Introdução à Software Básico: Introdução a Assembly IA-32

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Sumário

Lembrando aulas passadas

Escrevendo Programa em *Assembly*

- Editores de texto são usados para criar/modificar o código fonte
  - Unix: Vi, Emacs, xemacs, pico, etc
  - Windows: Notepad, word, edit, etc.
  - Alguns ambientes de desenvolvimento permitem programar em *Assembly* INTEL.
- Uma vez criado o código fonte, o **montador** gera o código objeto, e o ligador gera o código executável
- Nós utilizaremos o compilador **nasm** e o ligador **ld**. Disponível no ambiente Linux (existem versões para Windows).

Desenvolvendo Programas

Escrevendo Programa em *Assembly*

- Montado um programa chamado *hello.asm* em nasm:
  nasm -f elf -o hello.o hello.asm
  - *-f elf*, indica que o formato do arquivo objeto deve ser elf (*executable and linkable format*) 32 bits. Para compilar em 64 bits usar *elf*64.
  - *-hello.o*, nome do arquivo objeto de saída.
  - *-hello.asm*, nome do arquivo de entrada
- Gerando o executável com o ligador ld:
  ld -o hello hello.o
- Executando o programa em Linux: Basta digitar *./hello* no terminal.

Nota 1

Se o seu OS Linux é nativo de 64 bits, então para fazer a ligação deve utilizar:
ld -m elf_i386 -o hello hello.o

Nota 2

Também existe o montador GAS (comando **as**). Em formato GAS o código *Assembly* é ligeramente diferente que o NASM.

Estrutura de um programa NASM

```
;brief title of program          file name
;               Objectives:
;               Inputs:
;               Outputs:
section .data
 (initialized data go here)

section .bss
 (uninitialized data go here)

section .text
 (Directives and Instructions go here)
```

Figura: Anatomia de um programa Assembly Intel 32 bits, em formato NASM

Estrutura de um programa NASM

Seções

- A seção de texto (.section .text) é obrigatória em qualquer programa *Assembly*, já que nessa seção ficam os mnemônicos das instruções do programa.
- As seções de dados (section .data e section .bss) são opcionais e declaram elementos que são as variáveis do programa:
  - A seção de dados (section .data) declara elementos de dados cujo valor inicial é declarado no programa.
  - Já a seção de dados não inicializados (section .bss) declara elementos de dados inicializados com zero (ou para alguns montadores, não inicializados).

Estrutura de um programa NASM

Comentários

- Comentários em nasm: "';"' no início da linha

Formato

- Formato de uma declaração (linha de programa):
  [Símbolo] [Operação] [Operando(s)] [;Comentário]
- Símbolos podem ser:
  - Endereço de um ponto no código
  - Nome de uma posição de memória
  - Nome de constante

Antes de IA-32

- Intel 4004 (1971): Segundo CPU completamente integrado num único microchip da história (o primeiro foi o TMS 100)
  - 4 bits memory words, 640B of addressable memory, 740kHz
- Intel 8008 (1972):
  - 8 bits memory words, 16kB of addressable memory, 800kHz
- Intel 8086 (1978):
  - 16 bits memory words, 1MB of addressable memory, 10MHz
- Intel 80286 (1982):
  - 16 bits memory words, 16MB of addressable memory, 12.5MHz

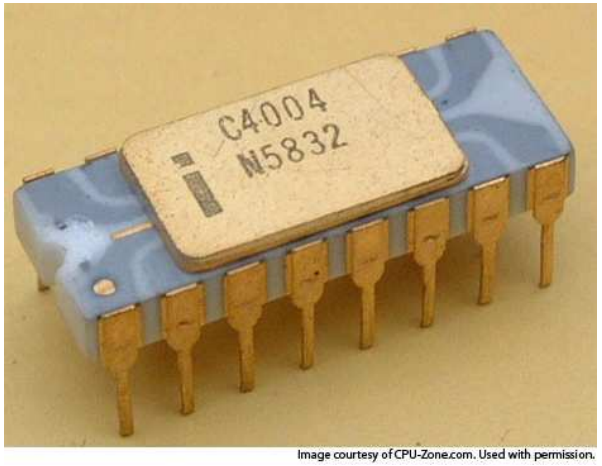
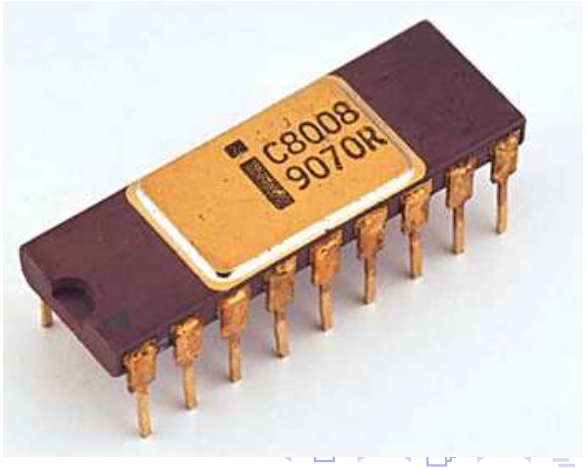
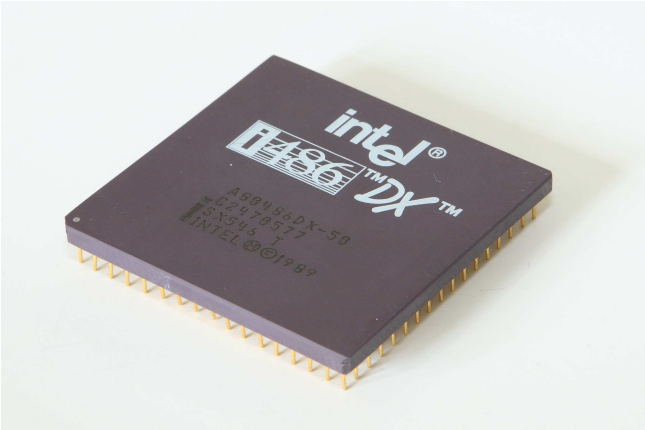


Image courtesy of CPU-Zone.com. Used with permission.



Primeiros Processadores IA-32

- Intel 80386(DX) (1985):
  - Introduz a *Memory Managment Unit* (MMU). 32 bits memory words, 4Gb of addressable memory, 16MHz
- Intel 80486(DX) (1989):
  - Processador especifico matemático embutido no mesmo chip. 32 bits memory words, 4Gb of addressable memory, 16MHz



Famila de Processadores IA-32

Processadores Compatíveis com IA-32

- Intel IA-32 Processors (x86-64):
  - Pentium (1993), Pentium II, Pentium III, Pentium 4, Pentium M, Celeron, Core, Core2, etc.
- AMD IA-32 Processors:
  - K5,K6,Duron, Athlon, Athlon XP.
- AMD AMD64 Processors:
  - Sempron, Athlon 64, Athlon 64 X2, Phenom, Turion, Phenom II, Athlon II, Opteron.

IA-64 vs. x86-64

- IA-64 (Intel Itanium architecture) é a primeira família de processadores de 64 bits, desenvolvida em conjunto pela HP e Intel.
- x86-64 é o família de processadores que pode ser conhecida também como AMD64 ou Intel 64.
- O conjunto de instruções x86-64 foi desenvolvido inicialmente pela AMD, sendo similar as instruções x86 (IA-32).
- O conjunto de instruções IA-64 é bem diferente do IA-32.
- Eventualmente a Intel incorporou as instruções desenvolvidas pela AMD. Ainda são fabricados chips Itanium porém os processadores x86-64 dominam o mercado.

Layout da Memória em LINUX

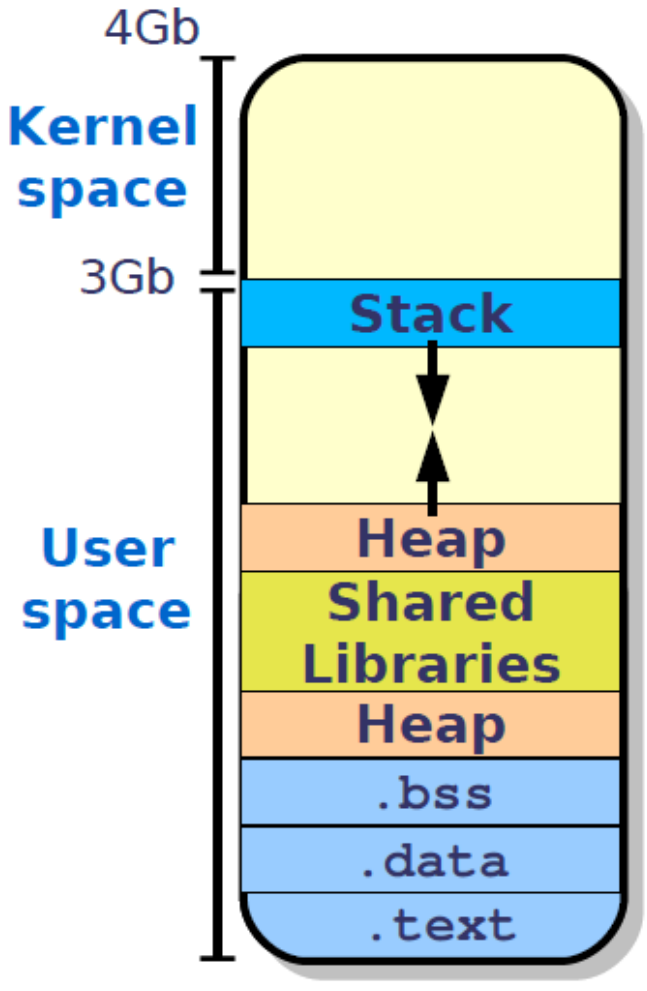


Figura: Memory Layout (Linux)

Layout da Memória em LINUX

- **Text**: Contem o código sendo executado. Esta parte da memória é compartilhada por vários programas. Esta parte da memória é tratado como de somente leitura (um programa não pode modificar suas próprias instruções).
- **Initialized Data Segment** (.data) e **Uninitialized Data Segment** (.bss): Contém variáveis globais e estáticas de um programa, as variáveis que não foram inicializadas com nenhum valor ficam na seção .bss (normalmente são inicializadas com zero).
- **Stack** (Pilha): Contém os *frames* necessários para chamada de funções. A pilha vai crescendo assim que necessário.
- **Heap**: Memória reserva para alocação dinâmica (*malloc*,*calloc*,*new*,etc). A Heap vai crescendo assim que necessário.
- **Shared/Dynamic Libraries**: Espaço reservado para bibliotecas compartilhadas.

Layout da Memória em LINUX

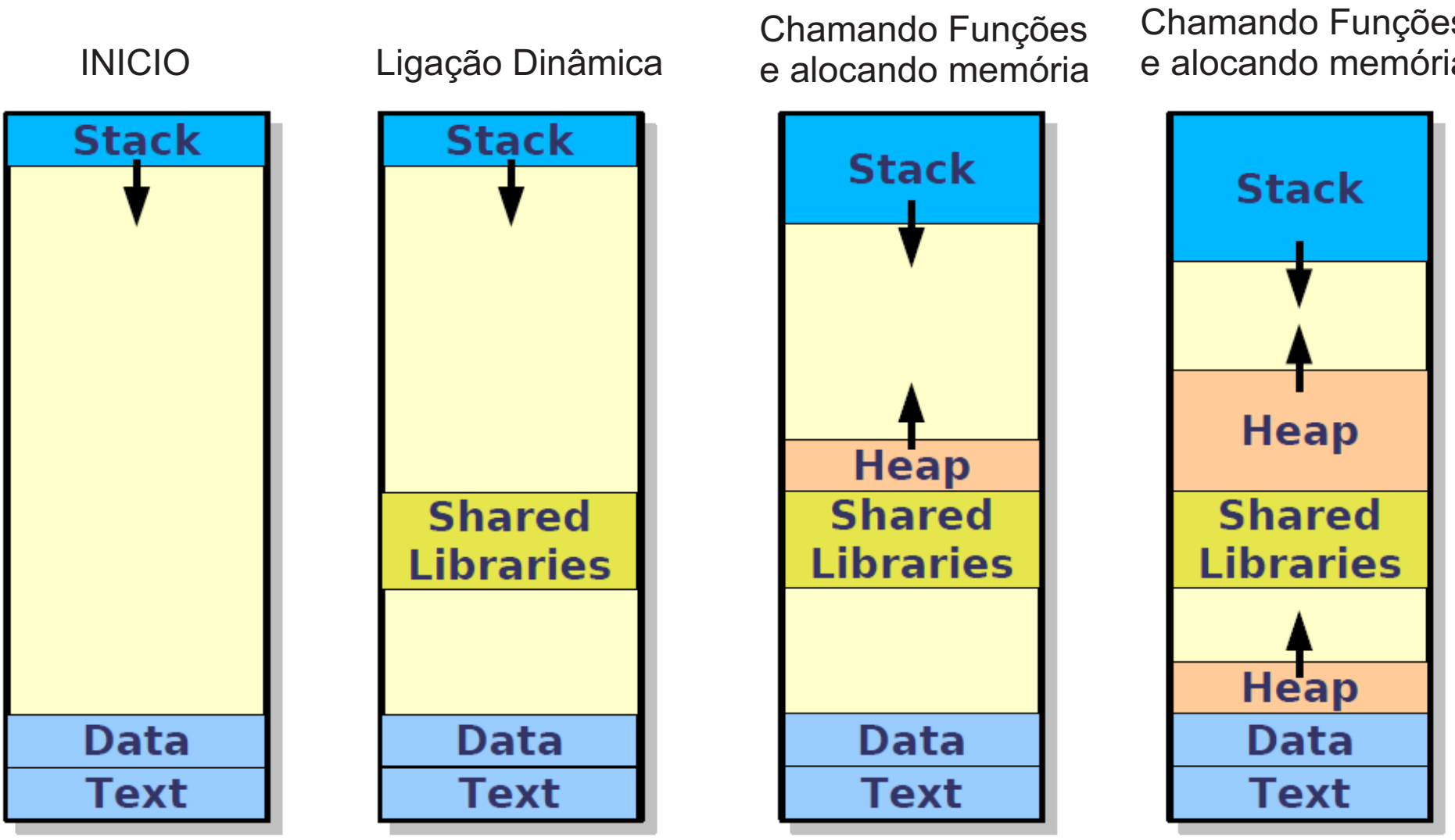


Figura: Memory Layout (Linux)

Fortmato da Pilha

- Cresce em direção a endereços menores.
- O registrador **esp** indica o menor enderço da pilha (topo da pilha)
  - esp: Stack Pointer
- O registrador **ebp** indica o maior enderço da pilha (final da pilha)
  - ebp: Base Pointer

Pilha IA-32

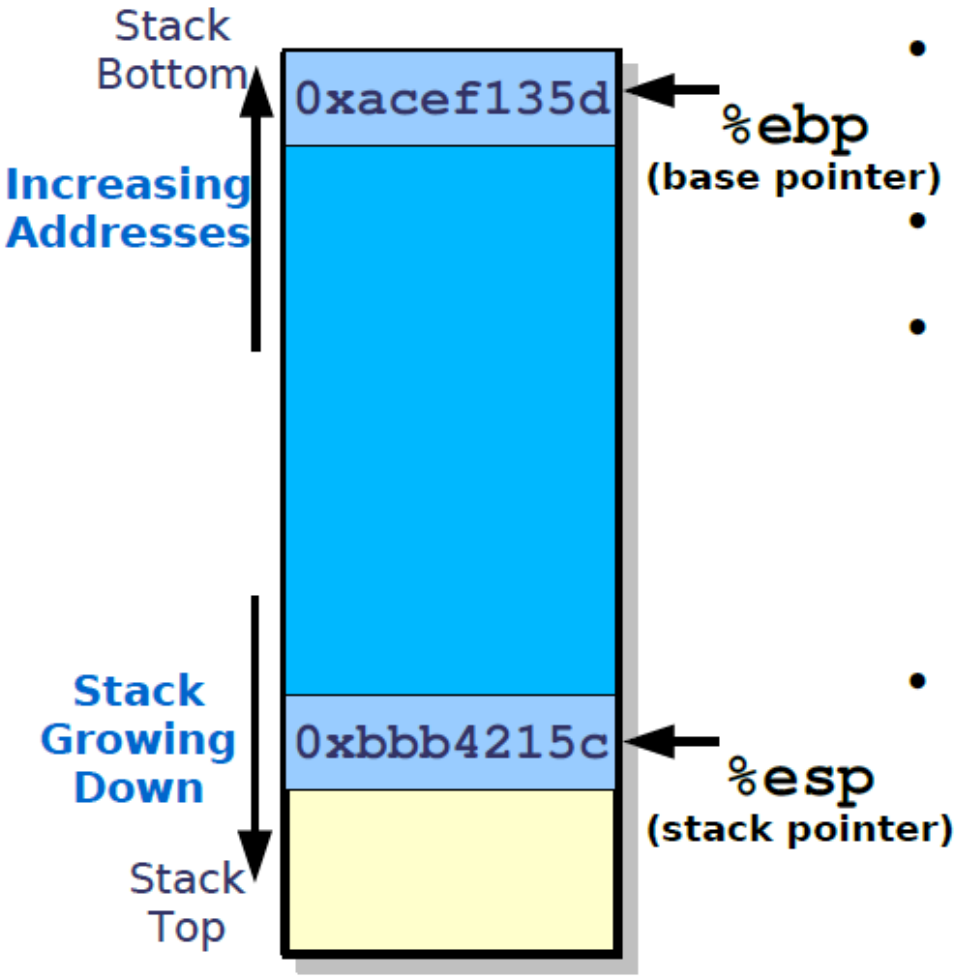
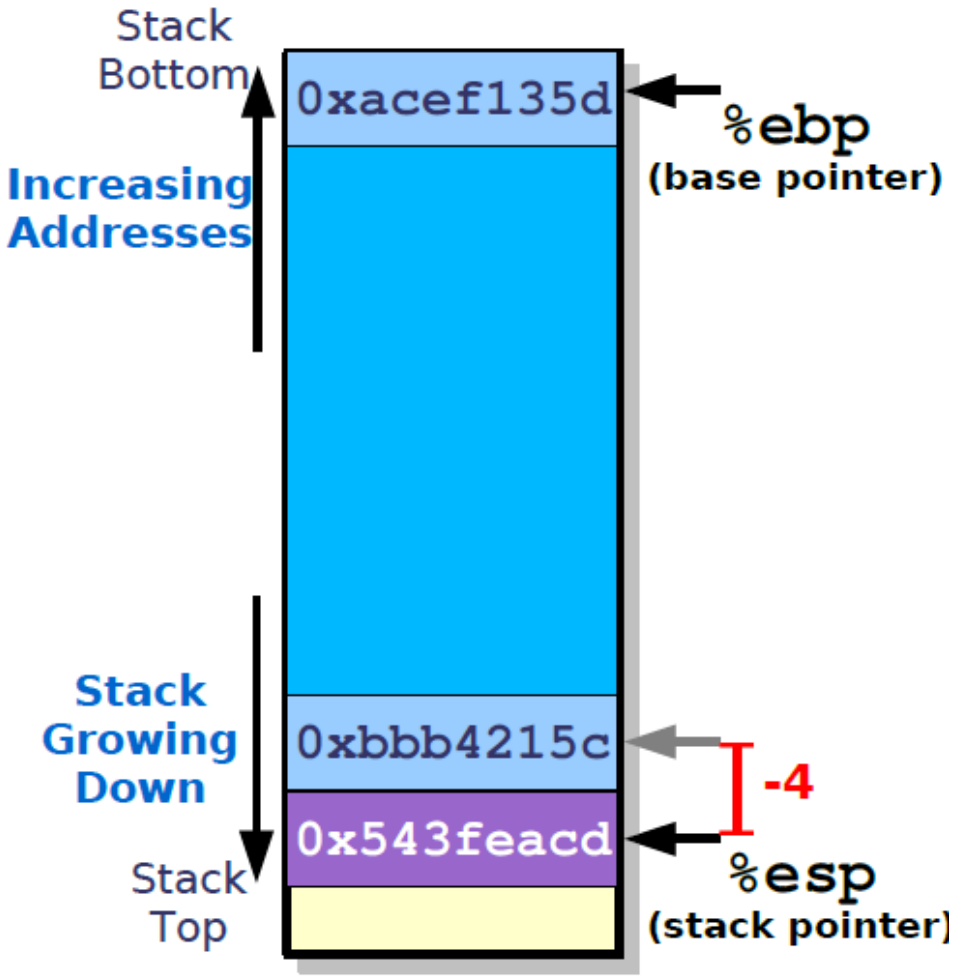


Figura: Pilha de um processador IA-32

Operação de Push

- push src
- src = endereço de memoria
- a operação inicialmente busca o dado indicado por src
- Decrementa **esp** em 4 posições (assumindo dados de 32 bits, é possível usar dados de 16 bits também).
- Escreve o valor do dado na posição indicada por **esp**

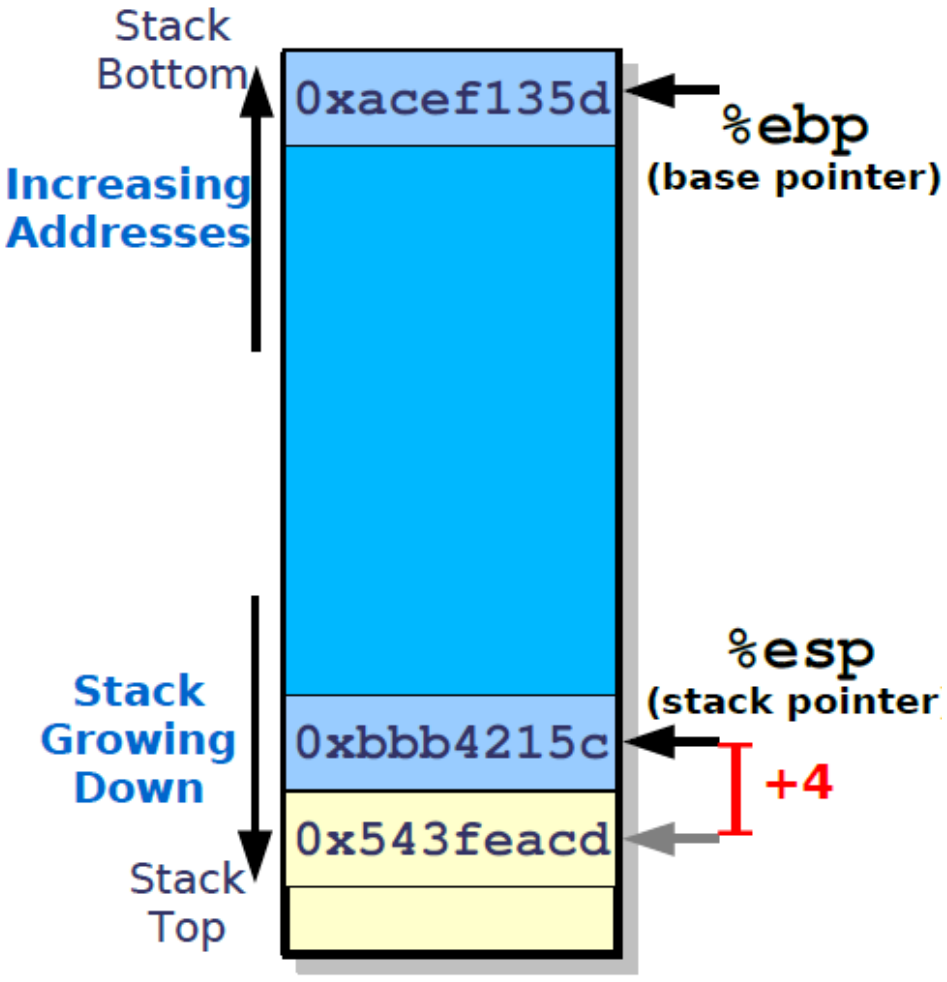
Pilha IA-32


Stack
Bottom
0xacef135d
%ebp
(base pointer)
Increasing
Addresses
Stack
Growing
Down
0xbbb4215c
-4
0x543feacd
%esp
(stack pointer)
Stack
Top

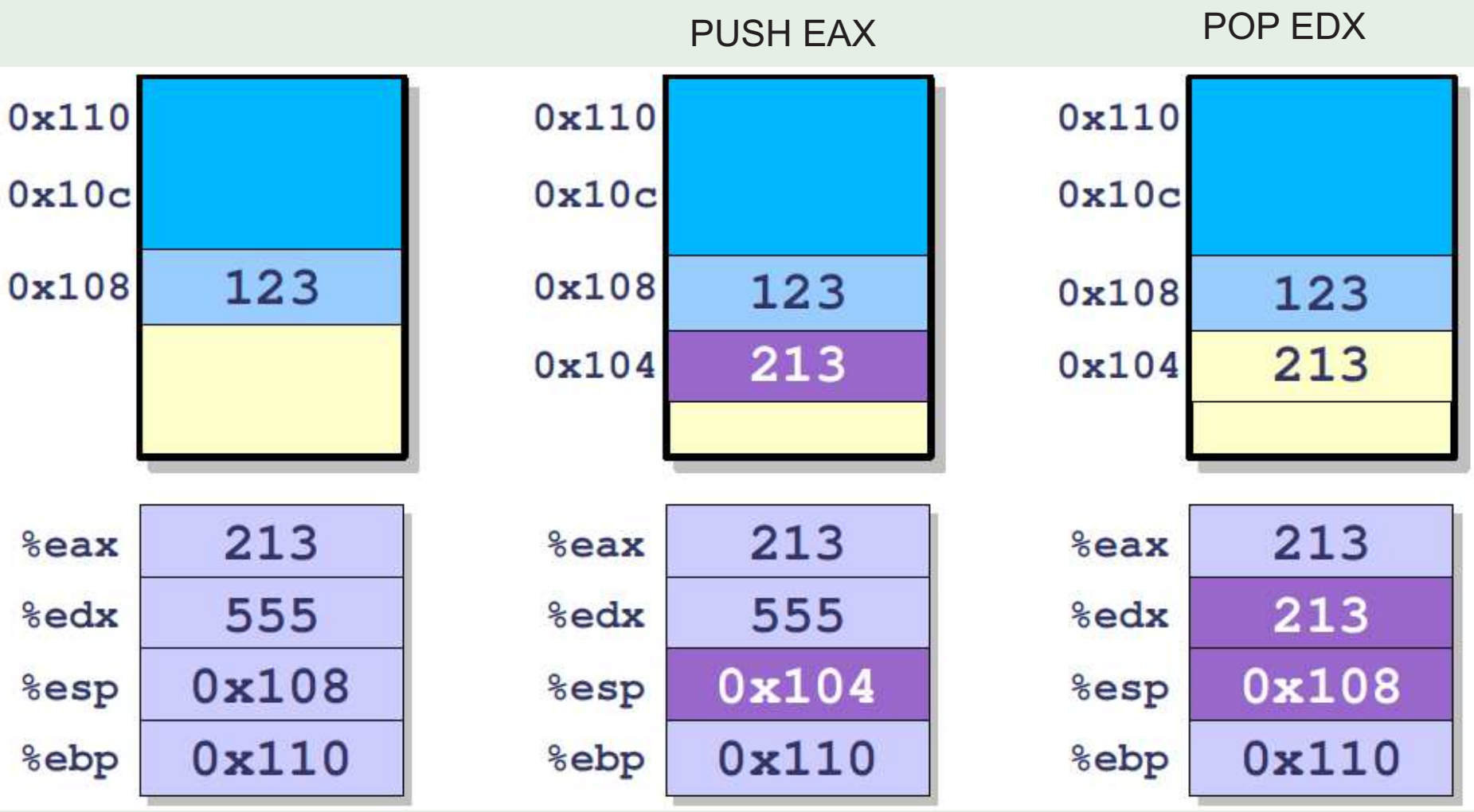
Operação de Pop

- pop dest
- dest = endereço de memoria
- Ler o dado apontado por **esp**
- Incrementa **esp** em 4.
- Escreve o o dado na posição dest

Pilha IA-32

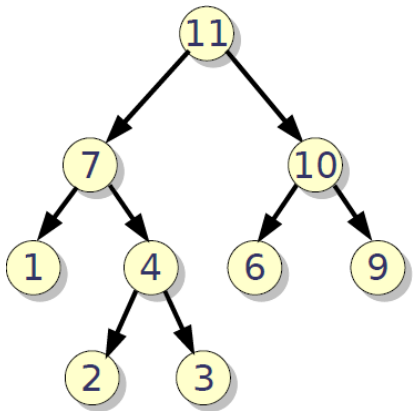

Stack
Bottom
0xacef135d
%ebp
(base pointer)
Increasing
Addresses
%esp
(stack pointer
Stack
Growing
Down
0xbbb4215c
+4
0x543feacd
Stack
Top

Exemplo


PUSH EAX
POP EDX
0x110
0x10c
0x108
123
0x104
213
%eax
%edx
%esp
%ebp
213
555
0x108
0x110
0x104

O que é uma heap binária?

- A heap binária é uma árvore binária onde o pai é sempre maior o igual aos filhos.
- Pode ser implementada por array ou árvore. No casso de array o no $a[i]$ tem filhos na posição $a[2i+1]$ e $a[2i+2]$ (com a primeira posição sendo 0).
- Aplicação: Acesso rápido a dados
- Grupos de Dados: Doug Lea desenvolveu o dlmalloc ("Doug Lea's Malloc") um alocador de memória de propósito geral. A Biblioteca GNU C utiliza a ptmalloc, a qual é baseada na dlmalloc. Os blocos de memória reservadas são classificados na heap pelo tamanho.

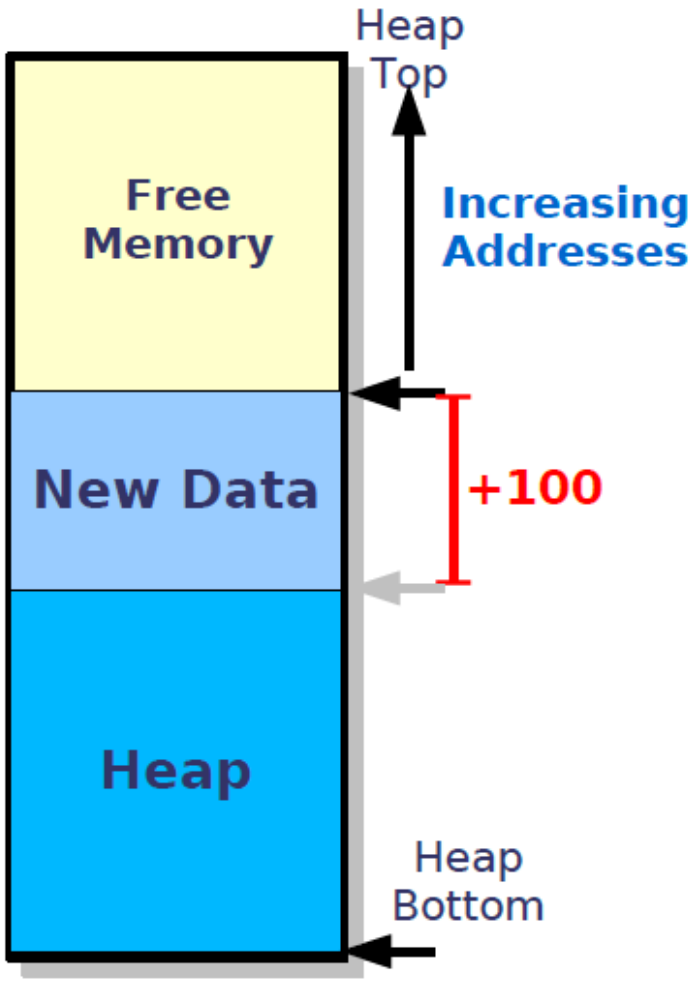


Heap de alocação de memória

Heap IA-32

- Cresce em direção de endereços maiores.
- Desde o ponto de vista do programados é admnistrada por a linguagem de programação independente da interface (C,C++, Java, etc).
- Desde o ponto de vista do sistema é administrada por chamadas ao sistema (*system calls*), como *mmap()*, *brk()*.

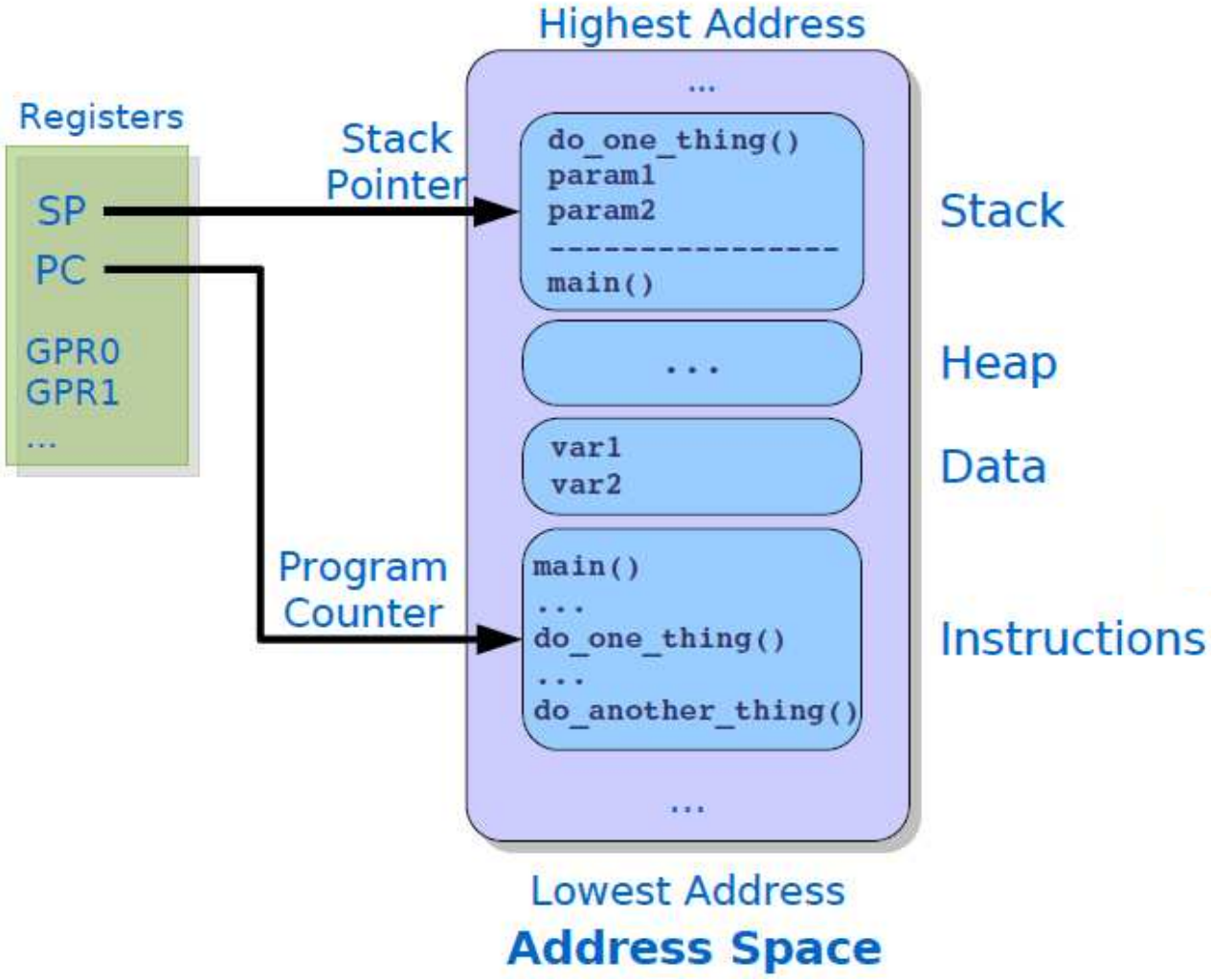
Heap IA-32


Heap
Top
Free
Memory
Increasing
Addresses
New Data
+100
Heap
Heap
Bottom

Modelo Computacional

- Registradores:
  - SP (Stack Pointer)
  - PC (Program Counter)
  - GPR (Registradores de uso Geral)
- Memoria:
  - Pilha (stack)
  - Heap
  - Dados
  - Instruções

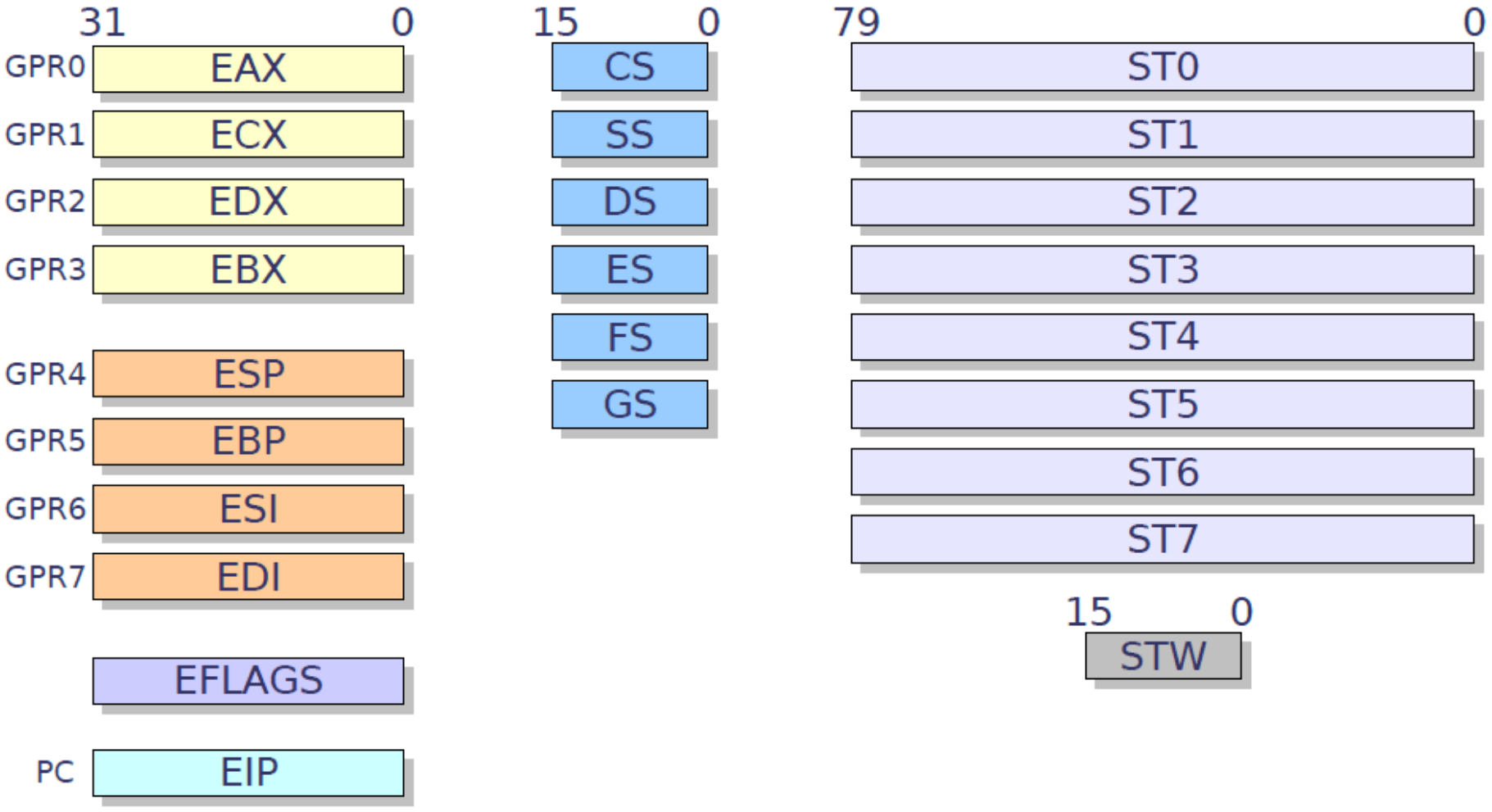
Modelo Computacional IA-32



Registradores

- Registradores de dados (read/write):
  - eax, ebx, ecx, edx
- Registradores de index e ponteiros (read/write):
  - ebp, esp, eip, esi, edi
- Registradores de segmentos (protegidos):
  - cs,ds,es,fs,gs,ss
- Registradores de flags (read):
  - eflags
- Registradores de ponto flutuante (read/write):
  - st0,...,st7

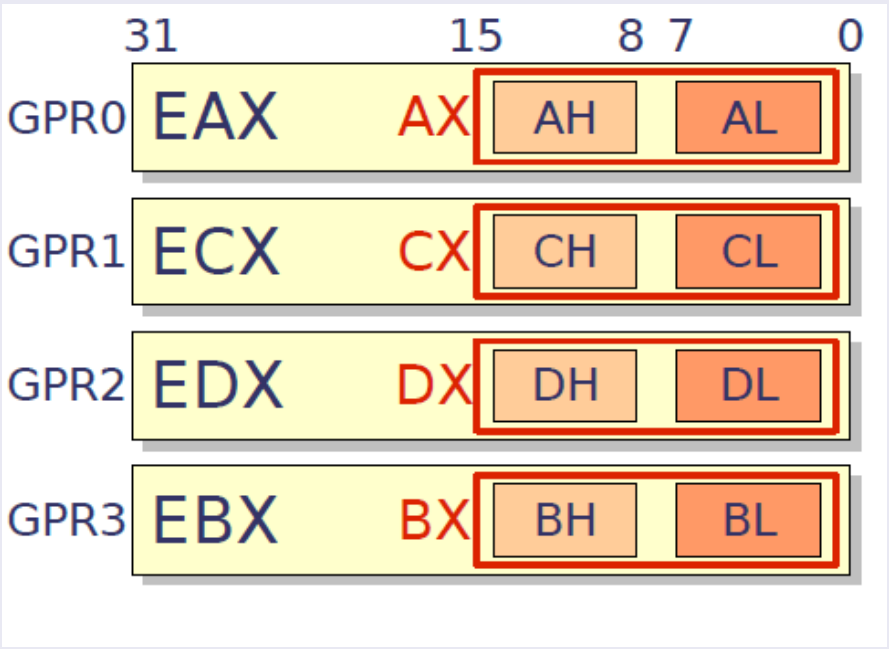
Registradores



Registradores de Dados

Os registradores de dados podem ser acessados como 32, 16 ou 8 bits.

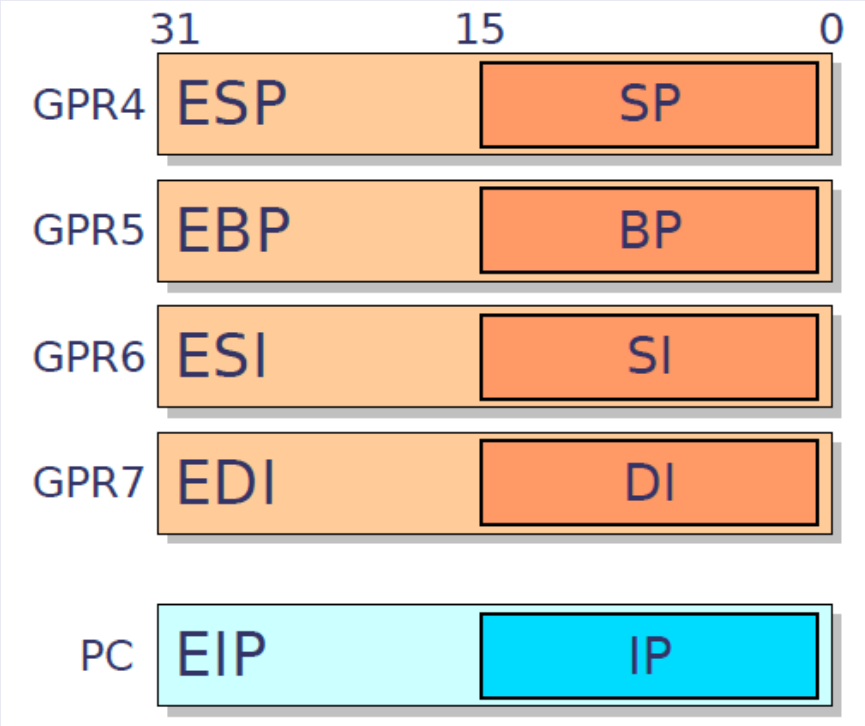
- EAX (Acumulador): Para operando e resultados (adição, subtração, etc).
- EBX (Base Register): Normalmente usado para indicar o endereço base da estrutura de dados
- ECX (Count Register): Normalmente usado como um contador para laços
- EDX (Data Register): Operandos e resultados de multiplicações e divisões



Registradores de Index e Ponteiros

Os registradores de dados podem ser acessados como 32 ou 16 bits.

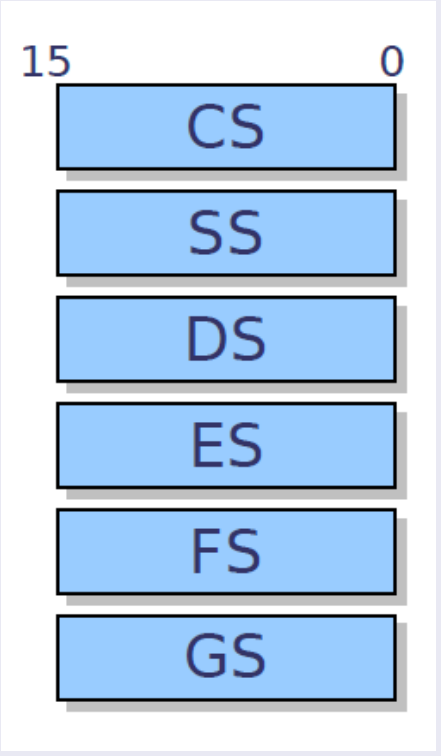
- ESP (stack pointer): Ponteiro ao topo da pilha
- EBP (base pointer): Ponteiro ao final da pilha
- ESI (Source index): Usado em operações com string (como source)
- EDI (Destination index): Usado em operações com string (como destino)
- EIP (Instruction Pointer): Ponteiro a seguinte instrução



Registradores de Segmento

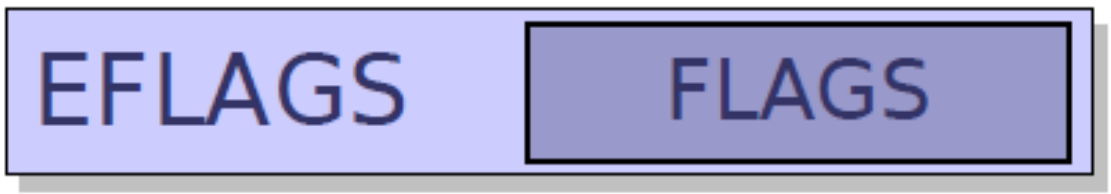
Os registradores de dados podem ser acessados como 16 bits.

- CS (Code Segment): Ponteiro para o segmento de código (text) atual
- SS (Stack Segment): Ponteiro para o segmento de pilha atual
- DS (Data Segment): Ponteiro para o segmente de dados atual
- ES,FS,GS (Extra data segments): Ponteiros adicionais para segmentos de memória distantes (video memory e outras).

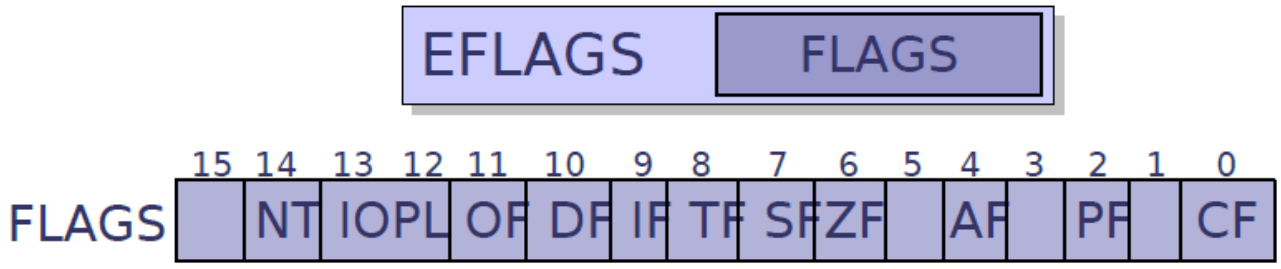


Registrado de Flags

- Status Flags (stat): Da o resultado de instruções aritméticas (add,sub,mul ou div)
- Control Flags (ctrl): Muda o comportamento do processador em algumas instruções (std, cld)
- System Flags (sys): Acessadas somente pelo Kernel



Registrado de Flags



CF (Carry Flag, stat): Left most bit of result;

PF (Parity Flag, stat): '1' if result is even;

AF (Auxiliary Carry Flag, stat): Extra carry flag;

ZF (Zero Flag, stat): '1' if result is '==0';

SF (Sign Flag, stat): '1' if result is '<0';

TF (Trap Flag, sys): Set CPU in single-step mode;

IF (Interrupt Flag, sys): '1' if interruptions are on;

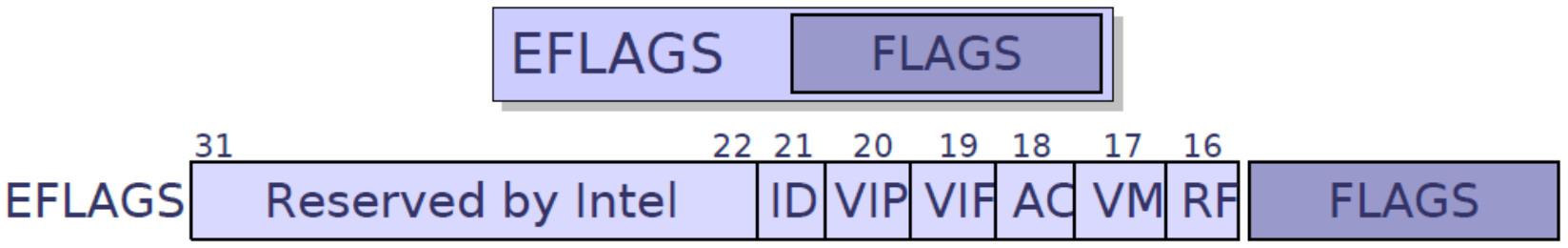
DF (Direction Flag, ctrl): Strings reading order. Modified by **`std`** ('1', go from higher to lower addresses) and **`cld`** ('0', reverse order);

OF (Overflow Flag, stat): '1' if an overflow occurs;

IOPL (I/O Privilege Level, sys): Current task privilege;

NT (Nested Task Flag, sys): '1' if current task is linked to the previous one.

Registrado de Flags



RF (Resume Flag, sys):
Set CPU in debug mode;

VM (Virtual Mode, sys):
Set the 8086 virtual mode (unset the protected mode);

AC (Alignment Check Flag, sys):
Set alignment checking mode for memory references;

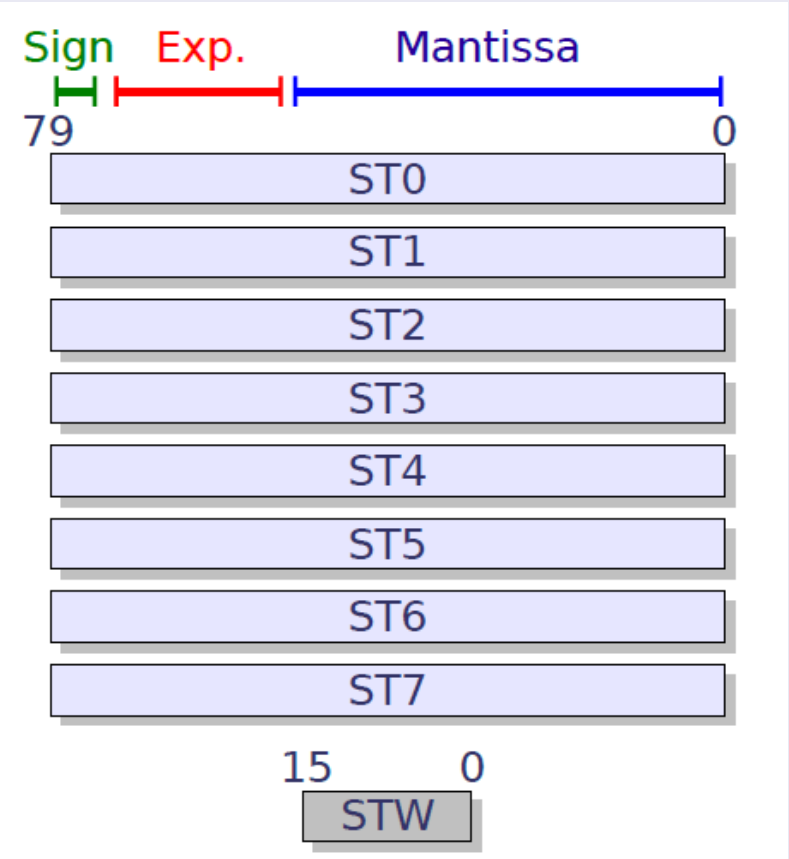
VIF (Virtual Interrupt Flag, sys):
Virtual image of the IF flag (used in conjunction with VIP);

VIP (Virtual Interrupt Pending Flag, sys):
Indicates pending interruptions ('1' if one is pending);

ID (ID Flag, sys):
Triggers the CPUID instruction support.

Registradores de Ponto Flutuante

- Tamanho de 80 bits
- Somente pode ser acessado como pilha (não diretamente)
- Decompostos em: $(sign)M * 10^e$.
  Onde: M = matissa (64 bits)
  e = Expoente (15 bits)
  sign = sinal (1 bit).



Status Word Register

B	C3	ST	C2	C1	C0	IR		EU	PE	OE	ZE	DE	IE
15	14	13	10	9	8	7	6	5	4	3	2	1	0

Exception Flags (bits 0-6)

IE Invalid Operation Exception (Uncertain result)

DE Denormalized Operation Exception (Operation on denormalized number)

ZE Zero Divide Exception (Division by zero)

OE Overflow Exception (Result is too large)

UE Underflow Exception (Result is too small)

PE Precision Exception (Loss of precision occured)

ST (Stack pointer): Counting free registers;

Condition Code (C0,C1,C2,C3): Store results of boolean tests on float in these bits;

Others (IR, B):

IR Interruption Request

B Busy (on-going operation)

Sintaxe Intel vs. AT&T

Intel

- Todos os manuais da Intel estão escritos nesta syntaxe. Muitas ferramentas utilizam esta syntaxe, incluindo todas as ferramentas da comunidade Microsoft.
- O compilador é o NASM

AT&T

- A maioria das ferramentas GNU e da comunidade UNIX utilizam esta sintaxe.
- O compilador é o GAS (comando AS). O gcc também é capaz de montar

Nota

Na disciplina vamos usar a sintaxe INTEL. Porém, voltado para o sistema LINUX

Sintaxe Intel vs. AT&T

Intel Syntax

- O primeiro operando é o destino e o segundo operando é a fonte

Intel Syntax

```
Instr.      dest, src
mov         eax,[ecx]
```

AT&T Syntax

- O primeiro operando é a fonte e o segundo operando é o destino

AT&T Syntax

```
Instr.      src, dest
mov         (%ecx),%eax
```

Intel Syntax:

- **10d**: Decimal value (**10** is ok);
- **10h**: Hexadecimal value;
- **1**: Immediate value;
- **eax**: Register;
- **byte ptr**: Address of a byte (8bits)
- **word ptr**: Address of a word (16bits)
- **dword ptr**: Address of a long (32bits)

AT&T Syntax:

- **0d10**: Decimal value (**10** is ok);
- **0x10**: Hexadecimal value;
- **$1**: Immediate value;
- **%eax**: Register;
- **movb**: Operand on bytes (8bits)
- **movw**: Operand on words (16bits)
- **movl**: Operand on longs (32bits)

Sintaxe Intel vs. AT&T

Intel Syntax

```
mov   eax, 1
mov   ebx,0ffh
int   80h
mov   al,bl
mov   ax,bx
mov   eax,ebx
mov   eax,dword ptr [ebx]
```

AT&T Syntax

```
movl      $1,%eax
movl      $0xff,%ebx
int       $0x80
movb      %bl,%al
movw      %bx,%ax
movl      %ebx,%eax
movl      (%ebx),%eax
```

Sintaxe Intel vs. AT&T

Intel

- Comentário começa com “;”
- Endereço é encapsulado em colchetes (“[”, “]”) e especificados por:
  - [base+index*scale+offset]
  - Exemplo: mov eax,[ebx+ecx*4h-20h]

AT&T

- Comentário começa com “#”
- Endereço é encapsulado em parênteses (“(”, “)”) e especificados por:
  - offset(base,index,scale)
  - Exemplo: subl -0x20(%ebx,%eax,0x4),%eax

Próxima Aula

Continução Introdução a IA-32