

# Assembly Language for Intel-Based Computers, 5<sup>th</sup> Edition

Kip R. Irvine

## Capítulo 6: Instruções Booleanas e Comparação Processamento Condicional

*Slides prepared by the author*

*Revision date: June 4, 2006*

(c) Pearson Education, 2006-2007. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.

# índice

- **Instruções Booleanas e de Comparação**
- Jumps condicionais
- Instruções de loop condicionais
- Estruturas condicionais
- Aplicação: máquinas de estado finito
- Diretivas de decisão

# Instruções booleanas e de comparação

- Flags de status da CPU
- Instrução AND
- Instrução OR
- Instrução XOR
- Instrução NOT
- Aplicações
- Instrução TEST
- Instrução CMP

# Flags de Status - Revisão

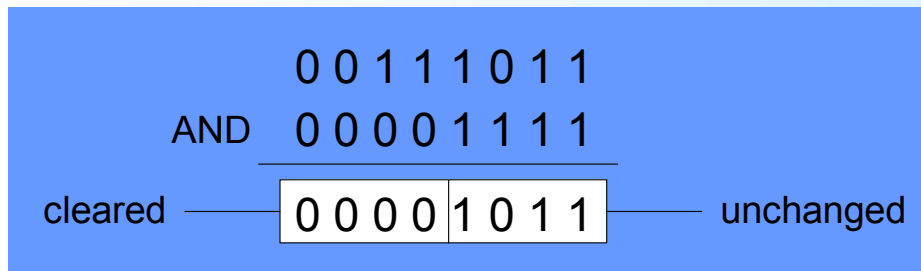
- O flag Zero é acionado quando o resultado de uma operação igual a zero.
- O flag de Carry é acionado quando uma instrução gera um resultado que é muito grande (ou muito pequeno) para o operando destino “nros sem sinal”.
- O flag de sinal é acionado quando o operando destino é negativo, e zerado quando o operando destino é positivo.
- O flag de Overflow é acionado quando uma instrução gera um resultado de sinal inválido ( carry do MSB XOR carry para o MSB) “nros com sinal”.
- O flag de paridade é acionado quando uma instrução gera um número par de bits 1 no byte mais à direita do operando destino;
- O flag Auxiliary Carry é acionado quando uma operação produz um vai-um do bit 3 para o bit 4

# Instrução AND

- Realiza uma operação booleana AND entre os bits correspondentes em dois operandos
- Sintaxe:

*AND destino, fonte*

(tipos de operando iguais a MOV)



AND

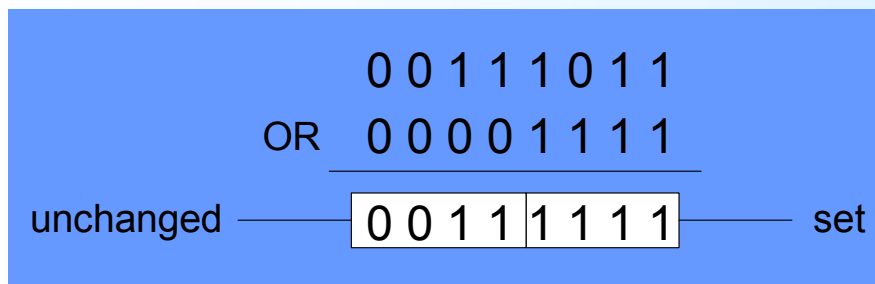
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

AND: uso típico → zerar determinados bits, preservando os outros

# Instrução OR

- Realiza uma operação booleana OR entre os bits correspondentes de dois operandos
- Sintaxe:

OR *destino, fonte*



OR

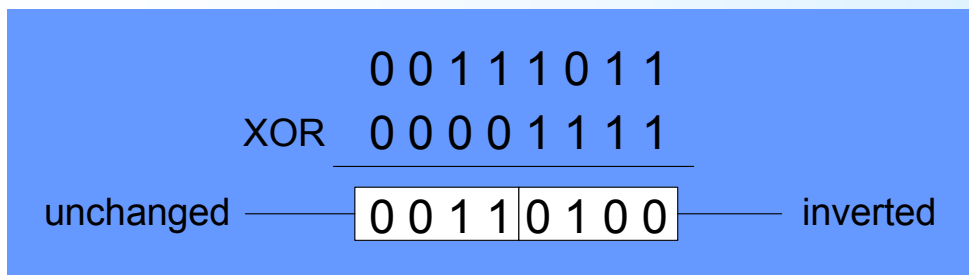
x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

OR: uso típico → setar determinados bits, preservando os outros

# Instrução XOR

- Realiza uma operação booleana OU-exclusivo entre os bits correspondentes de dois operandos
- Sintaxe:

*XOR destino, fonte*



XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR: uso típico → operações lógicas diversas  
Ex: trocar o valor de x e y, sem usar temp:

```
x = x ^ y ;  
y = x ^ y ;  
x = x ^ y ;
```

Propriedade:  
 $(A \text{ xor } B) \text{ xor } B = A$



# Instrução NOT

- Realiza uma operação booleana NOT no operando destino
- Sintaxe:

NOT *destino*

```
NOT  0 0 1 1 1 0 1 1
      ───────────
      1 1 0 0 0 1 0 0 ——— inverted
```

NOT

X	$\neg X$
F	T
T	F



# Aplicações - 1

- Tarefa: Converter o caractere em AL em maiúsculo.
- Solução: Usar a instrução AND para zerar o bit 5.

```
mov al, 'a'           ; AL = 01100001b  
and al, 11011111b     ; AL = 01000001b
```

*Código ASCII: letra minúscula = (letra maiúscula + 32)*

## Aplicações - 2

- Tarefa: Converter o valor binário de um byte no seu dígito decimal ASCII equivalente.
- Solução: Usar a instrução OR para acionar os bits 4 e 5.

```
mov al,6                      ; AL = 00000110b
or  al,00110000b             ; AL = 00110110b
```

Dígito ASCII '6' = 00110110b

*Código ASCII: dígito unitário= 48 + unidade*

## Aplicações - 3

- Tarefa: saltar a um label se um inteiro é par.
- Solução: fazer AND do bit menos significativo com 1. Se o resultado é zero, o número é par.

```
mov ax,wordVal
and ax,1                ; low bit set?
jz  EvenValue           ; jump if Zero flag set
```

Obs: instrução JZ = (jump se Zero)

Escrever um código que salta a um label se um inteiro é negativo.

## Aplicações - 4

- Tarefa: salta para um label se o valor em AL não é zero.
- Solução: fazer o OR do byte consigo mesmo, e usar a instrução JNZ (jump se not zero) .

```
or  al,al  
jnz IsNotZero          ; jump if not zero
```

Fazer o OR consigo mesmo não altera o valor.

# Instrução TEST

- Realiza uma operação **AND não-destrutiva** entre os bits correspondentes de dois operandos
- Nenhum operando é modificado, mas o flag de Zero é afetado.
- Exemplo: salta a um label se bit 0 ou bit 1 em AL é um.

```
test al,00000011b  
jnz  ValueFound
```

- Exemplo: salta a um label se nem bit 0 nem bit 1 em AL é um.

```
test al,00000011b  
jz   ValueNotFound
```

# Instrução CMP

- Compara o operando destino com o operando fonte
  - Subtração não-destrutiva , destino menos fonte (destino não é alterado)
- Sintaxe: `CMP destino, fonte`
- Exemplo: destino == fonte

```
mov al,5  
cmp al,5                ; Zero flag set
```

- Exemplo: destino < fonte

```
mov al,4  
cmp al,5                ; Carry flag set
```

# Instrução CMP

- Exemplo: destino > fonte

```
mov al,6  
cmp al,5 ; ZF = 0, CF = 0
```

(ambos os flags Zero e Carry são zerados)

# Instrução CMP

As comparações mostradas aqui são realizadas em inteiros com sinal.

- Exemplo: destino > fonte

```
mov al,5  
cmp al,-2           ; Sign flag == Overflow flag
```

- Exemplo: destino < fonte

```
mov al,-1  
cmp al,5           ; Sign flag != Overflow flag
```



# Próxima seção

- Instruções Booleanas e de Comparação
- **Jumps condicionais**
- Instruções de loop condicionais
- Estruturas condicionais
- Aplicação: máquinas de estado finito
- Diretivas de decisão

# Jumps condicionais

- Jumps são baseados em . . .
  - Flags específicos
  - Igualdade
  - Comparações sem sinal
  - Comparações com sinal
- Aplicações
- *Encrypting* de uma cadeia
- Instrução de teste de bit (BT)

# Instrução Jcond

- Uma instrução de jump condicional desvia a um label quando um valor específico de registrador ou condições de flag são encontrados
- Exemplos:
  - JB, JC jump se o Carry flag é acionado
  - JE, JZ jump se o Zero flag é acionado
  - JS jump se o Sign flag é acionado
  - JNE, JNZ jump se o Zero flag é zerado
  - JECXZ jump se ECX igual a 0

## Intervalo de Jcond

- Antes de 386:
  - jump deve ser entre  $-128$  a  $+127$  bytes do valor contido no ponteiro de instrução
- Processadores IA-32:
  - Offset de 32-bit offset permite jump em qualquer localização na memória

## Jumps baseados em flags específicos

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

# Jumps baseados na igualdade

Mnemonic	Description
JE	Jump if equal ( <i>leftOp = rightOp</i> )
JNE	Jump if not equal ( <i>leftOp <math>\neq</math> rightOp</i> )
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

# Jumps baseados em comparações sem sinal

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$ )
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if $leftOp \geq rightOp$ )
JNB	Jump if not below (same as JAE)
JB	Jump if below (if $leftOp < rightOp$ )
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if $leftOp \leq rightOp$ )
JNA	Jump if not above (same as JBE)

# Jumps baseados em comparações com sinal

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$ )
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightOp$ )
JNL	Jump if not less (same as JGE)
JL	Jump if less (if $leftOp < rightOp$ )
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if $leftOp \leq rightOp$ )
JNG	Jump if not greater (same as JLE)

**Lembrete: Usar jump logo após cmp**



# Aplicações - 1

- Tarefa: saltar ao label se o EAX sem sinal é maior que EBX
- Solução: Usar CMP, seguida de JA

```
cmp  eax,ebx  
ja   Larger
```

- Tarefa: saltar a um label se o EAX com sinal é maior que EBX
- Solução: Usar CMP seguida de JG

```
cmp  eax,ebx  
jg   Greater
```

## Aplicações - 2

- Saltar para o label L1 se EAX sem sinal é menor ou igual a Val1

```
cmp eax,Val1  
jbe L1           ; below or equal
```

- Saltar para o label L1 se EAX com sinal é menor ou igual a Val1

```
cmp eax,Val1  
jle L1
```

## Aplicações - 3

- Comparar AX com BX sem sinal e copiar o maior entre eles na variável denotada Large

```
mov Large,bx
cmp ax,bx
jna Next
mov Large,ax
Next:
```

Isto nada mais é  
que o comando “if”

- Compare AX com BX com sinal e copiar o menor dentre eles numa variável denotada Small

```
mov Small,ax
cmp bx,ax
jnl Next
mov Small,bx
Next:
```

## Aplicações - 4

- Saltar para o label L1 se a palavra de memória apontada por ESI é igual a Zero

```
cmp WORD PTR [esi],0  
je L1
```

- Saltar para o label L2 se o doubleword de memória apontado por EDI é par

```
test DWORD PTR [edi],1  
jz L2
```

## Aplicações - 5

Tarefa: saltar para o label L1 se os bits 0, 1 e 3 em AL estão todos acionados.

- Solução: Zerar todos os bits, exceto 0, 1 e 3. Então, comparar o resultado com 00001011 b.

```
and al,00001011b      ; clear unwanted bits
cmp al,00001011b      ; check remaining bits
je  L1                ; all set? jump to L1
```

## Sua vez . . .

- Escrever um código que salta ao label L1 se ou bit 4, 5 ou 6 é acionado no registrador BL.
- Escrever um código que salta ao label L1 se bits 4, 5 e 6 estão todos acionados no registrador BL.
- Escrever um código que salta ao label L2 se AL tem paridade par.
- Escrever um código que salta ao label L3 se EAX é negativo.
- Escrever um código que salta ao label L4 se a expressão  $(EBX - ECX)$  é maior que zero.

# Criptografia de uma cadeia

O seguinte loop usa a instrução XOR para transformar cada caractere da cadeia num novo valor.

```
KEY = 239                                ; can be any byte value
BUFMAX = 128
.data
buffer  BYTE BUFMAX+1 DUP(0)
bufSize DWORD BUFMAX

.code
    mov ecx,bufSize                      ; loop counter
    mov esi,0                            ; index 0 in buffer
L1:
    xor buffer[esi],KEY                  ; translate a byte
    inc esi                              ; point to next byte
    loop L1
```

# Programa de criptografia de uma cadeia

- Tarefas:
  - Entrar com uma mensagem (cadeia) pelo teclado
  - *criptografia* da mensagem
  - Mostrar a mensagem *criptografada*
  - *Decriptografia* da mensagem
  - Mostrar a mensagem *decriptografada*

Enter the plain text: Attack at dawn.

Cipher text: «ççÄîä-Äç-ïÄÿü-Gs

Decrypted: Attack at dawn.



# Instrução BT (Bit Test)

- Copia bit  $n$  de um operando no flag Carry
- Sintaxe: BT *bitBase*,  $n$ 
  - bitBase pode ser  $r/m16$  ou  $r/m32$
  - $n$  pode ser  $r16$ ,  $r32$  ou  $imm8$
- Exemplo: saltar ao label L1 se bit 9 é acionado no registrador AX:

<pre>bt AX,9</pre>	<pre>; CF = bit 9</pre>
<pre>jc L1</pre>	<pre>; jump if Carry</pre>

# Próxima seção

- Instruções Booleanas e de Comparação
- Jumps condicionais
- **Instruções de loop condicionais**
- Estruturas condicionais
- Aplicação: máquinas de estado finito
- Diretivas de decisão



*Intervalo ?*

# Instruções de loop Condicional

- LOOPZ e LOOPE
- LOOPNZ e LOOPNE

# LOOPZ e LOOPE

- Sintaxe:

LOOPE *destino* →  
LOOPZ *destino* → instruções idênticas

- Lógica:

- $ECX \leftarrow ECX - 1$
- se  $ECX > 0$  e  $ZF=1$ , salta para o destino
- Útil quando se rastreia um vetor para encontrar o primeiro elemento que **não coincide** com um dado valor.

semelhante à instrução loop, porém leva também em conta o flag sezo

# LOOPNZ e LOOPNE

- LOOPNZ (LOOPNE) é uma instrução de loop condicional
- Sintaxe:  
    LOOPNE *destino* →  
    LOOPNZ *destino* → *instruções idênticas*
- Lógica:
  - $ECX \leftarrow ECX - 1$ ;
  - se  $ECX > 0$  e  $ZF=0$ , salta para o destino
- Útil quando se rastreia um vetor em busca do primeiro elemento que *coincide* com um dado valor.

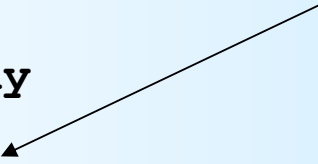
semelhante à instrução loop, porém leva também em conta o flag sezo

# LOOPNZ Exemplo

O seguinte código encontra o primeiro valor não negativo num vetor:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h ; test sign bit
    pushfd                    ; push flags on stack
    add esi,TYPE array
    popfd                     ; pop flags from stack
    loopnz next               ; continue loop
    jnz quit                  ; none found
    sub esi,TYPE array        ; ESI points to value
quit:
```

and c/ o bit de sinal +/-



## Sua vez . . .

Localizar o primeiro valor diferente de zero no vetor. Se nenhum valor for encontrado, fazer com que ESI aponte para o valor de nao\_achou:

```
.data
array SWORD 50 DUP(?)
nao_achou SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0           ; check for zero

    (fill in your code here)

quit:
```

## ... (solução)

```
.data
array  SWORD 50 DUP(?)
nao_achou  SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1:  cmp WORD PTR [esi],0           ; check for zero
    pushfd                        ; push flags on stack
    add esi,TYPE array
    popfd                         ; pop flags from stack
    loope L1                     ; continue loop
    jz quit                      ; none found
    sub esi,TYPE array           ; ESI points to value
quit:
```



## Próxima seção

- Instruções Booleanas e de Comparação
- Jumps condicionais
- Instruções de loop condicionais
- Estruturas condicionais
- Aplicação: máquinas de estado finito
- Diretivas de decisão

# Estruturas Condicionais

- Comando IF estruturado em blocos
- Expressões compostas com AND
- Expressões compostas com OR
- Loops WHILE
- Seleção baseada em tabela

# Comando IF estruturado em blocos

Programadores de linguagem Assembly podem facilmente traduzir comandos lógicos em C++/Java para linguagem Assembly. Por exemplo:

```
if( op1 == op2 )  
    X = 1;  
else  
    X = 2;
```

```
mov  eax,op1  
cmp  eax,op2  
jne  L1  
mov  X,1  
jmp  L2  
L1:  mov  X,2  
L2:
```

## Sua vez . . .

Implementar o código em linguagem Assembly. Todos os valores são sem sinal:

```
if( ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

(existem várias soluções corretas para esse problema)

## Sua vez . . .

Implementar o código em linguagem Assembly. Todos os valores são sem sinal:

```
if( ebx <= ecx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
cmp ebx,ecx  
ja  next  
mov eax,5  
mov edx,6  
next:
```

(existem várias soluções corretas para esse problema)

## Sua vez . . .

Implementar o seguinte código em linguagem assembly. Todos os valores são inteiros com sinal de 32-bits:

```
if( var1 <= var2 )  
    var3 = 10;  
else  
{  
    var3 = 6;  
    var4 = 7;  
}
```

(existem várias soluções corretas para esse problema.)

## Sua vez . . .

Implementar o seguinte código em linguagem assembly. Todos os valores são inteiros com sinal de 32-bits:

```
if( var1 <= var2 )  
    var3 = 10;  
else  
{  
    var3 = 6;  
    var4 = 7;  
}
```

```
mov eax,var1  
cmp eax,var2  
jle L1  
mov var3,6  
mov var4,7  
jmp L2  
L1: mov var3,10  
L2:
```

(existem várias soluções corretas para esse problema.)

# Expressão composta com AND - 1

- Quando se implementa o operador lógico AND, considerar que compiladores p/ linguagens de alto nível usam avaliações simplificadas.
- No exemplo seguinte, se a primeira expressão é falsa, a segunda expressão é desviada:

```
if (a1 > b1) AND (b1 > c1)  
    x = 1;
```





## Expressão composta com AND - 2

```
if (a1 > b1) AND (b1 > c1)
    X = 1;
```

Esta seria uma possível implementação ...

```
    cmp  a1,b1                ; first expression...
    ja   L1
    jmp  next
L1:
    cmp  b1,c1                ; second expression...
    ja   L2
    jmp  next
L2:                            ; both are true
    mov  X,1                  ; set X to 1
next:
```

Estratégia: continua tentando se estiver indo bem

[Web site](#) [Examples](#)

## Expressão composta com AND - 3

```
if (a1 > b1) AND (b1 > c1)
    X = 1;
```

Mas a seguinte implementação usa 29% (2/7) menos código revertendo o primeiro operador relacional. Permite-se ao programa desviar do segundo jump:

```
    cmp al,b1                ; first expression...
    jbe next                ; quit if false
    cmp bl,cl                ; second expression...
    jbe next                ; quit if false
    mov X,1                 ; both are true
next:
```

Estratégia: desiste assim que der errado  
técnica conhecida como fall-through

## Sua vez . . .

Implementar o seguinte pseudocódigo em linguagem assembly. Todos os valores são sem sinal:

```
if( ebx <= ecx
    && ecx > edx )
{
    eax = 5;
    edx = 6;
}
```

(existem várias soluções corretas para esse problema.)

## Sua vez . . .

Implementar o seguinte pseudocódigo em linguagem assembly. Todos os valores são sem sinal:

```
if( ebx <= ecx
    && ecx > edx )
{
    eax = 5;
    edx = 6;
}
```

```
cmp ebx,ecx
ja  next
cmp ecx,edx
jbe next
mov eax,5
mov edx,6
next:
```

(existem várias soluções corretas para esse problema.)

# Expressão composta com OR - 1

- No exemplo seguinte, se a primeira expressão é verdadeira, a segunda expressão é desviada:

```
if (a1 > b1) OR (b1 > c1)  
    x = 1;
```



## Expressão composta com OR - 2

```
if (a1 > b1) OR (b1 > c1)
    X = 1;
```

Pode-se usar a lógica "fall-through" fazendo o código mais curto possível:

```
    cmp al,b1                ; is AL > BL?
    ja  L1                   ; yes
    cmp bl,cl                ; no: is BL > CL?
    jbe next                 ; no: skip next statement
L1: mov X,1                   ; set X to 1
next:
```

# Loops WHILE

Um loop WHILE é na verdade um comando IF seguido de um corpo de loop, seguido de um salto incondicional ao topo do loop. Considerar o seguinte exemplo:

```
while( eax < ebx)
    eax = eax + 1;
```

Essa é uma possível implementação:

```
top: cmp  eax, ebx           ; check loop condition
     jae  next              ; false? exit loop
     inc  eax               ; body of loop
     jmp  top               ; repeat the loop
next:
```

## Sua vez . . .

Implementar o seguinte loop, usando inteiros de 32-bits sem sinal:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```



## Sua vez . . .

Implementar o seguinte loop, usando inteiros de 32-bits sem sinal:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
top: cmp ebx, val1          ; check loop condition
     ja  next               ; false? exit loop
     add ebx, 5              ; body of loop
     dec val1
     jmp top                 ; repeat the loop
next:
```

## Próxima seção

- Instruções Booleanas e de Comparação
- Jumps condicionais
- Instruções de loop condicionais
- Estruturas condicionais
- Aplicação: máquinas de estado finito
- Diretivas de decisão

# Diretiva de Decisão e Repetição

- Diretivas do MASM para facilitar a implementação de estruturas condicionais e de repetição;
- São traduzidas pelo Assembler em comandos assembly correspondentes;
- Facilitam a programação, mas diminuem a portabilidade do código
- Não utilizaremos esse tipo de diretivas neste curso
- *\*próximos slides servem apenas como referência*

# Usando a diretiva .IF

não utilizadas no curso

- Expressões de tempo de execução (Runtime)
- Operadores relacionais e lógicos
- Código gerado pelo MASM
- Diretiva .REPEAT
- Diretiva .WHILE

# Expressões de tempo de execução (Runtime)

não utilizadas no curso

- .IF, .ELSE, .ELSEIF e .ENDIF podem ser usados para avaliar expressões de tempo de execução e criar comandos IF estruturados em blocos .
- Exemplos:

```
.IF eax > ebx  
    mov edx,1  
.ELSE  
    mov edx,2  
.ENDIF
```

```
.IF eax > ebx && eax > ecx  
    mov edx,1  
.ELSE  
    mov edx,2  
.ENDIF
```

- MASM gera um código “escondido” que consiste de labels de código, instruções CMP e salto condicional.

# Operadores relacionais e lógicos

não utilizadas no curso

Operator	Description
<i>expr1</i> == <i>expr2</i>	Returns true when <i>expression1</i> is equal to <i>expr2</i> .
<i>expr1</i> != <i>expr2</i>	Returns true when <i>expr1</i> is not equal to <i>expr2</i> .
<i>expr1</i> > <i>expr2</i>	Returns true when <i>expr1</i> is greater than <i>expr2</i> .
<i>expr1</i> >= <i>expr2</i>	Returns true when <i>expr1</i> is greater than or equal to <i>expr2</i> .
<i>expr1</i> < <i>expr2</i>	Returns true when <i>expr1</i> is less than <i>expr2</i> .
<i>expr1</i> <= <i>expr2</i>	Returns true when <i>expr1</i> is less than or equal to <i>expr2</i> .
! <i>expr</i>	Returns true when <i>expr</i> is false.
<i>expr1</i> && <i>expr2</i>	Performs logical AND between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i>    <i>expr2</i>	Performs logical OR between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i> & <i>expr2</i>	Performs bitwise AND between <i>expr1</i> and <i>expr2</i> .
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

[Web Site](#) [Examples](#)

# Código gerado pelo MASM

não utilizadas no curso

```
.data
val1    DWORD 5
result  DWORD ?
.code
mov eax,6
.IF eax > val1
    mov result,1
.ENDIF
```

Código gerado:

```
mov eax,6
cmp eax,val1
jbe @C0001
mov result,1
@C0001:
```

MASM automaticamente gera um salto sem sinal porque val1 é sem sinal.

# Código gerado pelo MASM

não utilizadas no curso

```
.data  
val1    SDWORD 5  
result SDWORD ?  
.code  
mov eax,6  
.IF eax > val1  
    mov result,1  
.ENDIF
```

Código gerado:

```
mov eax,6  
cmp eax,val1  
jle @C0001  
mov result,1  
@C0001:
```

MASM automaticamente gera um salto com sinal (JLE) porque val1 é com sinal.



# Código gerado pelo MASM

não utilizadas no curso

```
.data
result DWORD ?
.code
mov ebx,5
mov eax,6
.IF eax > ebx
    mov result,1
.ENDIF
```

Código gerado :

```
mov ebx,5
mov eax,6
cmp eax,ebx
jbe @C0001
mov result,1
@C0001:
```

MASM automaticamente gera um salto sem sinal (JBE) quando ambos os operandos são registradores . . .

# Código gerado pelo MASM

não utilizadas no curso

```
.data
result SDWORD ?
.code
mov ebx,5
mov eax,6
.IF SDWORD PTR eax > ebx
    mov result,1
.ENDIF
```

Código gerado:

```
mov ebx,5
mov eax,6
cmp eax,ebx
jle @C0001
mov result,1
@C0001:
```

... a menos que um dos operandos de registradores seja prefixado com o operador SDWORD PTR . Então, um salto com sinal é gerado.

# Diretiva .REPEAT

não utilizadas no curso

Executa o corpo do loop antes do teste da condição associado com a diretiva .UNTIL .

Exemplo:

```
; Display integers 1 - 10:

mov eax,0
.REPEAT
    inc eax
    call WriteDec
    call Crlf
.UNTIL eax == 10
```

# Diretiva .WHILE

não utilizadas no curso

Teste a condição do loop antes de executar o corpo do loop. A diretiva .ENDW marca o fim do loop.

Exemplo:

```
; Display integers 1 - 10:

mov eax,0
.WHILE eax < 10
    inc eax
    call WriteDec
    call Crlf
.ENDW
```

# The End

