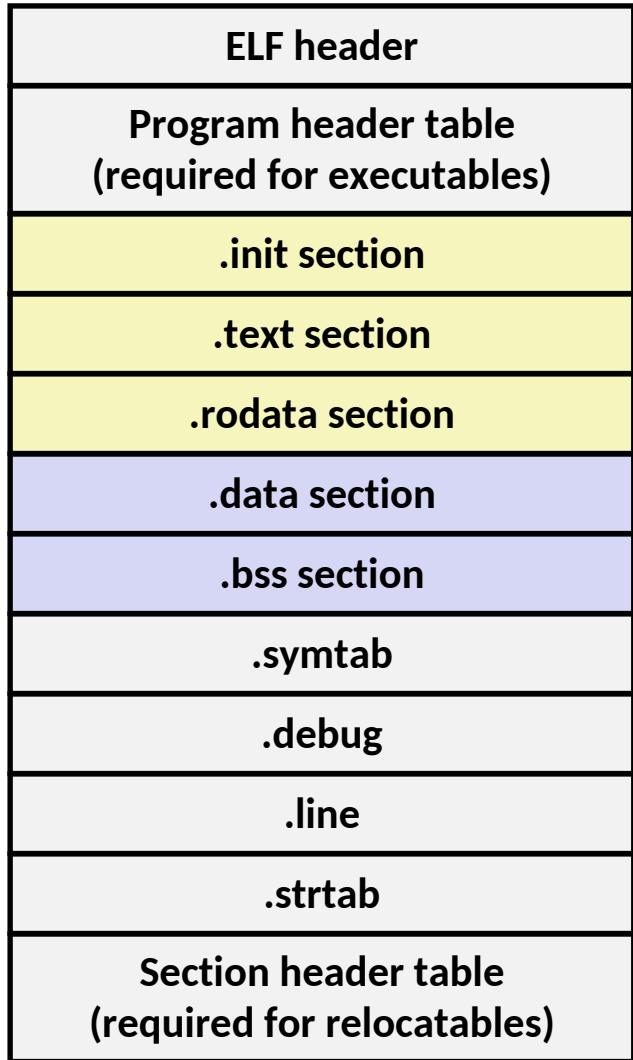
Significado original (obsoleto)

general purpose	%eax	%ax	%ah	%al	<i>accumulate</i>
	%ecx	%cx	%ch	%cl	<i>counter</i>
	%edx	%dx	%dh	%dl	<i>data</i>
	%ebx	%bx	%bh	%bl	<i>base</i>
	%esi	%si			<i>source index</i>
	%edi	%di			<i>destination index</i>
	%esp	%sp			<i>stack pointer</i>
	%ebp	%bp			<i>base pointer</i>

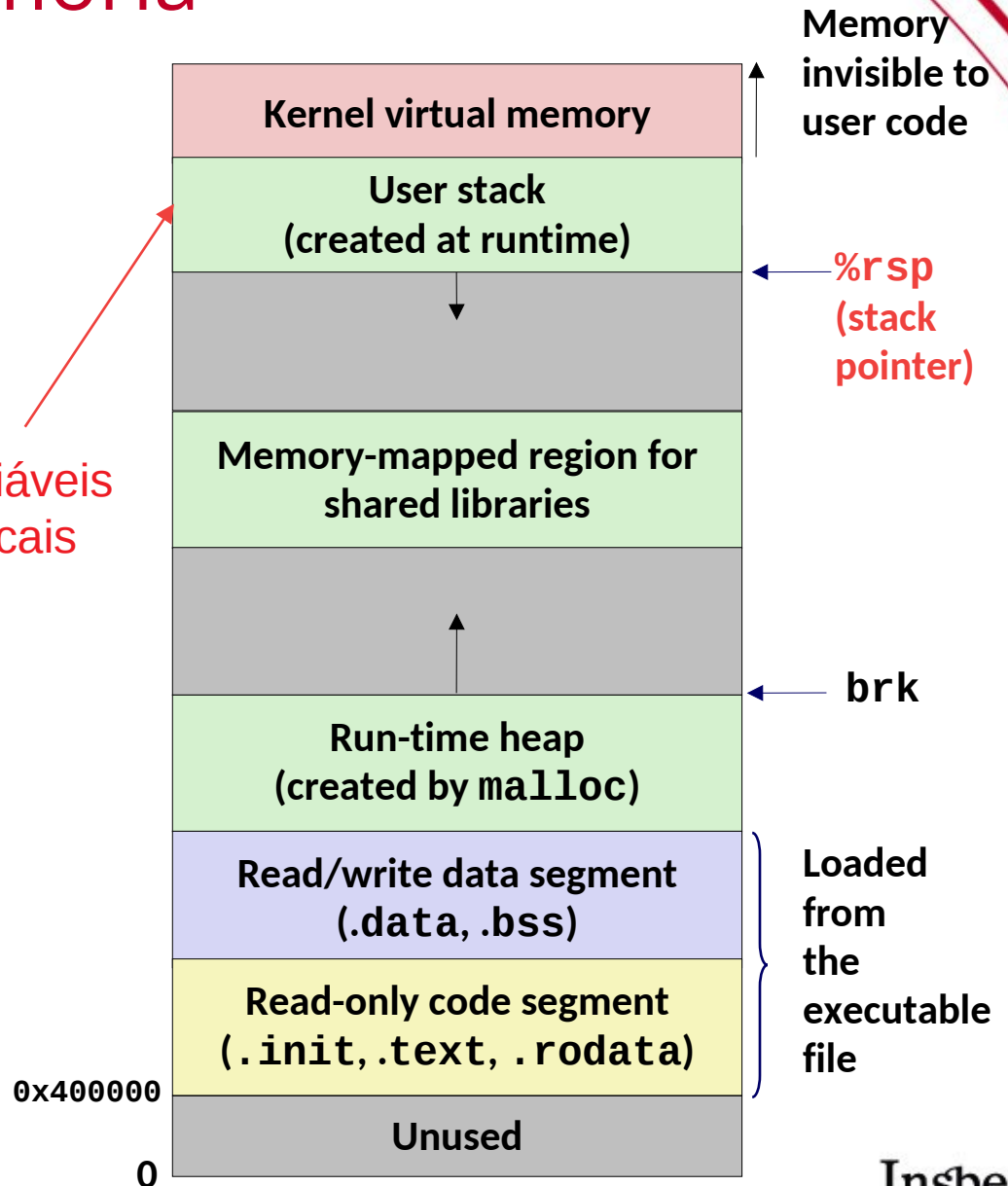
16-bit virtual registers (backwards compatibility)

Executável na memória

Executable Object File



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	<code>movq \$0x4, %rax</code>	<code>temp = 0x4;</code>
		Mem	<code>movq \$-147, (%rax)</code>	<code>*p = -147;</code>
	Reg	Reg	<code>movq %rax, %rdx</code>	<code>temp2 = temp1;</code>
		Mem	<code>movq %rax, (%rdx)</code>	<code>*p = temp;</code>
	Mem	Reg	<code>movq (%rax), %rdx</code>	<code>temp = *p;</code>

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 inicio da região de memória
- Constante de deslocamento D especifica offset

movq 8(%rbp),%rdx

Modo de endereçamento completo

Forma geral: $D(\mathbf{Rb}, \mathbf{Ri}, \mathbf{S})$

Representa o valor $\text{Mem}[\text{Reg}[\mathbf{Rb}] + \mathbf{S} * \text{Reg}[\mathbf{Ri}] + \mathbf{D}]$

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:

```
leq $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 `leaq` 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 assembly => C

Como ficaria a tradução da função **minhafunc** para linguagem C ?

Quais seriam as informações passadas para função como argumento na sua chamada e onde são armazenados ?

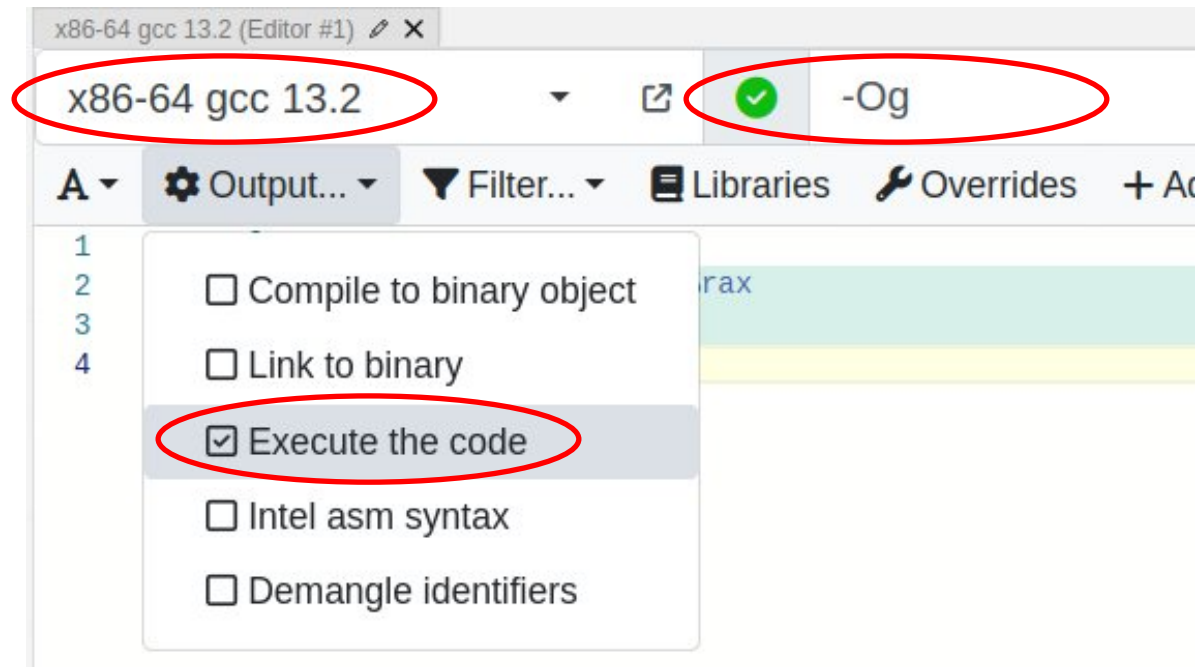
A função utiliza variáveis locais e retorna alguma coisa ?

```
(gdb) disas minhafunc
Dump of assembler code for function minhafunc:
0x00000000000001149 <+0>:    endbr64
0x0000000000000114d <+4>:    mov     %edi,%eax
0x0000000000000114f <+6>:    add    (%rsi),%eax
0x00000000000001151 <+8>:    mov    %eax,(%rsi)
0x00000000000001153 <+10>:   ret
End of _assembler dump.
```

Tradução de função assembly => C

Para ajudar na tradução podemos usar a ferramenta **Compiler Explorer**, um compilador online interativo que mostra como saía um código assembly gerado a partir de código na linguagem C.

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



Tradução de função Assembly => C

O programa completo traduzido de Assembly para linguagem C seria:

```
int minhafunc(int a, int *b) {
    int res;
    res = a + *b;
    *b = res;
    return res;
}
int main() {
    int res;
    minhafunc(5,&res);
    return res;
}
```

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>	
<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 shlq
<code>sarq S, D</code>	<code>D = D >> S</code>	# Aritmético: o sinal é mantido
<code>shrq S, D</code>	<code>D = D >> S</code>	# Lógico: o bit mais a esq é zerado
<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>	

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>

Insper

www.insper.edu.br