

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**.

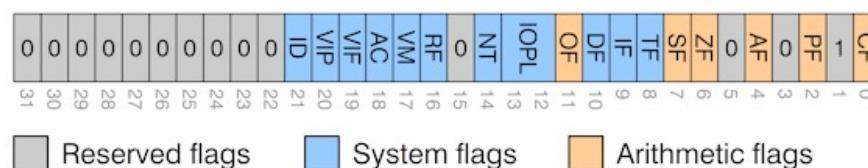
Status Flags - Review

- 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.

eflags register



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	$x \wedge y$
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	$x \vee y$
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	$x \oplus y$
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

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

$$\begin{array}{r} 00111011 \\ \text{XOR } 00001111 \\ \hline \text{unchanged } \boxed{0011|0100} \text{ inverted} \end{array}$$

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

XOR

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

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
BUFSIZE = 128
.data
buffer BYTE BUFSIZE+1 DUP(0)
bufSize DWORD BUFSIZE

.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: «¢¢Äiä-Ä¢-iÄÿü-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: «¢¢Äiää-Ä¢-iÄÿü-Gs
```

```
Decrypted: Attack at dawn.
```

NOT Instruction

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

Syntax: **NOT** *destination*

NOT *reg*

NOT *mem*

$$\begin{array}{r} \text{NOT} \quad 00111011 \\ \hline 11000100 \end{array}$$

— inverted

NOT

X	$\neg X$
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      ZF = ?  
jnz ValueFound
```

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**.
- Example1:** destination == source

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

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

- 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
```

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

(both the Zero and Carry flags are clear)

CMP Instruction (signed integers)

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

- 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
```

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

Outline

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

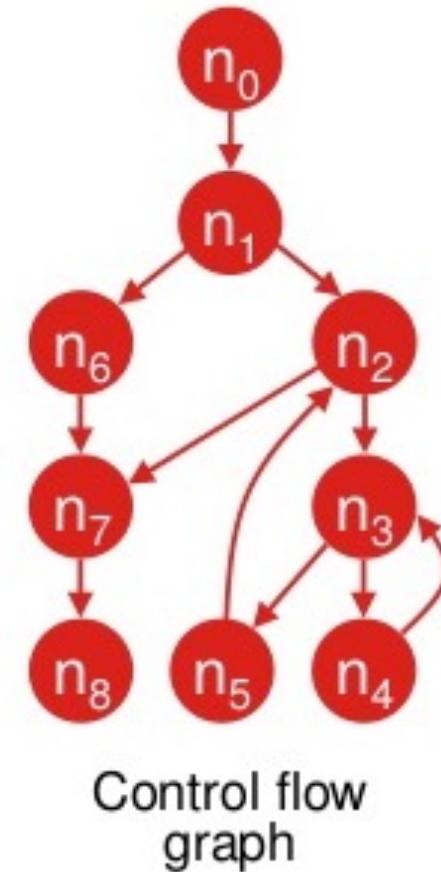
if A > B ...

while X > 0 and X < 200 ...

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

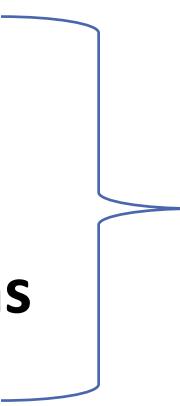
```
cmp eax,0  
jz L1           ; jump if ZF = 1
```

```
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
```



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.

J_{cond} Instruction

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

J**cond** 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)

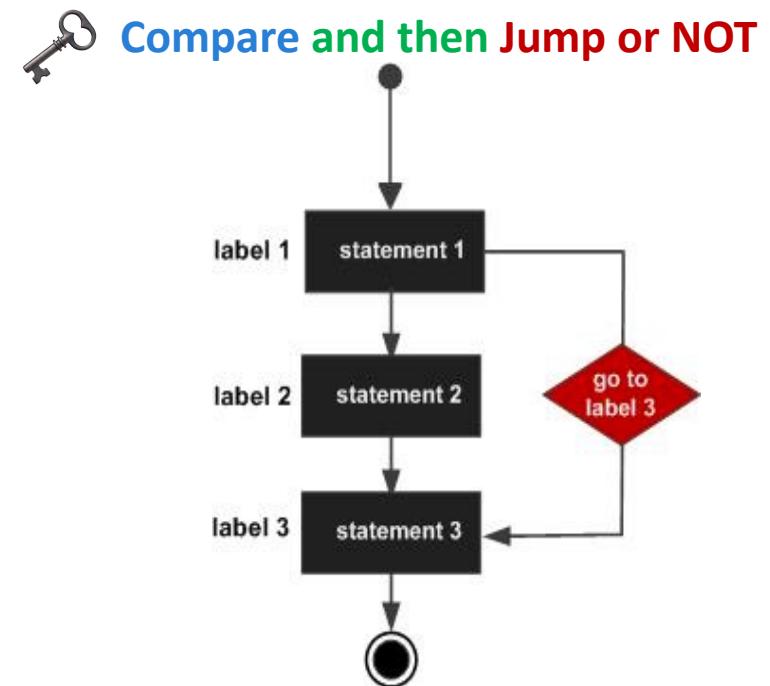
CPU status flags are most commonly set by **arithmetic**, **comparison**, and **boolean** instructions.

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:  
. .
```



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 ($\text{leftOp} = \text{rightOp}$)
JNE	Jump if not equal ($\text{leftOp} \neq \text{rightOp}$)
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 $leftOp \geq rightOp$)
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 $leftOp \leq rightOp$)
JNA	Jump if not above (same as JBE)

A and B

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 $leftOp \geq rightOp$)
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 $leftOp \leq rightOp$)
JNG	Jump if not greater (same as JLE)

G and L

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
```

A and B

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
```

G and L

Next:

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
```

Why use ebx and not esi?

```
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
```

[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

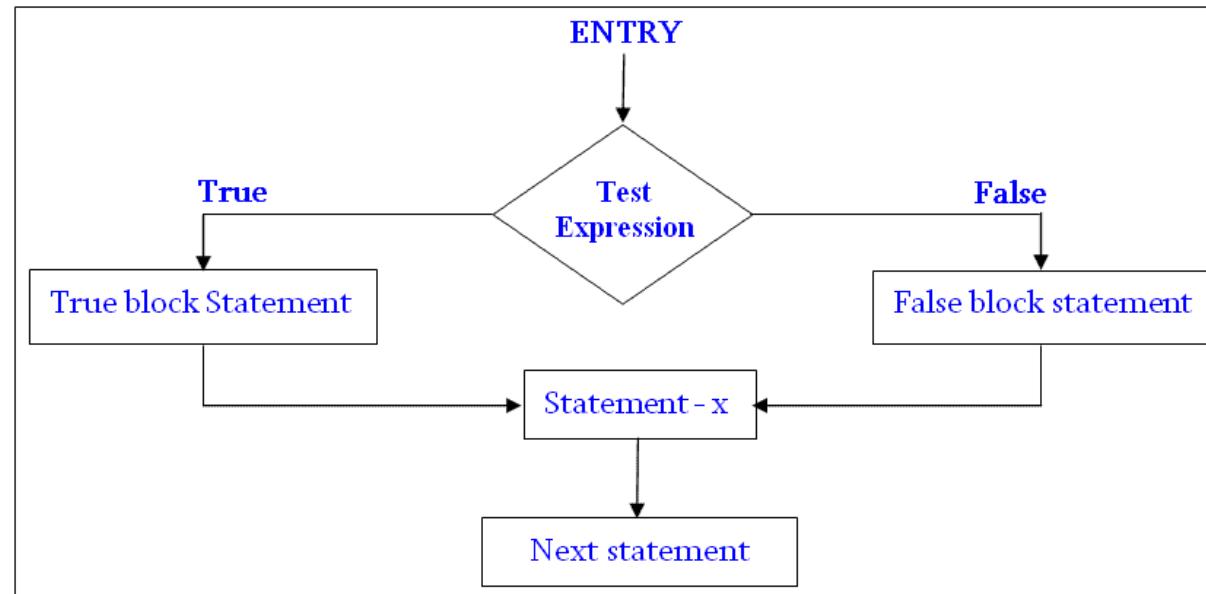
Conditional Structures

- **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

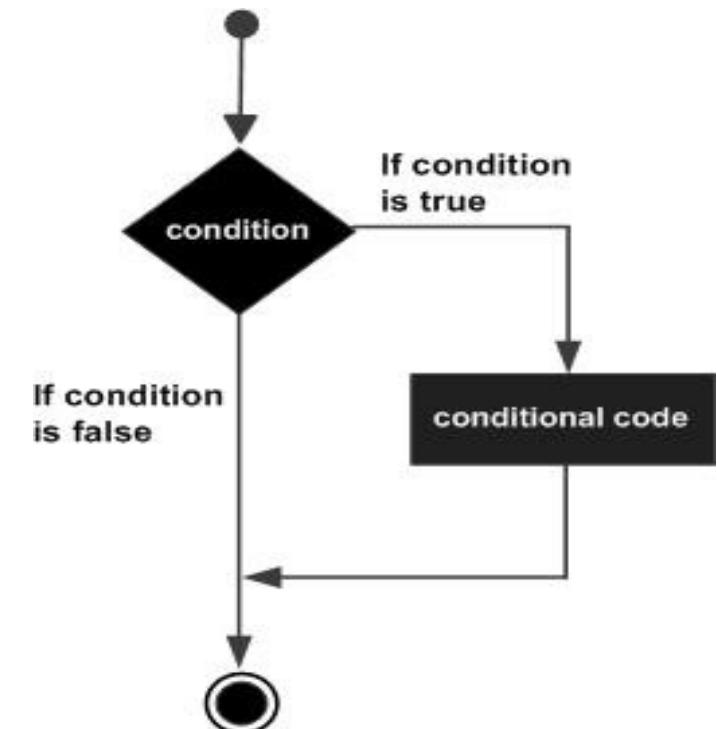
- An **IF structure** imply that **a boolean expression** is followed by **one or two lists of statements**:
- Two types:

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
```

```
    .
    .
    .
```



File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Server Explorer Toolbox

Source.asm* Project4.lst

```
1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7
8
9 .CODE
10 main PROC
11     cmp ax, bx
12     JG if_block
13     JMP next
14
15     if_block:
16         INC AX
17         Add BX, 2
18
19     Next:
20     INVOKE ExitProcess, 0
21 main ENDP
22 END main
```

JG if_block

if(AX > BX){

if(AX <= BX)

skip

AX = AX + 1;

BX = BX + 2;

3

Output

Show output from: Debug

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4\Debug\Project4.exe'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'
The program '[7724] Project4.exe' has exited with code 0 (0).

Solution Explorer

Project4

References

External Dependencies

Source.asm

Properties

120% No issues found Ln: 20 Ch: 7 Col: 10 TABS CRLF

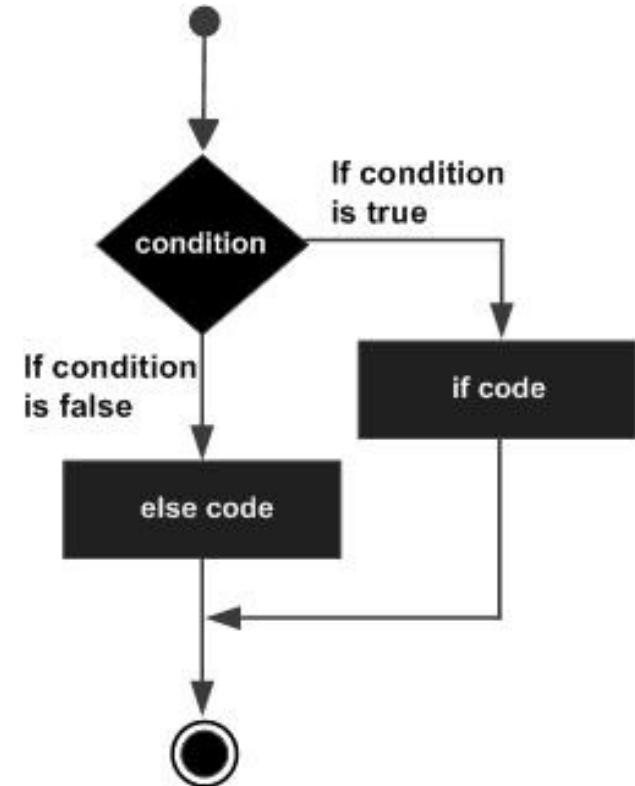
Block-Structured IF Statements

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
```

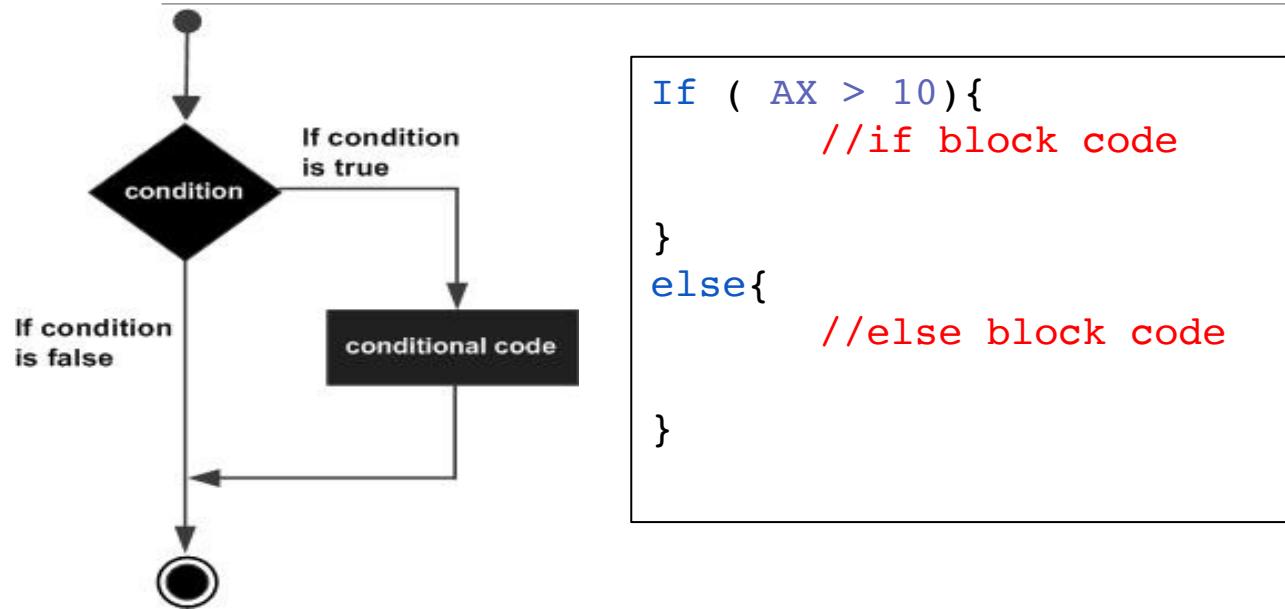
The else part
is optional.



Block-Structured IF Statements

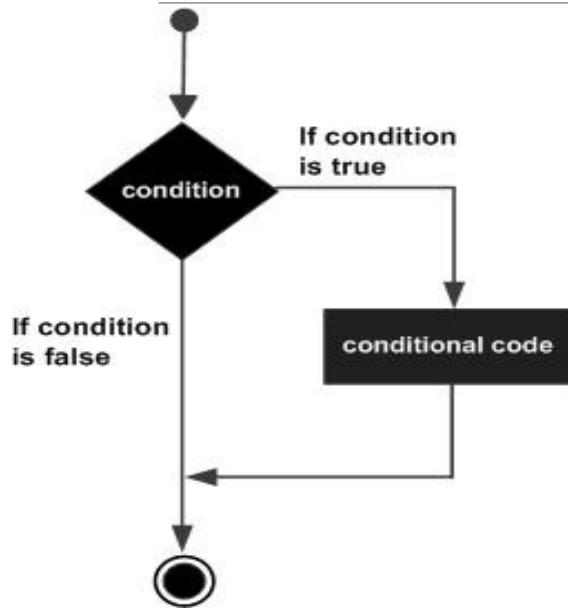
- IF statement is translated into assembly language with a
 - **CMP instruction** followed by
 - **Conditional jumps.**
- When op1 or op2 is a **memory operand** (a variable),
 - **One of them must be moved to a register** before executing **CMP**.

Block-Structured IF Statements



```
If ( AX > 10){  
    //if block code  
}  
else{  
    //else block code  
}
```

Block-Structured IF Statements

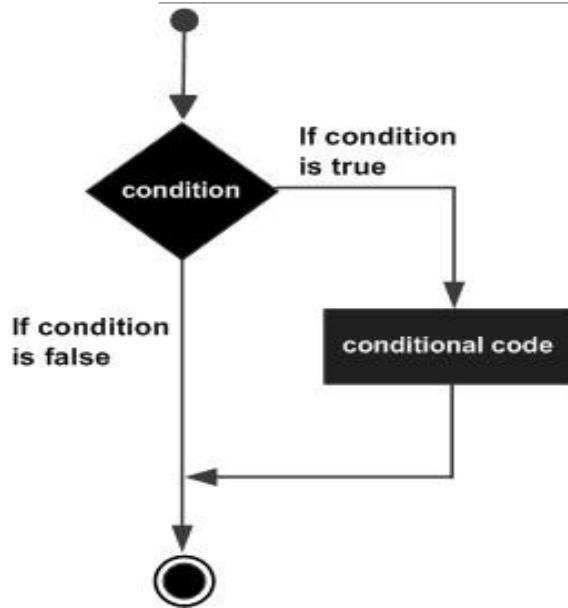


```
If ( AX > 10){  
    //if block code  
}  
else{  
    //else block code  
}
```

CMP AX, 10
Jump (conditional) to ELSE if (AX>10) is FALSE
// if block code

Label:
// ELSE block code

Block-Structured IF Statements



```
If ( AX > 10){  
    //if block code  
}  
else{  
    //else block code  
}
```

Problem! : Conditional Jumps trigger
when CONDITION is TRUE!

CMP AX, 10

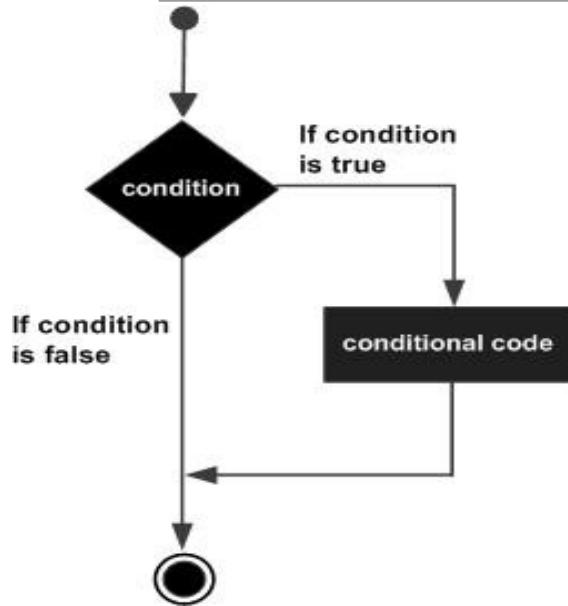
Jump (conditional) to ELSE if (AX>10) is FALSE

// if block code

Label:

// ELSE block code

Block-Structured IF Statements



```
If ( AX > 10){  
    //if block code  
}  
else{  
    //else block code  
}
```

CMP AX, 10
Jump (conditional) to ELSE if (AX>10) is FALSE
// if block code

Label:
// ELSE block code

Solution : Negate the condition
Now, it can jump when condition is TRUE.

Block-Structured IF Statements

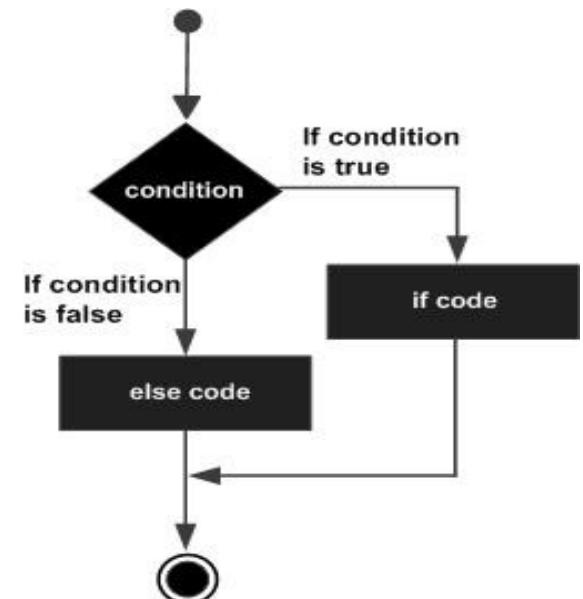
- When the **condition** is **true**, execute the **body**
- When the **condition** is **false**, don't execute the **body**, **jump** over it.
- This is the **jump** case

1

Thus, **jump** when the **condition** is **false**.



- x86 Jump instructions **jump** when the **condition** is **true**
- Thus, we often need to **reverse** the **condition**.



Block-Structured IF Statements

- How to reverse a condition?
 - o Using DeMorgan's Law
 - o In order to branch on a negated condition, we need to know the negation of various conditions

NOT [$(BL > CL)$ AND $(AL < BL)$]
≡ NOT $(BL > CL)$ OR NOT $(AL < BL)$ =
≡ $(BL \leq CL)$ OR $(AL \geq BL)$

Condition	Negated Condition
$x > y$	$x \leq y$
$x \geq y$	$x < y$
$x < y$	$x \geq y$
$x \leq y$	$x > y$
	\neq
	$\neg (cond1) \quad \neg (cond2)$
	$\neg (cond1) \quad \& \quad \neg (cond2)$

IF Statements: Example1 (IF-Then)

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

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

Reverse The IF Condition

(There are multiple correct solutions to this problem.)

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

IF Statements: Example1 (IF-Then)

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

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

Reverse The IF Condition

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

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

A and B

(There are multiple correct solutions to this problem.)

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4 MN - Live Share

Source.asm* Project4.lst

```
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, 5
12    cmp AX, BX
13    JLE else_block ; when if condition is false NOT (AX > BX) == (AX <= BX)
14
15    if_block:
16        INC AX
17        ADD BX, 2
18        JMP next
19
20    else_block:
21        DEC AX
22        SUB BX, 2
23
24    Next:
25
26
27    INVOKE ExitProcess, 0
28 main ENDP
```

No issues found

Ln: 18 Ch: 8 Col: 14 TABS CRLF

Output

Show output from: Debug

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4\Debug\Project4.exe' (Win32).
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'.
The program '[7724] Project4.exe' has exited with code 0 (0).

NOT $(AX > BX)$ = False
 $(AX \leq BX)$ = True.

if $(AX > BX)$ {
True $\rightarrow AX = AX + 1$; //if block
 $BX = BX + 2$; }
else {
 $AX = AX - 1$; //else block
 $BX = BX - 2$;
}

Solution Explorer

Search Solution Explorer (P)
Solution 'Project4' (1 of 1 projects)
Project4
- References
External Dependencies
Source.asm

Properties

Add to Source Control

IF Statements: Example2 (IF-Then-Else)

- Implement the following pseudocode in assembly language.

($op1 \neq op2$)

```
if( op1 == op2 )  
    X = 1;  
else  
    X = 2;
```

Reverse The IF Condition

MOV EAX, OP1

CMP eax, op2

X JNE elseblock

↳ ifblock:

MOV X, 1

JMP next

else block:

MOV X, 2

next:

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

IF Statements: Example2 (IF-Then-Else)

- Implement the following pseudocode in assembly language.

```
if( op1 == op2 )
    X = 1;
else
    X = 2;
```

Reverse The IF Condition

```
mov eax, op1
cmp eax, op2
jne L1
mov X, 1
jmp L2
L1: mov X, 2
L2:
```

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

IF Statements: Example3 (IF-Then-Else)

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

**Do not Reverse
the Condition**

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

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

(There are multiple correct solutions to this problem.)

IF Statements: Example3 (IF-Then-Else)

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

**Do not Reverse
the Condition**

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

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

1. Compare two operands
2. If condition TRUE
jump to IF BLOCK
3. ELSE Block
4. Jump over IF block
5. IF block

(There are multiple correct solutions to this problem.)

IF Statements: Example3 (IF-Then-Else)

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

**Do not Reverse
the Condition**

```
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:
```

G and L

(There are multiple correct solutions to this problem.)

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

Conditional Structures

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

Attendance

Compound Expression with AND (Example1)

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

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4 MN Live Share

Source.asm* Project4.lst

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 x DWORD 0
8
9 .CODE
10 main PROC
11     mov AL, 10
12     mov bl, 5
13
14 ; compare the first expression al>bl
15 cmp al, bl
16 JG exp2 ; when al > bl, check the second expression
17 JMP next ; when first expression is false, jump to next
18 exp2:
19 ; compare the second expression, bl > cl
20 cmp bl, cl
21 JG if_block ; when bl > cl , enter if block
22 JMP next ; when second exp is false (bl <= cl)
23
24 if_block:
25     add x, 1
26
27 next:
28     TINVOKED ExitProcess

```

Output

Show output from: Debug

```

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4.exe'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'
The program '[7724] Project4.exe' has exited with code 0 (0x0)

```

Handwritten notes:

- $\text{if } [AL > BL] \text{ AND } [BL > CL] \{$
- $x = x + 1;$ ←
- 3
- Diagram showing the flow of control through three comparison blocks:
 - (AL > BL)
 - (BL > CL)
 - (AND)
 The diagram shows the flow of control from the first comparison to the second, and then to the AND block. Red circles indicate the state of variables: AL, BL, and CL. The first comparison (AL > BL) has two states: 1 (true) and 0 (false). The second comparison (BL > CL) also has two states: 1 (true) and 0 (false). The AND block has two states: 0 (false) and 1 (true). The final state of x is shown as 1. A handwritten note "enter the if block" points to the AND block.

Compound Expression with AND (Example1)

- 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;
```

Do not Reverse

A and B

```
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 (Example1)

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

Reverse The IF Condition

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Local Windows Debugger

Source.asm* Project4.lst

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 x DWORD 0
8
9 .CODE
10 main PROC
11
12     cmp al, bl
13     JLE Next    ; when first condition is false (al <= bl), skip ifblock
14
15     cmp bl, cl ; when first exp is true, evaluate second exp.
16     JLE Next    ;when second exp is false, (bl <= cl) skip the ifblock
17
18     if_block:
19         mov x, 1
20
21     Next:
22
23     INVOKE ExitProcess, 0
24 main ENDP
25 END main
26
27
28

```

Output Show output from: Debug

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4\Source.asm' at 0x401000
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll' at 0x77E10000
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll' at 0x77E20000
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll' at 0x77E30000
The program '[7724] Project4.exe' has exited with code 0 (0x0)

If $(AL > BL)$ AND $(BL > CL)$ Σ

$x = x + 1;$

3

$(AL > BL) \quad | \quad (BL > CL) \quad | \quad AND$

0	0	0
1	0	0
1	1	1

skip ifblock.

enter ifblock

Server Explorer Toolbox Solution Explorer Solution 'Project4' (1 of 1 p) Project4 References External Dependencies Source.asm

Solution Explorer Git Changes Properties

Add to Source Control 1

Compound Expression with AND (Example2)

- 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:

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

Reverse The IF Condition

```
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:
.
.
```

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

Method 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:
```

Do not Reverse
the Condition

Method 2

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

Reverse the
Condition

Compound Expression with AND (Example3)

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

Reverse
The IF
Condition

```
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.)

Conditional Structures

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

Compound Expression with OR

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

Reverse the second Condition

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Local Windows Debugger

Source.asm* Project4.lst

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 x DWORD 0
8
9 .CODE
10 main PROC
11     cmp al, bl
12     JG if_block ;when first exp is true al > bl, enter the if-block
13
14     cmp bl, cl ;when first exp is false, al <= bl, evaluate second exp.
15     JLE Next ;when second exp is false bl<=cl, skip ifblock
16
17     if_block:
18         add x, 1
19
20     Next:
21     INVOKE ExitProcess, 0
22
23 main ENDP
24 END main
25
26
27

```

$al = 10$

$bl = 5$

$x = 0$

$(al > bl) \text{ OR } (bl > cl)$

$x = x + 1$

$bl <= cl$

\exists

$(al > bl) | (bl > cl) | OR$

Can enter if block

Cannot enter if block

Output

Show output from: Debug

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4\Debug\Project4.exe'

'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'

'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'

'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'

The program '[7724] Project4.exe' has exited with code 0 (0x0)

Solution Explorer

Search Solution Explorer

Project4

References

External Dependencies

Source.asm

Properties

Add to Source Control

Compound Expression with OR

- In the following implementation, the code branches to L1
 - if the first expression is true
 - otherwise, it falls through to the second **CMP instruction**.
- The second expression reverses the `>` operator and uses **JBE** instead:

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

Reverse the second Condition

```
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:
```

Conditional Structures

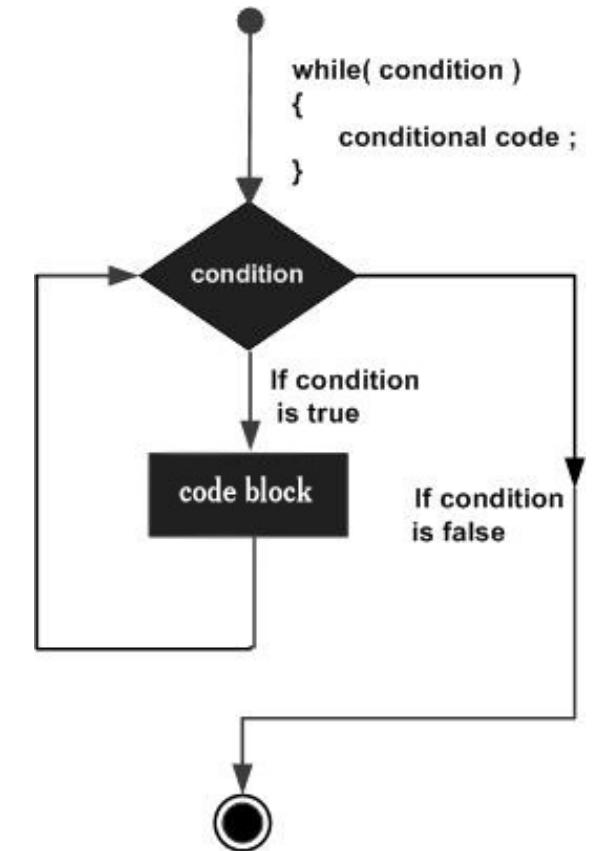
- Block-Structured IF Statements
 - Compound Expressions with AND
 - Compound Expressions with OR
- **WHILE Loops**

WHILE Loops

- A WHILE loop **tests a condition first** before performing a block of statements.
- As long as the loop condition remains true, the statements are repeated.

```
while( val1 < val2 )
{
    val1++;
    val2--;
}
```

- When implementing this structure in assembly language,
 - it is convenient to **reverse the loop condition**
 - and **jump** to endwhile if a condition becomes true



File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Local Windows Debugger

Source.asm* Project4.lst

```

1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 x DWORD 0
8 y DWORD 0
9
10 .CODE
11 main PROC
12
13 L1:
14     cmp ax, bx ; check the while loop condition
15     JLE next ; if condition is false (AX <= BX), break the loop
16     add x, 1    // loop body
17     add y, 2
18     JMP L1
19
20 Next:
21
22     INVOKE ExitProcess, 0
23
24 main ENDP
25 END main
26
27

```

Output Show output from: Debug

'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4\Source.asm'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'
The program '[7724] Project4.exe' has exited with code 0 (0x0)

Diagram annotations:

- Top Loop:** A blue bracket encloses the code from L1 to the first JMP instruction. Handwritten notes above it say "While (true) {". Below the first JMP are handwritten assignments: $x = x + 1$ and $y = y + 2$. To the right, a red circle highlights the condition $(AX \leq BX)$.
- Bottom Loop:** A red bracket encloses the code from the first JMP to the second JMP. Handwritten notes above it say "While (AX > BX) {". Below the second JMP are handwritten assignments: $AX = AX + 1$ and $BX = BX + 2$.

Server Explorer Toolbox Solution Explorer Properties Diagnostic Tools

You are screen sharing Stop Share

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Local Windows Debugger

Live Share

Source.asm* Project4.lst

```
1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 x DWORD 0
8 y DWORD 0
9
10 .CODE
11 main PROC
12     L1:    add x, 1
13         add y, 2
14         cmp ax, bx ; check the while loop condition
15         JLE next ; if condition is false (AX <= BX), break the loop
16         JMP L1
17     Next:
18     INVOKE ExitProcess, 0
19 main ENDP
20 END main
```

Output

Show output from: Debug

```
'Project4.exe' (Win32): Loaded 'C:\Users\Zulkar\Source\Repos\Project4\Project4.exe'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'
'Project4.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'
The program '[7724] Project4.exe' has exited with code 0 (0x0)
```

Do {
 $AX = AX + 1$
 $BX = BX + 2$
} While ($AX > BX$);

Server Explorer Toolbox Solution Explorer Diagnostic Tools

Solution Explorer (1 of 1 projects)
Solution 'Project4'
Project4
References
External Dependencies
Source.asm

Solution Explorer Git Changes Properties

WHILE Loops: Example1

```
while( val1 < val2 )  
{  
    val1++;  
    val2--;  
}
```

Reverse The loop Condition

G and L

WHILE Loops: Example1

```
while( val1 < val2 )
{
    val1++;
    val2--;
}
```

Reverse The loop Condition

G and L

	mov eax,val1	; copy variable to EAX?
beginwhile:		
	cmp eax,val2	; if not (val1 < val2)
	jge endwhile	; exit the loop
	inc eax	; val1++;
	dec val2	; val2--;
	jmp beginwhile	; repeat the loop
endwhile:		
	mov val1,eax	; save new value for val1