Lab 7 - Conditional Statements

A technique that lets you alter the flow of control is known as *conditional branching*. Every IF statement, switch statement, or conditional loop found in high-level languages has built-in branching logic. Assembly language also provides all the tools you need for decision-making logic.

7.1 Boolean Instructions

The Intel instruction set contains the AND, OR, XOR and NOT instructions, which directly implement Boolean operations on binary bits, shown in the table below.

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.

7.1.1 The CPU Status Flags

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

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an operation generates a carry out of the highest bit of the destination operand.
- The Sign flag is a copy of the high bit of the destination operand, indicating that it is negative if set and positive if clear. (Zero is assumed to be positive.)
- The Overflow flag is set when an instruction generates a result that is outside the signed range of the destination operand.
- The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

7.2 AND Instruction

The AND instruction performs a Boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand:

```
AND destination,source
```

The following operand combinations are permitted.

```
AND reg,reg         ;register vs register
AND reg,mem         ;register vs memory
AND reg,imm         ;register vs immediate operand
AND mem,reg         ;memory vs register
AND mem,imm         ;memory vs immediate operand
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. But the immediate operands can be no larger than 32 bits:

For each matching bit in the two operands, the following rule applies: If both bits equal 1, the result bit is 1; otherwise, it is 0.

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction lets you clear 1 or more bits in an operand without affecting other bits. The technique is called *bit masking*. Suppose, for example, that a control byte is about to be copied from the AL register to a hardware device. Further, we will assume that the device resets itself when bits 0 and 3 are cleared in the control byte. If we want to reset the device without modifying any other bits in AL, we can write the following:

```
and AL,11110110b                ;clear bits 0 and 3, leave others unchanged
```

For example, suppose AL is initially set to 10101110 binary. After ANDing it with 11110110, AL equals 10100110:

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

Flags: The AND instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, suppose the following instruction results in a value of Zero in the EAX register. In that case, the Zero flag will be set:

```
and eax,1Fh
```

7.2.1 Example to Convert Characters to Uppercase:

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

0 1 1 0 0 0 0 1 = 61h ('a')
0 1 0 0 0 0 0 1 = 41h ('A')

If we AND any character with 11011111 binary, all bits are unchanged except for bit 5, which is cleared. In the following example, all characters in an array are converted to uppercase:

```
Include Irvine32.inc
.data
      array BYTE "hello_world",0
.code
main PROC
      mov ecx,LENGTHOF array
      mov esi,OFFSET array
      L1:
            and BYTE PTR [esi],11011111b    ; clear bit 5
            inc esi
      loop L1
      mov EDX,OFFSET array
      call WriteString
```

```
exit
main ENDP
END main
```

7.3 OR Instruction

The OR instruction performs a Boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand:

```
OR destination,source
```

The OR instruction uses the same operand combinations as the AND instruction:

```
OR reg,reg          ;register vs register
OR reg,mem          ;register vs memory
OR reg,imm          ;register vs immediate operand
OR mem,reg          ;memory vs register
OR mem,imm          ;memory vs immediate operand
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1.

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

The OR instruction is particularly useful when you need to set 1 or more bits in an operand without affecting any other bits.

```
or AL,00000100b                 ;set bit 2, leave others unchanged
```

For example, if AL is initially equal to 11100011 binary and then we OR it with 00000100, the result equals 11100111:

```
mov al,11100011b
or al,00000100b                 ;result in AL = 11100111
```

Flags: The OR instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, you can OR a number with itself (or zero) to obtain certain information about its value:

```
or al,al
```

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

7.4 XOR Instruction

The XOR instruction performs a Boolean exclusive-OR operation between each pair of matching bits in two operands and stores the result in the destination operand:

```
XOR destination,source
```

The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions. For each matching bit in the two operands, the following applies: If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1.

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

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

7.4.1 Checking the Parity Flag

Parity checking is a function performed on a binary number that counts the number of 1 bits contained in the number; if the resulting count is even, we say that the data has even parity; if the count is odd, the data has odd parity.

In x86 processors, the Parity flag is set when the lowest byte of the destination operand of a bitwise or arithmetic operation has even parity. Conversely, when the operand has odd parity, the flag is cleared. An effective way to check the parity of a number without changing its value is to XOR 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)
```

Visual Studio uses PE = 1 to indicate even parity, and PE = 0 to indicate odd parity.

7.5 NOT Instruction

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

```
NOT reg
NOT mem
```

For example, the one's complement of F0h is 0Fh:

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

Flags: No flags are affected by the NOT instruction.

7.6 TEST Instruction

The TEST instruction performs an implied AND operation between each pair of matching bits in two operands and sets the Sign, Zero, and Parity flags based on the value assigned to the destination operand.

The only difference between TEST and AND is that TEST does not modify the destination operand. The TEST instruction permits the same operand combinations as the AND instruction. TEST is particularly valuable for finding out whether individual bits in an operand are set.

The TEST instruction can check several bits at once. Suppose we want to know whether bit 0 or bit 3 is set in the AL register. We can use the following instruction to find this out:

```
test al,00001001b               ;test bits 0 and 3
```

(The value 00001001 in this example is called a bit mask.) From the following example data sets, we can infer that the Zero flag is set only when all tested bits are clear:

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

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

Flags: The TEST instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in the same way as the AND instruction.

7.7 CMP Instruction

The most common Boolean expressions involve some type of comparison. The following pseudocode snippets support this idea:

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

In x86 assembly language we use the CMP instruction to compare integers. Character codes are also integers, so they work with CMP as well.

The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified:

```
CMP destination,source
```

CMP uses the same operand combinations as the AND instruction.

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. When two unsigned operands are compared, the 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

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 is a valuable tool for creating conditional logic structures. When you follow CMP with a conditional jump instruction, the result is the assembly language equivalent of an IF statement.

Examples:

When AX equals 5 and is compared to 10, the Carry flag is set because subtracting 10 from 5 requires a borrow:

```
mov ax,5
cmp ax,10                    ;ZF = 0 and CF = 1
```

Comparing 1000 to 1000 sets the Zero flag because subtracting the source from the destination produces zero:

```
mov ax,1000
mov cx,1000
cmp cx,ax                    ;ZF = 1 and CF = 0
```

Comparing 105 to 0 clears both the Zero and Carry flags because subtracting 0 from 105 generates a positive, nonzero value.

```
mov si,105
cmp si,0                     ;ZF = 0 and CF = 0
```

7.8 Setting and Clearing Individual CPU Flags

How can you easily set or clear the Zero, Sign, Carry, and Overflow flags? There are several ways, some of which require modifying the destination operand. To set the Zero flag, TEST or AND an operand with Zero; to clear the Zero flag, OR an operand with 1:

```
test al,0                    ;set Zero flag
and al,0                     ;set Zero flag
or al,1                      ;clear Zero flag
```

TEST does not modify the operand, whereas AND does. To set the Sign flag, OR the highest bit of an operand with 1. To clear the Sign flag, AND the highest bit with 0:

```
or al,80h                    ;set Sign flag
and al,7Fh                   ;clear Sign flag
```

To set the Carry flag, use the STC instruction; to clear the Carry flag, use CLC:

```
stc                          ;set Carry flag
clc                          ;clear Carry flag
```

To set the Overflow flag, add two positive values that produce a negative sum. To clear the Overflow flag, OR an operand with 0:

```
mov al,7Fh                    ;AL = +127
inc al                        ;AL = 80h (-128), OF=1
or eax,0                      ;clear Overflow flag
```

7.9 Conditional Jumps

Two steps are involved in executing a conditional statement:

- 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.

7.9.1 Examples

The CMP instruction in the following example 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:
```

The AND instruction in the following example 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:
```

7.9.2 J*cond* Instruction

A *conditional jump instruction* branches to a destination label when a status flag condition is true. Otherwise, if the flag condition is false, the instruction immediately following the conditional jump is executed. The syntax is as follows:

```
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 jump instructions evaluate the flag states, using them to determine whether or not jumps should be taken.

7.9.3 Using the CMP Instruction

Suppose you want to jump to label L1 when EAX equals 5. In the next example, if EAX equals 5, the CMP instruction sets the Zero flag; then, the JE instruction jumps to L1 because the Zero flag is set:

```
cmp eax,5
je L1                          ;jump if equal
```

The JE instruction always jumps based on the value of the Zero flag. If EAX were not equal to 5, CMP would clear the Zero flag, and the JE instruction would not jump.

In the following example, the JL instruction jumps to label L1 because AX is less than 6:

```
mov ax,5
cmp ax,6
jl L1                          ;jump if less
```

In the following example, the jump is taken because AX is greater than 4:

```
mov ax,5
cmp ax,4
jg L1                          ;jump if greater
```

7.9.4 Types of Conditional Jump Instructions

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

Below table shows a list of jumps based on the Zero, Carry, Overflow, Parity, and Sign flags.

Mnemonic	Description	Flags / Registers
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

7.9.5 Equality Comparisons

Below table lists the jump instructions based on evaluating equality. In the table, the notations *leftOp* and *rightOp* refer to the left (destination) and right (source) operands in a CMP instruction:

```
CMP leftOp,rightOp
```

Mnemonic	Description
JE	Jump if equal ($leftOp = rightOp$)
JNE	Jump if not equal ($leftOp \neq rightOp$)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX = 0 (64-bit mode)

Although the JE instruction is equivalent to JZ (jump if Zero) and JNE is equivalent to JNZ (jump if not Zero), it's best to select the mnemonic (JE or JZ) that best indicates your intention to either compare two operands or examine a specific status flag.

7.9.5.1 Examples

Following are code examples that use the JE, JNE, JCXZ, and JECXZ instructions. Examine the comments carefully to be sure that you understand why the conditional jumps were (or were not) taken.

Example 1:

```
mov edx,0A523h
cmp edx,0A523h
jne L5                          ;jump not taken
je L1                           ;jump is taken
```

Example 2:

```
mov bx,1234h
sub bx,1234h
jne L5                          ;jump not taken
je L1                           ;jump is taken
```

Example 3:

```
mov cx,0FFFFh
inc cx
jcxz L2                         ;jump is taken
```

Example 4:

```
xor ecx,ecx
jecxz L2                        ;jump is taken
```

7.9.6 Unsigned Comparisons

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if $leftOp \geq rightOp$)
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)

The operand names reflect the order of operands, as in the expression ($leftOp > rightOp$). The jumps in above table are only meaningful when comparing unsigned values. Signed operands use a different set of jumps.

7.9.7 Signed Comparisons

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightO$)
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)

The table displays a list of jumps based on signed comparisons. The following instruction sequence demonstrates the comparison of two signed values:

```
mov al,+127                    ;hexadecimal value is 7Fh
cmp al,-128                    ;hexadecimal value is 80h
ja IsAbove                     ;jump not taken, because 7Fh < 80h
jg IsGreater                   ;jump taken, because +127 > -128
```

The JA instruction, which is designed for unsigned comparisons, does not jump because unsigned 7Fh is smaller than unsigned 80h. The JG instruction, on the other hand, is designed for signed comparisons so it jumps because +127 is greater than -128.

7.9.7.1 Examples

In the following code examples, examine the comments to be sure you understand why the jumps were (or were not) taken:

Example 1:

```
mov edx,-1
cmp edx,0
jnl L5                         ;jump not taken (-1 >= 0 is false)
jnle L5                        ;jump not taken (-1 > 0 is false)
jl L1                          ;jump is taken (-1 < 0 is true)
```

Example 2:

```
mov bx,+32
cmp bx,-35
jng L5                         ;jump not taken (+32 <= -35 is false)
jnge L5                        ;jump not taken (+32 < -35 is false)
jge L1                         ;jump is taken (+32 >= -35 is true)
```

Example 3:

```
mov ecx,0
cmp ecx,0
jg L5                          ;jump not taken (0 > 0 is false)
jnl L1                         ;jump is taken (0 >= 0 is true)
```

Example 4:

```
mov ecx,0
cmp ecx,0
jl L5                                   ;jump not taken (0 < 0 is false)
jng L1                                  ;jump is taken (0 <= 0 is true)
```

Testing Status Bits:

Often, we do not want to change the values of the bits we're testing, but we do want to modify the values of CPU status flags. Conditional jump instructions often use these status flags to determine whether or not to transfer control to code labels. Suppose, for example, that an 8-bit memory operand named status contains status information about an external device attached to the computer. The following instructions jump to a label if bit 5 is set, indicating that the device is offline:

```
mov al,status
test al,00100000b                       ;test bit 5
jnz DeviceOffline
```

The following statements jump to a label if any of the bits 0, 1, or 4 are set:

```
mov al,status
test al,00010011b                       ;test bits 0,1,4
jnz InputDataByte
```

Jumping to a label if bits 2, 3, and 7 are all set requires both the AND and CMP instructions:

```
mov al,status
and al,10001100b                        ;mask bits 2,3,7
cmp al,10001100b                        ;all bits set?
je ResetMachine                         ;yes: jump to label
```

Larger of Two Integers:

The following code compares the unsigned integers in EAX and EBX and moves the larger of the two to EDX:

```
mov edx,eax                             ;assume EAX is larger
cmp eax,ebx                             ;if EAX is >= EBX
jae L1                                  ;jump to L1
mov edx,ebx                             ;else move EBX to EDX
L1:                                     ;EDX contains the larger integer
```

Smallest of Three Integers:

The following instructions compare the unsigned 16-bit values in the variables V1, V2, and V3 and move the smallest of the three to AX:

```
.data
      V1 WORD ?
      V2 WORD ?
      V3 WORD ?
.code
      mov ax,V1                                 ;assume V1 is smallest
      cmp ax,V2                                 ;if AX <= V2
      jbe L1                                    ;jump to L1
      mov ax,V2                                 ;else move V2 to AX
      L1:
```

```
        cmp ax,V3                       ;if AX <= V3
        jbe L2                          ;jump to L2
        mov ax,V3                       ;else move V3 to AX
        L2:
```

Loop until Key Pressed:

In the following 32-bit code, a loop runs continuously until the user presses a standard alphanumeric key. The *ReadKey* method from the Irvine32 library sets the Zero flag if no key is present in the input buffer:

```
.data
        char BYTE ?
.code
        L1:
        mov eax,10                      ;create 10 ms delay
        call Delay
        call ReadKey                    ;check for key
        jz L1                           ;repeat if no key
        mov char,AL                     ;save the character
```

The foregoing code inserts a 10-millisecond delay in the loop to give MS-Windows time to process event messages. If you omit the delay, keystrokes may be ignored.

Sequential Search of an Array:

A common programming task is to search for values in an array that meet some criteria. For 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:

```
Include Irvine32.inc
.data
        intArray SWORD  0,0,0,0,20,35,-12,66,4,0
        ;intArray SWORD  0,0,0,0
        noneMsg  BYTE "A non-zero value was not found",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,TYPE intArray         ; point to next index
        loop  L1                        ; continue the loop
        jmp   notFound                  ; none found
found:
        movsx eax,WORD PTR [ebx]        ; otherwise, display it
        call  WriteInt
        jmp   quit
notFound:
        mov   edx,OFFSET noneMsg        ; display "not found" message
        call  WriteString
quit:
        call  crlf
        exit
main ENDP
END main
```

Simple String Encryption

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

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

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

This simple program uses symmetric encryption, a process by which the same key is used for both encryption and decryption. The following steps occur in order at runtime:

- The user enters the plain text.
- The program uses a single-character key to encrypt the plain text, producing the cipher text, which is displayed on the screen.
- The program decrypts the cipher text, producing and displaying the original plain text.

```
Include Irvine32.inc
KEY = 239           ; any value between 1-255
BUFMAX = 128        ; maximum buffer size

.data
      sPrompt  BYTE  "Enter the plain text: ",0
      sEncrypt BYTE  "Cipher text:          ",0
      sDecrypt BYTE  "Decrypted:            ",0
      buffer   BYTE   BUFMAX+1 DUP(0)
      bufSize  DWORD  ?
.code
main PROC
      call   InputTheString            ; input the plain text
      call   TranslateBuffer           ; encrypt the buffer
      mov    edx,OFFSET sEncrypt       ; display encrypted message
      call   DisplayMessage
      call   TranslateBuffer           ; decrypt the buffer
      mov    edx,OFFSET sDecrypt       ; display decrypted message
      call   DisplayMessage

      exit
main ENDP

;-----------------------------------------------------
InputTheString PROC
;
; Prompts user for a plaintext string. Saves the string
; and its length.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
      pushad
      mov    edx,OFFSET sPrompt        ; display a prompt
      call   WriteString
      mov    ecx,BUFMAX                ; maximum character count
      mov    edx,OFFSET buffer         ; point to the buffer
```

```
	call	ReadString			; input the string
	mov	bufSize,eax			; save the length
	call	Crlf
	popad
	ret
InputTheString ENDP

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

;-----------------------------------------------------
TranslateBuffer PROC
;
; Translates the string by exclusive-ORing each
; byte with the encryption key byte.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
	pushad
	mov	ecx,bufSize			; loop counter
	mov	esi,0				; index 0 in buffer
L1:
	xor	buffer[esi],KEY			; translate a byte
	inc	esi				; point to next byte
	loop	L1

	popad
	ret
TranslateBuffer ENDP
END main
```