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**.

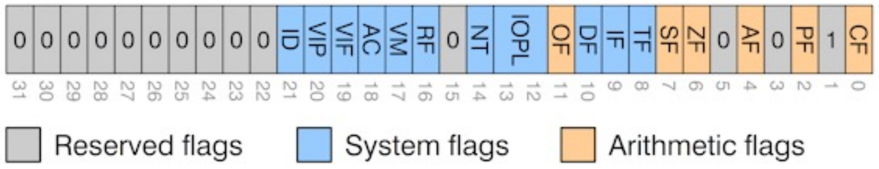
Status Flags - Review

- 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).

```
         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

- **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	$SF \neq OF$
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	$SF \neq OF$
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

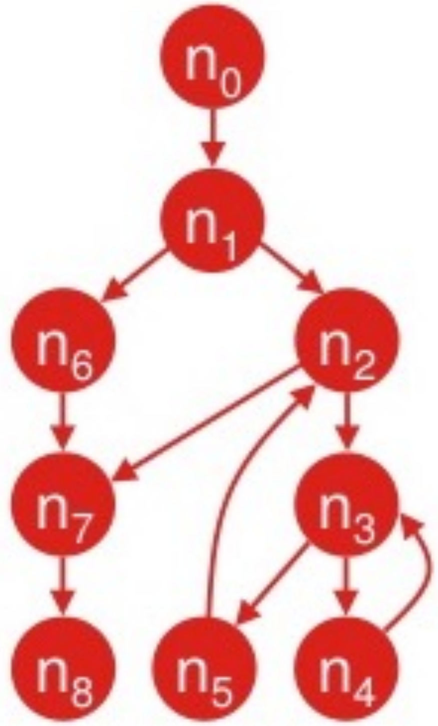
if A > B ...

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
```



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

- 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:

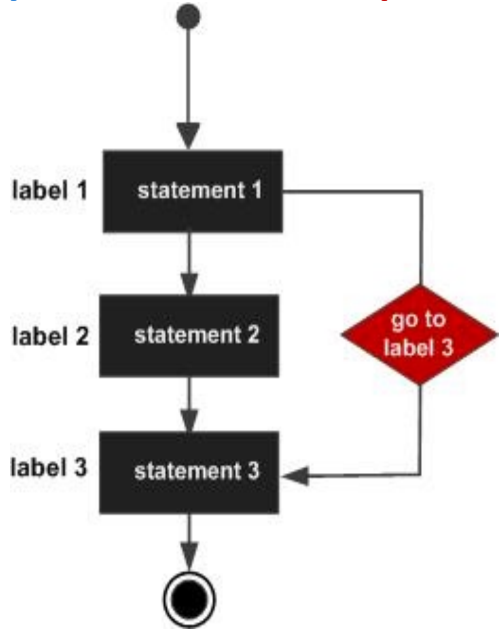
JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

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.

Compare and then Jump or NOT



- 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 (*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

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

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

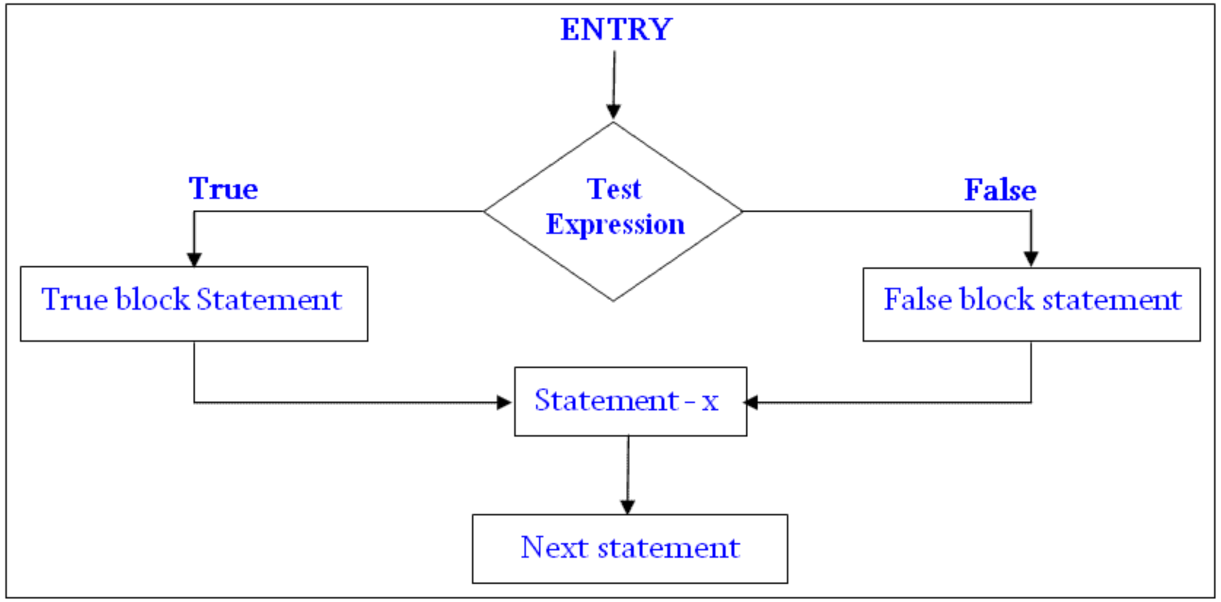
Conditional Structures

- **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.



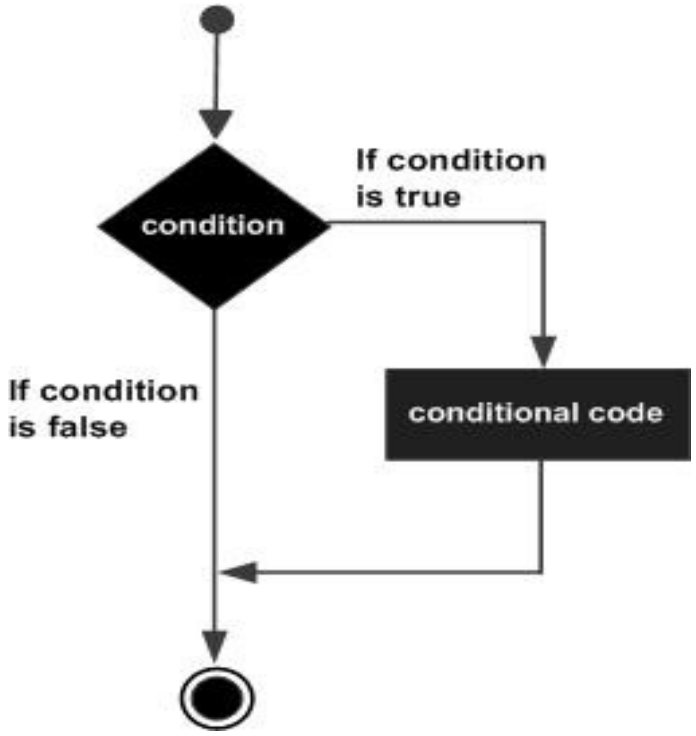
Block-Structured IF Statements

- 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

.
.
.
```



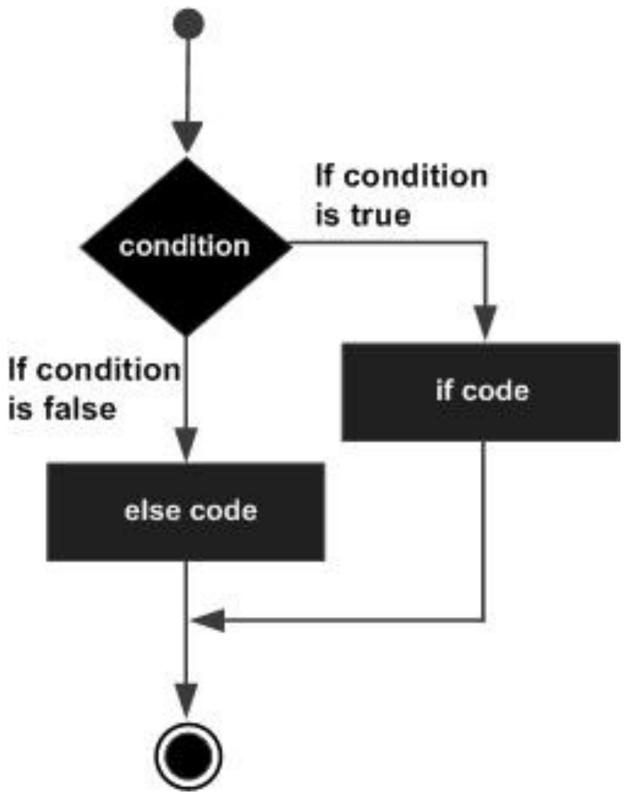
Block-Structured IF Statements

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
```

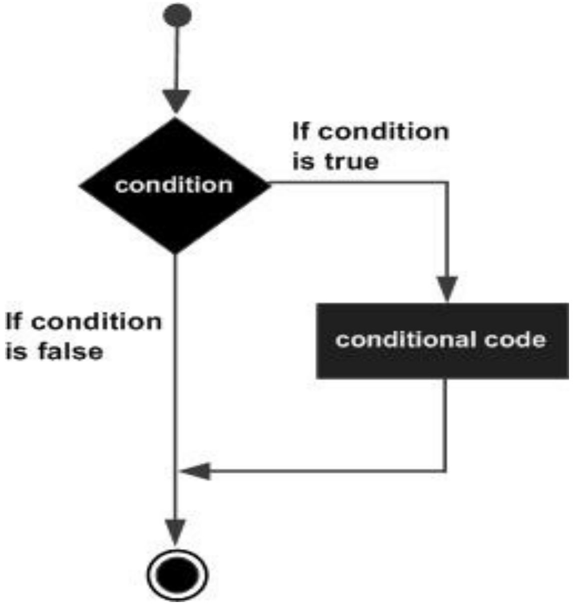
The else part is optional.



Block-Structured IF Statements

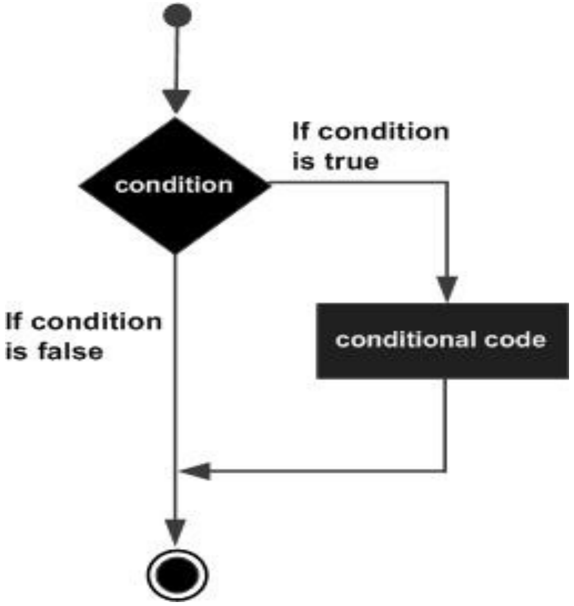
- IF statement is translated into assembly language with a
  - **CMP instruction** followed by
  - **Conditional jumps**.
- **When** **op1** or **op2** is a **memory operand** (a variable),
  - **One of them must be moved to a register before executing CMP.**

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

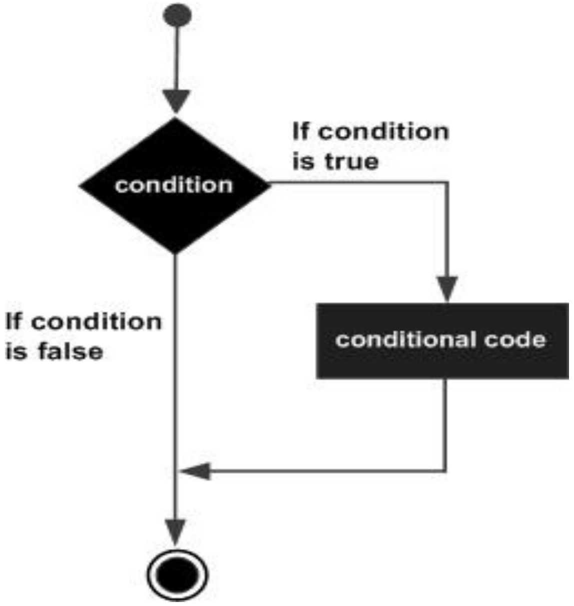
Jump (conditional) to ELSE if (AX>10) is FALSE

// if block code

Label:

    // ELSE block code

Block-Structured IF Statements




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

}
else{
        //else block code

}
```

CMP AX, 10

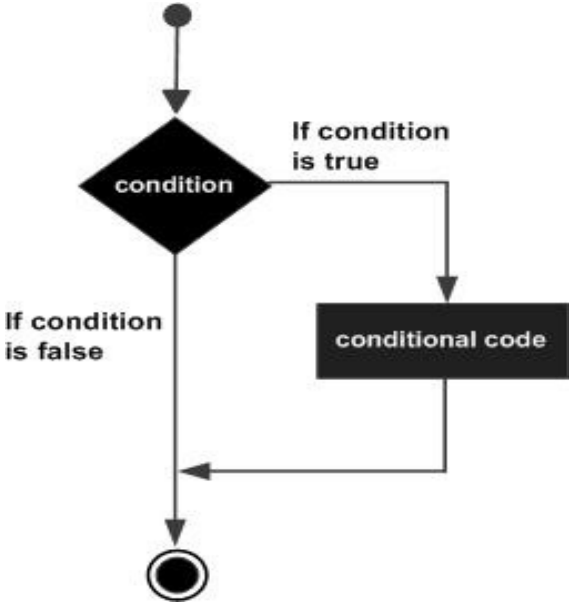
Jump (conditional) to ELSE if (AX>10) is FALSE

// if block code

Label:
    // ELSE block code

Problem! : Conditional Jumps trigger when CONDITION is TRUE!

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

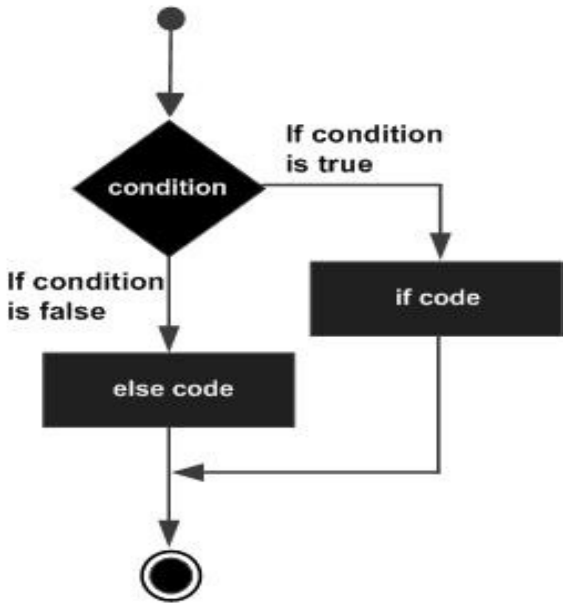
- When the condition is true, **execute** the body
- When the condition is false, don't execute the body, **jump** over it.
- This is the *jump* case

1

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

2

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

Condition	Negated Condition
x > y	x <= y
x >= y	x < y
x < y	x >= y
x <= y	x > y
=	!=
<cond1> && <cond1>	! <cond1> || ! <cond2>
<cond1> || <cond1>	! <cond1> && ! <cond2>

IF Statements: Example1 (IF-Then)

- 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**.

- 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 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:
```

A and B

- 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: Example2 (IF-Then-Else)

- 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**.

•Implement the following pseudocode in assembly language.

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

Reverse The IF Condition

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:

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

Do not Reverse the 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: Example3 (IF-Then-Else)

- 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**.

- 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;
}
```

```
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)

- 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**.

- 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.)

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;
```

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

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                 ; quit if false
    cmp bl,cl                ; second expression...
    jbe next                 ; quit if false
    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 <u>second</u> Condition

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:
```

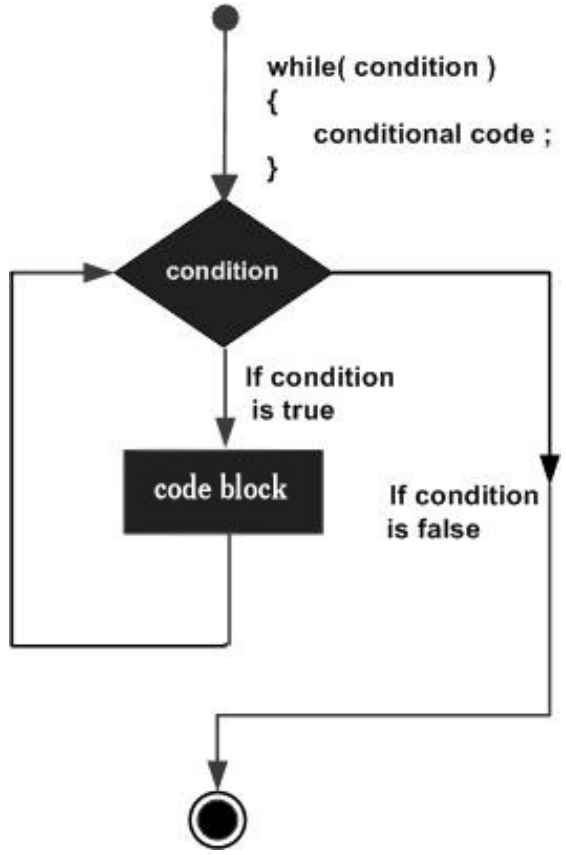
Conditional Structures

- 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**

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
```

WHILE Loops: Example2

```
while( eax < ebx)
    eax = eax + 1;
```

Reverse The loop Condition

A and B

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

This is a possible implementation

WHILE Loops: Example3

- 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:
```

Reverse The loop Condition

A and B

WHILE Loops: Example 4

- **IF** statement **Nested** in a **Loop**

Calculates the sum of all array elements greater than the value in sample (50)

```
int array[] = {10,60,20,33,72,89,45,65,72,18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;
while( index < ArraySize )
{
   if( array[index] > sample )
   {
      sum += array[index];
   }
   index++;
}
```

Lecture-20-Chapter-6-Conditional-Processing-with-annotations

WHILE Loops: Example 4

Calculate ... array ele... in sample

- **IF** statement **Nested** in a **Loop**

0 1 2 3 4 5 6 7 8 9

```
int array[] = {10,60,20,33,72,89,45,65,72,18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;
while( index < ArraySize )
{
   if( array[index] > sample )
   {
      sum += array[index];
   }
   index++;
}
```

windows10_fall21 [Running]

Source.asm* | Project4.lst

```
    mov ecx, LENGTHOF array
    mov esi, index ; esi contains index
    mov eax, sum ; eax contains sum

    mov edx, sample  ; edx contains sample

    begin_while:
        ; check while loop condition
        cmp esi, ecx
        JGE next     ; when index (esi) >= Arraysize (ecx), break the loop

        ; check if condition
        cmp array[esi*4], edx
        JG if_block    ; when array[esi] > sample

        jmp increase_index

        if_block:
            add eax, array[esi*4]

        increase_index:
            add esi, 1

    jmp begin_while


    Next:
```

esi	(esi*4)
0	0
1	(4)
2	(8)
3	(12)

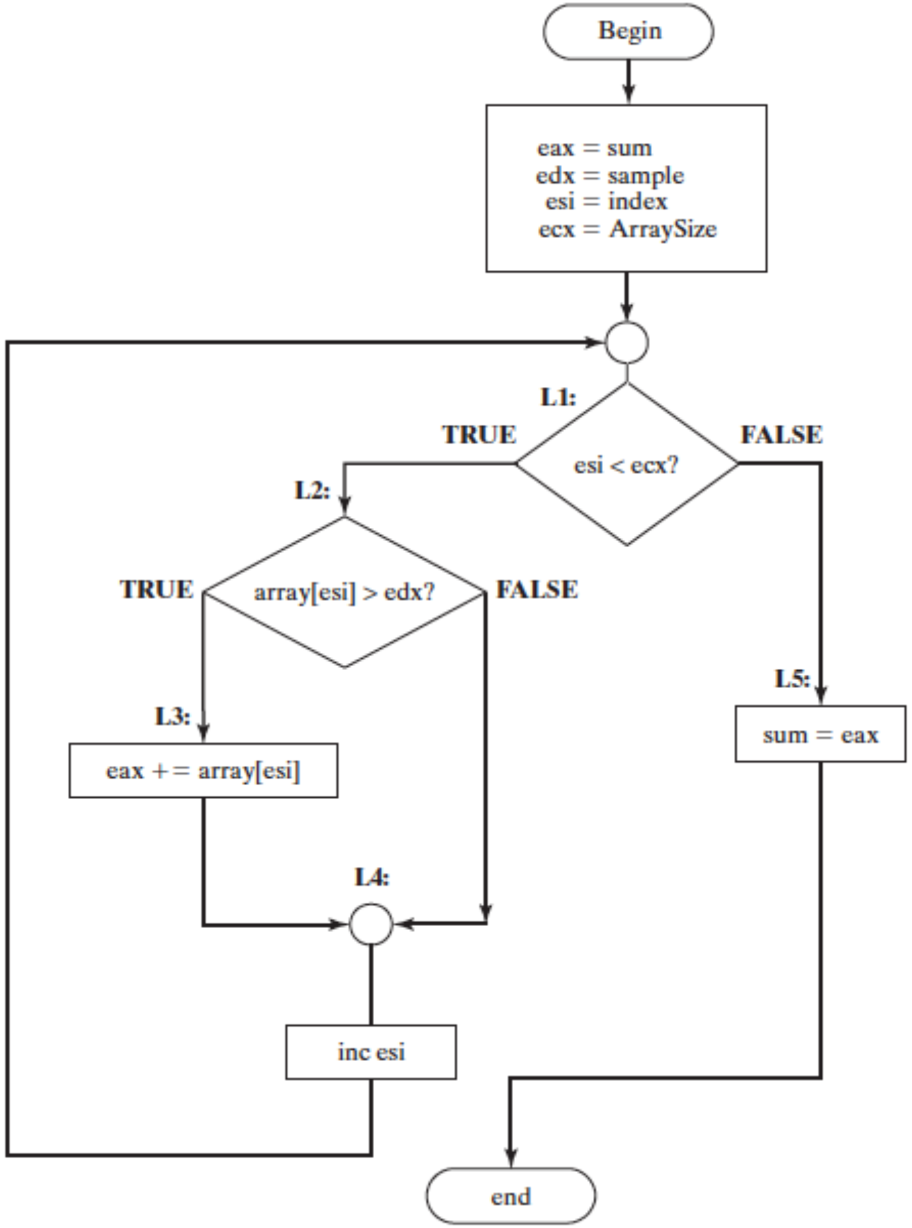
120 % | No issues found | Ln: 28 | Ch: 18 | Col: 24 | TABS | CRLF

Source.asm* | Project4.lst

```
ExitProcess Proto, dwExitCode:DWORD

.DATA
array DWORD 10, 60, 20, 33, 72, 89, 45, 65, 72, 18
sample DWORD 50
index DWORD 0
sum DWORD 0


.CODE
main PROC
    ; computer the size of the array
    mov ecx, LENGTHOF array
    mov esi, index ; esi contains index
    mov eax, sum ; eax contains sum

    mov edx, sample  ; edx contains sample

    begin_while:
        ; check while loop condition
        cmp esi, ecx
        JGE next    ; when index (esi) >= Arraysize (ecx), break the loop

        ; check if condition
        cmp array[esi*4], edx
        JG if_block    ; when array[esi] > sample
        jmp increase_index
        if_block:
            add eax, array[esi*4]
        increase_index:
            add esi, 1

    jmp begin_while

    Next:
```

WHILE Loops: Example 4

- **IF** statement **Nested** in a **Loop**



WHILE Loops: Example 4

- **IF** statement **Nested** in a **Loop**

```
int array[] = {10,60,20,33,72,89,45,65,72,18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;
while( index < ArraySize )
{
    if( array[index] > sample )
    {
       sum += array[index];
    }
    index++;
}
```

```
.data
sum DWORD 0
sample DWORD 50
array DWORD 10,60,20,33,72,89,45,65,72,18
ArraySize = ($ - Array) / TYPE array          40/4 = 10

.code
main PROC
     mov    eax,0                 ; sum
     mov    edx,sample
     mov    esi,0                 ; index
     mov    ecx,ArraySize

L1:  cmp    esi,ecx               ; if esi < ecx          Loop
     jl     L2
     jmp    L5

L2:  cmp    array[esi*4], edx     ; if array[esi] > edx   IF
     jg     L3
     jmp    L4
L3:  add    eax,array[esi*4]

L4:  inc    esi
     jmp    L1

L5:  mov    sum,eax
```

Attendance!

Outline

- **Boolean and Comparison Instructions**
- Conditional Jumps
- Conditional Structures
- Application: Finite-State Machines — **See the book**
- Conditional Control Flow Directives — **See the book**

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 (Just know them)
- **Conditional Structures**
  - IF statement
  - While loop