Topic 6
Lecture 6b
Conditional Processing

CSCI 150

Assembly Language / Machine Architecture

Prof. Dominick Atanasio

What's Next (2 of 5)

- Boolean and Comparison Instructions
- Conditional Jumps
- **Conditional Loop Instructions**
- Conditional Structures
- Application: Finite-State Machines

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:
  - LOOPE *destination*
  - LOOPZ *destination*
- Logic:
  - ECX ← ECX – 1
  - if ECX > 0 and ZF=1, jump to *destination*
- Useful when scanning an array for the first element that does not match a given value.

In 32-bit mode, ECX is the loop counter register and in 64-bit mode, RCX is the counter.

LOOPNZ and LOOPNE

- LOOPNZ (LOOPNE) is a conditional loop instruction
- Syntax:
  - LOOPNZ *destination*
  - LOOPNE *destination*
- Logic:
  - ECX $\leftarrow$ ECX – 1;
  - if ECX > 0 and ZF=0, jump to *destination*
- Useful when scanning an array for the first element that matches a given value.

LOOPNZ Example

The following code finds the first positive value in an array. If none exist, point to the sentinel value:

```
section .data
array:      dw  -3, -6, -1, -10,10, 30, 40, 4
arraysz:  equ  $ - array
sentinel:  equ  0FFFFh

section .text
        mov esi,  array
        mov ecx, arraysz
.loop:
        test        word [esi], 8000h             ; test sign bit
        pushfd                                    ; push flags on stack
        add         esi, 2                        ; increment pointer
        popfd                                     ; pop flags from stack
        loopnz      .loop                         ; continue loop

        jnz          .quit                        ; none found
        sub          esi, 2                       ; ESI points to value
.quit:
// print value found <this includes sentinel of no positive value is found>
```

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
section .bss
    array:       resw 50
    arraysz:    equ $ - array
section .data
    sentinel:    equ 0FFFFh

section .text
       mov esi, array
       mov ecx, arraysz
.loop
        cmp word [esi], 0                              ; check for zero

        ; your code goes here

quit:
```

What's Next (3 of 5)

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- **Conditional Structures**
- Application: Finite-State Machines

Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection

Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
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:
```

Your Turn . . . (3 of 8)

Implement the following pseudocode in assembly language. All values are unsigned:

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

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

(There are multiple correct solutions to this problem.)

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

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

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

(There are multiple correct solutions to this problem.)

Compound Expression with AND (1 of 3)

- When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is false, the second expression is skipped:

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

Compound Expression with AND (2 of 3)

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

This is one possible implementation but maybe not the most efficient . . .

```
        cmp al, bl                  ; first expression...
        ja  L1
        jmp next
L1:
        cmp bl, cl                  ; second expression...
        ja  L2
        jmp next
L2:                                 ; both are true
        mov X,1                     ; set X to 1
next:
```

Compound Expression with AND (3 of 3)

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

But the following implementation uses 29% less instructions by reversing the first relational operator. We allow the program to "fall through" to the second expression:

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

Implement the following pseudocode in assembly language. All values are unsigned:

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

Compound Expression with OR (1 of 2)

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

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

Compound Expression with OR (2 of 2)

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

We can use "fall-through" logic to keep the code as short as possible:

```
      cmp al, bl          ; 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:
```

WHILE Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

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

This is a possible implementation:

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

Your Turn . . . (6 of 8)

Implement the following loop, using unsigned 32-bit integers:

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

Table-Driven Selection (1 of 4)

- Table-driven selection uses a table lookup to replace a multiway selection structure
- Create a table containing lookup values and the offsets of labels or procedures
- Use a loop to search the table
- Suited to a large number of comparisons

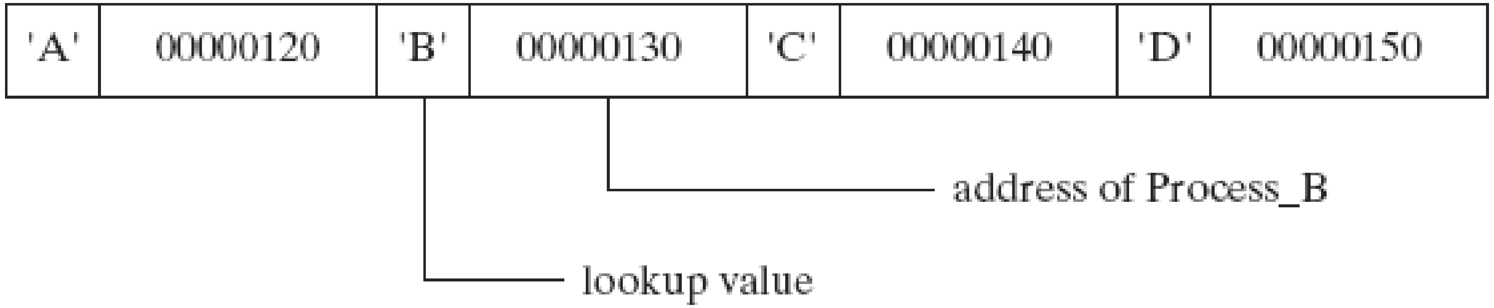
Table-Driven Selection (2 of 4)

Step 1: create a table containing lookup values and procedure offsets:

```
section .data
case_table:
        db 'A'                                ; lookup value
        dd process_a                          ; address of procedure
        entry_sz: equ $ - case_table
        db  'B'
        dw Process_B
        db 'C'
        dd Process_C
        db 'D'
        dd Process_D
        entry_qty: equ  $ - case_table / entry_sz
```

Table-Driven Selection (3 of 4)

Table of Procedure Offsets:



Step 2: Use a loop to search the table. When a match is found, call the procedure offset stored in the current table entry:

```
      mov ebx, case_table         ; point EBX to the table
      mov ecx, entry_qty          ; loop counter

L1:   cmp al,[ebx]                ; match found?
      jne L2                      ; no: continue
      call [ebx + 1]              ; yes: call the procedure
      call WriteString            ; display message
      call Crlf
      jmp L3                      ; and exit the loop
L2:   add ebx,EntrySize           ; point to next entry
      loop L1                     ; repeat until ECX = 0

L3:
```

required for procedure pointers

What's Next (4 of 5)

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- **Application: Finite-State Machines**

Application: Finite-State Machines (1 of 2)

- A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a state-transition diagram.
- We use a graph to represent an FSM, with squares or circles called nodes, and lines with arrows between the circles called edges.

Application: Finite-State Machines (2 of 2)

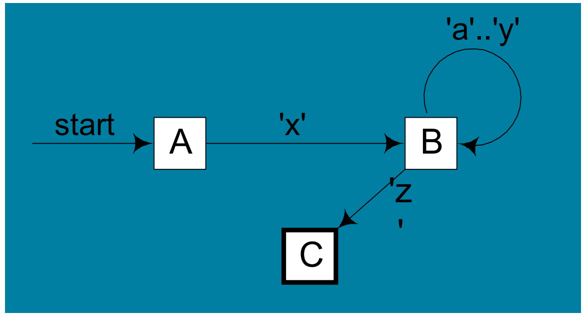
- A FSM is a specific instance of a more general structure called a directed graph.
- Three basic states, represented by nodes:
  - Start state
  - Terminal state(s)
  - Nonterminal state(s)

Finite-State Machine

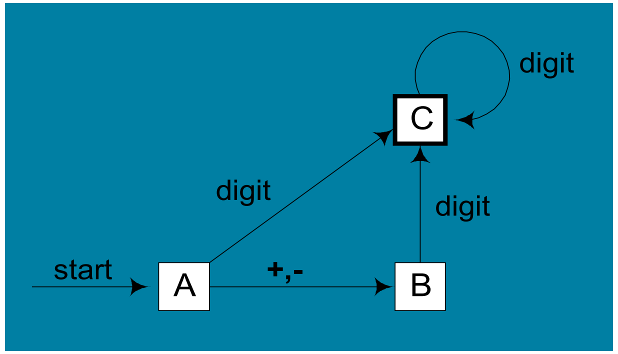
- Accepts any sequence of symbols that puts it into an accepting (final) state
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
- Advantages:
  - Provides visual tracking of program's flow of control
  - Easy to modify
  - Easily implemented in assembly language

Finite-State Machine Examples

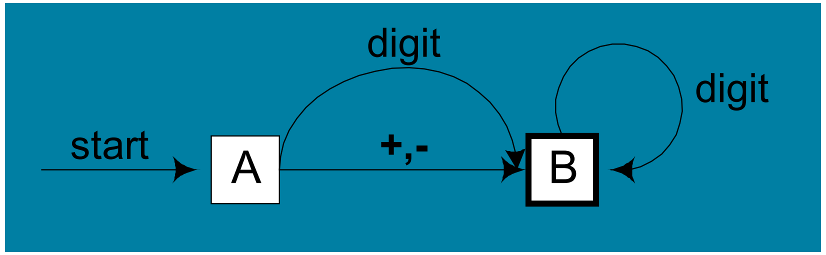
- FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':



- FSM that recognizes signed integers:



- Explain why the following FSM does not work as well for signed integers as the one shown on the previous slide:



Implementing an FSM

The following is code from State A in the Integer FSM:

```
state_a:
    call get_next            ; read next char into AL
    cmp al, '+'              ; leading plus sign?
    je state_b               ; go to State B
    cmp al, '-'              ; leading minus sign?
    je state_b               ; go to State B
    call is_digit            ; returns ZF = 1 if AL = digit
    jz state_c               ; go to State C
    call display_error       ; invalid input found
    jmp exit
```

View the [Finite.asm source code.]()

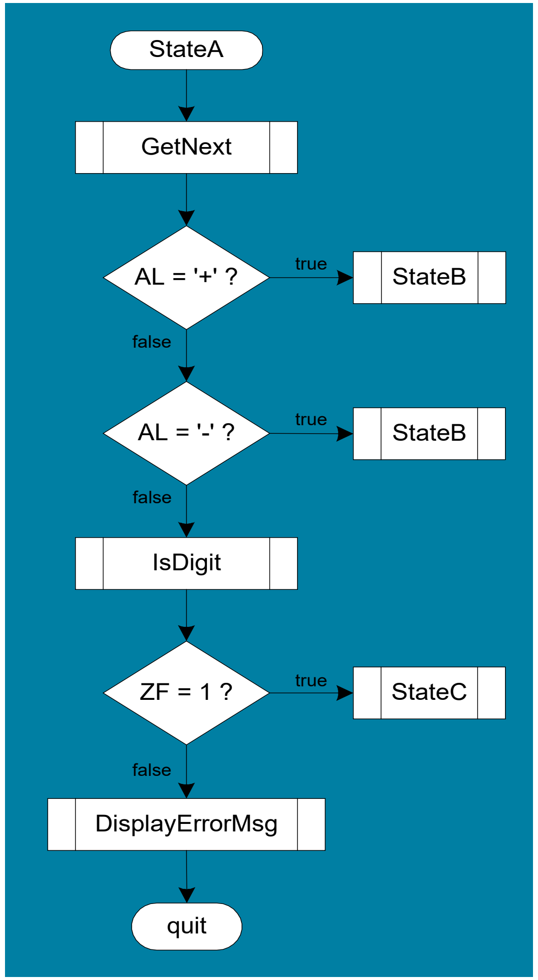
IsDigit Procedure

Receives a character in AL. Sets the Zero flag if the character is a decimal digit.

```
Is_digit
        cmp   al, '0'               ; ZF = 0
        jb    ID1
        cmp   al, '9'               ; ZF = 0
        ja    ID1
        test  ax, 0                 ; ZF = 1
ID1:
        ret
```

Flowchart of State A

State A accepts a plus or minus sign, or a decimal digit.



Your Turn . . . (8 of 8)

- Draw a FSM diagram for hexadecimal integer constant that conforms to NASM syntax.
- Draw a flowchart for one of the states in your FSM.
- Implement your FSM in assembly language. Let the user input a hexadecimal constant from the keyboard.

4C 6F 70 70 75 75 6E