서강대학교

Assembly Programming

Chapter 6: Conditional Processing

CSE3030

Prof. Youngjae Kim

Distributed Computing and Operating Systems Laboratory (DISCOS)

https://discos.sogang.ac.kr

Chapter 6: Conditional Processing

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

6.1 Conditional Branching

- A programming language that permits decision making lets you alter the flow of control, using a technique known as *conditional branching*.
  - Assembly language also provides all the tools you need for decision-making logic.

- This chapter will cover
  - How the binary foundations are behind programming logic
  - How the CPU compares instruction operands, using the CMP instruction and the process status flags
  - How to use assembly language to implement logic structures characteristic of high-level languages

6.2 Boolean and Comparison Instructions

- Basic operations of boolean algebra
  - AND, OR, XOR and NOT
    - These operations are carried out at the binary bit level.

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand
OR	Boolean OR operation between a source operand and a destination operand
XOR	Boolean exclusive-OR operation between a source and a destination operand
NOT	Boolean NOT operation on a destination operand
TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately

Table 6-1 Selected Boolean Instructions

6.2 Boolean and Comparison Instructions

- Boolean instructions
  - Affects the Zero, Carry, Sign, Overflow, and Parity flags.

- CPU Status Flags
  - The Zero flag is set when the result of an operation equals zero.
  - The Carry flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
  - The Sign flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
  - The Overflow flag is set when an instruction generates an invalid signed result.
  - The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

AND and OR Instructions

- ## AND Instruction
  - Performs a Boolean AND operation between each pair of matching bits in two operands
  - Syntax: **AND *destination, source*** (same operand types as MOV)

```
             0 0 1 1 1 0 1 1
        AND  0 0 0 0 1 1 1 1
             ---------------
cleared  --  0 0 0 0|1 0 1 1  --  unchanged
```

  - OR Instruction
    - Syntax: **OR *destination, source***

```
               0 0 1 1 1 0 1 1
           OR  0 0 0 0 1 1 1 1
               ---------------
unchanged  --  0 0 1 1|1 1 1 1  --  set
```

XOR and NOT Instructions

- XOR Instruction
  - Syntax: **`XOR destination, source`**

```
          0 0 1 1 1 0 1 1
XOR       0 0 0 0 1 1 1 1
          ---------------
unchanged 0 0 1 1 0 1 0 0  inverted
```

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

- NOT Instruction
  - Syntax: **`NOT destination`**

```
NOT  0 0 1 1 1 0 1 1
     ---------------
     1 1 0 0 0 1 0 0  inverted
```

Applications

- Application 1
  - Task: Convert the character in AL to upper case.
  - Solution: Use the AND instruction to clear bit 5.

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

- Application 2
  - Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
  - Solution: Use the OR instruction to set bits 4 and 5.

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

Applications

- Application 3
  - Task: Turn on the keyboard CapsLock key.
  - Solution: Use the OR instruction to set bit 6 in the keyboard flag byte at `0040:0017h` in the BIOS data area (Only in real address mode. Not work under Win 2000, NT, XP, Win7).

```
mov ax,40h                    ; BIOS segment
mov ds,ax
mov bx,17h                    ; keyboard flag byte
or BYTE PTR [bx],01000000b ; CapsLock on
```

Applications

- Application 4
  - Task: Jump to a label if an integer is even.
  - Solution: AND the lowest bit with a 1. If the result is zero, the number was even.

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

- Application 5
  - Task: Jump to a label if the value in AL is not zero.
  - Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

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

  - ORing any number with itself does not change its value.

TEST Instruction

- TEST Instruction
  - Performs a nondestructive AND operation between each pair of matching bits in two operands
  - No operands are modified, but the Zero flag is affected.
  - Example: jump to a label if either bit 0 or bit 1 in AL is set.

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

  - Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

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

```
0 0 1 0 0 1 0 1 <- input value
0 0 0 0 1 0 0 1 <- test value
0 0 0 0 0 0 0 1 <- result: ZF=0

0 0 1 0 0 1 0 0 <- input value
0 0 0 0 1 0 0 1 <- test value
0 0 0 0 0 0 0 0 <- result: ZF=1
```

More examples

CMP Instruction

- CMP Instruction
  - Compares the destination operand to the source operand. Nondestructive subtraction of source from destination (destination operand is not changed)
  - Syntax: CMP *destination, source*
  - Example: destination == source

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

  - Example: destination < source

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

  - Example: destination > source

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

CMP Instruction

- Example: destination > source (signed)

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

- Example: destination < source (signed)

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

6.3 Conditional Jumps

- A conditional jump instruction branches to a label when specific register or flag conditions are met.

- Examples:
  - JB, JC jump to a label if the Carry flag is set.
  - JE, JZ jump to a label if the Zero flag is set.
  - JS jumps to a label if the Sign flag is set.
  - JNE, JNZ jump to a label if the Zero flag is clear.
  - JECXZ jumps to a label if ECX equals 0.

- Jumps Based on Specific Flags

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 Based on Equality

  - JE; jump if equal (leftOp = rightOp)
  - JNE ; jump if not equal
  - JCXZ ; jump if CX = 0
  - JECXZ ; jump if ECX = 0
  - JRCXZ ; jump if RCX = 0 (64-bit mode)

- Jumps Based on Unsigned Comparisons

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)

- Jumps Based on Signed Comparisons

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)

Applications

- Jump to a label if unsigned EAX is greater than EBX.

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

- Jump to a label if signed EAX is greater than EBX.

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

- Jump to label L1 if unsigned EAX is less than or equal to Val1.

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

- Jump to label L1 if signed EAX is less than or equal to Val1.

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

Applications

- Compare unsigned AX to BX, and copy the larger of the two into a variable named **Large**.

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

- Compare signed AX to BX, and copy the smaller of the two into a variable named **Small**.

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

Applications

- Jump to label **L1** if the memory word pointed to by ESI equals Zero.

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

- Jump to label **L2** if the doubleword in memory pointed to by EDI is even.

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

- Jump to label L1 if bits 0, 1, and 3 in AL are all set.
  - Clear all bits except bits 0, 1,and 3. Then compare the result with 00001011 binary.

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

Encrypting a String

- The following loop uses the XOR instruction to transform every character in a string into a new value.

```
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
```

String Encryption Program

- Input a message (string) from the user
- Encrypt the message
- Display the encrypted message
- Decrypt the message
- Display the decrypted message

- Sample Output

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

• A Sample Program(1/5)

```
TITLE Encryption Program        (Encrypt.asm)
; This program demonstrates simple symmetric
; encryption using the XOR instruction.
; Chapter 6 example.

INCLUDE Irvine32.inc

KEY = 239             ; any value between 1-255
BUFMAX = 128          ; maximum buffer size

.data
sPrompt  BYTE  "Enter the plain text: ",0
sEncrypt BYTE  "Cipher text:          ",0
sDecrypt BYTE  "Decrypted:            ",0

buffer   BYTE   BUFMAX+1 DUP(0)
bufSize  DWORD  ?
```

- A Sample Program(2/5)

```
.code
main PROC

   call InputTheString    ; input the plain text
   call TranslateBuffer   ; encrypt the buffer
   mov  edx,OFFSET sEncrypt ; display encrypted msg
   call DisplayMessage
   call TranslateBuffer      ; decrypt the buffer
   mov  edx,OFFSET sDecrypt ; display decrypted msg
   call DisplayMessage

   exit
main ENDP
```

- A Sample Program(3/5)

```
;------------------------------------------------
InputTheString PROC
; Asks the user to enter a string from the
; keyboard. Saves the string and its length
; in variables.
; Receives: nothing. Returns: nothing
;------------------------------------------------
      pushad
      mov  edx,OFFSET sPrompt ; display a prompt
      call WriteString
      mov  ecx,BUFMAX         ; maximum char count
      mov  edx,offset buffer  ; point to the buffer
      call ReadString         ; input the string
      mov  bufSize,eax        ; save the length
      call Crlf
      popad
      ret
InputTheString ENDP
```

- A Sample Program(4/5)

```
;-------------------------------------------------
DisplayMessage PROC
;
; Display the encrypted or decrypted message.
; Receives: EDX points to the message
; Returns:  nothing
;-------------------------------------------------
      pushad
      call WriteString
      mov  edx,OFFSET buffer  ; display the buffer
      call WriteString
      call Crlf
      call Crlf
      popad
      ret
DisplayMessage ENDP
```

- A Sample Program(5/5)

```
;---------------------------------------------------
TranslateBuffer PROC
;
; Translates the string by XORing each byte
; with the same integer.
; Receives: nothing.  Returns: nothing
;---------------------------------------------------
      pushad
      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

      popad
      ret
TranslateBuffer ENDP
END main
```

6.4 Conditional Loop Instructions

- LOOPZ and LOOPE
  - Syntax: **LOOPE *dest*** **LOOPZ *dest***
  - Logic: **ECX ← ECX – 1**

    if ECX > 0 and ZF = 1, jump to *dest*
  - Useful when scanning an array for the first element that does not match a given value.

- LOOPNZ and LOOPNE
  - Syntax: **LOOPNZ *dest*** **LOOPNE *dest***
  - Logic: **ECX ← ECX – 1**

    if ECX > 0 and ZF = 0, jump to *dest*
  - Useful when scanning an array for the first element that matches a given value.

Example

- LOOPNZ Example: find 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:
```

Why do we push the flags on the stack before the ADD instruction?

Because ADD will modify the flags.

Example

- Locate the first nonzero value in the array.

```
.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:
```

6.5 Conditional Structures

- 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:
```

Example

- Examples of Block-Structured IF Statements
  - Unsigned Case

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

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

  - Signed Case

```
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:
```

Compound Expression with AND

- Compound Expression with Logical AND Operator
  - 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;
```

```
    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

- Better translation: the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

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

```
    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:
```

29% less code

```
    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:
```

7 instructions down to
5 instruction (29% down)

Compound Expression

- 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:
```

Compound Expression with OR

- Compound Expression with Logical OR Operator
  - 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;
```

  - 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

- 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;
```

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

While Loops

- Example: signed case

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

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

Table-Driven Selection

- 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

Example

- Let’s assume Process_A, Process_B, Process_C, and Process_D are located at addresses 120h, 130h, 140h, and 150h, respectively. The table would be arranged in memory as shown in Figure 6-2.

‘A’	00000120	‘B’	00000130	‘C’	00000140	‘D’	00000150

Address of Process_B

Lookup value

Figure 6-2 Table of procedure offsets

Example Program

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

```
.data
CaseTable BYTE 'A'      ; lookup value          1 Byte
   DWORD Process_A      ; address of procedure  4 Bytes
   EntrySize = ($ - CaseTable)  Entry size: 5 Bytes
   BYTE 'B'
   DWORD Process_B
   BYTE 'C'
   DWORD Process_C
   BYTE 'D'
   DWORD Process_D

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

Number of Entries = 4

Example Program

- Step 2: Use a loop to search the table. When a match is found, we 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
    jmp L3                   ; and exit the loop
L2: add ebx,EntrySize        ; point to next entry
    loop L1                  ; repeat until ECX = 0
L3:
```

This CALL instruction calls the procedure whose address is stored in the memory location referenced by EBX+1.
An indirect call such as this requires the NEAR PTR operator.

required for procedure pointers

```
JMP [operator] destination
```

where operator can be:

- SHORT: the target address is encoded with one byte and it is in the same segment
- NEAR PTR: the target address is encoded with two bytes and it is in the same segment
- FAR PTR: the target address is encoded with four bytes and it is in a different segment

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

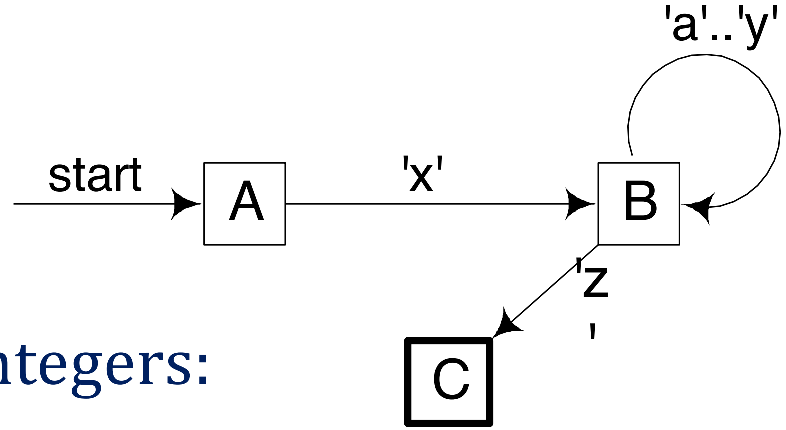
Finite-State Machine

- 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)

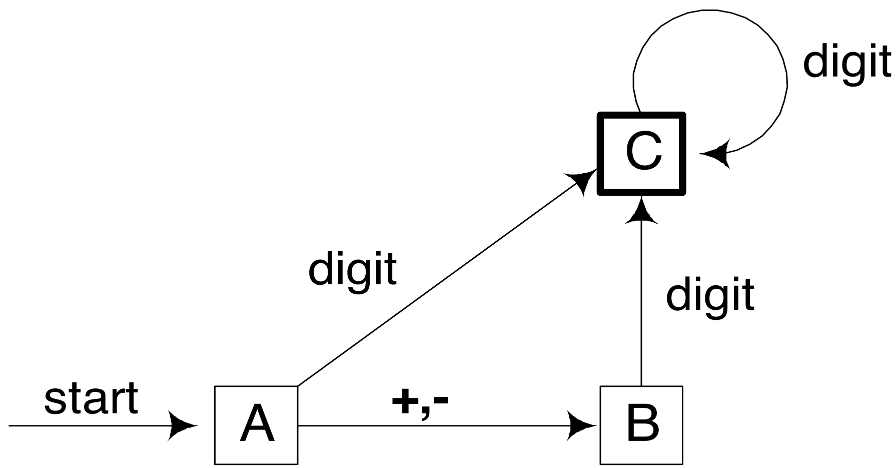
- Advantages:
  - Provides visual tracking of program's flow of control
  - Easy to modify
  - Easily implemented in assembly language

FSM Examples

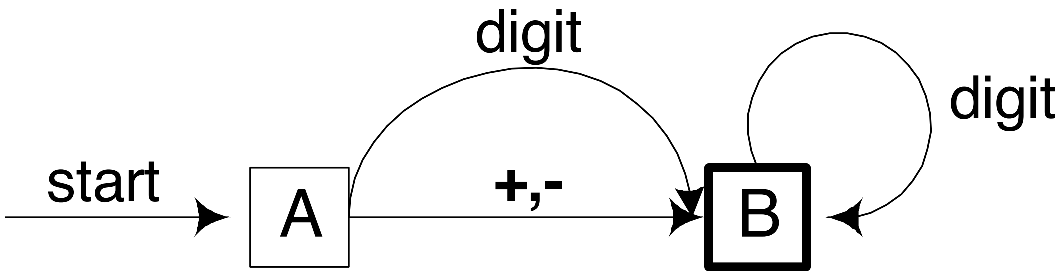
- FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':



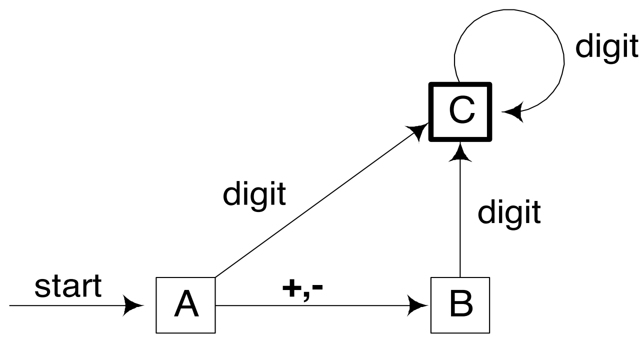
- FSM that recognizes signed integers:



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



Implementing an FSM




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

6.7 Conditional Control Flow Directives

- MASM includes a number of high-level *conditional control flow directives* to help to simply the coding of conditional statements.
- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime express [illegible] s.

Table 6-7 Conditional Control Flow Directives.

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF *condition*	Generates code that tests *condition* and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF *condition*	Generates code that executes the block of statements if *condition* is true.
.REPEAT	Generates code that repeats execution of the block of statements until *condition* becomes true
.UNTIL *condition*	Generates code that repeats the block of statements between .REPEAT and .UNTIL until *condition* becomes true
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until *CX* equals zero
.WHILE *condition*	Generates code that executes the block of statements between .WHILE and .ENDW as long as *condition* is true

Examples for Runtime Expressions

- 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.

MASM-Generated Code

```
.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     ; unsigned
   mov result,1
@C0001:
```

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

Generated code:

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

MASM-Generated Code

```
.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   ;both are regs
   mov result,1
@C0001:
```

```
.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 ; signed
   jle @C0001
   mov result,1
@C0001:
```

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

```
; 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.

```
; Display integers 1 – 10:
mov eax,0
.WHILE eax < 10
   inc eax
   call WriteDec
   call Crlf
.ENDW
```