CS221
Booleans, Comparison, Jump Instructions
Chapter 6

Assembly language is a great choice when it comes to working on individual bits of data. While some languages like C and C++ include bitwise operators, several high-level languages are missing these operations entirely (e.g. Visual Basic 6.0). At times these operations can be quite useful. First we will describe some common bit operations, and then discuss conditional jumps.

AND Instruction

The AND instruction performs a Boolean AND between all bits of the two operands. For example:

```
mov al, 00111011b
and al, 00001111b
```

The result is that AL contains 00001011. We have "multiplied" each corresponding bit. We have used AND for a common operation in this case, to clear out the high nibble. Sometimes the operand we are AND'ing something with is called a bit mask because wherever there is a zero the result is zero (masking out the original data), and wherever we have a one, we copy the original data through the mask.

For example, consider an ASCII character. "Low" ASCII ignores the high bit; we only need the low 7 bits and we might use the high bit for either special characters or perhaps as a parity bit. Given an arbitrary 8 bits for a character, we could apply a mask that removes the high bit and only retains the low bits:

```
and al, 01111111b
```

OR Instruction

The OR instruction performs a Boolean OR between all bits of the two operands. For example:

```
mov al, 00111011b
or  al, 00001111b
```

As a result AL contains 00111111. We have "Added" each corresponding bit, throwing away the carry.

A useful place for the OR instruction is to set specific bits in a status variable. For example, lets say you want 8 different Boolean variables. You could represent these 8 booleans using the bits in AL. Set a specific bit to 1 by using OR. Set that bit to 0 by using AND:

```
or al, 00001000b          ;  set bit 4 to 1
and al, 11110111b         ; set bit 4 to 0
```

XOR Instruction

The XOR instruction results in 0 if the two bits being compared are identical, and 1 if the two bits being compared are different.   For example:

```
mov al, 00111011b
xor al, 00001111b
```

Results in 00110100 in al.

A special quality of XOR is that it reverses itself when applied twice. If we apply another xor to the above sequence:

```
xor al, 00001111b
```

Then we get back 00111011 or the original value we moved into AL in the first place. This is sometimes used as a simple method to encode data, where the encryption key becomes the number that we XOR everything with.

Another place XOR is useful is to reverse a status bit:

```
xor al, 01000000          ;  Reverses bit 7, all others stay the same
```

Finally, XOR can be used in a trick to swap two numbers:

```
mov ax, 01010101b
mov bx, 11110000b

xor ax, bx              ; AX contains 10100101
xor bx, ax              ; BX contains 01010101
xor ax, bx              ; AX contains 11110000
```

Here we were able to swap two values without the need for a temporary variable! We get the same effect as:

```
mov temp, bx
mov bx, ax
mov ax, temp
```

NOT instruction

The NOT instruction reverses all bits in an operand, changing ones to zeros and vice versa. The result is the one's complement. For example:

```
mov al, 11110000b
not al
```

AL now contains 00001111b.

NEG instruction

The NEG instruction negates a number by converting it to its twos complement. For example:

```
mov al, -1          ; contains 11111111
neg al              ; contains 00000001
```

Since there is one more negative value possible than positive values, there is a chance we will have an overflow after performing a NEG operation. If this happens, the OF flag will be set to 1.

TEST instruction

The TEST instruction performs an implied AND between two operands. Neither operand is modified, but if *any* of the bits between the operands are identical, then the zero flag is set to 0. Otherwise the zero flag is set to 1.

A common use of this instruction is to test if at least one of some specific bits are set:

```
mov ah,  00001111b
test ah, 00010001b
```

ZF is set to 0 since bit 0 matches.

However:

```
mov ah,  00001110b
test ah, 00010001b
```

The zero flag here is set to 1 since none of the bits match.

CMP Instruction

The CMP (Compare) instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified. The result is reflected in the FLAGS registers. The first operand is the source, and the second is the destination.

The flags are set according to the following table for unsigned operands:

Dest < Src	Carry Flag = 1	Zero Flag = 0
Dest = Src	Carry Flag = 0	Zero Flag = 1
Dest > Src	Carry Flag = 0	Zero Flag = 0

For signed operands we have the following table:

Dest < Src	Carry Flag = ?	Zero Flag <> Overflow Flag
Dest = Src	Carry Flag = 1	Zero Flag = ?
Dest > Src	Carry Flag = 0	Zero Flag = Overflow Flag

You don't have to remember all these possible assignments to the different flags. The CMP instruction is called compare because when used preceding one of the conditional JUMP instructions, the jump will compare the proper flags and make the desired jump.

There are other Boolean operators, but we have covered the basic ones!

Conditional Jumps

You have already used some conditional jumps on your homework assignments. We will look at a few more here. As you have seen, there are no high-level logic structures like WHILE, FOR, CASE, or IF-THEN-ELSE statements in assembly. Instead we have to create these structures ourselves with jumps, which basically amounts to GOTO statements. (There are MACROS that we can use that model these conditional structures, but they are turned into Jump instructions).

A conditional jump transfers control to a destination address when some kind of flag condition becomes true. The syntax is:

Jcond destination

There are many variants of the conditional jump. Let's start with the General Comparisons. These are jumps that are made based on general comparisons of various flags. They have nothing to do with signed or unsigned numbers, but can generally be used on unsigned numbers:

General Comparisons:

Mnemonic		Taken if
JZ	Jump if Zero	ZF = 1
JE	Jump if Equal	ZF = 1
JNZ	Jump if not zero	ZF = 0
JNE	Jump if not equal	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JCXZ	Jump if CX=0	CX = 0

Since CMP will set ZF to 1 if two unsigned operands are the same, then if JZ or JE is followed by a CMP, we will take the branch if the operands are equal. Similarly, we will take the branch if two operands of a compare are different but followed by JNZ:

```
mov ax, 10
cmp ax, 10
je TakeBranch

mov ax, 10
cmp ax, 20
jne AlsoTakeBranch
```

We also have jumps based specifically on comparing unsigned values:

Mnemonic		Taken if
JA	Jump if Above (o1 > o2)	ZF = 0 and CF = 0
JAE	Jump if above or equal (o1 >= o2)	CF = 0
JB	Jump if below (o1 < o2)	CF = 1
JBE	Jump if below or equal (o1 <= o2)	CF =1 or ZF = 1

These jumps should be made directly after a CMP instruction. If we perform any other intermediate instructions, they might change the value of the flags.

Finally, we also have jumps based specifically on comparing signed values:

Mnemonic		Taken if
JG	Jump if Greater (o1 > o2)	SF=OF and ZF=0
JGE	Jump if Greater or Eq (o1 >= o2)	SF=OF
JL	Jump if Less (o1 < o2)	SF <> OF
JLE	Jump if less or eq (o1 <= o2)	ZF =1 or OF <> SF

There are other jumps for the NOT conditions; see the book if you would like to use them.

Here are some examples of using various JUMPs:

Larger of two numbers in BX and AX gets copied into DX:

```
        mov dx, ax              ; Assume AX is bigger
        cmp ax, bx              ; Compare
        jae L1                  ; Jump to L1 if AX >= BX
        mov dx, bx              ; if it's not copy BX to DX
L1:     …
```

Smallest of three numbers stored in AL, BL, CL copied into "small":

```
        mov small, AL           ; Assume AL is smallest
        cmp small, BL
        jbe L1                  ; Jump to L1 if AL < BL
        mov small, BL           ; BL smaller, so move into small
L1:     cmp small, CL           ; Compare smallest so far with CL
        jbe L2
        mov small, cl           ; CL smaller so move into small
L2:     …
```

Repeatedly reading from the keyboard, remembering the minimum entered so far, until the user types the sentinel value of "-9999":

```
        mov edx, -65535         ; current minimum
L1:     call readint
        call crlf
        cmp eax, -9999
        je ExitLoop
        cmp eax, edx
        jg L1
        mov edx, eax            ; Set new min
        jmp L1
ExitLoop:
        mov eax, edx
        call Writeint
```

Here is a program that finds the MAX out of an array of values. It has a subtle bug. Can you find it?

```
.data
array word 40, -10, 45, 910, 100, -45, 3
count word 7
```

```
.code
main proc
        movzx ecx, count                ; Loop through all except first
        dec ecx
        mov dx, array                   ; current max = 40
        mov ebx, offset array
        add ebx, 2
L1:     mov ax, [ebx]
        add ebx, 2
        cmp ax, dx
        jl L1                           ; If AX < DX jump to L1 to compare next number
        mov dx, ax                      ; new max
        loop L1

        move ax, 0
        mov ax, dx
        call Writeint

        exit
main endp
```

Here is the relevant portion with the bug fixed:

```
L1:     mov ax, [ebx]
        add ebx, 2
        cmp ax, dx
        jl SkipNewMax           ; If AX < DX jump to LOOP to compare next number
        mov dx, ax              ; new max
SkipNewMax:
        loop L1
```

We were skipping the LOOP instruction, so ECX wasn't being decremented properly.

There are several other useful sample programs you should look at in the textbook!

High Level Logic Structures

It may be useful to point out how assembly instructions can be used as a template to corresponding high-level logic structures. Here we will only look at the IF, Compound IF, and WHILE loop.

IF Statement

The template for the if statement is:

```
If (operand1 = operand2) then
        Statement;
        Statement;
End if
```

Here is an assembly template that corresponds to the same thing:

```
        cmp operand1, operand2
        jne FalseBranch
        <statement>
        <statement>
FalseBranch:
```

Compound-IF Statement using OR

The compound OR statement template looks like:

```
If ( X > op1) or (X >= op2) or ( X = op3) or (X< op4) then
        Statement;
        Statement;
End if
```

Here is an assembly template for the same thing:

```
        cmp X, op1                  ; If any condition true, jump to statements
        jg L1
        cmp al, op2
        jge L1
        cmp al, op3
        je L1
        cmp al, op4
        jl L1
        jmp L2
L1:     <statement>
        <statement>
L2:     …                  ; end if
```

Compound-IF using AND

The compound AND statement template looks like:

```
If (X > op1) and (X>=op2) and (X=op3) and (X<op4) then
        Statement;
        Statement;
End If
```

There are two approaches to the compound-AND. The first approach is to make a label that we jump to for each AND statement:

```
        cmp x, op1
        jg L1
        jmp EndStatement
L1:     cmp x, op2
        jge L2
        jmp EndStatement
L2:     cmp x, op3
        je L3
        jmp EndStatement
L3:     cmp x, op4
        jl L4
        jmp EndStatement
L4:     <statement>
        <statement>
EndStatement:
```

A more compact approach is to reverse the conditions and jump to an exit label when any condition is true. The condition X>op1 becomes NOT (X>op1) or (X<=op1).

```
        cmp x, op1
        jle L1
        cmp x, op2
        jl L1
        cmp x, op3
        jne L1
        cmp x, op4
        jge L1
        <statement>
        <statement>
L1:                             ; End If
```

While Structure

The WHILE structure tests a condition first before performing a block of statements. As long as the while condition remains true, the statements are repeated. For example:

```
While (op1 < op2)
        Statement;
        Statement;
End While
```

To translate this into assembly we can reverse the condition and jump to the label EndWhile when the condition becomes true:

```
WhileLabel:
        cmp op1, op2
        jge EndWhile                ; Condition false, end loop!
        <statement>
        <statement>
        jmp Whilelabel
EndWhile:
```

These loops are enough to also construct repeat-while or for loops. As you may recall from CS101 or CS201, we really only need one type of loop since all of these loops can be converted into oen another.