Introdução à Software Básico: Introdução a Assembly IA-32

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

Sumário

Operadores Lógicos em Assembly

Instruções de operadores lógicos

- And
- Or
- Xor
- Not

Instruções de Deslocamento

- Shl, shr
- Sal, sar: inclui o bit de sinal no deslocamento

Deslocamento Aritmético

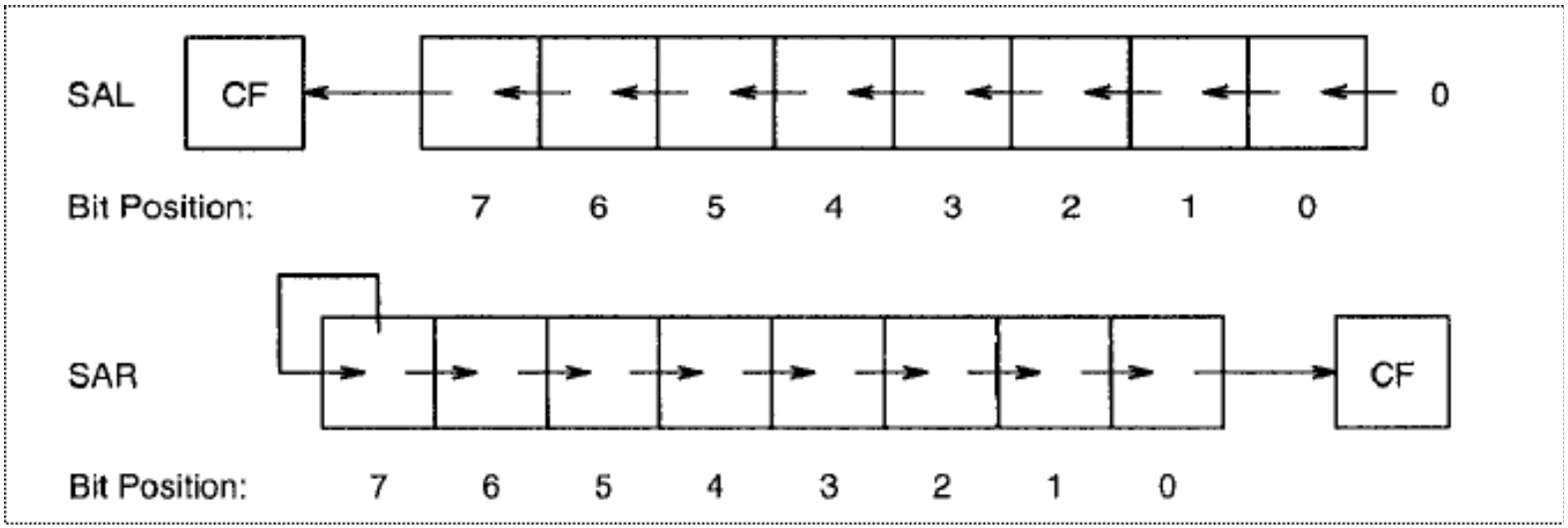


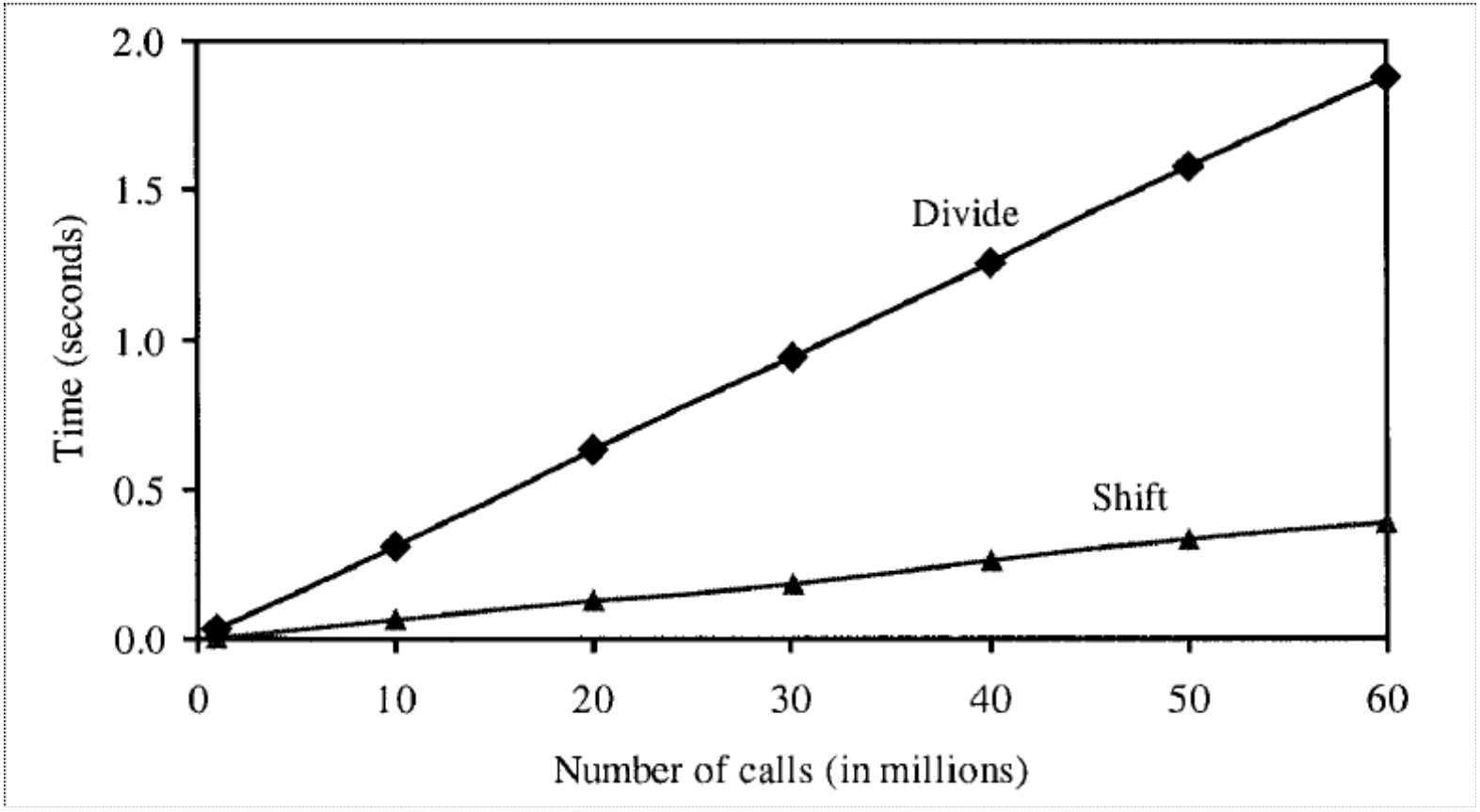
Figura: :

Deslocamento com SAL e SAR

Deslocamento

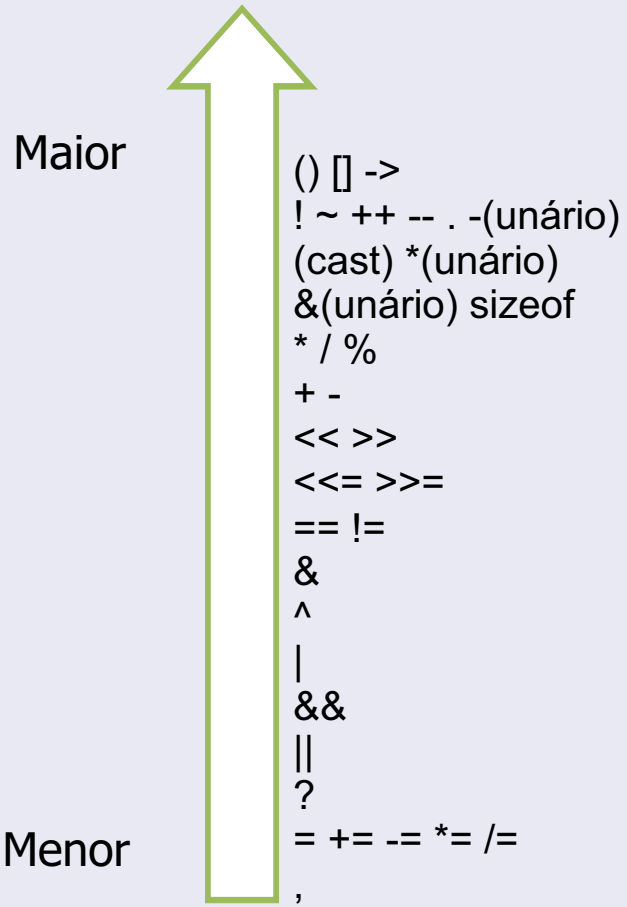
Porque deslocar?

- O Deslocamento pode ser utilizado para manipular BITS
- Além, disso o deslocamento é muito utilizado para multiplicar/dividir por 2



Precedência dos operadores

Em C



Em Assembly ia-32

A precedência é determinada pela ordem de execução da instrução

```
int main(){
int x = 0;
int a = 30;
int y = 17;
x = a+y*x;
}
```

```
int main(){
int x = 0;
int a = 30;
int y = 17;
x = (a+y)*x;
}
```

```
%include "io.mac"
.DATA
X DW 0
B DW 30
Y DW 17
.CODE
.STARTUP
MOV AX,[B]
ADD WORD AX,[Y]
MUL WORD [X]
.EXIT
```

Qual o valor do resultado da operação?

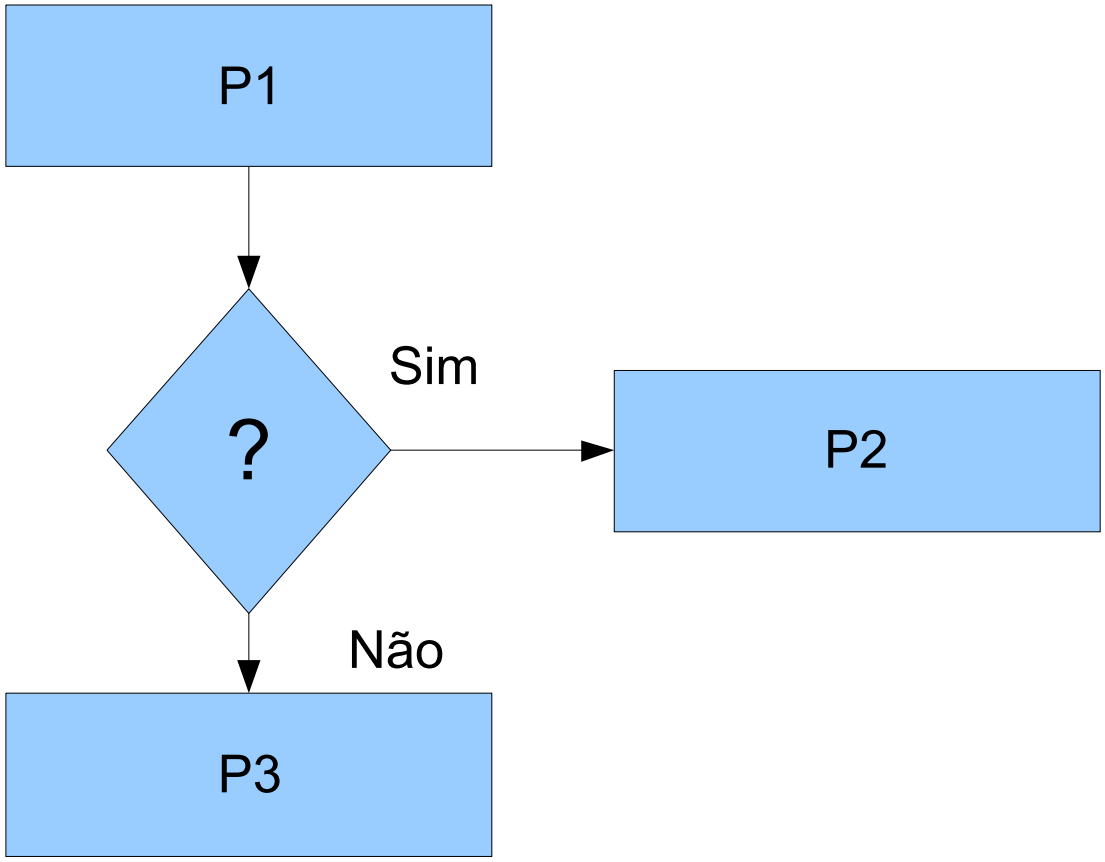
Onde está armazenado o resultado?

A qual versão em C corresponde o programa em Assembly?

O que fazer para tornar o programa idêntico ao programa em C?

Definição

Os comandos de controle de fluxo são comandos que permitem ao programador alterar a sequência de execução do programa.



Saltos

- A instrução **JMP** altera a sequência de forma incondicional
- Mas o controle de fluxo if necessita de uma condição a ser testada, logo existem saltos condicionais
- Condição é verificada pelo valor dos flags, alterados na última instrução aritmética ou lógica executada (normalmente **TEST** ou **CMP**).

Comando de fluxo de controle

Table 9-7: Arithmetic Tests Useful After a CMP Instruction

CONDITION	PASCAL OPERATOR	UNSIGNED VALUES	JUMPS WHEN	SIGNED VALUES	JUMPS WHEN
Equal	=	JE	ZF=1	JE	ZF=1
Not Equal	<>	JNE	ZF=0	JNE	ZF=0
Greater than	>	JA	CF=0 and ZF=0	JG	ZF=0 or SF=OF
Not Less than or equal to		JNBE	CF=0 and ZF=0	JNLE	ZF=0 or SF=OF
Less than	<	JB	CF=1	JL	SF<>OF
Not Greater than or equal to		JNAE	CF=1	JNGE	SF<>OF
Greater than or equal to	>=	JAE	CF=0	JGE	SF=OF
Not Less than		JNB	CF=0	JNL	SF=OF
Less than or equal to	<=	JBE	CF=1 or ZF=1	JLE	ZF=1 or SF<>OF
Not Greater than		JNA	CF=1 or ZF=1	JNG	ZF=1 or SF<>OF

Exemplos em Assembly de IF..ELSE..

```
if (count =100){
   <statement 1>
}
 <next statement>
```



```
cmp     count,100
jnz cont     ;or jne cont
<statement1 code here>
cont:
<next statement here>
```

Exemplos em Assembly de IF..ELSE..

```
if (count =100){
   <statement 1>
}
 <next statement>
```

→

```
cmp     count,100
jnz cont     ;or jne cont
<statement1 code here>
cont:
<next statement here>
```

```
if ( AX >= 5 )
BX = 1;
else
BX = 2;
```

→

```
cmp      ax, 5
jge      thenblock
mov      bx, 2
jmp      next
thenblock:
mov      bx, 1
next:
```

Exemplos em Assembly de IF..ELSE..

```
if (count =100){
   <statement 1>
}
 <next statement>
```

→

```
cmp     count,100
jnz cont     ;or jne cont
<statement1 code here>
cont:
<next statement here>
```

```
if ( AX >= 5 )
BX = 1;
else
BX = 2;
```

→

```
cmp     ax, 5
jge     thenblock
mov     bx, 2
jmp     next
thenblock:
mov     bx, 1
next:
```

Comparação com sinal: AX pode conter valor negativo!!

Exemplos Condicional aninhado

```
...
char str[] = "10";
if (str[0]=='0')
  if (str[1]=='0')
     printf("Zero");
  else
     printf("Um");
else /* str[0] == '1' */
  if (str[1]=='0')
     printf("Dois");
  else
     printf("Tres");
```

```
%include "io.mac"
.DATA
msg db 'Please enter binary number: ',0
msg0 db 'Zero!',0
msg1 db 'Um!',0
msg2 db 'Dois',0
msg3 db 'Tres',0
.UDATA
number resb 3
.CODE
.STARTUP
     PutStr  msg
     GetStr  number,3
     cmp BYTE [number],30h
     jne doistres
     cmp BYTE [number+1],30h
     jne um
zero:PutStr  msg0
     jmp Fim
um:  PutStr  msg1
     jmp Fim
doistres: cmp BYTE [number+1],30h
     jne tres
dois:PutStr  msg2
     jmp Fim
tres:PutStr msg3
Fim: .EXIT
```

Exemplos Condicional Encadeado

```
char s[2];
...
if (s[0] == '0')
 printf("Zero");
else if (s[0]=='1')
 printf("Um");
else if (s[0]=='2')
 printf("Dois");
else if ...
else if (s[0]=='9')
 printf("Nove");
else
printf("Nao era um digito!");
```

```
%include "io.mac"
.DATA
msg db 'Please enter number digit: ',0
msg0 db 'Zero!',0
msg1 db 'Um!',0
...
msg8 db 'Oito!',0
msg9 db 'Nove!',0
msg_ndig db 'não eh digito!',0
.UDATA
number resb 2
.CODE
.STARTUP
    PutStr  msg
    GetStr  number,2
    cmp BYTE [number],30h
    jne naozero
zero:PutStr  msg0
    jmp Fim
naozero:
    cmp BYTE [number],31h
    jne naoum
um:  PutStr  msg1
    jmp Fim

naooito:
    cmp BYTE [number],39h
    jne naodigito
nove: PutStr  msg9
        jmp Fim
naodigito:
    PutStr msg_ndig
Fim: .EXIT
```

Instrução LOOP em Assembly

- Grupo de instruções que utilizam CX ou ECX como contador para realizar repetições
  - *loop target*: decrementa ECX, se ECX for diferente de zero, salta para a posição do label target
  - *loope target*: decrementa ECX, e se ECX for diferente de zero e o flag Zero for um, salta para a posição do label target (também usada como loopz)
  - *loopne target*: decrementa ECX, e se ECX for diferente de zero e o flag Zero for zero, salta para a posição do label target (também usada como loopnz)
- Decrementar ECX não altera os flags

Comando de fluxo de controle

Mnemonic		Meaning	Action
`loop`	`target`	loop	ECX = ECX − 1 if CX ≠ 0 jump to target
`loope` `loopz`	`target` `target`	loop while equal loop while zero	ECX = ECX − 1 if (ECX ≠ 0 and ZF = 1) jump to target
`loopne` `loopnz`	`target` `target`	loop while not equal loop while not zero	ECX = ECX − 1 if (ECX ≠ 0 and ZF =0) jump to target

Exemplos loop

```
sum = 0;
for ( i=MAX; i >0; i-- )
   sum += i;
```

```
%include "io.mac"
.DATA
soma dw 0
MAX equ 10
.UDATA
i resw 1
.CODE
.STARTUP
      mov WORD [i],MAX
      mov WORD CX,[i]
lp:  add [soma],CX
      loop lp
fim: .EXIT
```

While usando Loop

```
int main (){
        int i = 0;
        while ( i < 100){
        printf(" %d", i);
                i++; }
        return(0); }
```

```
%include "io.mac"
.DATA
i dw 0
MAX equ 100
.CODE
    .STARTUP
    mov ECX,MAX
lp: PutCh 20
    PutInt [i]
    inc WORD [i]
    loop lp
.EXIT
```

Laços Aninhados em Assembly

- Pode-se escolher utilizar a instrução LOOP ou atualizar o contador e fazer o salto explicitamente

```
main() {
int i, j;
for (i = 0; i < 4; i++)
 for (j = 0; j < 2; j++)
  printf("i: %d j: %d\n",
    i, j);
}
```

```
i: 0 j: 0
i: 0 j: 1
i: 1 j: 0
i: 1 j: 1
i: 2 j: 0
i: 2 j: 1
i: 3 j: 0
i: 3 j: 1
```

```
%include "io.mac"
.DATA
i   db    0
MAXI equ   4
MAXJ equ   2
.UDATA
j   resb   1
.CODE
.STARTUP
lpi:
    mov BYTE [j],0
lpj:
    PutInt [i]
    PutCh  0x20
    PutInt [j]
    PutCh  0xA
    inc BYTE [j]
    cmp BYTE [j],MAXJ
    jb   lpj
    inc BYTE [i]
    cmp BYTE [i],MAXI
    jb   lpi
.EXIT
```

Break e Continue em Assembly

- Comandos de parada e continuação são realizados utilizando as instruções de saltos condicionais.
- Primeiro testa-se a condição, depois realiza o salto: para dentro do laço no caso do continue; para fora do laço no caso do break

Exemplo: BREAK

```
main () {
   int i, j;
   for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
     if (i == 1) break;
     else printf("i: %d j: %d\n",
   i, j);
   }
```

```
%include "io.mac"
.DATA
i   db     0
MAXI equ    4
MAXJ equ    2
.UDATA
j   resb   1
.CODE
.STARTUP
lpi: mov BYTE [j],0
lpj: cmp BYTE [i],1
    je    lpi2
    PutInt [i]
    PutCh  0x20
    PutInt [j]
    PutCh  0xA
    inc BYTE [j]
    cmp BYTE [j],MAXJ
    jb   lpj
lpi2:inc BYTE [i]
    cmp BYTE [i],MAXI
    jb   lpi
.EXIT
```

Exemplo: CONTINUE

```
main() {
  int i;
    for (i = 0; i < 5; i++)
      if (i == 1) continue;
      else printf("i: %d \n", i);
  }
```

```
%include "io.mac"
.DATA
msg  db     "i: ",0
i    db     0
MAXI equ    5
.CODE
.STARTUP
    mov  CX,MAXI
lpi: cmp BYTE [i],1
    je     lpi2
    PutStr msg
    PutInt [i]
    PutCh  0xA
lpi2:inc BYTE [i]
    loop lpi
.EXIT
```

Detalhamento das Instruções de Salto

- Tipos de instruções de salto por distância:
  - Salto curto (*short jump*)
  - Salto próximo (*near jump*)
  - Salto distante (*far jump*)
- Tipos de instruções de salto por endereço:
  - Relativo
  - Absoluto
- Tipos de instruções de salto por especificação do endereço:
  - Direto
  - Indireto

Detalhamento das Instruções de Salto

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump short, relative, displacement relative to next instruction
E9 *cw*	JMP *rel16*	Jump near, relative, displacement relative to next instruction
E9 *cd*	JMP *rel32*	Jump near, relative, displacement relative to next instruction
FF /4	JMP *r/m16*	Jump near, absolute indirect, address given in *r/m16*
FF /4	JMP *r/m32*	Jump near, absolute indirect, address given in *r/m32*
EA *cd*	JMP *ptr16:16*	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Jump far, absolute indirect, address given in *m16:32*

Near and Short Jumps

- **Near Jump**: Um salto dentro do mesmo segmento de código (o segmento apontado por CS).
- **Short Jump**: Um salto NEAR sendo que esta limitado a saltar entre -128 a +127 endereços da posição autal do EIP.

Funcionamento

Ao executar um salto próximo (NEAR), o processador pula para o endereço, dentro do segmento atual, indicado pelo operando. O operando indica um deslocamento *absoluto* ou *relativo*.

- O deslocamento absoluto é indicado mediante um offset para a base do segmento atual (CS).
- O deslocamento relativo é indicado mediante um offset com sinal que vai ser somado ao contador de instruções (EIP).

Comando de fluxo de controle

Near and Short Jumps

- **Near Jump**: Um salto dentro do mesmo segmento de código (o segmento apontado por CS).
- **Short Jump**: Um salto NEAR sendo que esta limitado a saltar entre -128 a +127 endereços da posição autal do EIP.

Funcionamento

- Um deslocamento próximo absoluto é especificado de forma *indireta* mediante um registrador ou posição de memória (r/m16 ou r/m32). O valor do operando é colocado no registrador EIP (se o operando é de 16 bits, os primeiros 16 bits de EIP são zerados).
- Um deslocamento próximo relativo é normalmente especificado mediante uma etiqueta ou rótulo. O rótulo é codificado como um número **com sinal** de 8,16 ou 32 bits. Este número é somado ao valor atual do EIP. EIP possue sempre o endereço da **próxima** instrução a ser exeutada. Se o operando é de 8 bits, o salto é do tipo curto (Short). Os saltos relativos são no geral mais rápidos que os absolutos. Os saltos curtos são os mais rápidos a serem executados.

Near and Short Jumps

- **Far Jump**: Um salta para uma instrução em um outro segmento de código.

Funcionamento em *Real Mode*

- Quando executado a instrução salta para o endereço indicado pelo operando. O operando especifica o endereço de forma direta mediante um ponteiro (ptr16:16 ou ptr16:32), ou de forma indireta por endereço de memória (m16:16 ou m16:32). No método direto o ponteiro deve ser indicado por um registrador como CS, ED, etc., e o offset indicado é carregado no registrador EIP. De forma indireta o endereço indicado pela memória é carregado no registrador CS e o offset no EIP.

Near and Short Jumps

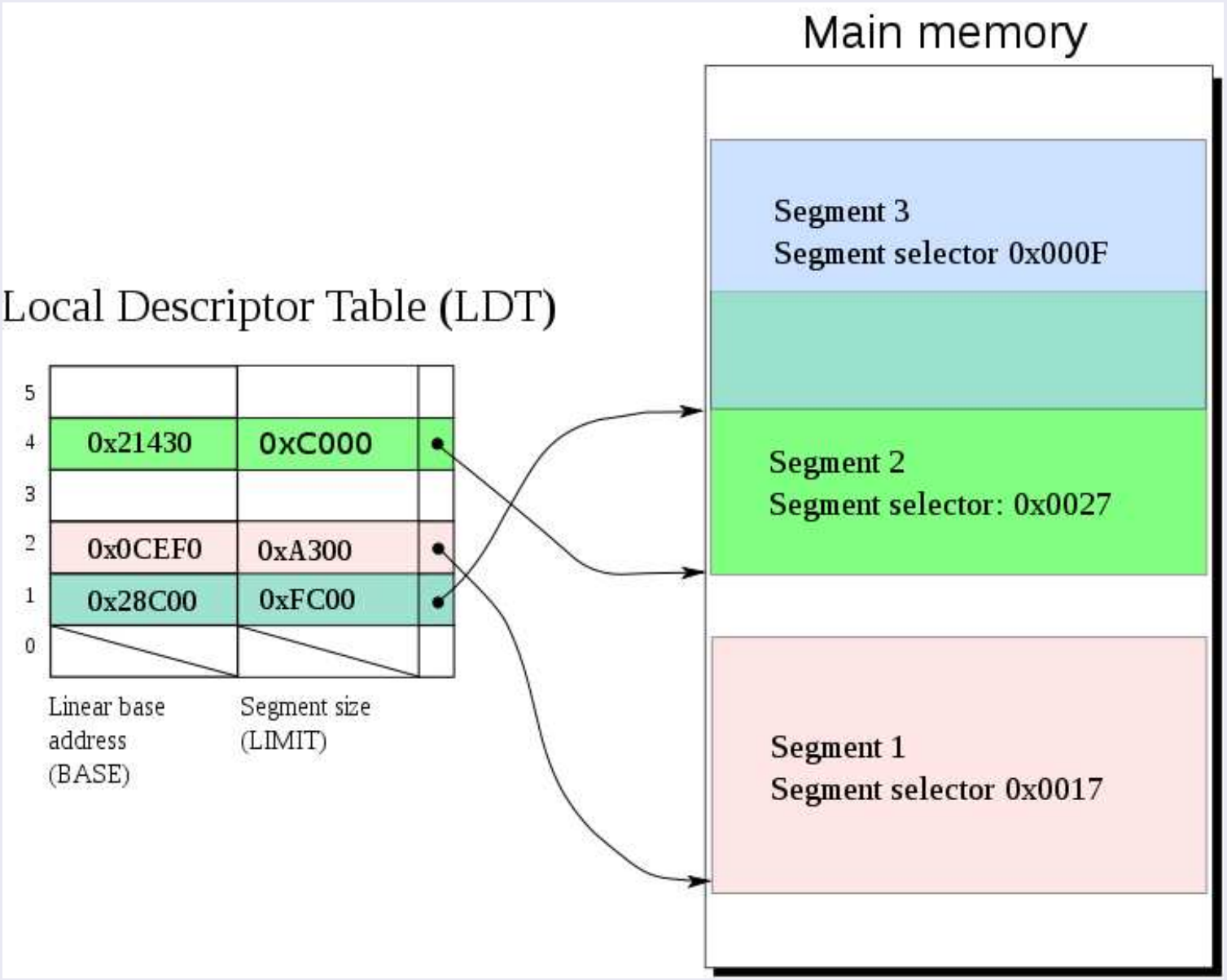
- **Far Jump**: Um salta para uma instrução em um outro segmento de código.

Funcionamento em *Protected Mode*

- Em modo nativo o salto distante pode ser utilizado para:
  - Pula distante similar ao Real Mode
  - *Call Gate*: Usados para deixar que códigos com menor privilegio possam acessar segmentos de códigos com maior privilegio. Um Call Gate é essencial nos sistemas operacionais atuais, por exemplo permitem que um código utilize funções e chamadas ao sistema do kernel.
  - *Task Switch*: Funcionamento similar ao CALL GATE mas utilizado para acessar segmentos de códigos de outras tarefas.

    No Call Gate e Task Switch o operando indica um ponteiro para a LDT(Local Description Table) que vai indicar o endereço de segmento base que vai ser somado ao offset do operando.

Detalhamento das Instruções de Salto



Próxima Aula

Continuação IA-32 (Procedimentos e Funções)