Assembly Language
組合語言-資訊工程二年級
Lecture slides(2018 – 2019)

Fu Jen Catholic University, Dept of Computer Science and Information Engineering –CSIE

資訊工程系-輔仁大學

教授: 周賜福

107年第一學期

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 6: Conditional Processing

Slides prepared by the author

Revision date: 2/15/2015

(c) Pearson Education, 2010. All rights reserved. You may modify and copy this slide show for
use in the classroom, as long as this copyright statement, the author's name, and the title are

Chapter 6 will address the following issues related to Assembly programming

- How can I use the Boolean operations introduced in Chapter1 (AND, OR,NOT)?

- How do I write an IF statement in assembly language?

- How are nested IF statements translated by compilers into machine language?

- How can I set and clear individual bits in a binary number?

- How can I perform simple binary data encryption?

- How are signed numbers differentiated from unsigned numbers in Boolean expressions?

Chapter Overview

- **Boolean and Comparison Instructions (AND, OR, NOT)**
- Conditional Jumps **(IF)**
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Boolean and Comparison Instructions

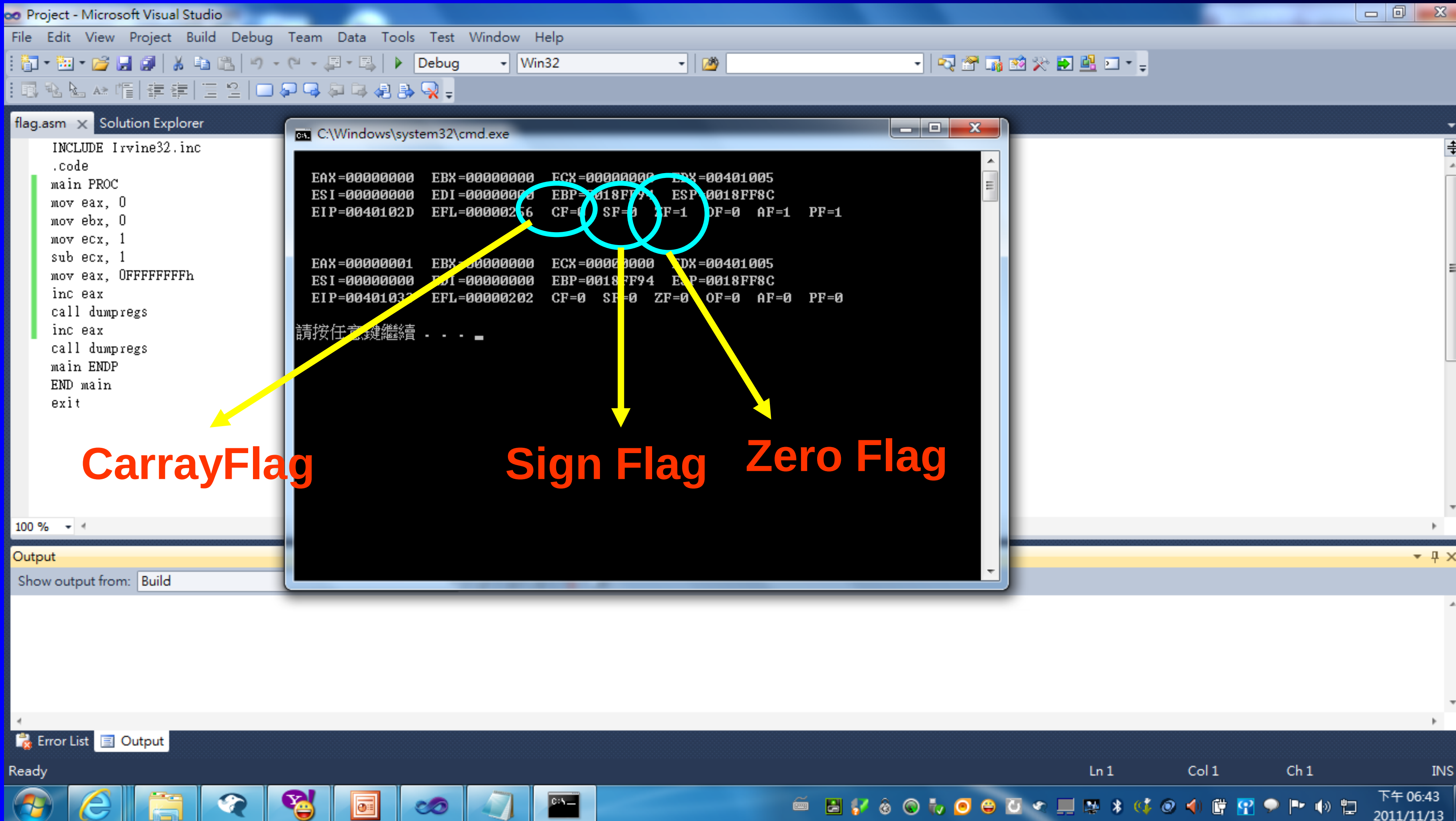
- CPU Status Flags (ZF, CF, SF, OF, PF)
- AND Instruction (*and destination, source*)
- OR Instruction (or *destination, source*)
- XOR Instruction (xor *destination, source*)
- NOT Instruction (not *destination, source*)
- Applications
- TEST Instruction
- CMP Instruction (Comparison instruction)

Status Flags - Review

- 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 (bit 7 carry is XORed with bit 6 Carry).
- The **Parity flag** is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The **Auxiliary Carry** flag is set when an operation produces a carry out from bit 3 to bit 4

Status Flags (We have seen in ch:4)

- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number



Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2015.

Status Flags - Review

What is Zero Flag?

The zero flag is set when the result of an arithmetic operation is zero.

Let us see how Zero flag works

Here is an example:

```
.code
mov ecx, 1
sub ecx, 1          ← ECX =0, ZF=1

mov eax, 0FFFFFFFFh
inc eax             ← EAX =0, ZF=1
inc eax             ← EAX =1, ZF=0

dec eax             ← EAX =0, ZF=1
```

Status Flags - Review

What is Carry Flag

The carry flag's operation is easiest to explain if we consider **addition** and **subtraction** separately.

When adding two unsigned integers, the Carry flag is a copy of the carry out of the MSB of the destination operand. Intuitively, we can say CF=1 when the sum exceeds the storage size of its destination operand.

Here is an example:

```
mov al 0FFh
add al, 1      ; AL =00, CF=1
```

See page 224

Status Flags - Review

Similarly we can show for Carry flag:

Here is another example:

```
mov ax, 00FFh
add ax, 1            ; AX =0100h, CF=0
```

But adding 1 to FFFFh in the AX register generates a Carry out of the high bit position of AX:

```
mov ax, 0FFFFh
add  ax, 1           ; AX =0000, CF=1
```

Status Flags - Review

What is sign flag?

The sign flag is set when the result of a signed operation is negative.

Here is an example:

```
mov eax, 4
sub  eax, 5                ; EAX =-1, SF= 1
```

Here is another example:

When a negative result is generated, the SF is set it 1

```
        mov bl, 1                 ; BL=01h
        sub b1, 2                 ; BL= FFh (-1), SF=1
```

Status Flags - Review

What is overflow flag?

The **overflow flag** is set when the result of a signed arithmetic operation overflows or underflows the destination operand.

Here is an example:

```
mov al, +127
add al, 1            ; OF=1 cannot store 128.
```

Here is another example:

```
mov al, -128
sub  al,1            ; OF=1
```

Status Flags - Review

What is parity flag?

The parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits. The following ADD and Sub instructions alter the parity of AL:

Here is an example:

```
mov al, 10001100b
add  al, 00000010b          ; AL =10001110, PF=1
```

Even number of bits

```
sub  al, 10000000b          ; AL = 00001110, PF=0
```

Odd number of bits

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
```

AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2015.

AND Instruction

The AND instruction lets you clear 1 or more bits in an operand without affecting other bits. This technique is called masking, much as you might use masking tape when painting a house to cover areas (such as windows) that should not be painted.

Here is an example:

With any bits we do “ and AL, 11110110” the 0th and 3rd bit left unchanged.

```
mov al, 10101110b
and  al, 11110110b          ; Result AL = 10100110
```

The AND instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flag.

AND Instruction

The AND instruction provides an easy way to translate a letter from lowercase to uppercase.
If we compare the ASCII codes of capital A and lowercase a, it becomes clear that only bit 5 is different:

01100001 = 61h (‘a)
01000001 = 41h (‘A)

Only one bit is different

```
.data
Array BYTE 50 “This Sentence is in Mixed case”,0
.code
mov ecx, LENGTHOF array
mov esi,  OFFSET array
L1:   and BYTE PTR [esi], 11011111b        ; clear bit 5
          inc esi
          loop L1
```

OR Instruction

- **Performs a Boolean OR operation between each pair of matching bits in two operands**
- **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
```

OR

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

OR Instruction

The OR instruction is particularly useful when you need to set 1 or more bits in an operand without affecting any other bits. Suppose, for example, that your computer is attached to a servo motor, which is activated by setting bit 2 in its control byte.

```
or AL, 00000100b        ; set bit 2, leave others unchanged
    Here is an example:
    mov AL, 11100011b   ;
    or  AL,  00000100b  ; result in AL = 11100111
```

The OR instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flag.

XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
- 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
```

XOR

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

XOR is a useful way to toggle (invert) the bits in an operand.

XOR Instruction

XOR instruction has the property of **"reversible"** which makes ideal tool for a simple form of Symmetric encryption.

We will see in example program later in the chapter.

The XOR instruction always clears the **Overflow** and **Carry flags**. XOR modifies the **Sign**, **Zero,** and **Parity flags** in a way that is consistent with the value assigned to the destination operand.

NOT Instruction

- Performs a Boolean NOT operation on a single destination operand
- **Syntax:**

NOT *destination*

NOT	0 0 1 1 1 0 1 1		
	1 1 0 0 0 1 0 0	—	inverted

NOT

X	¬X
F	T
T	F

NOT Instruction

The NOT instruction toggles (inverts) all bits in an operand. The result is called the one's complement.

Look at an example:

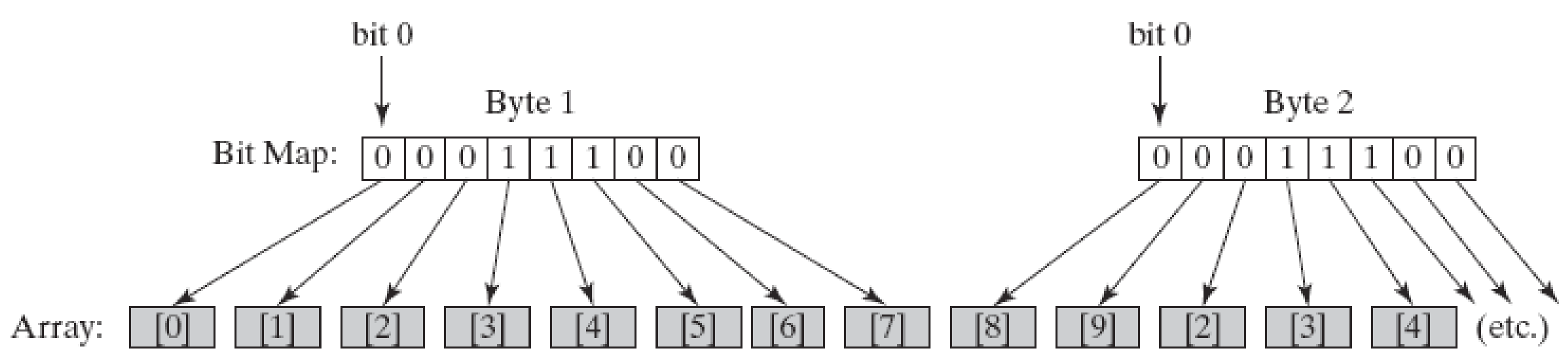
```
mov AL, 11110000b
not AL                      ; AL = 00001111b
```

For NOT instruction, No flags are affected

Bit-Mapped Sets

- Binary bits indicate set membership
- Efficient use of storage
- Also known as *bit vectors*

Figure 6–1 Mapping Binary Bits to an Array.



Bit-Mapped Sets

In **Bit-Mapped Sets**, some applications manipulate sets of items selected from a limited sized universal set.

Ex:

Employees within a company,

Environment monitoring from weather monitoring.

Example:

SetX = 10000000 00000000 00000000 0000111

(in the above bit 0,1,2 and 31 are set.)

```
mov eax, SetX
and eax, 10000b    ; is element [16] a member of a set.
```

Bit-Mapped Set Operations

- **Set Complement**

```
mov eax,SetX
not eax
```

- Set Intersection

```
mov eax,setX
and eax,setY
```

- Set Union

```
mov eax,setX
or  eax,setY
```

Set Complement

Set Complement

The complement of a set can be generated using the NOT instruction, which reverses all bits.

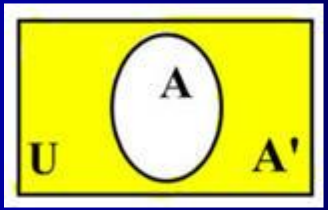
Therefore, the complement of the SetX that we introduced is generated in EAX using the following instructions:

Example:

SetX = 10000000 00000000 00000000 0000111

mov eax, SetX
not eax, SetY

The complement of a set is the set of all elements in the universal set that are not in the initial set.



A' is the complement

Set Intersection

Set Intersection:

The AND instruction produces a bit vector that represents the intersection of two sets.

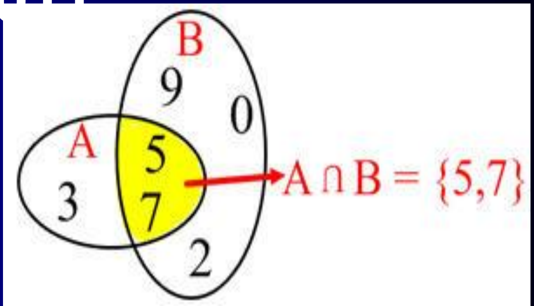
The following code generates and stores the intersection of SetX and SetY in EAX:

Example:

SetX = 10000000 00000000 00000000 00000111
SetY = 10000001 01010000 00000111 01100011
10000000 00000000 00000000 00000011

```
mov eax, SetX
and eax, SetY
```

A set formed by all **common elements** to two or more sets is called **intersection of sets**.



A larger domain would require more bits than could be held in a single register, making it necessary to use a loop to AND all of the bits together.

Set Union

Set Union:

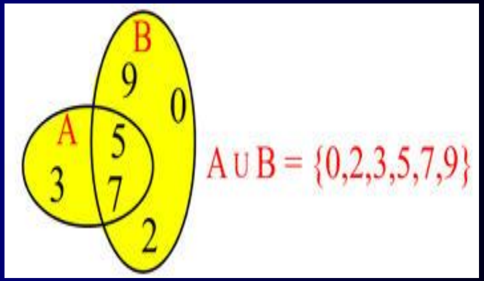
The OR instruction produces a bit map that represents the union of two sets. The following code generates the union of SetX and SetY in EAX:

Example:

SetX = 10000000 00000000 00000000 00000111
SetY = 10000001 01010000 00000111 01100011
10000001 01010000 00000111 01100111

```
mov eax, SetX
or eax, SetY
```

A set formed by all elements of two or more sets is called the union of those sets.



Applications (1 of 5)

- **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 = 0100001b
```

Applications (2 of 5)

- **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
```

The ASCII digit '6' = 00110110b

Applications (3 of 5)

- **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.

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

This code only runs in Real-address mode, and it does not work under Windows NT, 2000, or XP.

Applications (4 of 5)

- **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
```

JZ (jump if Zero) is covered in Section 6.3.

Your turn: Write code that jumps to a label if an integer is negative.

Applications (5 of 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

- **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
```

Jump if not zero
Zero flag clear

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

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

Jump if zero
Zero flag Set

CMP Instruction (1 of 3)

- 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
```

CMP Instruction (2 of 3)

- **Example: destination > source**

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

(both the Zero and Carry flags are clear)

CMP Instruction (3 of 3)

The comparisons shown here are performed with signed integers.

- **Example: destination > source**

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

- **Example: destination < source**

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

What's Next

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

Conditional Jumps

- Jumps Based On . . .
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed Comparisons
- Applications
- Encrypting a String
- Bit Test (BT) Instruction

J*cond* Instruction

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

Specific jumps: (B – Below, E –Equal NE-NotEqual

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 - jump to a label if the Sign flag is set

JNE, JNZ - jump to a label if the Zero flag is clear

JECXZ - jump to a label if ECX = 0

J*cond* Ranges

- **Prior to the 386:**
  - jump must be within –128 to +127 bytes from current location counter
- **x86 processors:**
  - 32-bit offset permits jump anywhere in memory

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

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

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)

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2015.

Applications (1 of 5)

- **Task:** Jump to a label if **unsigned** EAX is greater than EBX
- **Solution:** Use CMP, followed by JA

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

- **Task:** Jump to a label if **signed** EAX is greater than EBX
- **Solution:** Use CMP, followed by JG

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

Applications (2 of 5)

- **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 (3 of 5)

- **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 (4 of 5)

- 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
```

Applications (5 of 5)

- **Task:** Jump to label L1 if bits 0, 1, and 3 in AL are **all set**.
- **Solution:** 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
```

Your turn . . .

- Write code that jumps to label L1 if either bit 4, 5, or 6 is set in the BL register.
- Write code that jumps to label L1 if bits 4, 5, and 6 are all set in the BL register.
- Write code that jumps to label L2 if AL has even parity.
- Write code that jumps to label L3 if EAX is negative.
- Write code that jumps to label L4 if the expression (EBX – ECX) is greater than zero.

Application: Sequential search of an Array

Look at an example on page 224 and 225 about this sequential search

Go through the code and try to understand yourself.
Scanning an array for the first nonzero value.

Take a look at the program!
Try to run the program and understand how it is written!

String Encryption Program

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

View the Encrypt.asm program's source code. Sample output:

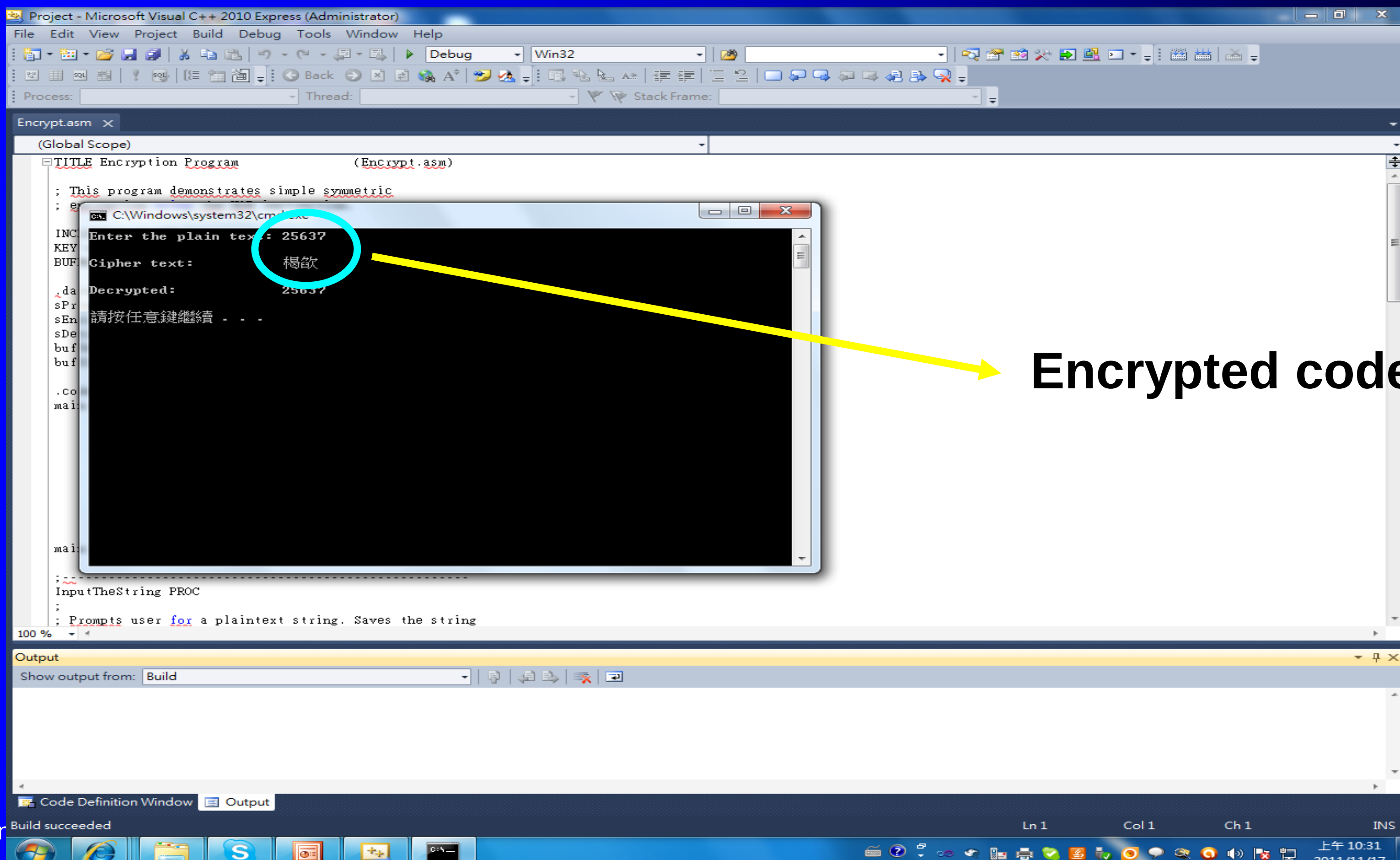
```
Enter the plain text: Attack at dawn.

Cipher text: «¢¢Äîä-Ä¢-ïÄÿ -Gs

Decrypted: Attack at dawn.
```

Application: Encryption Program

Look at an example on page 226 and 227 about this Encryption program



Encrypted code

Application: Encryption Program

Let us look at the code and see what is going on?

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

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  ?
```

Application: Encryption Program cont…

```
.code
main PROC
callInputTheString              ; input the plain text
callTranslateBuffer             ; encrypt the buffer
movedx,OFFSET sEncrypt          ; display encrypted message
callDisplayMessage
callTranslateBuffer             ; decrypt the buffer
movedx,OFFSET sDecrypt          ; display decrypted message
callDisplayMessage
exit
main ENDP
```

Application: Encryption Program cont…

```
;----------------------------------------------------
InputTheString PROC
; Prompts user for a plaintext string. Saves the string
; and its length.
; Receives: nothing
; Returns: nothing
;----------------------------------------------------
Pushad                          ; Notice that it does pushad
movedx,OFFSET sPrompt           ; display a prompt
callWriteString
movecx,BUFMAX                   ; maximum character count
movedx,OFFSET buffer            ; point to the buffer
callReadString                  ; input the string
movbufSize,eax                  ; save the length
callCrlf
Popad                           ; before leaving you need to popad
ret
InputTheString ENDP
```

Application: Encryption Program cont…

```
;-----------------------------------------------------
DisplayMessage PROC
; Displays the encrypted or decrypted message.
; Receives: EDX points to the message
; Returns:  nothing
;-----------------------------------------------------
Pushad                          ; push the address in stack
callWriteString
movedx,OFFSET buffer            ; display the buffer
callWriteString
callCrlf
callCrlf
popad                           ; pop the address in stack
ret
DisplayMessage ENDP
```

Application: Encryption Program cont…

```
;-----------------------------------------------------
TranslateBuffer PROC
; Translates the string by exclusive-ORing each
; byte with the encryption key byte.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
pushad
movecx,bufSize          ; loop counter
movesi,0                ; index 0 in buffer
L1:
xorbuffer[esi],KEY      ; translate a byte
Incesi                  ; point to next byte
loopL1
popad
ret
TranslateBuffer ENDP
```

Encrypting a String **Page 226**

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
```

APPLICATION: SIMPLE STRING ENCRYPTION

XOR instruction has an interesting property. If an integer X is XORed with Y and the resulting value is XORed with Y again, the value produced is X:

$$((X \otimes Y) \otimes Y) = X$$

This *reversible* property of XOR provides an easy way to perform a simple form of data encryption: A plain text message is transformed into an encrypted string called *cipher text* by XORing each of its characters with a character from a third string called a key. The intended viewer can use the key to *decrypt* the cipher text and produce the original plain text.

BT (Bit Test) Instruction

- Copies bit *n* from an operand into the Carry flag
- Syntax: BT *bitBase, n*
  - bitBase may be *r/m16* or *r/m32*
  - n may be *r16, r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

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

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* *(Loop if equal)*

  LOOPZ *destination* *(Loop if Zero)*
- **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. 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* *(loop if not Zero)*

  LOOPNE *destination* *(loop if not equal)*

- Logic:
  - ECX ← 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:
```

Your turn . . .

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:
```

Your turn . . .

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.)

Your turn . . .

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:
```

Your turn . . .

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:
```

Your turn . . .

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:

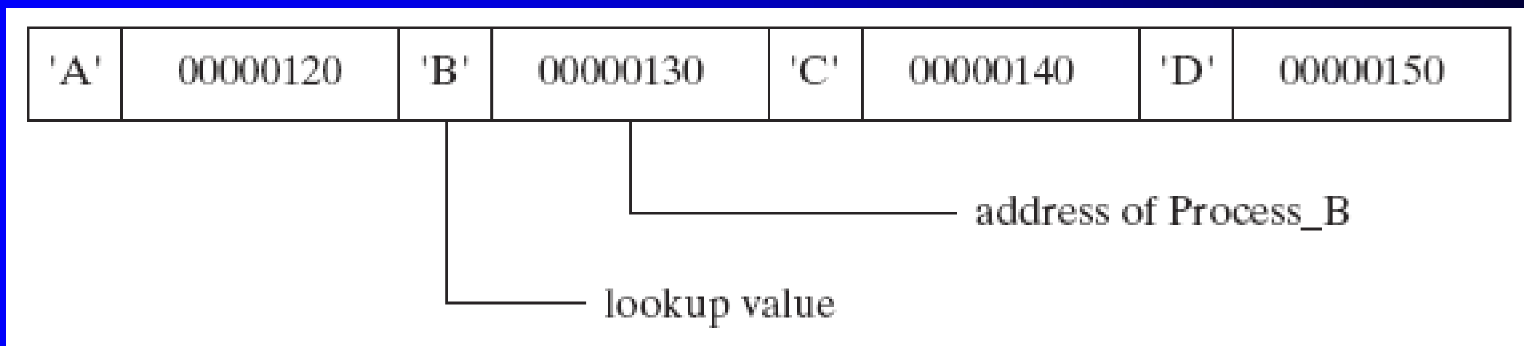


Table-Driven Selection (4 of 4)

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

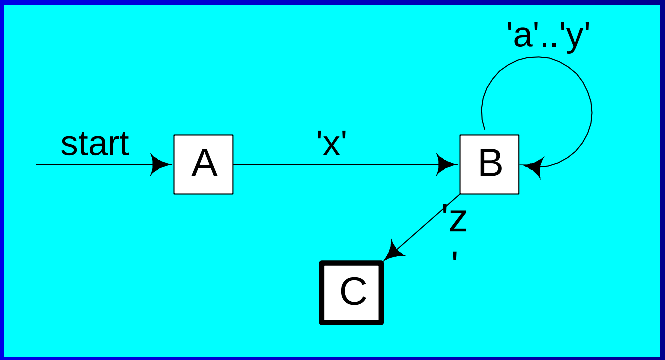
- A FSM is a specific instance of a more general structure called a directed graph.
- Three basic states, represented by nodes:
  - Start state
  - Terminal state(s)
  - Nonterminal state(s)

Finite-State Machine

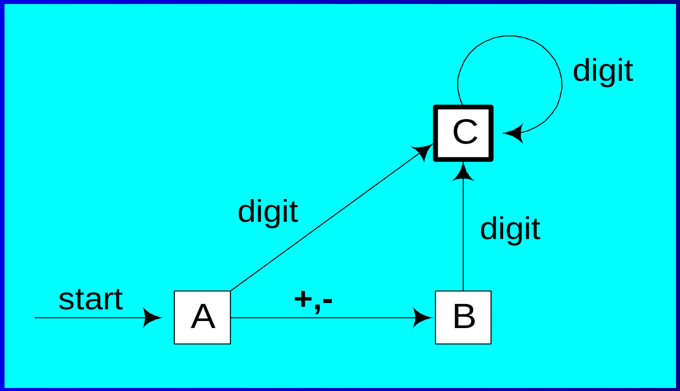
- **Accepts any sequence of symbols that puts it into an accepting (final) state**
- **Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)**
- Advantages:
  - Provides visual tracking of program's flow of control
  - Easy to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

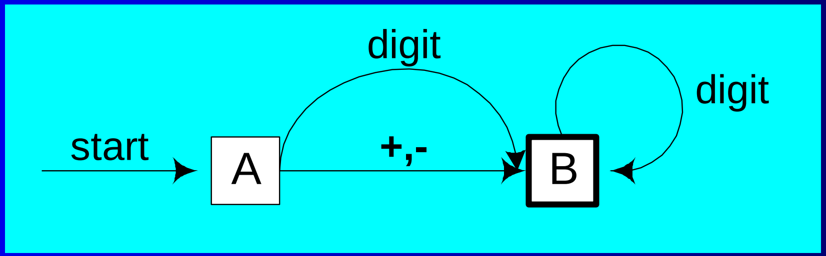


- FSM that recognizes signed integers:



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
```

View the Finite.asm source code.

Examine a complete program on Chapter 6 Ex: Finite state machine 1/5

```
TITLE Finite State Machine              (Finite.asm)

; This program implements a finite state machine that
; accepts an integer with an optional leading sign.

INCLUDE Irvine32.inc

ENTER_KEY = 13
.data
InvalidInputMsg BYTE "Invalid input",13,10,0
```

Ex: Finite state machine 2/5

```
.code
main PROC
        call Clrscr

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 contains a digit
        jz      StateC              ; go to State C
        call    DisplayErrorMsg     ; invalid input found
        jmp     Quit
```

Ex: Finite state machine 3/5

```
StateB:
        call    Getnext             ; read next char into AL
        call    IsDigit             ; ZF = 1 if AL contains a digit
        jz      StateC
        call    DisplayErrorMsg     ; invalid input found
        jmp     Quit
StateC:
        call    Getnext             ; read next char into AL
        call    IsDigit             ; ZF = 1 if AL contains a digit
        jz      StateC
        cmp     al,ENTER_KEY              ; Enter key pressed?
        je      Quit                      ; yes: quit
        call    DisplayErrorMsg     ; no: invalid input found
        jmp     Quit
Quit:
        call    Crlf
        exit
main ENDP
```

Ex: Finite state machine 4/5

```
;-----------------------------------------------
Getnext PROC
;
; Reads a character from standard input.
; Receives: nothing
; Returns: AL contains the character
;-----------------------------------------------
        call ReadChar                   ; input from keyboard
        call WriteChar              ; echo on screen
        ret
Getnext ENDP
```

Ex: Finite state machine 5/5

```
;-----------------------------------------------
DisplayErrorMsg PROC
;
; Displays an error message indicating that
; the input stream contains illegal input.
; Receives: nothing.
; Returns: nothing
;-----------------------------------------------
        push  edx
        mov   edx,OFFSET InvalidInputMsg
        call  WriteString
        pop   edx
        ret
DisplayErrorMsg ENDP
END main
```

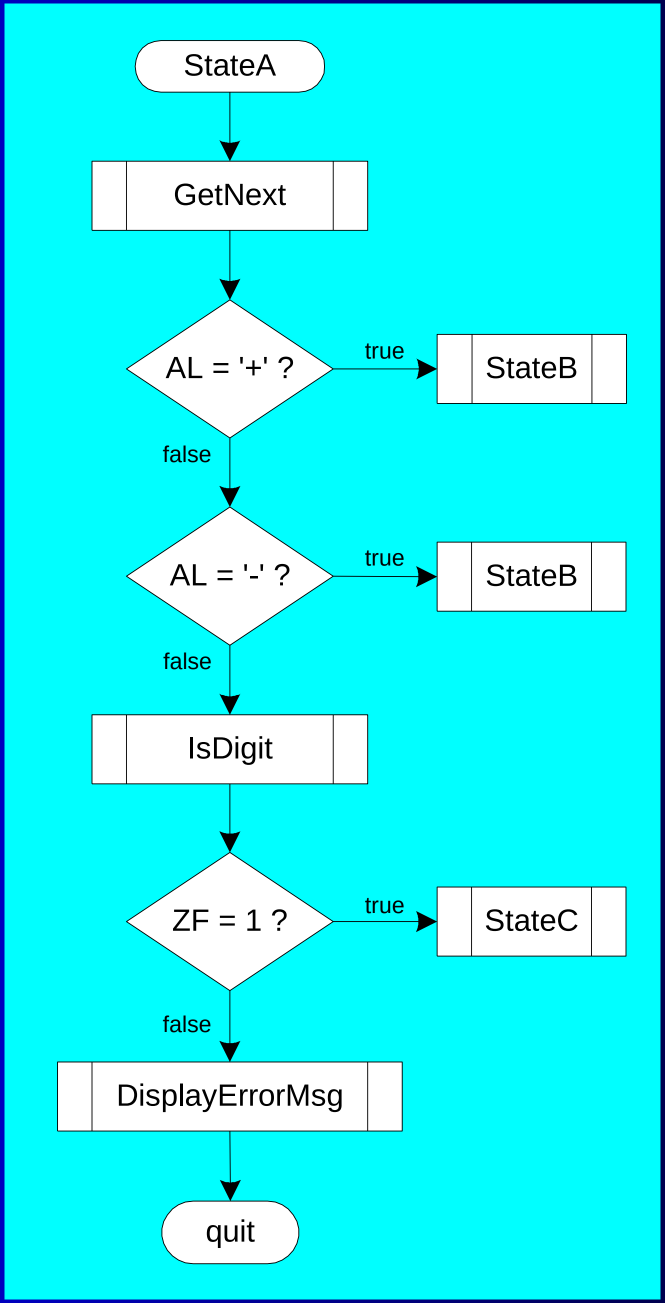
IsDigit Procedure

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  ax,0         ; ZF = 1
ID1: ret
IsDigit ENDP
```

Flowchart of State A

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



Your Turn . . .

- 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.

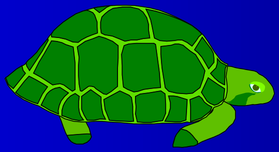
Example:

```
; 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**



4C 6F 70 70 75 75 6E