CSC 3210 Computer Organization and Programming

CHAPTER 6: CONDITIONAL PROCESSING

```
if A > B ...

while X > 0 and X < 200 ...

if check_for_error( N ) = true
```

Outline

- **Boolean and Comparison Instructions**
- Conditional Jumps
- Conditional Structures
- Conditional Control Flow Directives

Boolean and Comparison Instructions

- CPU Status Flags
- AND Instruction
- OR Instruction
- XOR Instruction
- NOT Instruction
  - Applications
- TEST Instruction
- CMP Instruction

```
if A > B ...
while X > 0 and X < 200 ...
if check_for_error( N ) = true
```

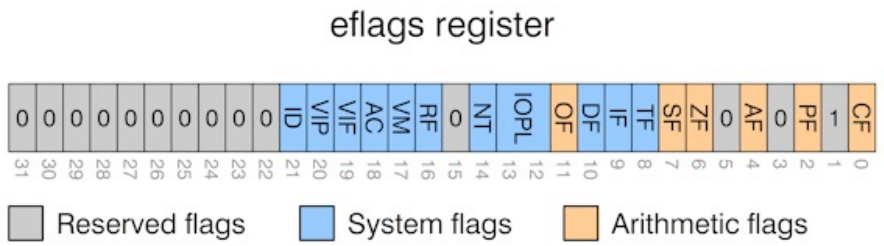
Boolean operations are the core of all **decision statements** because they **affect** the **CPU status flags**.

Status Flags - Review

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

Table 6-1 Selected Boolean Instructions.

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 operand 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.



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.
- The Parity flag is set when an instruction generates an **even number** of **1** bits in the **low byte** of the destination operand.

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)

```
AND reg,reg
AND reg,mem
AND reg,imm
AND mem,reg
AND mem,imm
```

AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

AND Instruction

- **Flags**
  - The AND instruction always clears the Overflow and Carry flags.
  - It modifies the Sign, Zero, and Parity flags
    - Consistent with the value assigned to the **destination operand**.

- **Example:**
  - Suppose the following instruction **results in** a value of Zero in the al register.
  - Thus, the Zero flag will be **set**:

```
and al,1Fh
```

AND Example

- **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.

OR Instruction

- Performs a **Boolean OR** operation between each pair of matching bits in two operands
- **Syntax**: `OR destination, source`

```
OR reg,reg
OR reg,mem
OR reg,imm
OR mem,reg
OR mem,imm
```

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

OR Instruction

- **Flags**
  - The OR instruction always clears the Carry and Overflow flags.
  - It modifies the Sign, Zero, and Parity flags
    - consistent with the value assigned to the destination operand.

- **Example**: OR a number with itself (or zero) to **obtain certain information about its value**:

```
or al,al
```

OR Instruction

- The values of the Zero and Sign flags indicate the following about the contents of AL:

Zero Flag	Sign Flag	Value in AL Is . . .
Clear	Clear	Greater than zero
Set	Clear	Equal to zero
Clear	Set	Less than zero

OR Example

- 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**.

XOR Instruction

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

Syntax: `XOR destination, source`

XOR

x	y	x ⊕ 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

- **Example:**
  - **bit masking:**
    - A bit exclusive-ORed with 0 **retains its value**,
    - A bit exclusive-ORed with 1 is **toggle** (complemented).

```
          0 0 1 1 1 0 1 1
XOR       0 0 0 0 1 1 1 1
          -----------------
          0 0 1 1 | 0 1 0 0
unchanged ————————|         |———————— inverted
```

XOR

x	y	x ⊕ 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

- **Flags**
  - The XOR instruction always clears the Overflow and Carry flags.
  - XOR modifies the Sign, Zero and Parity flags
    - consistent with the value assigned to the destination operand.

- **Example:**
  - **An effective way to check the parity of a number** without changing its value is to exclusive-OR the number with zero:

```
mov  al,10110101b      ; 5 bits = odd parity
xor  al,0              ; Parity flag clear (odd)
mov  al,11001100b      ; 4 bits = even parity
xor  al,0              ; Parity flag set (even)
```

The Parity flag (PF) is set when The least significant byte of the **destination** has an **even number of 1 bits**.

XOR Instruction : Another Property

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

XOR Application: Encrypting a String

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

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

```
KEY = 239                          ; can be any byte value between 1-255
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
```

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

XOR Application: Encrypting a String

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¢-ïÄÿü-Gs
Decrypted: Attack at dawn.
```

NOT Instruction

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

Syntax: **NOT** *destination*

```
NOT reg
NOT mem
```

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

- Flags
  - No flags are affected by the NOT instruction.

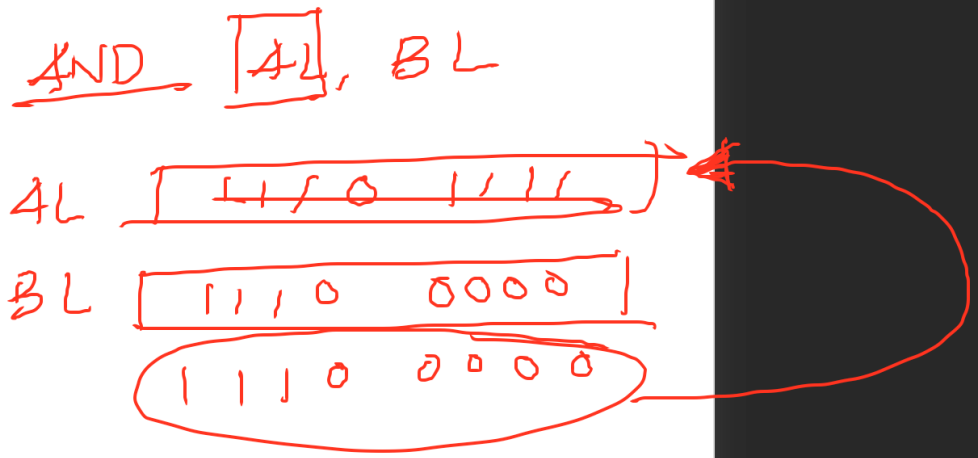
TEST Instruction

- Performs a **nondestructive AND operation** between **each pair of matching bits** in two operands
- No operands are modified
- **Sets** the Sign, Zero, and Parity flags based on the value assigned to the destination operand.

  - **Example 1**: jump to a label if either **bit 0** or **bit 1** in AL is **set**.

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

ZF = ?



CMP Instruction

- The most common **boolean expressions** involve some type of **comparison**:

```
if A > B ...
while X > 0 and X < 200 ...
if check_for_error( N ) = true
```

- **CMP** instruction is used to compare integers (signed and unsigned)

- Compares the **destination** operand to the **source** operand (HOW 🤔)
  - **CMP** performs **implied subtraction** of **source operand** from **destination operand** 🗝️
  - **Nondestructive subtraction** of source from destination (destination operand is not changed)

- **Syntax**:

  CMP *destination, source*

CMP Instruction (unsigned)

- **Flags**
  - The **CMP** instruction changes the Overflow, Sign, Zero, Carry, Auxiliary Carry, and Parity flags
    - According to the value the destination operand would have had if actual subtraction had taken place.

1) When two **unsigned operands** are compared,
   - Zero and Carry flags indicate the following relations between operands:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

Why CF is 1?

CMP Instruction (signed)

- **Flags**

2) When two **signed operands** are compared,
   - the Sign, Zero, and Overflow flags indicate the following relations between operands:

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

CMP Instruction (unsigned integers)

- The comparisons shown here are performed with **unsigned integers**.

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

- **Example1**: destination == source

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

- **Example2**: destination < source

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

CMP Instruction (unsigned integers)

- **Example3**: destination > source

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

(both the Zero and Carry flags are clear)

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

CMP Instruction (signed integers)

- The comparisons shown here are performed with **signed integers**

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

- **Example1**: destination > source

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

- **Example2**: destination < source

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

Outline

- Boolean and Comparison Instructions
- **Conditional Jumps**
- Conditional Structures
- Conditional Control Flow Directives

JZ = Jump if ZF = 1

```
MOV AX,5
MOV X,5
SUB AX,BX
JZ L1
ADD AX,1
ADD BX,2
L1:
ADD CX,2
```

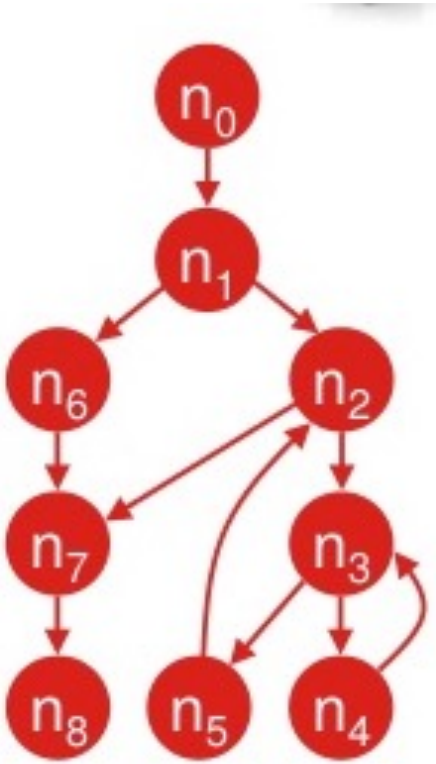
, ZF = 1

Implement **high-level logic structures** using a combination of comparisons and jumps.

```
cmp eax,0
jz L1              ; jump if ZF = 1
```

```
if A > B ...
while X > 0 and X < 200 ...
```

```
int rst(int r, int s, int t){  //n0
    if (r > 0 || s > 0) {      //n1
      while (r != s) {         //n2
          while (r > s) {      //n3
            r = r - s;         //n4
          }
          r = s;               //n5
      }
    } else { r = t; }          //n6
    return r;                  //n7
}                              //n8
```



Control flow graph

Conditional Jumps

- **Jumps Based On . . .**
  - **Specific flags**
  - **Equality**
  - **Unsigned comparisons**
  - **Signed Comparisons**
- Applications
- Search of an Array

Compare and then Jump

- **First,** an operation such as CMP, AND, or SUB modifies the **CPU status flags**.
- **Second**, a conditional jump instruction **tests** the flags and **causes a branch** to a new address.

Jcond Instruction

Jcond

→ JC = Jump if CF = 1

JNC = Jump if NOT (CF = 1)

CF = 0

- A conditional jump instruction **branches** to a label
  - When specific **status flag** condition **is true**
- Syntax

Jcond destination

- **cond** refers to a flag condition identifying the state of one or more flags.
- The following examples are based on the **Carry** and **Zero** flags:

JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

CPU status flags are most commonly **set** by arithmetic, comparison, and boolean instructions.

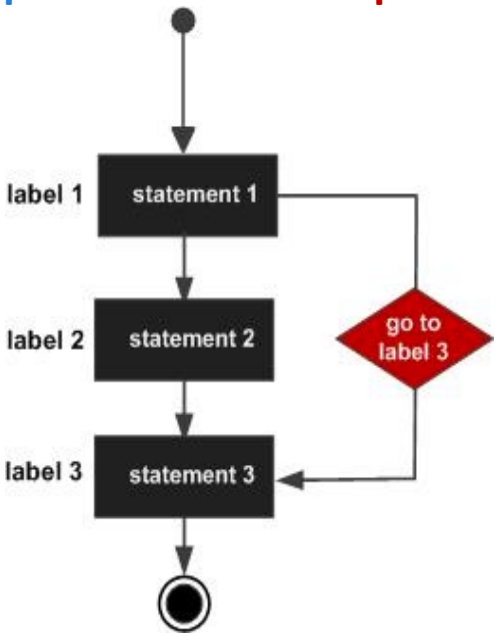
Conditional Jumps (Example1)

- **First,** an operation such as CMP, AND, or SUB modifies the **CPU status flags**.
- **Second**, a conditional jump instruction **tests** the flags and **causes a branch** to a new address.

- The CMP instruction compares EAX to Zero.
- The JZ (Jump if zero) instruction jumps to label L1 **if** the Zero flag was set **by the CMP** instruction:

```
        cmp eax,0
        jz L1           ; jump if ZF = 1
        .
        .
    L1:
        .
        .
```

Compare and then Jump or NOT



Conditional Jumps (Example2)

- **First,** an operation such as CMP, AND, or SUB modifies the **CPU status flags**.
- **Second**, a conditional jump instruction **tests** the flags and **causes a branch** to a new address.

Compare and then Jump

- The AND instruction performs a bitwise AND on the DL register, **affecting** the Zero flag.
- The JNZ (jump if not Zero) instruction jumps if the Zero flag is clear:

```
        and dl,10110000b
        jnz L2                  ; jump if ZF = 0
        .
        .
L2:
```

Types of Conditional Jumps Instructions

- **Conditional jump** instructions are able to
  - **Compare** signed and unsigned integers and
    - **Perform actions** based on the values of individual CPU flags.
- The conditional jump instructions can be divided into **four groups**:
  - Jumps based on **specific flag** values
  - Jumps based on **equality** between operands or the value of (E)CX
  - Jumps based on comparisons of **unsigned** operands
  - Jumps based on comparisons of **signed** operands

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

CMP leftOp, rightOp

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
JRCXZ	Jump if RCX = 0 (64-bit mode)

Jumps Based on Unsigned Comparisons

CMP leftOp, rightOp

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)

A and B

37

Jumps Based on Signed Comparisons

CMP leftOp, rightOp

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)

G and L

38

Attendance!

Conditional Jumps

- Jumps Based On . . .
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed Comparisons
- **Applications**
- Search of an Array

Applications 1

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

```
cmp eax,ebx
ja  Larger          ;jump if above
```

A and B

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

```
cmp eax,ebx
jg  Greater         ; jump if greater
```

G and L

Applications 2

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

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

A and B

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

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

G and L

Applications 3

- 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:
```

A and B

- 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:
```

G and L

Applications 4

- 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

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

  2. 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
```

Conditional Jumps

- Jumps Based On . . .
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed Comparisons
- Applications
- **Search of an Array**

Search of an Array

- A common programming task is to **search for values in an array** that meet some criteria
- **Example**:
  - The following program looks for the first nonzero value in an array of **16-bit integers**
    - If it finds one, it displays the value
    - Otherwise, it displays a message stating that a nonzero value was not found

```
; Scanning an Array                          (ArrayScan.asm)
; Scan an array for the first nonzero value.

INCLUDE Irvine32.inc

.data
intArray  SWORD  0,0,0,0,1,20,35,-12,66,4,0
;intArray SWORD  1,0,0,0                    ; alternate test data
;intArray SWORD  0,0,0,0                    ; alternate test data
;intArray SWORD  0,0,0,1                    ; alternate test data
noneMsg  BYTE "A non-zero value was not found",0
```

```
intArray  SWORD  0,0,0,0,1,20,35,-12,66,4,0

.code
main PROC
    mov   ebx,OFFSET intArray       ; point to the array
    mov   ecx,LENGTHOF intArray     ; loop counter

L1: cmp   WORD PTR [ebx],0          ; compare value to zero
    jnz   found                     ; found a value
    add   ebx,2                     ; point to next
    loop  L1                        ; continue the loop
    jmp   notFound                  ; none found

found:                              ; display the value
    movsx eax,WORD PTR[ebx]         ; sign-extend into EAX
    call  WriteInt
    jmp   quit

notFound:                           ; display "not found" message
    mov   edx,OFFSET noneMsg
    call  WriteString

quit:
    call Crlf
    exit
main ENDP
END main
```

Why use ebx and not esi?

WriteInt
Writes a **signed** 32-bit integer to the console window in decimal format.

WriteString
Writes a **null-terminated string** to the console window.

Crlf:
Writes an end-of-line sequence to the console window.

Outline

- Boolean and Comparison Instructions
- Conditional Jumps
- **Conditional Structures**
- Conditional Control Flow Directives

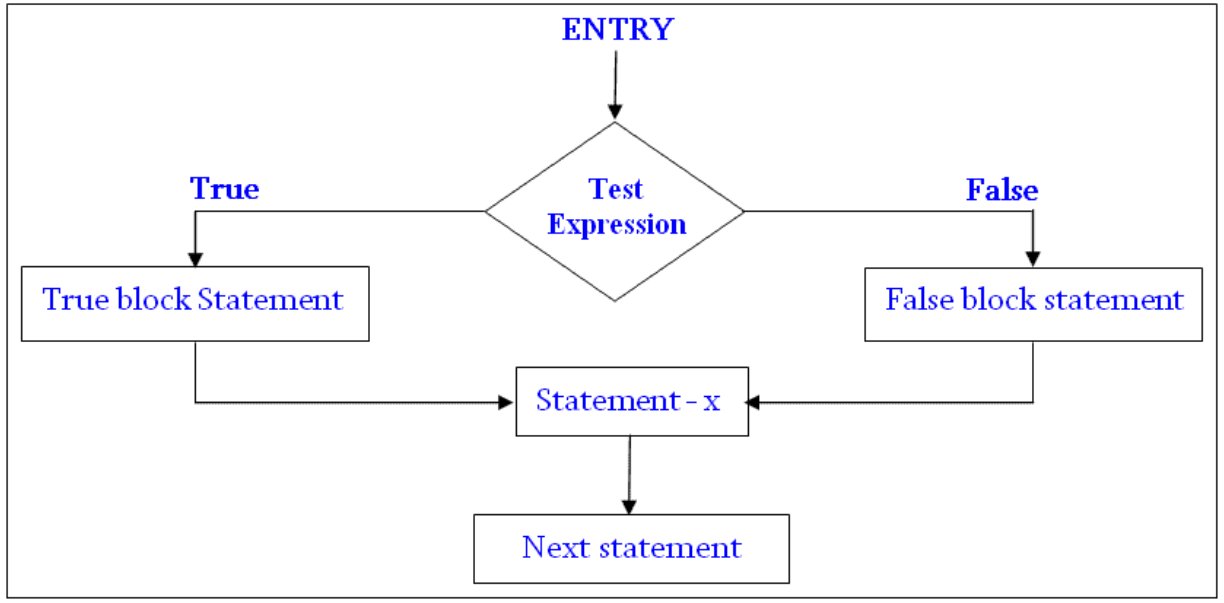
Conditional Structures

- **Block-Structured IF Statements**
  - Compound Expressions with AND
  - Compound Expressions with OR
- WHILE Loops

Conditional Structures

- A **conditional structure** is defined to be one or more **conditional expressions**
- Those **conditional expressions trigger** **a choice** between different logical **branches**
- Each **branch** **causes a different sequence of instructions** to execute.

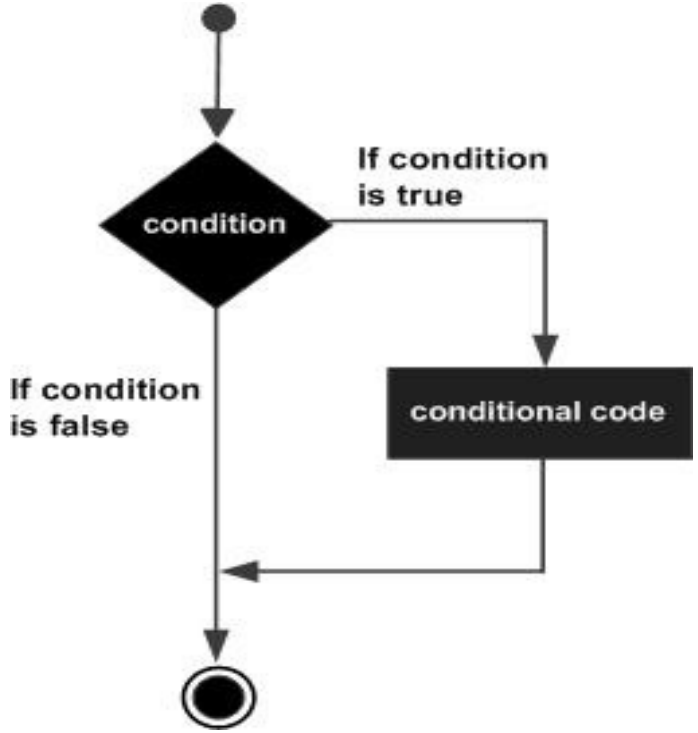
- When the **condition** is **true**, **execute** the **body**
- When the **condition** is **false**, don't execute the **body**, **jump** over it.



Block-Structured IF Statements

- An **IF structure** **imply** that **a boolean expression** is followed by **one** or **two lists of statements**:
- Two types:
  1. **IF-Then**
     - **Statement/s** performed when the expression is true
     - **Next statement/s** performed when the expression is false:

```
if( boolean-expression )
  statement-list-1
.
.
.
```



```
if (AX > BX) {
    AX = AX + 1;
    BX = BX + 2;
} // if block
next:
```

if (AX > BX) is True.

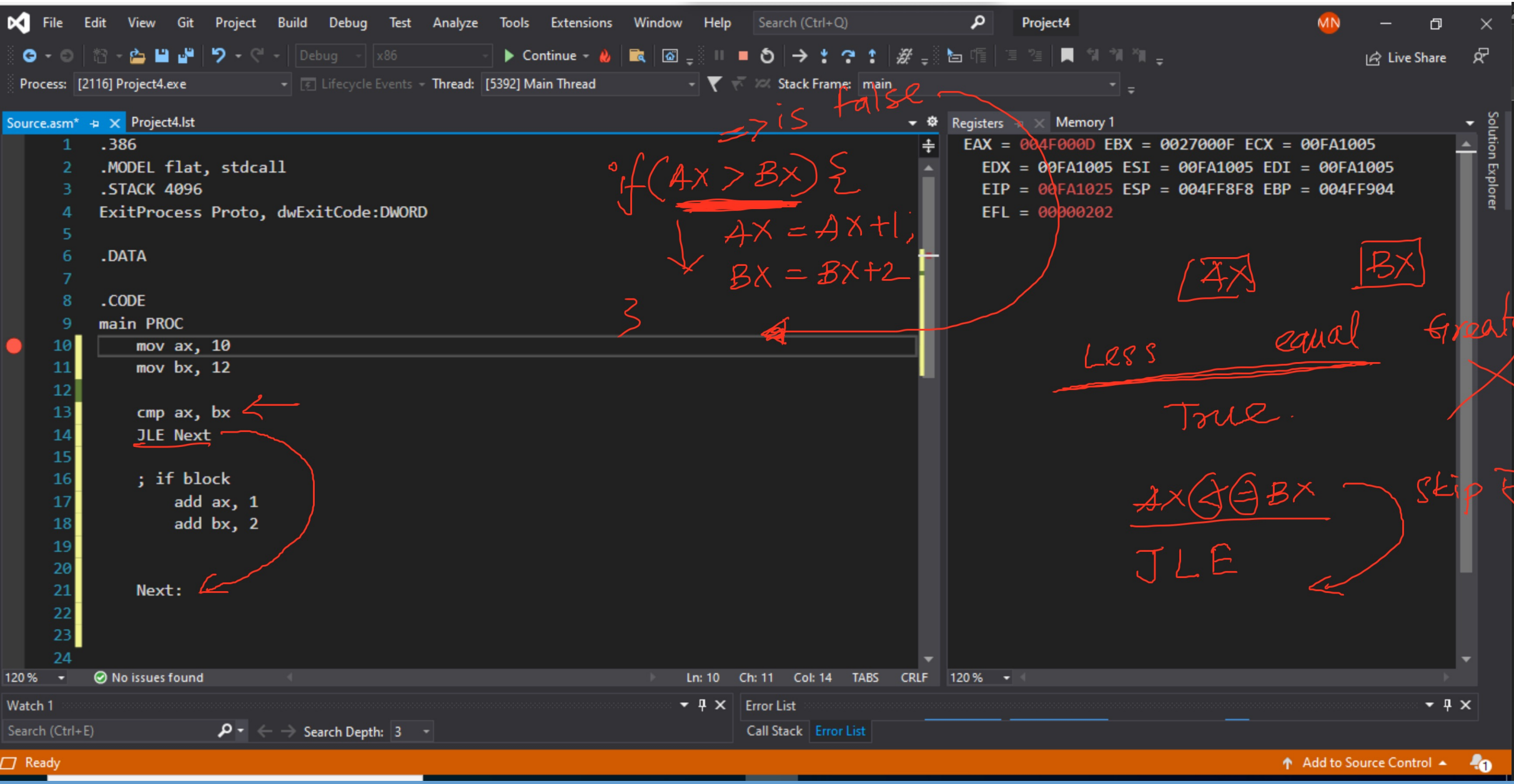
execute if block

otherwise

(skip the if block.)

```
CMP AX, BX
JNA next
    AX = AX + 1;
    BX = BX + 2;
next:
```

When condition is false skip the block.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Continue
Live Share
Process: [2116] Project4.exe
Lifecycle Events
Thread: [5392] Main Thread
Stack Frame: main
Source.asm*
Project4.lst
Registers
Memory 1
Solution Explorer
1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7
8 .CODE
9 main PROC
10 mov ax, 10
11 mov bx, 12
12
13 cmp ax, bx
14 JLE Next
15
16 ; if block
17 add ax, 1
18 add bx, 2
19
20
21 Next:
22
23
24
EAX = 004F000D EBX = 0027000F ECX = 00FA1005
EDX = 00FA1005 ESI = 00FA1005 EDI = 00FA1005
EIP = 00FA1025 ESP = 004FF8F8 EBP = 004FF904
EFL = 00000202
120 %
No issues found
Ln: 10 Ch: 11 Col: 14 TABS CRLF
Watch 1
Search (Ctrl+E)
Search Depth: 3
Error List
Call Stack
Error List
Ready
Add to Source Control
if(AX > BX) {
AX = AX+1;
BX = BX+2
}
is false
AX
BX
Less
equal
Greater
True.
AX ≤ BX
JLE
skip if block

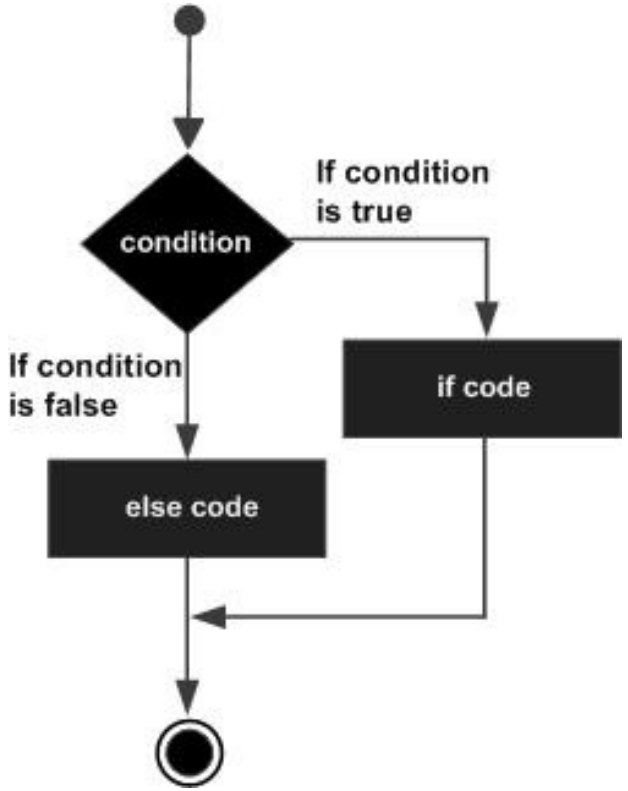
Block-Structured IF Statements

2. IF-Then-Else

- **Statement/s** performed when the expression is true
- **Another** performed when the expression is false:

```
if( boolean-expression )
  statement-list-1
else
  statement-list-2
```

The else part is optional.



Block-Structured IF Statements

- IF statement is translated into assembly language with a
  - **CMP instruction** followed by
  - **Conditional jumps**.
- **When op1** or **op2** is a **memory operand** (a variable),
  - **One of them must be moved to a register** before executing **CMP**.