Lab 5

Code Labels

A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
 mov    ax,bx
 ...
 jmp    target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
 L1: mov    ax,bx
 L2:
```

Label names are created using the rules for identifiers. You can use the same code label more than once in a program if each label is unique within its enclosing procedure. (A procedure is like a function.)

```
Include Irvine32.inc

.data
msg1 byte "Hello",0
msg2 byte "World",0
msg3 byte "Exit",0
.code

main proc
            mov edx, offset msg1
            call writestring
            jmp M3
      M2:
            mov edx, offset msg2
            call writestring
            jmp M4
      M3:
            mov edx, offset msg3
            call writestring
            jmp M2
      M4:
      call readint
      exit
main endp
end main
```

Loop Instruction

Loop instruction uses ECX register as iterator. Loop instruction keeps on executing a specific label until the value of ECX register become zero. With each iteration, value of ECX get decremented automatically.

```
LOOP label
```

Example: Print "Hello" five times:

```
Include Irvine32.inc
.data
msg1 byte "Hello",0
.code
main proc
            mov ecx, 5
      M1:
            mov edx, offset msg1
            call writestring
            loop M1

      call readint
      exit
main endp
end main
```

Nested Loops

When dealing with nested loops, ECX Is used for both loops: inner and outer. Therefore, it is mandatory to store the status of outer loop before entering in inner and restore the status after exiting.

Example:

```
INCLUDE Irvine32.inc

.data

count Dword ?
prompt Byte 0dh,0ah,"Pakistan  ",0
prompt1 Byte 0dh,0ah,"ZindaBd  ",0

.code

main PROC
      mov ecx, 5
      Loop1:
            mov edx, OFFSET prompt
            call WriteString
            mov count, ecx
            mov ecx, 3
            Loop2:
                  mov edx, OFFSET prompt1
```

```
                call WriteString
            LOOP Loop2
            mov ecx, count
      LOOP Loop1

      call readInt
      exit
main ENDP

END main
```

PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you're trying to access the variable using a size attribute that's different from the one used to declare the variable. Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the operand sizes do not match:

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble ; error
```

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

```
mov ax,WORD PTR myDouble
```

Why wasn't 1234h moved into AX? x86 processors use the *little-endian* storage format in which the low-order byte is stored at the variable's starting address.

Example: Print an array of Five integers:

```
            Include Irvine32.inc

            .data
            arr byte 1,2,3,4,5

            .code

            main proc
                        mov ecx, lengthof arr
                        mov esi, offset arr
                  L1:
                        movzx eax,byte ptr [esi]
```

```
        call writeint
        inc  esi
        loop L1


      call readint
      exit
main endp
end main
```

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

The operands can be 8, 16, or 32 bits, and they must be the same size. 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 ∧ 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*, much as you might use masking tape when painting a house to cover areas (such as windows) that should not be painted. 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 1 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:

Example: 11111111b AND 11111100b

```
Include Irvine32.inc
.data
.code
main proc
      mov al, 11111111b
      call writebin           ; writing Bin
      call crlf               ; new line
      AND al, 11111100b
      call writebin           ; writing Bin
      exit
main endp
end main
```

Result:

```
0000 0000 0001 1001 1111 1111 1111 1111
0000 0000 0001 1001 1111 1111 1111 1100
```

Flags

The AND instruction always clears the Overflow and Carry flags.

Converting Characters to Upper Case

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')
```

The rest of the alphabetic characters have the same relationship.

```
Include Irvine32.inc

.data
char byte "a"

.code

main proc
            mov al, char
            and al, 11011111b
            call writechar
      call readint
```

```
      exit
main endp
end main
```

Output: A

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

The operands can be 8, 16, or 32 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 ∨ 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.

Example: 11001101 OR 11111100

```
Include Irvine32.inc
.data
.code
main proc
      mov al, 11001101b
```

```
      call writebin           ; writing Bin
      call crlf               ; new line
      OR al, 11111100b
      call writebin           ; writing Bin
      call readint
      exit
main endp
end main
```

Output

```
0000 0000 0001 1001 1111 1111 1100 1101
0000 0000 0001 1001 1111 1111 1111 1101
```

Flags

The OR instruction always clears the Carry and Overflow flags.

Converting Characters to lower Case

In the previous example of converting a character from lower case to upper. By ORing `00100000` with the character will result it to become lower case.

Sets

AND instruction can be used as intersection of sets. Where sets are binary numbers. Similarly, OR instruction can be used as Union of sets.

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. The following truth table describes the boolean expression

x ⊕ y:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

Flag

The XOR instruction always clears the Overflow and Carry flags.

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.

CMP Instruction

The CMP (compare) compares destination with source and sets some flags. Neither operand is modified:

```
CMP destination,source
```

Following registers are modified.

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

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

Conditional Structures

There are no explicit high-level logic structures in the x86 instruction set, but you can implement them

using a combination of comparisons and 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.

```
Jcond destination
```

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)

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

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)

Example:

```
Include Irvine32.inc
```

```
.data
char byte "c"
msg1 byte "Both are equal",0
msg2 byte "entered char is smaller",0
msg3 byte "entered char is bigger",0
.code

main proc
            call readchar
            cmp al, char
            je L1
            jl L2
            jg L3
      L1:
            mov edx, offset msg1
            call writestring
            jmp _exit
      L2:
            mov edx, offset msg2
            call writestring
            jmp _exit
      L3:
            mov edx, offset msg3
            call writestring
            jmp _exit

      _exit:
      call readint
      exit
main endp
end main
```

Additional conditional Loops

These loop statement act similar to ECX value, additionally they have following condition to meet in order to execute.

LOOPZ	**loop if Zero flag = 0**
LOOPE	**loop if Zero Flag = 1**
LOOPNZ	**loop if Zero flag = 1**
LOOPNE	**loop if Zero flag = 0**