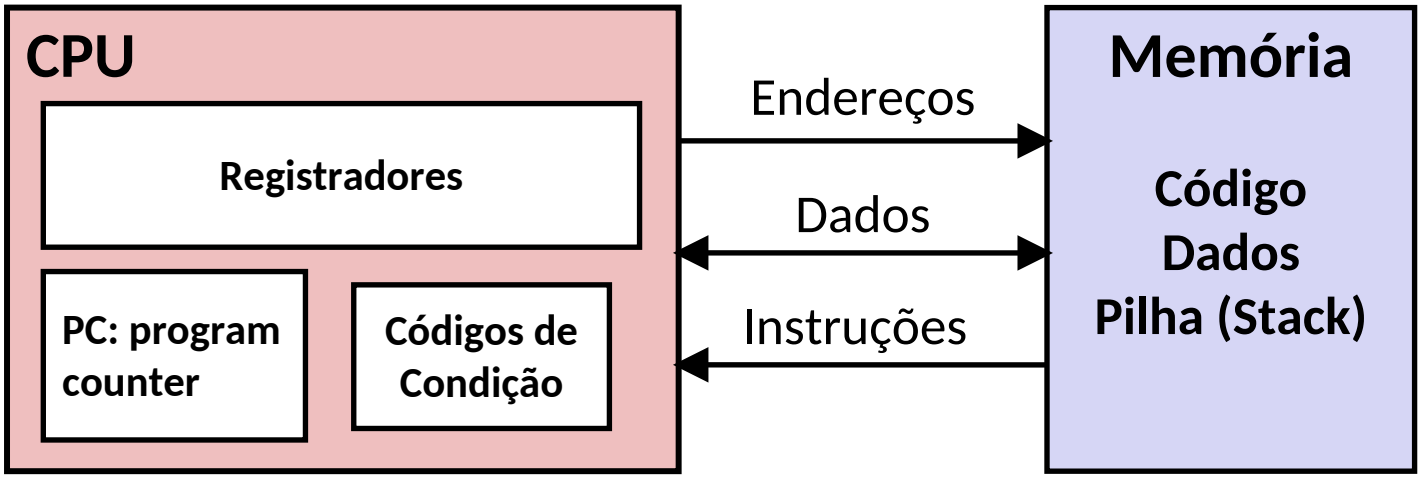
Insper

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
- Usado para saltos condicionais

Memória

- Um vetor de bytes
- Armazena código e dados
- Armazena estado atual do programa (pilha)

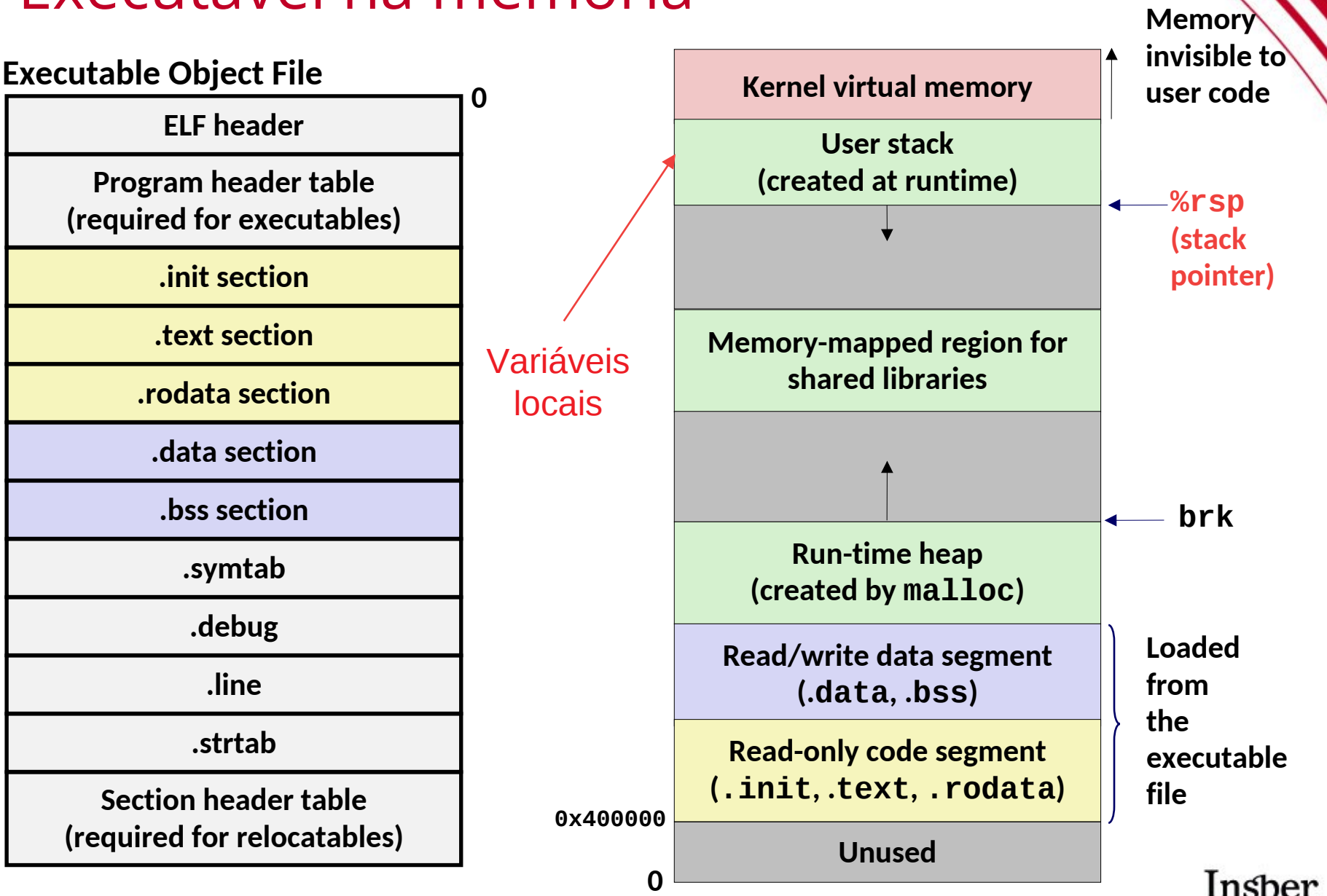
Registradores inteiros 64/32 bits

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%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

Executável na memória



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`	`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 inicio 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

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

Compare com:

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

que resulta em

$$R[\%\text{rsi}] = M\left[4 + R[\%\text{rax}] + 8 \times R[\%\text{rbx}]\right]$$

(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 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 ?

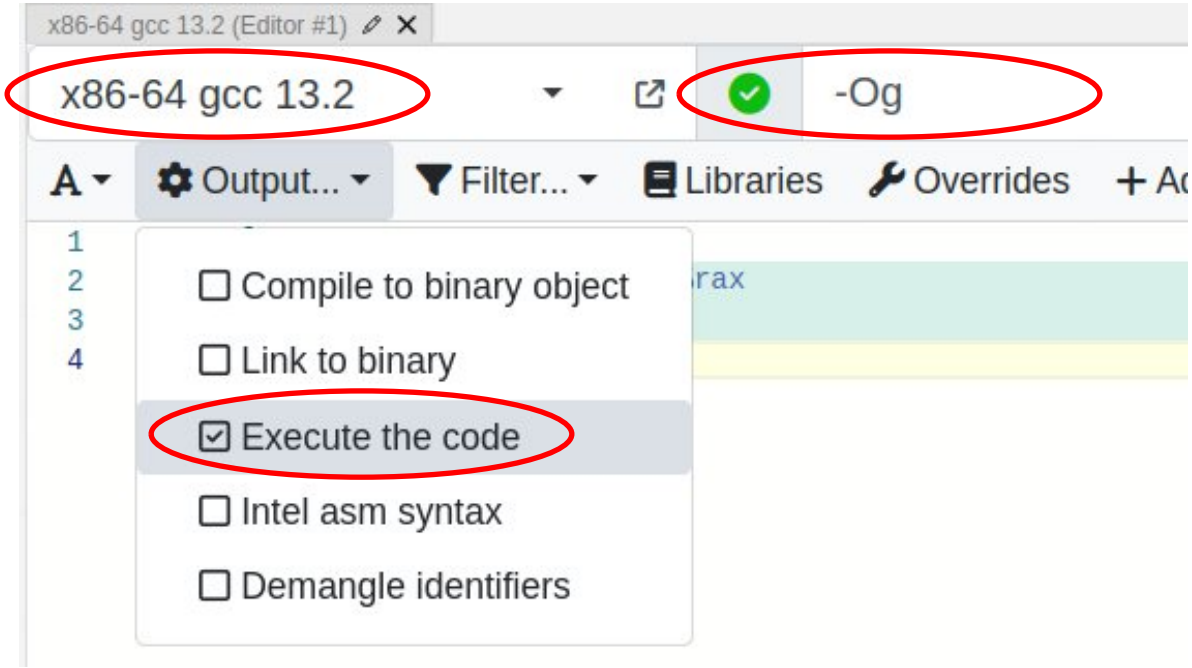
A função utiliza variáveis locais e retorna alguma coisa ?

```
(gdb) disas minhafunc
Dump of assembler code for function minhafunc:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     mov    %edi,%eax
   0x000000000000114f <+6>:     add    (%rsi),%eax
   0x0000000000001151 <+8>:     mov    %eax,(%rsi)
   0x0000000000001153 <+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 saída um código assembly gerado a partir de código na linguagem C.

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



Insper

Tradução de função Assembly => C

O programa completo traduzido de Assembly para linguagem C seria:

```c
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:

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: o sinal é mantido`
`shrq S, D`	`D = D >> S`	`# Lógico: o bit mais a esq é zerado`
`xorq S, D`	`D = D ^ S`	
`andq S, D`	`D = D & S`	
`orq S, D`	`D = D | S`	

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:

Instrução	*Cálculo*	
`incq D`	`D = D + 1`	`# Incremento.`
`decq D`	`D = D – 1`	`# Decremento.`
`negq D`	`D = -D`	`# Negativo.`
`notq D`	`D = ~D`	`# Operador “not” bit-a-bit.`

- Ver livro para mais instruções da bibligrafia 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