

Sistemas Hardware-Software

Aula 4 – Introdução a Assembly x64

2020 – Engenharia

Igor Montagner igorsm1@insper.edu.br

Fábio Ayres

Aula passada

- Representação em memória de vários tipos em C
- Estrutura de um arquivo executável
 - .text guarda nosso código
 - .data guarda globais inicializadas
 - .rodata guarda constantes
 - .bss reserva espaço para globais não inicializadas
- Variáveis locais só existem em tempo de execução

Agenda

- Arquitetura de computadores I
 - Arquitetura x86-64
 - Registradores
 - Instruções de movimentação de dados

Processadores Intel x86

- Dominam o mercado
 - Quase 90% de market share de PCs!
 - AMD Ryzen é bom, mas não compete em volume de vendas
- Estado atual
 - Core i3: entry-level
 - Core i5: mainstream
 - Core i7: high-end
 - Core i9: very high-end
 - Core m: mobile (tablets)
 - Xeon: servidores e estações de trabalho

Processadores Intel x86

- Evoluiu aos poucos
 - Retro-compatível desde o processador 8086, de 1978!
 - Features adicionadas com o tempo
- **Complex-instruction-set computer (CISC)**
 - Intel conseguiu uma proeza: fazer um CISC com desempenho de **RISC (Reduced-instruction-set computer)**
 - Mas nem tanto para a potência dissipada

Evolução dos processadores Intel

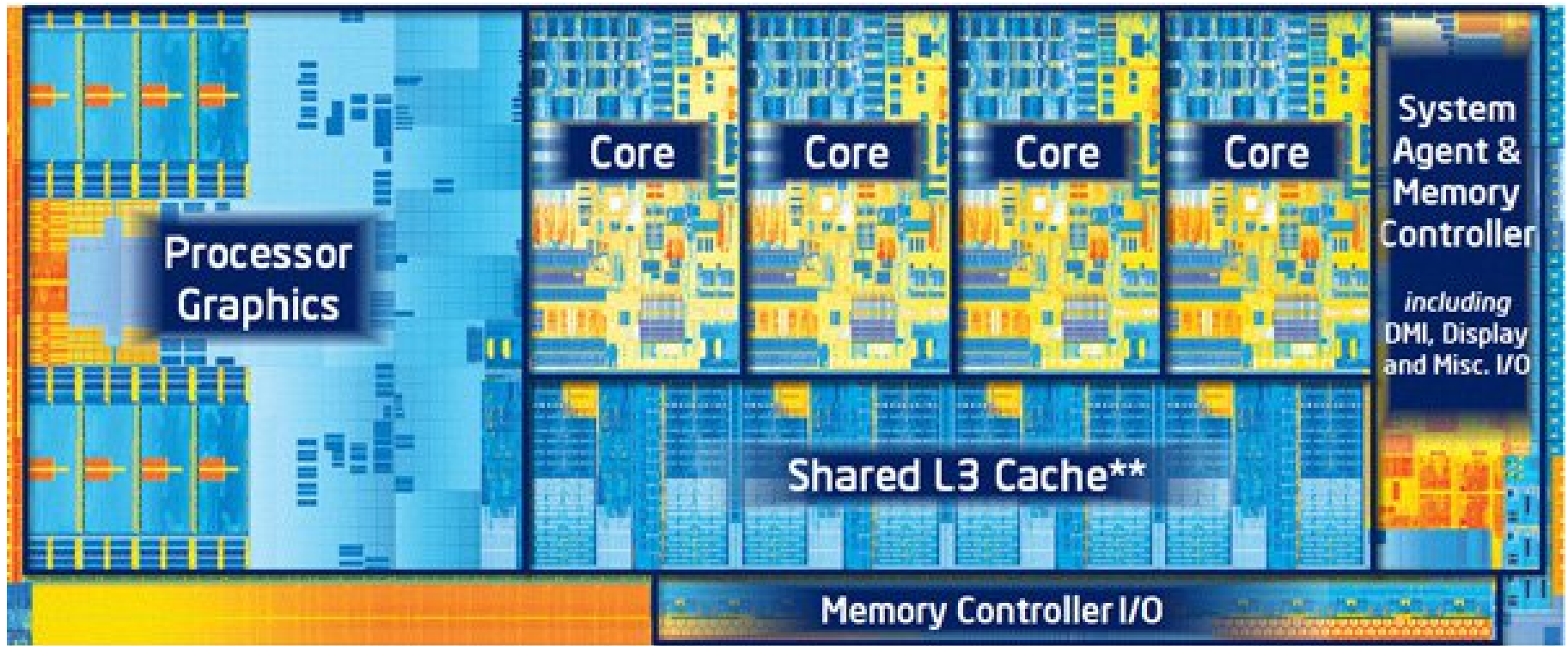
Nome	Data	Transistores	MHz
8086	1978	29K	5-10
<ul style="list-style-type: none">• Primeiro processador Intel 16-bit• Espaço de endereçamento: 1MB			
386	1985	275K	16-33
<ul style="list-style-type: none">• Primeiro processador Intel 32-bit (IA32)• Adicionou modo de endereçamento “flat”, capaz de rodar Unix			
Pentium 4E	2004	125M	2800-3800
<ul style="list-style-type: none">• Primeiro processador Intel 64-bit (x86-64)			
Core 2	2006	291M	1060-3500
<ul style="list-style-type: none">• Primeiro processador Intel multi-core			
Core i7	2008	731M	1700-3900
<ul style="list-style-type: none">• Quad-core			

Evolução dos processadores Intel

Características adicionadas com o tempo:

- Instruções de suporte à operações multimídia
 - MMX – Multimedia Extensions
 - SSE – Streaming SIMD Extensions
 - SSE, SSE2, SSE3, SSE
 - AVX – Advanced Vector Extensions
 - AVX, AVX2, AVX-512 (futuro)
- Instruções mais eficientes de operação condicional
- Transição 32 para 64 bits
- Mais cores

Exemplo: Intel Ivy Bridge (Core i7 3770K)



Agenda

- Arquitetura de computadores I
 - Arquitetura x86-64
 - Registradores
 - Instruções de movimentação de dados

Definições

Arquitetura (também conhecida como **ISA: instruction set architecture**):

- registradores, instruções
- Exemplos de ISAs:
 - Intel: x86, IA32, Itanium, x86-64
 - ARM

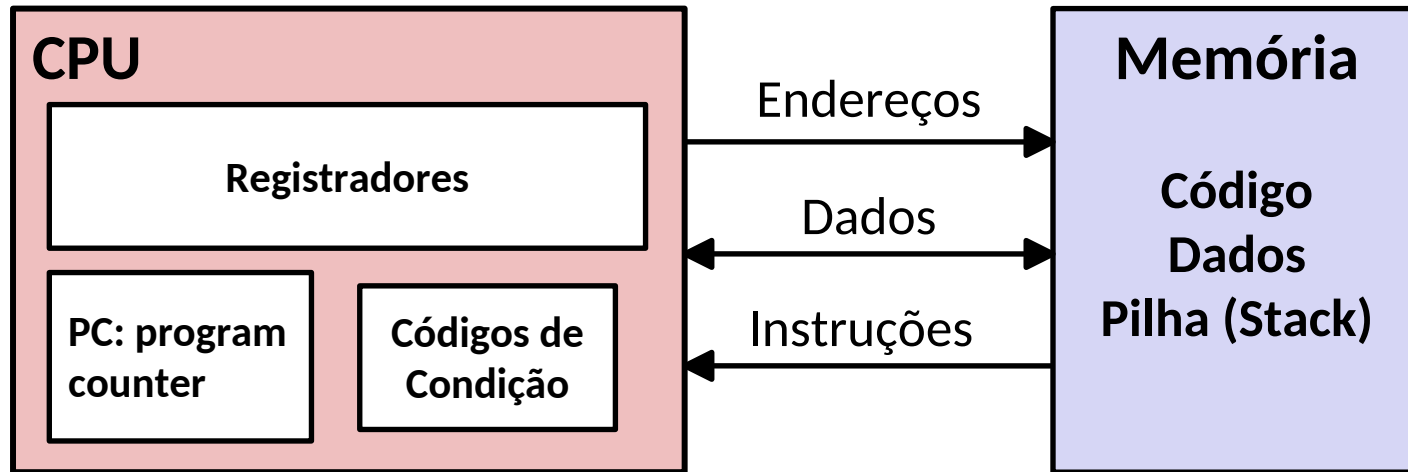
Microarquitetura: Implementação da arquitetura

- Tamanho de cache, número de cores, frequência de clock

Código:

- **Código de máquina:** sequência de bytes que o processador executa
- **Código assembly:** representação textual mais “amigável” do código de máquina

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

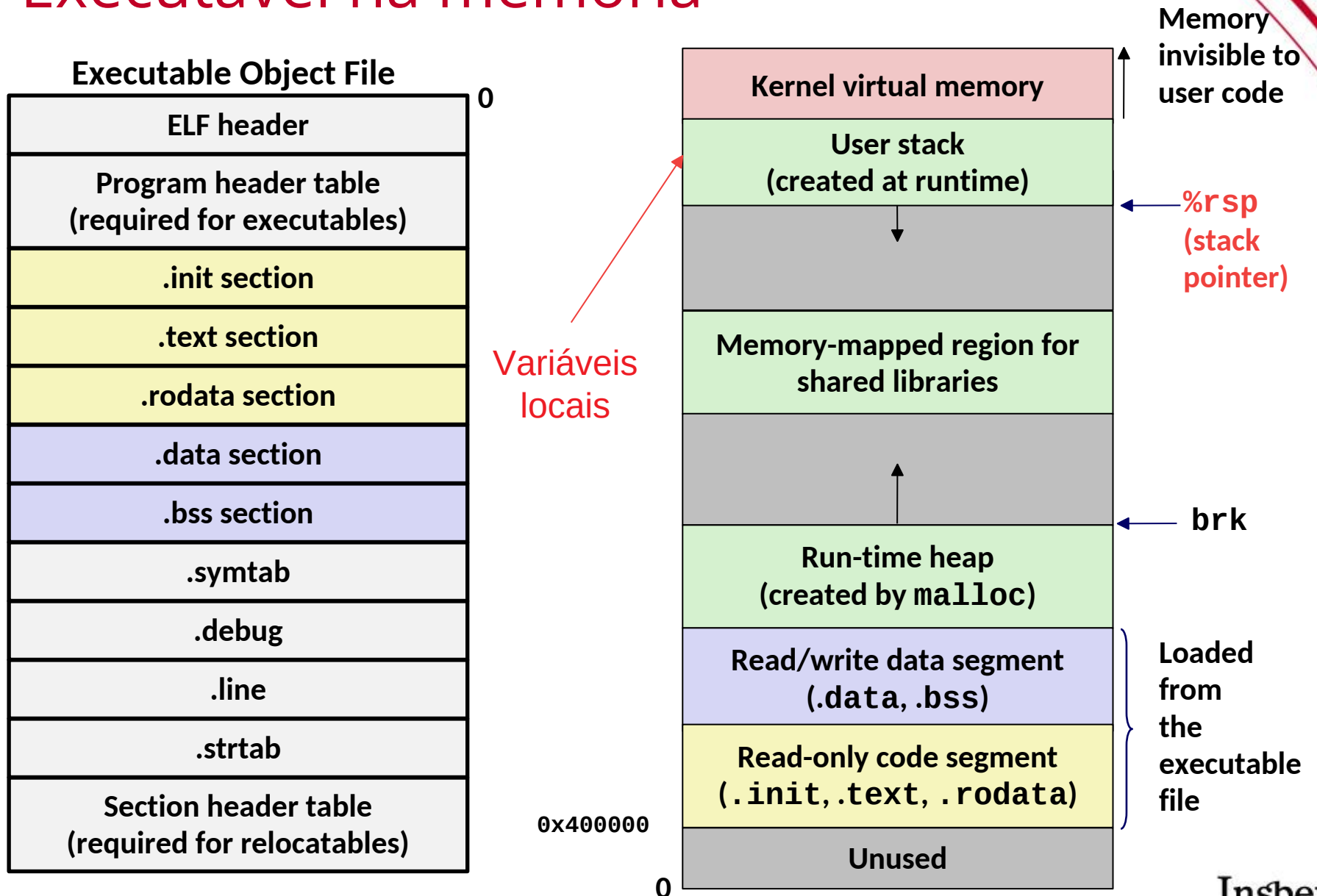
Memória

Um vetor de bytes

Armazena código e dados

Armazena a pilha: essencial para usar funções

Executável na memória



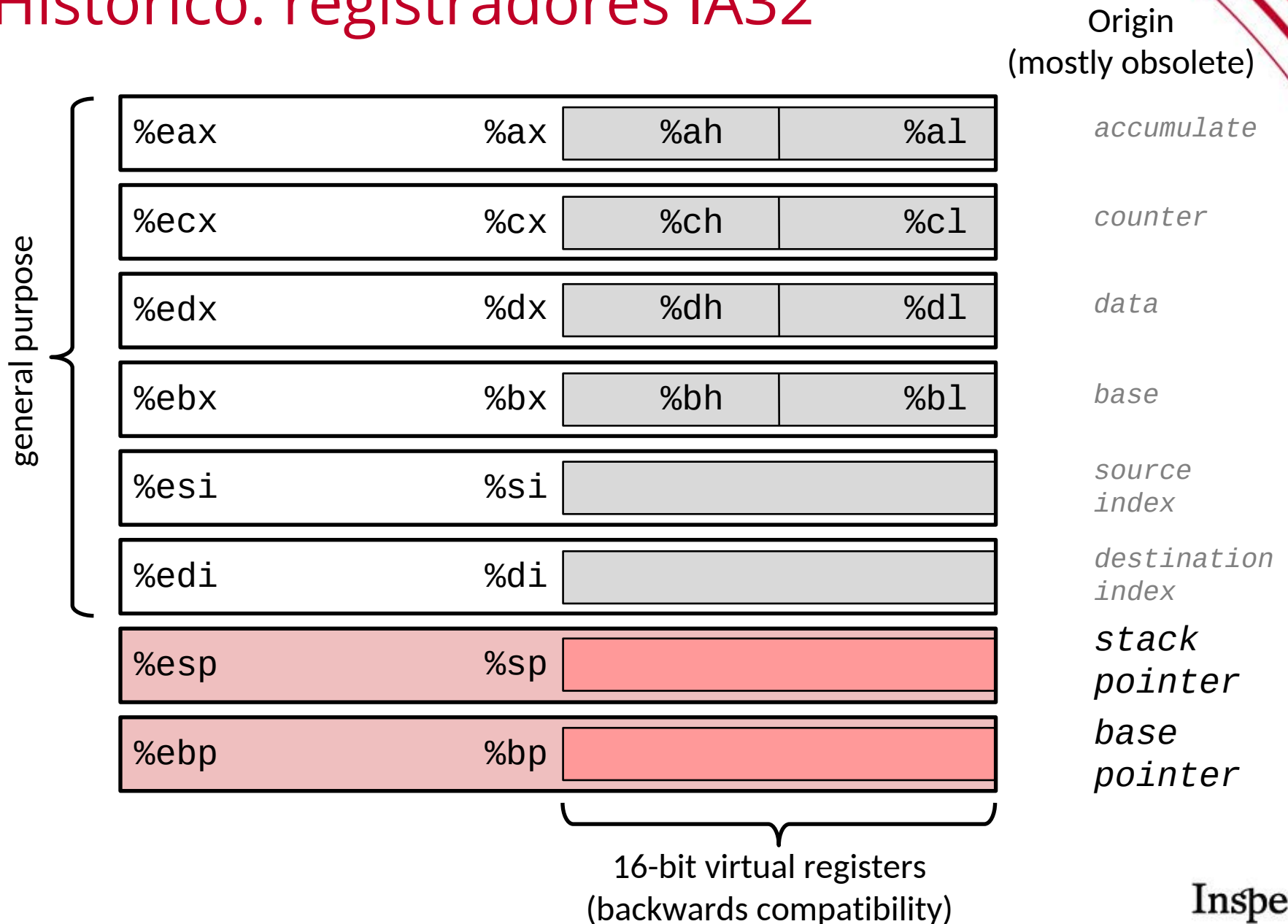
Registradores inteiros x86-64

%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)

Histórico: registradores IA32



Registradores de ponto flutuante

	255	128	0
YMM0		XMM0	
YMM1		XMM1	
YMM2		XMM2	
YMM3		XMM3	
YMM4		XMM4	
YMM5		XMM5	
YMM6		XMM6	
YMM7		XMM7	
YMM8		XMM8	
YMM9		XMM9	
YMM10		XMM10	
YMM11		XMM11	
YMM12		XMM12	
YMM13		XMM13	
YMM14		XMM14	
YMM15		XMM15	

Fonte: https://commons.wikimedia.org/wiki/File:AVX_registers.svg

Agenda

- Arquitetura de computadores I
 - Arquitetura x86-64
 - Registradores
 - Instruções de movimentação de dados

push, pop

Instrução **push S**:

- Cria espaço na pilha
- Move **S** para o espaço recém-criado

Ou seja:

push %rbx

Equivale a

```
sub $8, %rsp  
mov %rbx, (%rsp)
```

Instrução **pop S**:

- Move os dados do topo da pilha para **S**
- Remove espaço da pilha

Ou seja:

pop %rbx

Equivale a

```
mov (%rsp), %rbx  
add $8, %rsp
```

push, pop

Initially

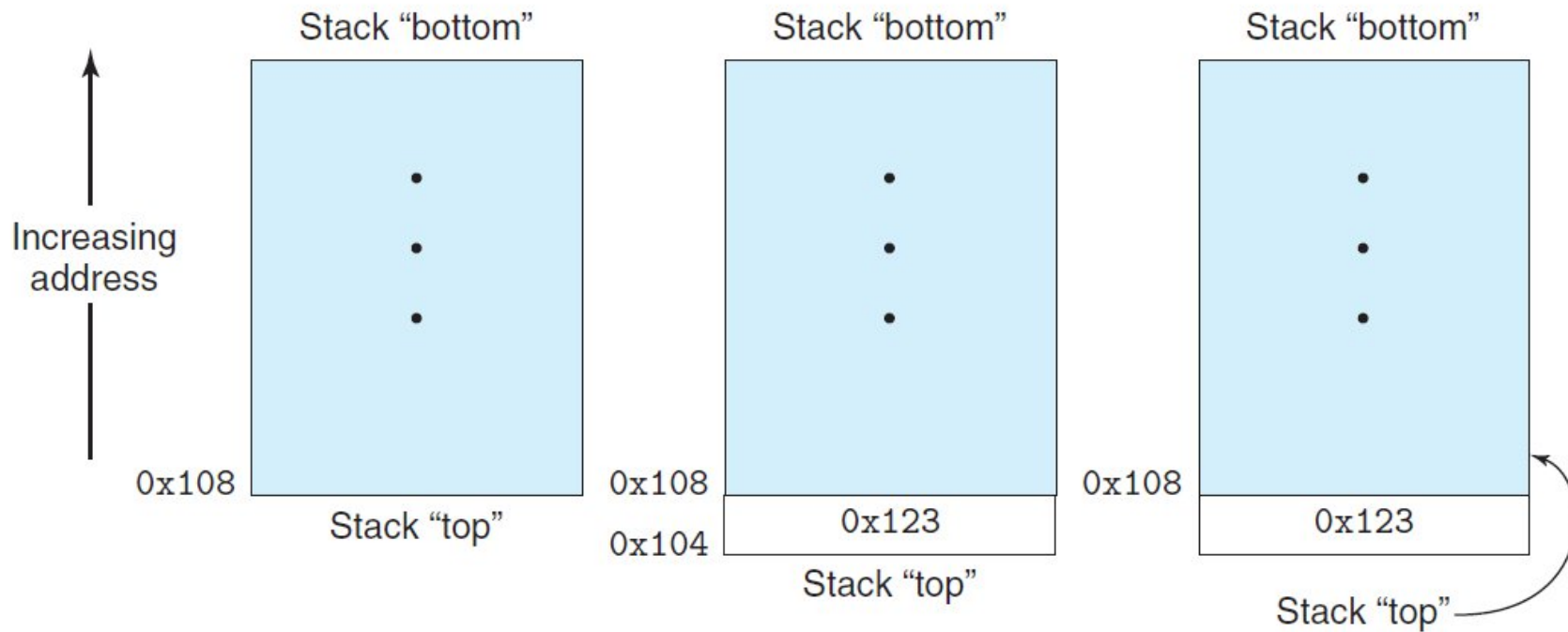
%eax	0x123
%edx	0
%esp	0x108

pushl %eax

%eax	0x123
%edx	0
%esp	0x104

popl %edx

%eax	0x123
%edx	0x123
%esp	0x108



Código de funcao1

00000000000000006da <funcao1>:

6da:	83 c7 01	add	\$0x1,%edi
6dd:	89 f8	mov	%edi,%eax
6df:	03	add	(%rsi),%eax
6e1:	c3	retq	

Código de funcao1

00000000000000006da <funcao1>:

6da: 83 c7 01

6dd: 89 f8

6df: 03

6e1: c3

add

mov

add

retq

\$0x1,%edi

%edi,%eax

(%rsi),%eax

O quê faz MOV

O quê significa esse ()?

Atividade prática

GDB: parando programas e examinando registradores

1. usar GDB para acompanhar a execução de um programa
2. examinar valores dos registradores

Código de funcao2

```
0x1145 <+0>:    mov     0x2ec9(%rip),%eax
0x114b <+6>:    add     $0x1,%eax
0x114e <+9>:    mov     %eax,0x2ec0(%rip)
0x1154 <+15>:   add     %edi,%eax
0x1156 <+17>:   retq
```

Código de funcao2

```
0x1145 <+0>:    mov    0x2ec9(%rip),%eax
0x114b <+6>:    add     $0x1,%eax
0x114e <+9>:    mov     %eax,0x2ec0(%rip)
0x1154 <+15>:   add     %edi,%eax
0x1156 <+17>:   retq
```

Quem
é %rip?

O quê significa
0x2ec0(%rip)?

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

movq 8(%rbp),%rdx

E os tamanhos?

O tamanho do dado é especificado na instrução! MOV não converte tipos!

Usamos um sufixo com o tamanho do tipo:

Q = quad word (8 bytes)

L = long word (4 bytes)

W = word (2 bytes)

B = byte (1 bytes)

Também podemos ver o tamanho dos registradores usados!

E os tamanhos?

Cuidado com acessos à memória!

```
movb $-1, (%rsp)
```

Copia um byte no endereço do topo da pilha.

```
movq $-1, (%rsp)
```

Copia 8 bytes no endereço do topo da pilha.

Exemplo

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

```

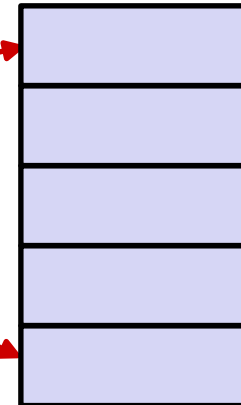
void swap
(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

```

Registers

%rdi	
%rsi	
%rax	
%rdx	

Memory



Register Value

%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Registers

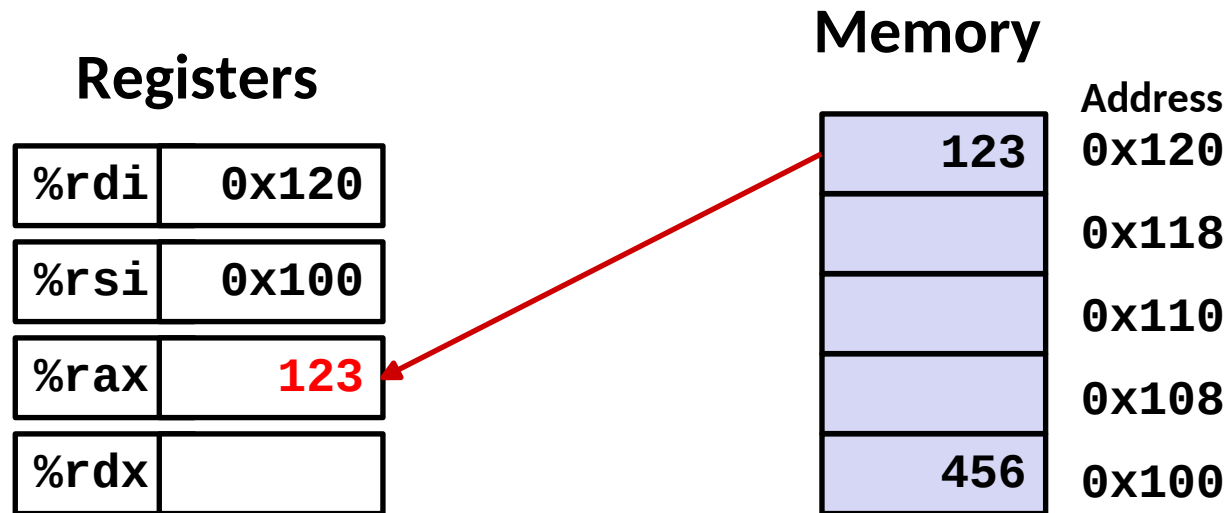
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

Address
123
456

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```


Registers

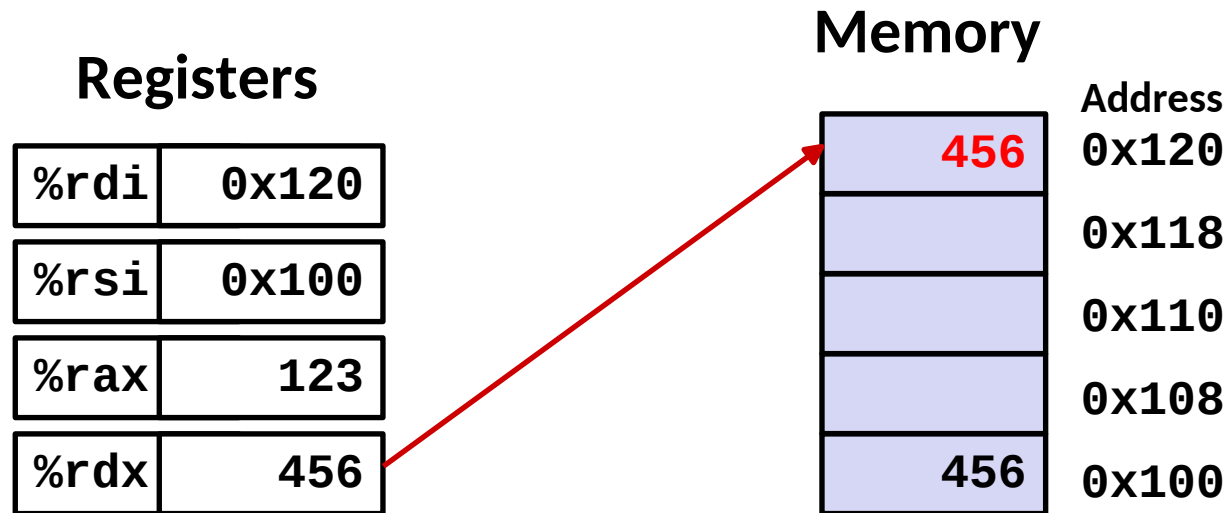
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Address
0x120
123
0x118
0x110
0x108
0x100
456

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```



swap:


```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Address
0x120
0x118
0x110
0x108
0x100



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

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

Exemplo

%rdx	0xf000
%rcx	0x0100

Expressão	Calculo de endereço	Resultado
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080



Atividade prática

Analisando operações de memória

1. Entender como variáveis globais são acessadas em Assembly

Inspere

www.insper.edu.br