

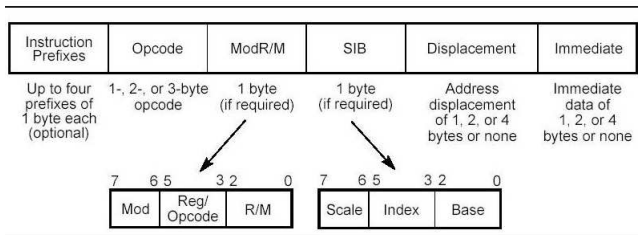
Introdução à Software Básico: Introdução a Assembly IA-32

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

Sumário

- 1 Formato das Instruções em IA-32
- 2 Grupo Básico de Instruções
- 3 Interrupções e Chamadas ao Sistema
- 4 Acesso à Memória

Formato das Instruções



Formato das Instruções em Código Máquina

- Tamanho das Instruções variam de 1 a 12 bytes
 - Apenas o Opcode é obrigatório (1 ou 2 bytes)
 - Os bytes seguintes dependem da instrução
- Além da instrução e seus operandos, prefixos podem ser acrescentados para alterar formato padrão das instruções
 - Máximo de 4 prefixos (tamanho de 1 byte cada)

ModR/M

- Mod=00
 - Primeiro operando é o número de um registrador
 - Segundo operando é um endereço de memória armazenado num registrador ($\text{Memory}[\text{R/M}]$).
- Mod=01
 - Igual ao mode 00 com deslocamento (*displacement*) de 8-bits
 - Segundo operando é um endereço de memória armazenado num registrador ($\text{Memory}[\text{displacement} + \text{R/M}]$).
- Mod=10
 - Igual ao mode 01 com deslocamento (*displacement*) de 32-bits
- Mod=11
 - O segundo operador é também um número de um registrador

SIB

- Indica como um endereço de memória é calculado
- $Endereco = Reg[base] + Reg[Index] * 2^{scale}$

Tipos de Instruções

- Operadores de Memória
 - Movimentação de Dados
 - Gerenciamento de Pilha
- Operações Lógicas/Aritméticas
 - Operações bitwise
 - Aritmetica inteira
 - Aritmética ponto flutuante
- Controle de Fluxo
 - Testes
 - Laços
 - Pulos

Mnemonic	Operand	Operand	Operation
<code>mov</code>	<code>dst</code>	<code>src</code>	<code>src → dst</code>
<code>xchg</code>	<code>dst</code>	<code>dst</code>	<code>dst ↔ dst</code>
<code>lea</code>	<code>reg</code>	<code>mem</code>	<code>mem → reg</code>
<code>movz</code>	<code>reg16</code>	<code>src8</code>	zero-extended <code>src → reg</code>
	<code>reg32</code>	<code>src8</code>	
	<code>reg32</code>	<code>src16</code>	
<code>movsz</code>	<code>reg16</code>	<code>src8</code>	sign-extended <code>src → reg</code>
	<code>reg32</code>	<code>src8</code>	
	<code>reg32</code>	<code>src16</code>	
<code>cbw</code>	-	-	sign-extended <code>%al → %ax</code>
<code>cwd</code>	-	-	sign-extended <code>%ax → %dx.%ax</code>
<code>cdq</code>	-	-	sign-extended <code>%eax → %edx.%eax</code>
<code>cwde</code>	-	-	sign-extended <code>%ax → %eax</code>

`lea` = Load Effective Address

Mnemonic	Operand	Operation
push	src8	%esp-1 → %esp ; src → (%esp)
	src16	%esp-2 → %esp ; src → (%esp)
	src32	%esp-4 → %esp ; src → (%esp)
pop	dst8	(%esp) → dst ; %esp+1 → %esp
	dst16	(%esp) → dst ; %esp+2 → %esp
	dst32	(%esp) → dst ; %esp+4 → %esp
pushf	-	%esp-4 → %esp ; %eflags → (%esp)
popf	-	(%esp) → %eflags ; %esp+4 → %esp
pusha	-	Push %eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi
popa	-	Pop %eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi
enter	-	Create a stack frame
leave	-	Restore the previous stack frame

Mnemonic	Operand	Operand	Operation	Touched Flags
and	dst	src	$\text{src} \& \text{dst} \rightarrow \text{dst}$	SF,ZF,PF
or	dst	src	$\text{src} \text{dst} \rightarrow \text{dst}$	
xor	dst	src	$\text{src} \wedge \text{dst} \rightarrow \text{dst}$	
test	dst	src	$\text{src} \& \text{dst}$ (result discarded)	
not	dst		$\sim \text{dst} \rightarrow \text{dst}$	-
cmp	dst	src	sub src, dst (result discarded)	OF,SF,ZF,AF,CF,PF

Mnemonic	Operand	Operand	Operation	Touched Flags
shl/sal	dst	src	left shift dst of src bits	CF,OF
shr/sar	dst	src	right shift dst of src bits	
rol	dst	src	left rotate dst of src bits	
ror	dst	src	right rotate dst of src bits	

Mnemonic	Operand	Operand	Operation	Touched Flags
add	dst	src	$\text{src} + \text{dst} \rightarrow \text{dst}$	OF,SF,ZF, AF,CF,PF
adc	dst	src	$\text{src} + \text{dst} + \text{CF} \rightarrow \text{dst}$	
sub	dst	src	$\text{dst} - \text{src} \rightarrow \text{dst}$	
sbb	dst	src	$\text{dst} - \text{src} - \text{CF} \rightarrow \text{dst}$	
inc	dst		$\text{dst} + 1 \rightarrow \text{dst}$	OF,SF,ZF,AF,PF
dec	dst		$\text{dst} - 1 \rightarrow \text{dst}$	
neg	dst		$-\text{dst} \rightarrow \text{dst}$	OF,SF,ZF,AF,PF CF=0 if $\text{dst}==0$

Mnemonic	Operand	Operation	Flags
mul (unsigned)	reg8	$\%al * reg8 \rightarrow \%ax$	CF, OF
	reg16	$\%ax * reg16 \rightarrow \%dx.\%ax$	
	reg32	$\%eax * reg32 \rightarrow \%edx.\%eax$	
imul (signed)	reg8	$\%al * reg8 \rightarrow \%ax$	
	reg16	$\%ax * reg16 \rightarrow \%dx.\%ax$	
	reg32	$\%eax * reg32 \rightarrow \%edx.\%eax$	
div (unsigned)	reg8	$\%ax / reg8 \rightarrow \%al ; \%ax \bmod reg8 \rightarrow \%ah$	OF, SF, ZF, AF, CF, PF
	reg16	$\%dx.\%ax / reg16 \rightarrow \%ax ; \%dx.\%ax \bmod reg16 \rightarrow \%dx$	
	reg32	$\%edx.\%eax / reg32 \rightarrow \%eax ; \%edx.\%eax \bmod reg32 \rightarrow \%edx$	
idiv (signed)	reg8	$\%ax / reg8 \rightarrow \%al ; \%ax \bmod reg8 \rightarrow \%ah$	
	reg16	$\%dx.\%ax / reg16 \rightarrow \%ax ; \%dx.\%ax \bmod reg16 \rightarrow \%dx$	
	reg32	$\%edx.\%eax / reg32 \rightarrow \%eax ; \%edx.\%eax \bmod reg32 \rightarrow \%edx$	

Teste

- Os condicionais são feitos utilizando uma instrução de pulo e uma de comparação
- A instrução de comparação **cmp**, compara dois operandos fazendo a subtração do primeiro pelo segundo e ativando as *flags* necessárias.

Exemplo

- **cmp eax, ebx**
- **je LO**
- Se eax e ebx são iguais então o contador de programa vai pular para o rótulo LO.

Mnemonic	Operand	Operation	Notes
jmp	lbl	jump to lbl	-
ja/jne	lbl	Jump if above / not below or equal	unsigned operands
jae/jnb	lbl	Jump if above or equal / not below	
jbe/jna	lbl	Jump if below or equal / not above	
jb/jnae	lbl	Jump if below / not above or equal	
jb/jnle	lbl	Jump if greater / not less or equal	signed operands
jge/jnl	lbl	Jump if greater or equal / not less	
jle/jng	lbl	Jump if less or equal / not greater	
jl/jnge	lbl	Jump if less / not greater or equal	
je/jz	lbl	Jump if equal / zero (ZF=1)	equality testing
jne/jnz	lbl	Jump if not equal / not zero (ZF=0)	
jc	lbl	Jump if (CF=1)	-
jnc	lbl	Jump if (CF=0)	
js	lbl	Jump if (SF=1)	
jns	lbl	Jump if (SF=0)	

Mnemonic	Operand	Operation
loop	lbl	$\%cx-1 \rightarrow \%cx$; if ($\%cx \neq 0$) jump to lbl
loope	lbl	$\%cx-1 \rightarrow \%cx$; if ($\%cx \neq 0$) and ($ZF=1$) jump to lbl
loopne	lbl	$\%cx-1 \rightarrow \%cx$; if ($\%cx \neq 0$) and ($ZF=0$) jump to lbl
loopz	lbl	$\%cx-1 \rightarrow \%cx$; if ($\%cx \neq 0$) and ($ZF=1$) jump to lbl
loopnz	lbl	$\%cx-1 \rightarrow \%cx$; if ($\%cx \neq 0$) and ($ZF=0$) jump to lbl

Mnemonic	Operand	Operation
finit	-	FPU Initialization
fincstp	-	Increment the FPU stack pointer
fdecstp	-	Decrement the FPU stack pointer
ffree	st(i)	Free the content of st(i)
fldz	-	Load zero in st(0)
fld1	-	Load one in st(0)
fldpi	-	Load π in st(0)
fld	?	Load a float in st(0)
fild	?	Load an int in st(0)
fst	?	Write a float in main memory
fstp	?	Write a float in main memory and pop
fxch	?	Exchange two registers content

Mnemonic	Operand	Operation
fadd	-	$st(0)+st(1) \rightarrow st(0)$
	mem/st(i)	$st(0)+mem/st(i) \rightarrow st(0)$
	mem/st(i) mem/st(j)	$st(0)+mem/st(i) \rightarrow st(0)$
fsub	-	Similar to fadd but for subtraction
fmul/fdiv	-	Similar to fadd but for multiplication/division
fchs	-	Change sign
fabs	-	Absolute value
fsqr	-	Square
fsqrt	-	Square root
fcos	-	Cosine
fyl2x	-	$y * \log_2(x)$
frndint	-	Round to integer value

Mnemonic	Operand	Operand	Operation
fcom	mem/st(i)	mem/st(j)	Compare two operands, store result in STW
fcomp	mem/st(i)	mem/st(j)	Compare two operands, store result in STW and pop
fcomi	mem/st(i)	mem/st(j)	Compare two int operands, store result in STW
fcomip	mem/st(i)	mem/st(j)	Compare two int operands, store result in STW and pop

O que uma interrupção faz?

- Salva o status atual da CPU e para a atividade atual
- Chama uma subrotina específica
 - Dependendo da chamada da interrupção (0-255) o gerenciador de interrupções carrega um vetor de interrupções e pula para a subrotina correspondente.
- Se várias interrupções acontecem ao mesmo tempo existe uma lista de prioridades
- Quando a subrotina da interrupção termina deve-se retornar o status anterior do CPU e continuar com o processamento anterior.

Tipos de Interrupções

- Hardware Interno
 - Evento que ocorre durante a execução de um programa (e.g. divisão por zero, overflow, etc.)
- Hardware Externo
 - Evento que ocorre por um controlador de dispositivos externos (e.g. PCI/AGP bus, hard-drive, graphic cards, teclado, etc.)
- Software
 - Eventos produzidos por programas (normalmente o SO). Esses eventos podem ser chamados mediante a instrução **int**.

ID	Message
0x00	Division Error
0x01	Single Step Mode (Debug)
0x02	NMI Interrupt
0x03	Breakpoint
0x04	Overflow
0x05	Bound Range Exceeded
0x06	Invalid Opcode
0x07	Coprocessor Not Available
0x08	Double Exception
0x09	Coprocessor Segment Overrun
0x0a	Invalid Task State Segment
0x0b	Segment Not Present
0x0c	Stack Fault
0x0d	General Protection
0x0e	Page Fault
0x0f	Reserved
0x10	Coprocessor Error
0x11-0x1f	Reserved
0x12-0xffffffff	Coprocessor Error

Chamadas ao Sistema (System Calls)

System Call

- A chamada ao sistema é uma chamada para que o kernel do SO realize uma determinada tarefa.
- A chamada ao sistema é na verdade uma interrupção, porém utilizada para chamar uma subrotina do kernel do SO.

Formato

- Argumentos:
 - eax: Identificador da chamada ao sistema
 - ebx,ecx,edx,esi,edi: argumentos da chamada
- Chamada:
 - int 80h
- Exemplo:
 - mov eax,1
 - mov ebx,0
 - int 80h

%eax	Name	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char*	size_t	-	-
4	sys_write	unsigned int	const char*	size_t	-	-
5	sys_open	const char*	int	int	-	-
6	sys_close	unsigned int	-	-	-	-
7	sys_waitpid	pid_t	unsigned int	int	-	-
8	sys_create	const char*	int	-	-	-
9	sys_link	const char*	const char*	-	-	-
10	sys_unlink	const char*	-	-	-	-
11	sys_execve	struct pt_regs	-	-	-	-
12	sys_chdir	const char*	-	-	-	-
13	sys_time	int*	-	-	-	-
14	sys_mknod	const char*	mode_t	dev_t	-	-
15	sys_chmod	const char*	mode_t	-	-	-
...

Exemplo 1

```
section .data
msg db 'Hello Word!',0dh,0ah ;0dh+0ah é código do CR+LF
section .text
global _start
_start:  mov eax,4      ;system call ID (sys_write)
         mov ebx,1      ;primeiro argumento: file handler (stdout)
         mov ecx,msg    ;segundo argumento: ponteiro à string
         mov edx,13     ;terceiro argumento: tamanho da string
         int 80h        ;chamada à system call
         mov eax,1
         mov ebx,0
         int 80h
```

Exemplo 2

```
section .data
Snippet db 'KANGAROO',0dH,0ah
section .text
global _start
_start:  mov eax,4
         mov ebx,1
         mov ecx,Snippet
         mov edx,10
         int 80h
         mov ebx,Snippet
         mov eax,8
DoMore:  add byte [ebx],32
         inc ebx
         dec eax
         Jnz DoMore
         mov eax,4
         mov ebx,1
         mov ecx,Snippet
         mov edx,10
         int 80h
         mov eax,1
         mov ebx,0
         int 80h
```


Exemplo 3 - parte 1

```
section .data
name_msg      db      'Please enter your name: '
NAME_SIZE     EQU     $-name_msg
query_msg     db      'How many times to repeat welcome message? '
QUERY_SIZE    EQU     $-query_msg
confirm_msg1  db      'Repeat welcome message '
CONF1_SIZE    EQU     $-confirm_msg1
confirm_msg2  db      ' times? (y/n) '
CONF2_SIZE    EQU     $-confirm_msg2
welcome_msg   db      'Welcome to Assembly Language Programming '
WELC_SIZE     EQU     $-welcome_msg
nwnln         db      0Dh,0Ah
NWLN_SIZE     EQU     $-nwnln
section .bss
user_name     resb    16
response      resb    1
...
```

Exemplo 3 - parte 2

```
...  
section .text  
global _start  
_start:  
    mov eax,4  
    mov ebx,1  
    mov ecx,name_msg  
    mov edx,NAME_SIZE  
    int 80h  
    mov eax,3  
    mov ebx,0  
    mov ecx,user_name  
    mov edx,16  
    int 80h  
ask_count:  
    mov eax,4  
    mov ebx,1  
    mov ecx,query_msg  
    mov edx,QUERY_SIZE  
    int 80h  
    mov eax,3  
    mov ebx,0  
    mov ecx,response  
    mov edx,1  
    int 80h  
...
```

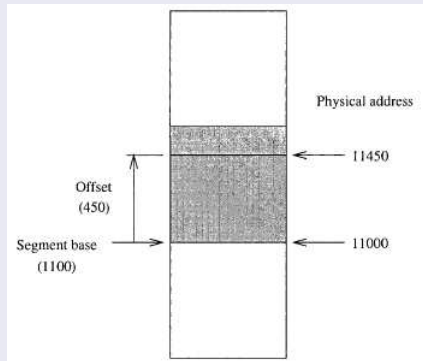
```
...  
    mov ECX,0  
    mov CL,[response]  
    sub CL,0x30  
display_msg:  
    push ECX  
    mov eax,4  
    mov ebx,1  
    mov ecx,welcome_msg  
    mov edx,WELCSIZE  
    int 80h  
    mov eax,4  
    mov ebx,1  
    mov ecx,user_name  
    mov edx,16  
    int 80h  
    mov eax,4  
    mov ebx,1  
    mov ecx,nwln  
    mov edx,NWLNSIZE  
    int 80h  
    pop ECX  
    loop display_msg  
    mov eax,1  
    mov ebx,0  
    int 80h
```

Acesso à Memória

- Os endereços da memória física são chamados **endereços físicos**, enquanto os endereços utilizados nos programas são chamados **endereços lógicos**, e dependem da forma de acessar a memória.
- A arquitetura IA-32 possui capacidade de endereçamento 4Gbytes (endereços físicos de 32 bits)
- Dois modos de acessar a memória:
 - Real mode (16 bits, compatibilidade com 8086)
 - Protected mode (32 bits, modo nativo de IA-32)

Real Mode

- Compatibilidade com 8086, somente 1M de memória é acessível (Endereço físico de 20 bits)
- A memória é acessada de forma segmentada, com segmentos de 16 bits e offsets de 16 bits.
- Utilizado somente quando é necessário guardar compatibilidade com 16 bits.



Endereço Físico e Lógico

- Os segmentos são diferenciados no uso: código, dados, pilha.
- Cada segmento tem tamanho de 64k
- Os segmentos podem estar em qualquer posição na memória
- Um programa pode acessar até 6 segmentos simultâneos

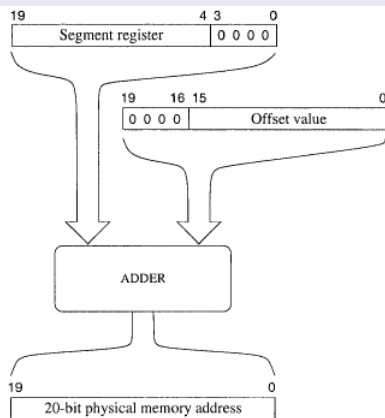
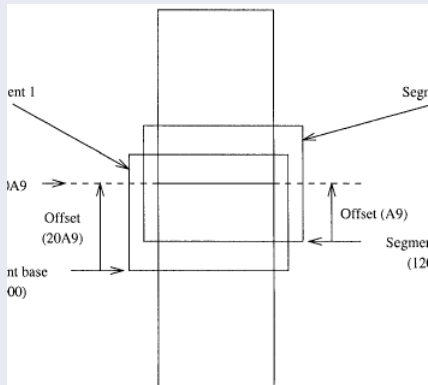


Figure 4.12 Physical address generation in the real mode.

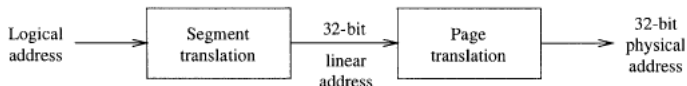
Segmentos

- Os segmentos são independentes, e podem ser contíguos, disjuntos, parcialmente ou totalmente sobrepostos

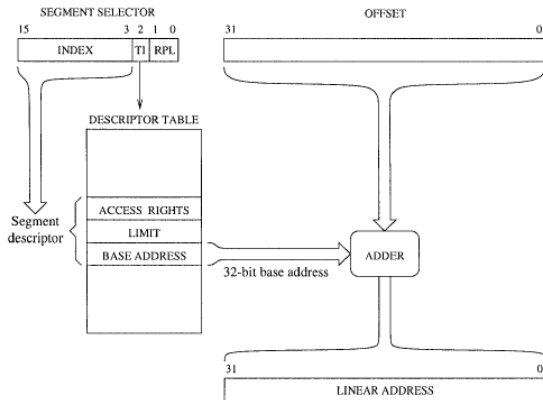


Modo Nativo - Protected Mode

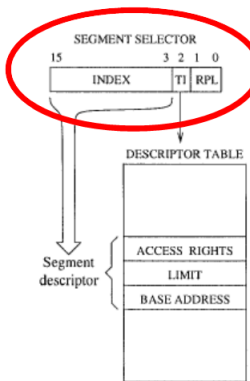
- Neste modo existe tanto paginação quanto segmentação
- Segmentação traduz endereços lógicos em endereços lineares.
- Paginação é utilizada para memória virtual
 - Transparentes para programas aplicativos.
 - Traduz endereços lineares em endereços físicos.



Segmentação



Segmentação

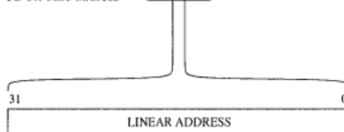


Seletor de Segmento

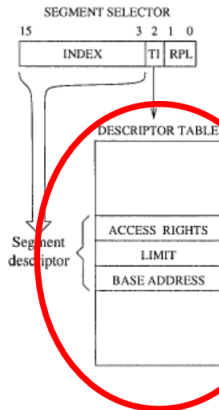
Contéudo armazenado em um registrador de segmentos (16 bits) (CS,SS, DS, etc).

- Índice na *descriptor table*, das informações do segmento de memória desejado.
- *Table indicator*, indicando se é a tabela local ou global
- *Requester privilege level*, indicando o nível de prioridade no acesso a memória.

32-bit base address



Segmentação



Descriptor Table

Global Descriptor Table: somente uma.
Dados disponíveis a todos os programas

Local Descriptor Table: cada programa tem sua tabela.

Um programa somente consegue acessar segmentos descritos na sua tabela local (ou na tabela global)

Cada item da tabela contém campos que descrevem um segmento:

Endereço inicial do segmento

Limite (tamanho) do segmento

Direitos de acesso ao segmento

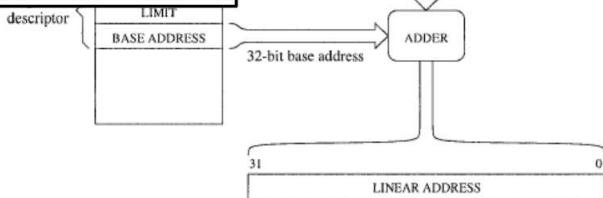
LINEAR ADDRESS

Segmentação

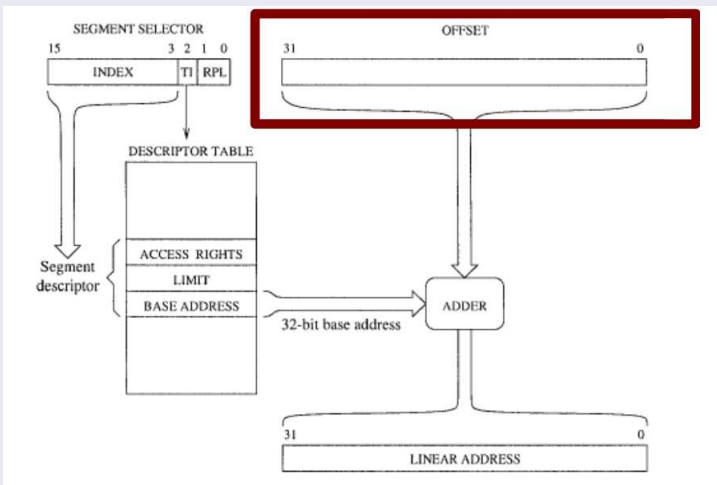
Cálculo do endereço linear

Valor do offset é somado ao endereço de base do segmento.

Offset pode ter 16 ou 32 bits.
Endereço de base sempre tem 32 bits.

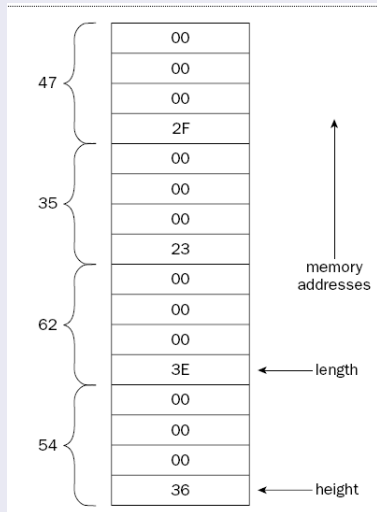


Offset: Determinado pelo Programa/aplicativo



Segmentos

- Utiliza *little endian*. O que significa que o byte menos significativo no endereço menor.
- Exemplo: height dd 54, length 62,35,47



Tamanhos Padrão

Nome	Tamanho em Bits	Tamanho em Bytes
Bit	1	-
Byte	8	1
Word	16	2
Double Word	32	4
Quad Word	64	8
Ten Byte	80	10
Parágrafo	480	16
Página	2048	256
Segmento	524280	65535

Próxima Aula

Modos de Endereçamento