Week 7
Chapter 6: Conditional Processing

What's Next

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

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:

  LOOPE *destination*

  LOOPZ *destination*
- Logic:
  - ECX $\leftarrow$ 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. In 16-bit real-address mode, CX is the counter, 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:

```
.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:
```

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

```
.data
array SWORD 50 DUP(?)
sentinel 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:
```

. . . (solution)

```
.data
array SWORD 50 DUP(?)
sentinel 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:
```

What's Next

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

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:
```

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 var3,6
    mov var4,7
    jmp L2
L1: mov 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 . . .

```
   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 code 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;
}
```

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

(There are multiple correct solutions to this problem.)

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:
```

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

```
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:
```

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:

```
.data
CaseTable BYTE 'A'                  ; lookup value
    DWORD Process_A                 ; address of procedure
    EntrySize = ($ - CaseTable)
    BYTE 'B'
    DWORD Process_B
    BYTE 'C'
    DWORD Process_C
    BYTE 'D'
    DWORD Process_D

NumberOfEntries = ($ - CaseTable) / EntrySize
```

Table-Driven Selection (3 of 4)

Table of Procedure Offsets:

'A'	00000120	'B'	00000130	'C'	00000140	'D'	00000150

address of Process_B

lookup value

Table-Driven Selection (4 of 4)

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,OFFSET CaseTable        ; point EBX to the table
    mov ecx,NumberOfEntries         ; loop counter

L1: cmp al,[ebx]                    ; match found?
    jne L2                          ; no: continue
    call NEAR PTR [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

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

Application: Finite-State Machines

- 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

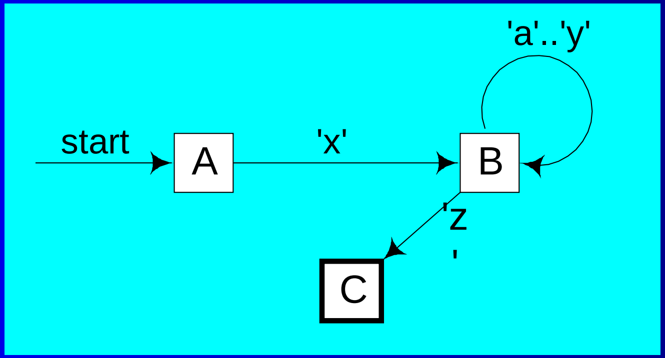
- A FSM is a specific instance of a more general structure called a directed graph.
- Three basic states, represented by nodes:
  - Start state
  - Terminal state(s)
  - Nonterminal state(s)

Finite-State Machine

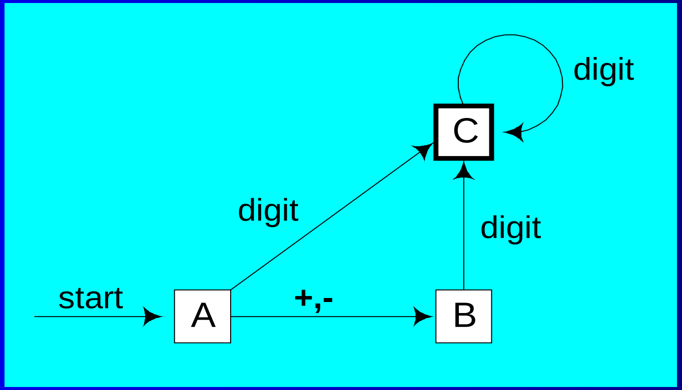
- Accepts any sequence of symbols that puts it into an accepting (final) state
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
- Advantages:
  - Provides visual tracking of program's flow of control
  - Easy to modify
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

Finite-State Machine Examples

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

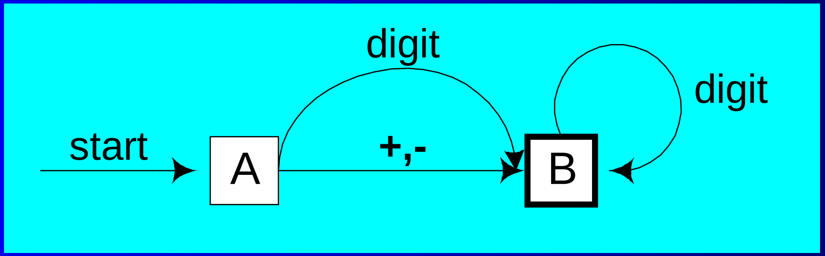


- FSM that recognizes signed integers:



Your Turn . . .

- 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:

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

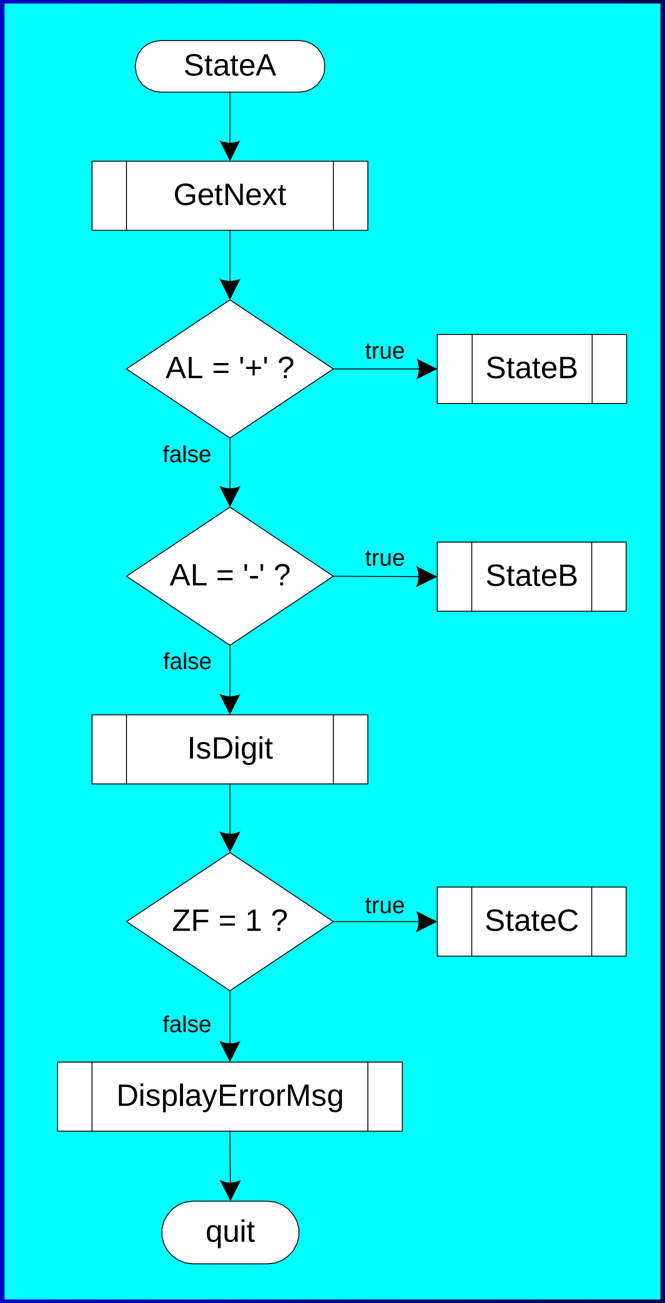
IsDigit Procedure

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

```
IsDigit PROC
     cmp   al,'0'           ; ZF = 0
     jb    ID1
     cmp   al,'9'           ; ZF = 0
     ja    ID1
     test  al,0             ; ZF = 1
ID1: ret
IsDigit ENDP
```

Flowchart of State A

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



- Draw a FSM diagram for hexadecimal integer constant that conforms to MASM 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.

What's Next

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

Creating IF Statements

- Runtime Expressions
- Relational and Logical Operators
- MASM-Generated Code
- .REPEAT Directive
- .WHILE Directive

Runtime Expressions

- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.
- Examples:

```
.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 generates "hidden" code for you, consisting of code labels, CMP and conditional jump instructions.

Relational and Logical Operators

Operator	Description
expr1 == *expr2*	Returns true when *expression1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
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.

Signed and Unsigned Comparisons

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

Generated code:

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

MASM automatically generates an unsigned jump (JBE) because val1 is unsigned.

Signed and Unsigned Comparisons

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

Generated code:

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

MASM automatically generates a signed jump (JLE) because val1 is signed.

Signed and Unsigned Comparisons

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

Generated code:

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

MASM automatically generates an unsigned jump (JBE) when both operands are registers . . .

Signed and Unsigned Comparisons

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

Generated code:

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

. . . unless you prefix one of the register operands with the SDWORD PTR operator. Then a signed jump is generated.

.REPEAT Directive

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

Example:

```
; Display integers 1 – 10:

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

.WHILE Directive

Tests the loop condition before executing the loop body The .ENDW directive marks the end of the loop.

Example:

```
; Display integers 1 – 10:

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

Summary

- Bitwise instructions (AND, OR, XOR, NOT, TEST)
  - manipulate individual bits in operands
- CMP – compares operands using implied subtraction
  - sets condition flags
- Conditional Jumps & Loops
  - equality: JE, JNE
  - flag values: JC, JZ, JNC, JP, ...
  - signed: JG, JL, JNG, ...
  - unsigned: JA, JB, JNA, ...
  - LOOPZ, LOOPNZ, LOOPE, LOOPNE
- Flowcharts – logic diagramming tool
- Finite-state machine – tracks state changes at runtime