Aula 7 -Processamento Condicional
Arquitetura de Computadores I

Prof. MSC. Wagner Guimarães Al-Alam

Universidade Federal do Ceará
Campus de Quixadá

2017-1

Agenda

- Instruções de Comparação e Booleanas
- Saltos Condicionais
- Instruções de Laço Condicional
- Estruturas Condicionais
- Aplicação: Máquinas de Estados Finitos
- Diretivas de Controle de Fluxo Condicional

Instruções de Comparação e Booleanas

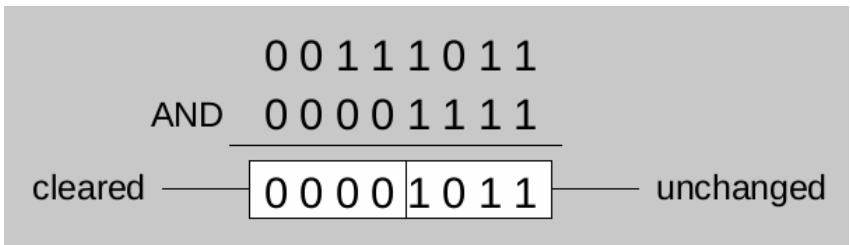
- Flags de Status de CPU
- Instrução AND
- Instrução OR
- Instrução XOR
- Instrução NOT
- Aplicações
- Instrução TEST
- Instrução CMP

Revisão - Status Flags

- A **Zero flag** é definida quando o resultado da operação é zero.
- A **Carry flag** é definida quando uma instrução gera um resultado que é muito grande (ou muito pequeno) para o operando destino.
- A **Sign flag** é definida se o operando destino é negativo após a operação, e está limpo quando o operando destino é positivo.
- A **Overflow flag** é definida quando uma instrução gera um resultado com sinal inválido.
- A **Parity flag** é definida quando uma instrução gera uma quantidade par de bits 1 no byte menos significativo do operando de destino.
- A **Auxiliary Carry flag** é definida quando uma operação produz um deslocamento do bit 3 para o bit 4

Instrução AND

- Efetua a operação Booleana AND entre cada par de respectivos bits em dois operandos
- Sintaxe:
  AND destination, source
  (same operand types as MOV)

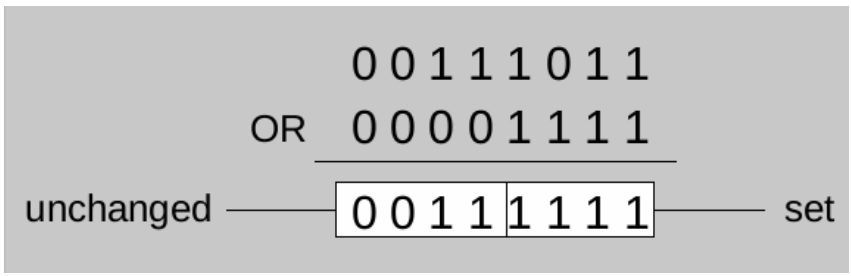


AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

Instrução OR

- Efetua a operação Booleana OR entre cada par de respectivos bits em dois operandos
- Sintaxe:
  OR destination, source

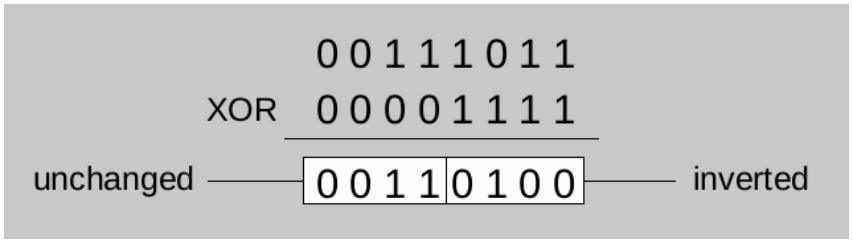


OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

Instrução XOR

- Efetua a operação Booleana exclusive-OR entre cada par de respectivos bits em dois operandos
- Sintaxe:
  XOR destination, source



XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR é um caminho útil para inveret os bits em um operando.

Instrução NOT

- Efetua a operação Booleana NOT em um único operando de destino
- Sintaxe:
  NOT destination

```
NOT   00111011
      ---------
      11000100 ——— inverted
```

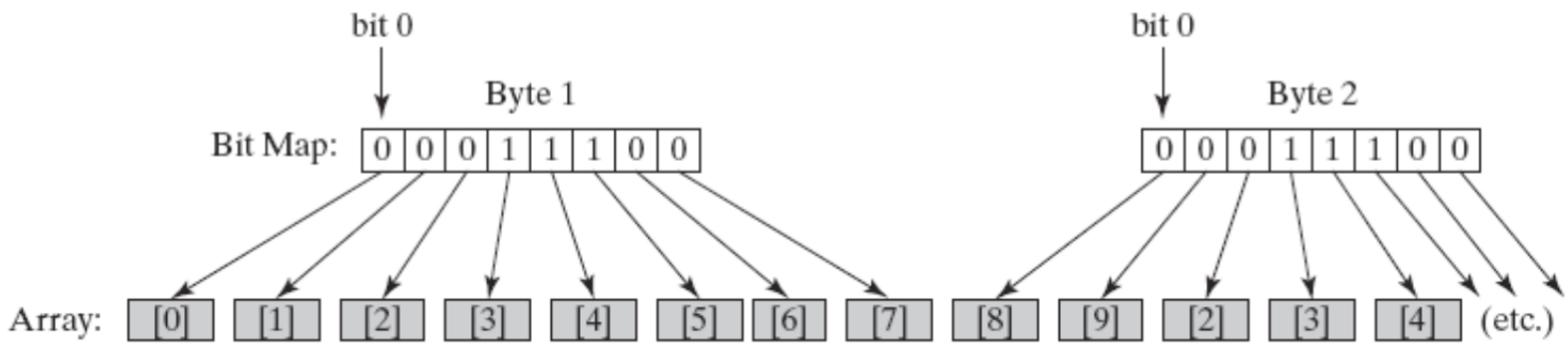
NOT

X	¬X
F	T
T	F

Conjuntos de Mapeamento de Bits

- Bits binários indicam membros de conjuntos
- Uso eficiente de armazeamento
- Também conhecido como vetore de bits

FIGURE 6–1 Mapping Binary Bits to an Array.



Operações sobre Conjuntos de Mapeamento de Bits

- Complemento de conjunto
  mov eax,SetX
  not eax
- Interseção de conjuntos
  mov eax,setX
  and eax,setY
- União de conjuntos
  mov eax,setX
  or eax,setY

Aplicações (1/5)

- Tarefa: Converter o caractere em AL para caixa alta.
- Solução: Usar a instrução AND para limpar o bit 5.

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

Aplicações (2/5)

- Tarefa: Converter um byte decimal em binário em seu digito equivalente ASCII Decimal.
- Solução: Usar a instrução OR para definir os bits 4 e 5.

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

O dígito '6' em ASCII = 00110110b

Aplicações (3/5)

- Tarefa: Acender a luz da tecla CapsLock no teclado
- Solução: Usar a instrução OR para definir o bit 6 do byte da flag de teclado em 0040:0017h na área de dados da BIOS.

```
1 mov ax,40h                ; BIOS segment
2 mov ds,ax
3 mov bx,17h                ; keyboard flag byte
4 or [bx],01000000b         ; CapsLock on
```

Este código somente executa em modo de endereçamento real, e não funciona no Windows NT, 2000, or XP.

Aplicações (4/5)

- Tarefa: Salta para uma etiqueta se um inteiro é par.
- Solução: AND o bit menos significativo e 1. Se o resultado é zero, o número é par.

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

JZ (jump if Zero)

Sua Vez

Escreva um código que salta para uma etiqueta se o inteiro é negativo.

Aplicações (5/5)

- Tarefa: Saltar para uma etiqueta se o valor em AL não é zero.
- Solução: OR do byte com ele mesmo, então usar a instrução JNZ (jump if not zero).

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

Cerque qualquer número com ele mesmo e seu valor não muda.

Instrução TEST

- Efetua a operação não destrutiva AND entre cada par de bits de máquina em dois operandos.
- Nenhum operando é modificado, mas a ZeroFlag é afetada.
- Examplo: Salte para uma etiqueta se se o bit 0 ou bit 1 em AL é definido.

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

- Examplo: salte para uma etiqueta se ambos bits 0 nor 1 em AL não estão definidos.

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

Instrução CMP 1/3)

- Compara o operando de destino ao operando de origem.
- Subtração não destrutiva da fonte pelo destino(operando destino não é alterado)
- Sintaxe:
  CMP destination, source
- Exemplo: destination == source

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

- Exemplo: destination < source

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

Instrução CMP (2/3)

- Exemplo: destination > source

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

(ambas as flags Zero e Carry são limpas)

Instrução CMP (3/3)

As comparações mostram que aqui são efetuadas usando inteiros com sinal integers.

- Exemplo: destination > source

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

- Exemplo: destination < source

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

Instruções Booleanas no modo 64-Bit

- instruções boobleanas de 64-bit, na maioria dos casos trabalham do mesmo modo que as instruções de 32 bits.
- Se o operando fonte é constante cujo tamanho é menor que 32 bits e o destno é a menor parte dos registradores de 64-bit em operandos de memória, todos os bits no operando de destino são afetados.
- Quando a fonte é uma constante de 32-bit ou registrador, somente os 32 bits menos significativos do operando destino são afetados.

Saltos Condicionais

- Saltos baseados em . . .
  - Flags Específicas
  - Igualdade
  - Comparações Unsigned
  - Comparações Signed
- Aplicações
- Encriptando uma String
- Instrução Bit Test (BT)

Jcond Instruction

- Uma instrução de salto condicional salta para uma etiqueta quando um registrador específico ou condição de flag é encontrada.
- Saltos Específicos:
  - JB, JC - salta para uma etiqueta se a Carry flag está ativada
  - JE, JZ - salta para uma etiqueta se a Zero flag está ativada
  - JS - salta para uma etiqueta se a Sign flag está ativada
  - JNE, JNZ - salta para uma etiqueta se a Zero flag está desativada
  - JECXZ - salta para uma etiqueta se a ECX $= 0$

Intervalos de Endereços par Jcond

- Antes de 386:
  - saltos deviam ser entre os bytes –128 to +127 a partir do contador da posição corrente
- Processadores x86:
  - Offset de 32-bit permite saltar para qualquer posição da memória

Saltos Baseados em Flags Específicas

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

Saltos Baseados em Igualdade

Mnemonic	Description
JE	Jump if equal ($leftOp = rightOp$)
JNE	Jump if not equal ($leftOp \neq rightOp$)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Saltos Baseados em Comparações Unsigned

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* >= *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* <= *rightOp*)
JNA	Jump if not above (same as JBE)

Saltos Baseados em Comparações Signed

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* >= *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* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Aplicações (1/ 5)

- Tarefa: Salta para uma etiqueta se EAX unsigned é maior que EBX
- Solução: Use CMP, seguido por JA

```
1 cmp eax,ebx
2 ja Larger
```

- Tarefa: Salta para uma etiqueta se EAX signed é maior que EBX
- Solução: Use CMP, seguido por JG

```
1 cmp eax,ebx
2 jg Greater
```

Aplicações (2/ 5)

- Salta para a etqueta L1 se EAX unsigned é menor ou igual a Val1

```
1 cmp eax, Val1
2 jbe L1                    ; menor ou igual
```

- Salta para a etiqueta L1 se EAX *signed* é menor ou igual a Val1

```
1 cmp eax, Val1
2 jle L1
```

Aplicações (3/ 5)

- Compara AX *unsigned* com BX, e copia o maior dos dois numa variável chamada *Large*

```
1 mov [Large],bx
2 cmp ax,bx
3 jna Next
4 mov [Large],ax
5     Next:
```

- Comparar AX *signed* com BX, e copiar o menor dos dois em uma variável chamada *Small*

```
1 mov [Small],ax
2 cmp bx,ax
3 jnl Next
4 mov [Small],bx
5     Next:
```

Aplicações (4/ 5)

- Salta para a etiqueta L1 se a *word* na memória apontada por ESI é igual a zero

```
1 cmp WORD [esi],0
2 je L1
```

- Salta para a etiqueta L2 se a *doubleword* na memória, apontada por EDI é par

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

Aplicações (5/ 5)

- Tarefa: Saltar para a etiqueta L1 se os bits 0, 1, e 3 em AL são todos ativados.
- Solução: Limpar todos bits exceto 0, 1,e 3. Então comparar o resultado com o binário 00001011.

```
1 and al,00001011b              ; limpa os bits indesejaveis
2 cmp al,00001011b              ; verifica os bits remanecentes
3 je L1                         ; tudo ativado? salta para L1
```

Sua Vez . . .

- Escreva um código que salte para a etiqueta L1 se qualquer dos bits 4, 5, ou 6 está ativo no registrador BL.
- Escreva um código que salte para a etiqueta L1 se os bits 4, 5, e 6 estão todos ativos no registrador BL.
- Escreva um código que salte para a etiqueta L2 se AL tem paridade par.
- Escreva um código que salte para a etiqueta L3 se EAX é negativo.
- Escreva um código que salte para a etiqueta L4 se a expressão (EBX - ECX) é maior que zero.

Encriptando uma String

- O laço que segue usa a instrução XOR para transformar qualquer caractere da *String* em um novo valor.

```
1  %include "io.inc"
2
3  section .data
4  KEY DB 239                ; can be any byte value
5  BUFMAX DB 128
6  buffer DB "teste",0
7  bufSize EQU $-buffer
8
9  section .text
10 global main
11 main:
12 mov ebp, esp              ; for correct debugging
13
14 mov ecx, bufSize          ; loop counter
15 dec ecx
16 mov esi, buffer           ; index 0 in buffer
17
18 L1:
19 mov bl, [KEY]
20 xor byte [esi], bl        ; translate a byte
21 inc esi                   ; point to next byte
22 loop L1
23
24 xor eax, eax
25 ret
```

Programa de Criptografia de String

- Tarefas:
  - Recebe uma mensagem de entrada do usuário(string)
  - Encripta a mensagem
  - Mostra a mensagem criptografada
  - Decodifica a mensagem
  - Mostra a mensagem descriptografada

Veja o código fonte do arquivo Encrypt.asm. Exemplo de saída:

```
Enter the plain text: Attack at dawn.
Cipher text: «¢¢ÄîäÄ¢-ïÄÿü-Gs
Decrypted: Attack at dawn.
```

Instrução BT (Bit Test)

- Copia o bit *n* de um operando na Carry flag
- Sintaxe: BT *bitBase*, *n*
  - *bitBase* pode ser r/m16 ou r/m32
  - *n* pode ser r16, r32, ou imm8
- Exemplo: salta para etiqueta L1 se o bit 9 está ativo no registrador AX:

```
1 bt AX,9              ; CF = bit 9
2 jc L1                ; jump if Carry
```

Instruções Condicionais de Laço

- LOOPZ e LOOPE
- LOOPNZ e LOOPNE

LOOPZ e LOOPE

- Sintaxe:
  - LOOPE destination
  - LOOPZ destination
- Logic:
  ECX $\leftarrow$ ECX - 1
  se ECX $>$ 0 e ZF=1, salta para o destino
- Útil quando se estiver varrendo um array em busca do primeiro elemento que não esteja de acordo com um dado valor.

Nota

No modo 32-bit, ECX é o contador de laço. No modo de endereçamento real de 16-bit, CX é o contador, e no modo 64-bit RCX é o contador.

LOOPNZ e LOOPNE

- LOOPNZ (LOOPNE) é uma instrução de laço condicional
- Sintaxe:
  LOOPNZ destination
  LOOPNE destination
- Logic:
  - ECX $\leftarrow$ ECX - 1;
  - se ECX $>$ 0 e ZF=0, salta para o destino
- Útil quando se varre um array em busca de um elemento que satisfaz um dado valor.

Exemplo de LOOPNZ

- O seguinte código busca o primeiro valor positivo em um array:

```
1  %include "io.inc"
2
3  section .data
4  array DB -3,-6,-1,-10,10,30,40,4
5  sizeOfArray EQU $-array
6  sentinel DB 0
7
8  section .text
9  global main
10 main:
11 mov ebp, esp                    ; for correct debugging
12
13 mov esi, array
14 mov ecx, sizeOfArray
15 next:
16 test WORD [esi],8000h ; test sign bit
17 pushfd                          ; push flags on stack
18 add esi, 1
19 popfd                           ; pop flags from stack
20 loopnz next                     ; continue loop
21 jnz quit                        ; none found
22 sub esi,1          ; ESI points to value
23 quit:
24
25 xor eax, eax
26 ret
```

Sua Vez . . .

- Localize o primeiro valor não zero em um array.

```
1  %include "io.inc"
2  section .data
3      array DW 50 times 0
4      sizeOfArray EQU ($-array)/2
5  section .text
6  global CMAIN
7  CMAIN:
8      mov esi, array
9      mov ecx,sizeOfArray
10 L1: cmp WORD [esi],0
11     ; check for zero
12     (complete aqui com seu codigo)
13 quit:
14     xor eax, eax
15     ret
```

. . . (solução)

```
1  %include "io.inc"
2  section .data
3  array times 50 DW 0
4  sizeOfArray EQU ($-array)/2
5  section .text
6  global CMAIN
7  CMAIN:
8  mov esi, array
9  mov ecx,sizeOfArray
10 mov WORD [esi+10], 5
11 mov edx, 0
12 L1: cmp WORD [esi],0
13 pushfd                    ; push flags on stack
14 add esi, 2
15 inc edx
16 popfd                     ; pop flags from stack
17 loope L1                  ; continue loop
18 jz quit                   ; none found
19 sub esi, 2                ; ESI points to value
20 quit:
21 PRINT_UDEC 4, EDX
22 xor eax, eax
23 ret
```

Estruturas Condicionais

- Bloco estruturado IF
- Compondo Expressões com AND
- Compondo Expressões com OR
- Laços WHILE
- Seleção dirigida por tabela

Bloco estruturado IF

Programadores de Assembly podem facilmente traduzir uma estrutura escrita em C++/Java em linguagem Assembly. Por exemplo:

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

```
1 mov eax,op1
2 cmp eax,op2
3 jne L1
4 mov X,1
5 jmp L2
6 L1: mov X,2
7 L2:
```

Sua Vez . . .

Implemente o seguinte pseudocódigo na linguagem Assembly. Todos os valores são *unsigned*:

```
1 if( ebx <= ecx )
2 {
3 eax = 5;
4 edx = 6;
5 }
```

```
1 cmp ebx,ecx
2 ja next
3 mov eax,5
4 mov edx,6
5 next:
```

(Existem múltiplas soluções corretas para esse problema.)

Sua Vez . . .

Implemente o seguinte pseudocódigo na linguagem Assembly. Todos os valores são inteiros *signed* de 32-bit:

```
1 if( var1 <= var2 )
2 var3 = 10;
3 else
4 {
5 var3 =6;
6 var4 =7;
7 }
```

```
1 mov eax,var1
2 cmp eax,var2
3 jle L1
4 mov var3,6
5 mov var4,7
6 jmp L2
7 L1: mov var3,10
8 L2:
```

(Existem múltiplas soluções corretas para esse problema.)

Expressões Compostas com AND (1/3)

- Quando se está implementando com o operador lógico AND, considere que HLLs usam avaliação curto-circuito (segunda expressão não é avaliada se a primeira for falsa).
- No exemplo que segue, se a primeira expressão é false, a segunda é pulada:

if $(al > bl) AND (bl > cl)$
$X = 1;$

Expressões Compostas com AND (2/3)

$if(al > bl) AND (bl > cl)$
$X = 1;$
Essa é uma possível implementação . . .

```
1  cmp al,bl          ; first expression...
2  ja L1
3  jmp next
4  cmp bl,cl          ; second expression...
5  ja L2
6  jmp next
7  L1:
8  L2:                ; both are true
9  mov X,1            ; set X to 1
10 next:
```

Expressões Compostas com AND (3/3)

if $(al > bl) AND (bl > cl)$
$X = 1;$
Porém, a seguinte implementação usa 29% menos código por inverter o primeiro operador relacional. Permitimos que o programa atravesse para a segunda expressão.

```
1 cmp al,bl      ; first expression...
2 jbe next       ; quit if false
3 cmp bl,cl      ; second expression...
4 jbe next       ; quit if false
5 mov X,1        ; both are true
6 next:
```

Your turn . . .

Implemente o seguinte pseudocódigo em assembly. Todos valores são unsigned:

```
1 if( ebx <= ecx
2 && ecx > edx )
3 {
4 eax = 5;
5 edx = 6;
6 }
```

```
1       cmp ebx,ecx
2       ja next
3       cmp ecx,edx
4       jbe next
5       mov eax,5
6       mov edx,6
7       next:
```

(Existem múltiplas soluções corretas para esse problema.)

Expressões Compostas com OR (1/2)

- Quando se está implementando com o operador lógico OR, considere que HLLs usam avaliação curto-circuito.
- No exemplo que segue, se a primeira expressão é verdadeira, a segunda é pulada:

```
1 if (al > bl) OR (bl > cl)
2 X = 1;
```

Expressões Compostas com OR (2/2)

```
1 if (al > bl) OR (bl > cl)
2     X = 1;
```

- Podemos usar a lógica de atravessar para manter o código menor:

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

Laços WHILE

Um laço WHILE é na verdade uma estrutura IF seguida pelo corpo do laço, e por fim por um salto incondicional para o topo do laço. Considere o seguinte exemplo:

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

Esta é uma implementação possível:

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

Sua vez . . .

- Implemente o seguinte laço, usando inteiros de 32-bit unsigned:

```
1 while( ebx <= val1 )
2 {
3 ebx = ebx + 5;
4 val1 = val1 - 1
5 }
```

Nota

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:

Seleção Baseada em Tabela (1/5)

- Seleção baseada em tabela usa uma tabela para buscar e substituir uma seleção múltipla (equivalente ao SWITCH das linguagens HLL)
- Crie uma tabela contendo os valores a sere buscados e os offsets das etiquetas ou procedimentos
- Use um laço pra buscar na tabela
- Adequado para grande quantidade de comparações

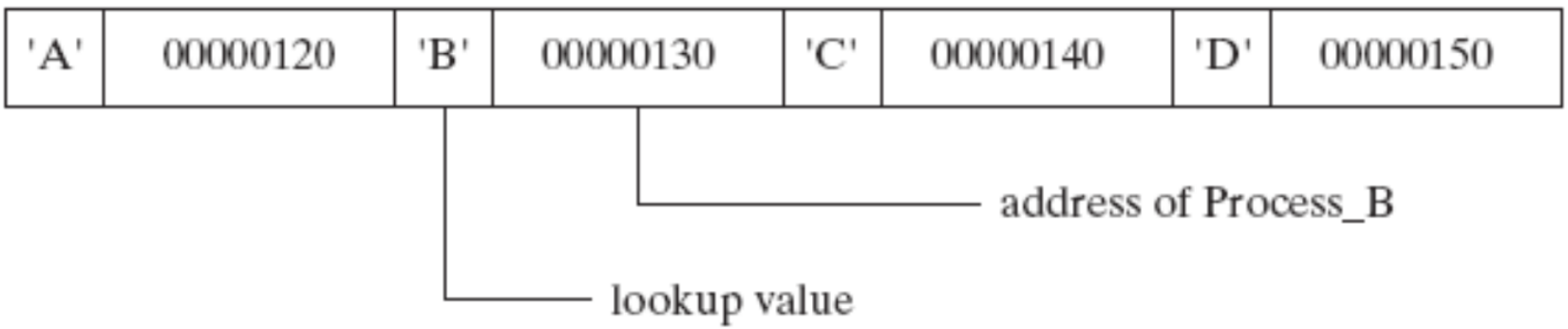
Seleção Baseada em Tabela (2/5)

- Passo 1: Criar uma tabela contendo os valores a sere buscados e os offsets das etiquetas ou procedimentos:

```
1  %include "io.inc"
2
3  section .data
4  CaseTable DB 'A'          ; lookup value
5  DD Process_A             ; address of procedure
6  EntrySize EQU ($ - CaseTable)
7  DB 'B'
8  DD Process_B
9  DB 'C'
10 DD Process_C
11 DB 'D'
12 DD Process_D
13 NumberOfEntries EQU ($ - CaseTable) / EntrySize
14 msgA DB "Process_A",0
15 msgB DB "Process_B",0
16 msgC DB "Process_C",0
17 msgD DB "Process_D",0
```

Seleção Baseada em Tabela (3/5)

- Tabela de Offsets dos procedimentos:



Seleção Baseada em Tabela (4/5)

Passo 2: Usar um laço para buscar na tabela. Quando um valor correspondente é encontrado, chame o procedimento pelo offset armazenado na entrada corrente da tabela:

```
1  section .text
2  global CMAIN
3  CMAIN:
4  mov al, 'D'
5  mov ebx, CaseTable              ; point EBX to the table
6  mov ecx, NumberOfEntries        ; loop counter
7  L1: cmp al,[ebx]                  ; match found?
8  jne L2                          ; no: continue
9  call [ebx + 1]                  ; yes: call the procedure
10 PRINT_STRING [EDX]                ; display message
11 NEWLINE
12 jmp L3                          ; and exit the loop
13 L2: add ebx,EntrySize             ; point to next entry
14 dec ecx
15 JNZ L1                          ; repeat until ECX = 0
16 L3:
17 xor eax, eax
18 ret
```

Seleção Baseada em Tabela (5/5)

Procedimentos:

```
1  Process_A:
2  mov edx, msgA
3  ret
4  Process_B:
5  mov edx, msgB
6  ret
7  Process_C:
8  mov edx, msgC
9  ret
10 Process_D:
11 mov edx, msgD
12 ret
```

Aplicação: Máquinas de Estados Finitos

- Uma máquina de estados finitos, em inglês, *finite-state machine* (FSM) é uma estrutura de grafo que muda de estado baseada em alguma entrada. Também é chamada de diagrama de transição de estados.
- Usamos um grafo para representar uma FSM, com quadrados ou círculo chamados nós e linhas com setas entre os círculos, chamadas aresta.

Aplicação: Máquinas de Estados Finitos

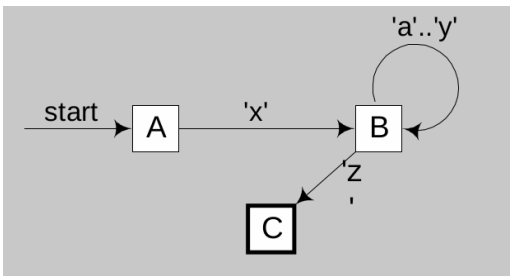
- Uma FSM é uma instância específia de uma estrutura mais genérica chamada grafo direcionado.
- Três estados básicos são representados por nós:
  - Estado inicial
  - Estados terminais
  - Estados não terminais

Máquina de Estados Finitos

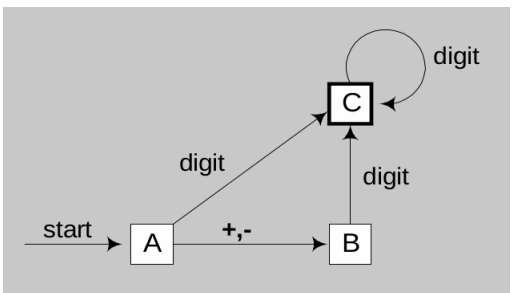
- Aceita qualquer sequência de símbolos que colocam em um estado de aceitção (final)
- Pode ser usado para reconhecer ou validar uma sequência de caracteres governadso por regras de linguagem (chamado de expressões regulares)
- Advantages:
  - Provê um rastreamento visual do fluxo de controle do programa
  - Fácil de modificar
  - Fácil de implementar na linguagem Assembly

Máquina de Estados Finitos - Exemplo

- FSM que reconhece strings que começam por 'x', seguido pelas letras'a'..'y', e terminando com 'z':

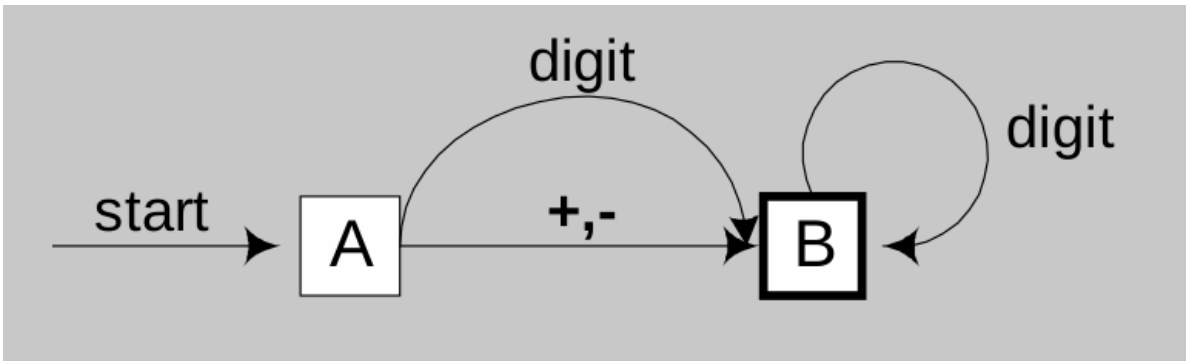


- FSM que reconhece inteiros com sinal:



Sua Vez . . .

- Explique por que o seguinte FSM não funciona para inteiros com sinal como o exemplo do slide anterior:



Implementando um FSM

O código que segue parte do estado A no inteiro:

```
1  StateA:
2  call Getnext                ; read next char into AL
3  cmp al,'+'                  ; leading + sign?
4  je StateB                   ; go to State B
5  cmp al,'-'                  ; leading - sign?
6  je StateB                   ; go to State B
7  call IsDigit                ; ZF = 1 if AL = digit
8  jz StateC                   ; go to State C
9  call DisplayErrorMsg        ; invalid input found
10 jmp Quit
```

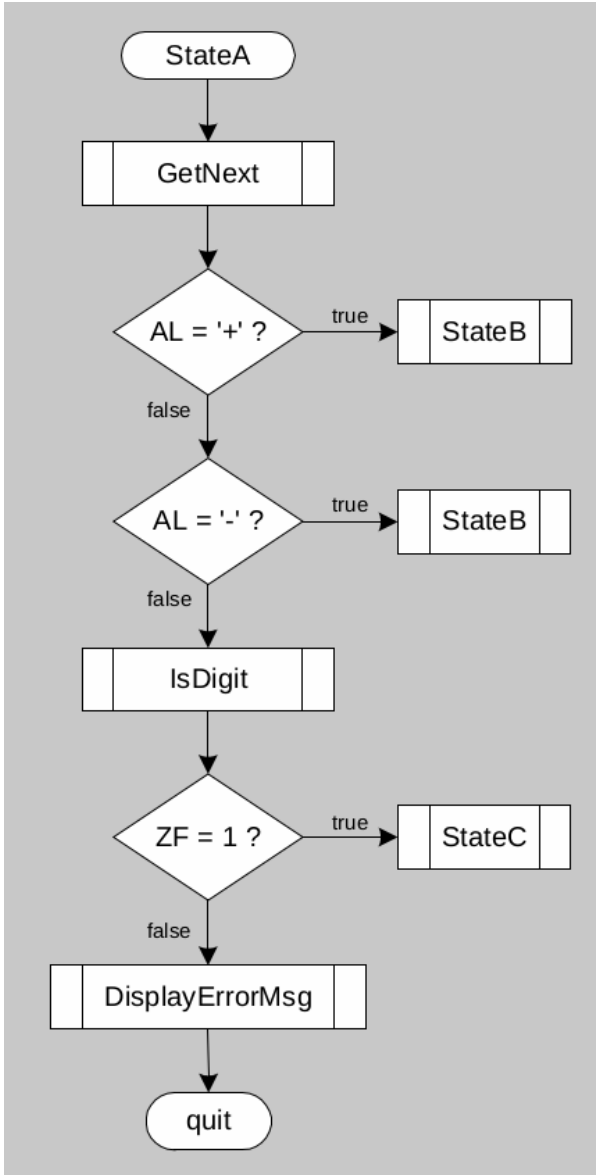
Procedimento IsDigit

Recebe um caractere em AL. Define ZF se o caractere é um dígito decimal.

```
1 IsDigit PROC
2 cmp al,'0'            ; ZF = 0
3 jb ID1
4 cmp al,'9'            ; ZF = 0
5 ja ID1
6 test ax,0             ; ZF = 1
7 ID1: ret
8 IsDigit ENDP
```

Diagrama de Estado A

Estado A aceita + ou -, ou um dígito decimal.



Sua Vez . . .

- Desenhe um diagrama FSM para reconhecer um inteiro hexadecimal.
- Desenhe um diagrama de fluxo para um dos estados do seu FSM.
- Implemente seu FSM na linguagem Assembly. O usuário deve digitar no teclado uma constante hexadecimal.