Module 3: Basic Instruction Set, Addressing, Program Logic and Control

Lesson 1 Data transfer, Basic Arithmetic, INT Instruction and Addressing Modes

Learning Outcomes

- LO 3.1.1 Explain the operations of data transfer instruction, and implement it in moving various data in the program.
- LO 3.1.2 Explain the uses of the basic arithmetic operation and utilize these instructions in changing register/memory contents.
- LO 3.1.3 Introduce what are interrupts
- LO 3.1.4 Explain each addressing mode; and implement data transfer of each addressing mode.

Data Transfer Instruction

This instruction moves data and addresses between registers or between a register and a memory location or I/O port. Instructions in this group can be further divided to subgroups namely: general purpose, input/output, address transfer and flag transfer.

The MOV Instruction

MOV transfers (or copies) data referenced by the address of the second operand to the address of the first operand (MOV *destination, source*). The sending field is unchanged. The format is:

[Label:]	MOV	Register/memory, register/memory/immediate

Rules:

The operands that reference memory or registers must agree in size (both must be bytes, both words, and both doublewords). Example:

```
WORD_VAL DW ?
BYTE_VAL DB ?
MOV AH, WORD_VAL  ; Invalid
MOV AX, BYTE_VAL  ; Invalid
```

You cannot move data between two memory locations directly. Instead, you must move the source data into a general-purpose register first, then move that register into the destination.

```
MOV DATA1, DATA2  ; Invalid MOV operation
MOV AX, DATA1           ; Valid MOV
MOV DATA2, AX           ; operations
```

You cannot load an immediate into a segment register directly. Use first the general-purpose register. Example:

```
MOV DS, 5   ; Invalid MOV operation
MOV AX, 5         ; Valid MOV
MOV DS, AX  ; Operation
```

You cannot move the contents of one segment register into another directly. Use the general-purpose register first. Example

```
MOV ES, DS; Invalid MOV operation
MOV AX, DS; Valid MOV
MOV ES, AX; Operation
```

The XCHG Instruction

XCHG performs another type of data transfer, but rather than simply copy the data from one location to another, XCHG swaps the two data items. The format for XCHG is

[Label:]	XCHG	Register/memory, register/memory

Valid XCHG operations involve exchanging data between two registers and between a register and memory. Examples:

```
WORDQ DW ?  ;Word data item
…
XCHG CL, BH
XCHG CX, WORDQ
```

The LEA Instruction

LEA is useful for initializing a register with an offset address. The format for LEA is

[Label:]	LEA	Register, memory

A common use for LEA is to initialize an offset in BX, DI, SI for indexing an address in memory.

```
        DATATBL DB 25 DUP (?)    ; Table of 25 bytes
        BYTEFLD DB ?                    ; One Byte
….
        LEA BX, DATATBL          ; Load offset address
        MOV BYTEFLD, [BX]        ; Move first byte of DATATBL
```

And equivalent operation to LEA is MOV with the offset operator, which generates slightly shorter machine code equivalent.

```
        MOV BX, OFFSET DATATBL  ; Load offset address
```

PUSH and POP

PUSH puts the contents of a word-sized (16-bit) register or memory operand on the top of the stack, it decrements SP by 2 or ESP by 4 and transfer a doubleword from the specified operand to the new top of the stack. POP takes a word off from the stack and puts it in memory or register, it increments SP 2, the 32-bit operand denotes a doubleword value, and ESP is incremented by 4.

[Label:]	PUSH	Source
[Label:]	POP	Destination

```
PUSH AX      ; Push data on top of the stack
POP DX       ; transfer the top of the stack to DX
```

This instruction is also convenient for copying one segment register into another. Nevertheless, its advantage is that it does not require you to use general-purpose register for immediate storage.

The INT Instruction

The INT instruction enables a program to interrupt its own processing, for example, to initialize the mouse driver.

```
MOV AX, 00H ; Initialize
INT 33H           ; mouse driver
```

INT exits normal processing and accesses the Interrupt Vector Table in low memory to determine the address of the requested routine. The operation then transfers to BIOS or the operating system for specified action and returns to the program to resume

processing. Most often, the interrupt has to perform the complex steps of an input or output operation. An interrupt requires a trail that facilitates exiting a program and, on successful completion, returning to it. For this purpose, INT performs the following:

- Pushes the contents of the Flag register onto the stack
- Clears the interrupt and trap flags
- Pushes the CS register onto the stack
- Pushes Instruction Pointer (containing the address of the next instruction) onto the stack
- Performs the required operation

The return from the interrupt, the operation issues an IRET (Interrupt return), which pops the register off the stack. The restored CS:IP causes a return to the instruction immediately following INT.

Because the preceding process is entirely automatic, your concerns are to define a large stack enough for the necessary pushing and popping and to use the appropriate INT operations.

Addressing Modes

An operand address provides a source of data for an instruction to process. Some instruction, such as CLC and RET, do not require an operand whereas other instruction may have one, two, or three operands. Where there are two operands, the first operand is the destination, which contains data in a register or in memory, and which is to be processed. The second operand is the source, which contains either the data to be delivered (immediate) or the address (in memory or of a register) of the data. The source data for most instructions is unchanged by the operation. Three basic modes of addressing are register, immediate, and memory; memory addressing consists of six types, for eight modes in all.

Register Addressing – for this mode; a register provides the name of any 8-, 16-, or 32 bit registers. Depending on the instruction, the register may appear in the first operand, the second operand, or both, as the following examples

```
mov DX, word_mem        ; Register in first operand
mov word_mem, DX        ; Register in the second operand
mov EBX, EDX            ; Register in both operands
```

Because the processing of data between registers involves no reference to memory, it is the fastest type of operation.

Immediate Addressing – contains value or an expression. Here are some examples of valid constants

a. Hexadecimal: 0148H
b. Decimal: 328 (converts the assembler converts to 148H)
c. Binary: 101001000B (which converts to 148H)

For many instructions with two operands, the first operand may be a register or memory location, and the second may be an immediate constant. The destination fields (first operand) define the length of the data.

```
BYTE_VAL    DB 150
WORD_VAL    DW 300
DBWD_VAL DD 0
```

```
SUB BYTE_VAL, 50        ; Immediate to memory (byte)
MOV WORD_VAL, 40H ; Immediate to memory (word)
MOV DBWD_VAL, 0   ; Immediate to memory (doubleword)
MOV AX, 0245H           ; Immediate to register (word)
```

The use of an immediate operand provides faster processing than defining a numeric constant in the data segment and referencing it in an operand.

The length of an immediate constant cannot exceed the length defined by the first operand.

Ex. `MOV AL, 0245H    ; Invalid immediate length`

However, if an immediate operand is shorter than a receiving operand, as in

Ex. `ADD AX, 48H      ; Valid immediate length`

The assembler expands the immediate operand to two bytes, 0048H, and store it in object code as 4800H

Direct memory addressing – In this format, one of the operands references a memory location and the other references a register. DS is the default segment register for addressing data in memory, as DS:offset.

```
ADD BYTE_VAL, DL  ; Add register to memory (byte)
MOV BX, WORD_VAL  ; Move memory to register (word)
```

Direct-Offset addressing – This addressing mode, a variation of direct addressing, uses arithmetic operators to modify an address.

```
BYTE_TBL DB  12, 15, 16, 22   ; Table of bytes
WORD_TBL DW 163, 227, 485,    ; Table of bytes
DBWD_TBL DD 456, 562, 897,    ; Table of bytes
```

Byte Operations. These instruction access byte from BYTE_TBL:

```
MOV CL, BYTE_TBL [2]    ; Get byte from BYTE_TBL
MOV CL, BYTE_TBL+2      ; Same operation
```

The first MOV uses an arithmetic operator to access the third byte (16) from BYTE_TBL. (BYTE_TBL [0] is the first byte, BYTE_TBL [1] the second, and BYTE_TBL[2] the third.) The second MOV uses a plus (+) operator for exactly the same effect.

Word Operations. These instructions access the words from WORD_TBL:

```
MOV CX, WORD_TBL [4]
MOV CX, WORD_TBL+4
```

The MOVs access the third word of WORD_TBL. (WORD_TBL[0] is the first word, WORD_TBL[2] the second, and WORD_TBL[4] the third)

Doubleword Operations. These instructions access doublewords from DBWD_TBL:

```
MOV CX, DBWD_TBL [8]    ; Get doubleword from DBWD_TBL
MOV CX, DBWD_TBL +8           ; Same operation
```

The MOVs access the third double word of DBWD_TBL. (DBWD_TBL[0] is the first double word, DBWD_TBL[4] the second, and DBWD_TBL[8] the third)

Indirect Memory Addressing - Indirect addressing takes advantage of the computer's capability for segment:offset addressing. The registers used for this purpose are base registers (BX or BP) and index registers (DI and SI), coded within this square brackets, which indicate a reference memory. If you code the .386, .486, or .586 directive, you can also use any of the general purpose registers(EAX, EBX, ECX, and EDX).

An indirect address such as [DI] tells the assembler that the memory address to use will be DI when the program subsequently executes. BX, DI, and SI are associated with DS and DS:BX, DS:DI, DS:SI for processing data in the data segment. BP is associated with SS as SS:BP, for handling data in the stack.

When the operand contains an indirect address, the second operand references a register or immediate value; when the second operand contains an indirect address, the first operand references a register. Note that a reference is square brackets to BP, BX, DI, or SI implies indirect operand, and the processor treats contents of the register as an offset address when the program is executing.

```
DATA_VAL DB 50                ;Define byte
          ….
      LEA BX, DATA_VAL  ; Load BX with offset
      MOV [BX], CL            ; Move CL to DATA_VAL
```

LEA first initializes BX with the offset address DATA_VAL. MOV then uses the address now in BX to store CL in the memory location to which points, in this case, DATA_VAL.

Base Displacement Addressing – This addressing mode also uses base registers (BX or BP) and index registers (DI and SI) but combined with a displacement (a number of offset value) to form an effective address.

```
DATA_TBL 365 DUP(?)
          ….
      LEA BX, DATA_TBL        ; Load BX with offset
      MOV BYTE PTR [BX+2], 0  ; Move 0 to DATA_TBL+2
```

The following MOV instruction moves zero to a location two bytes immediately following the start of DATA_TBL

Base-Index Addressing – This addressing modes combines a base register (BX or BP) with an index register (DI or SI) to form an effective address; for example, [BX+DI] the address BX plus the address in DI. A common use for this mode is in addressing a 2 dimensional array, where, say BX references the row and SI the column.

```
MOV AX, [BX+SI]
ADD [BX+DI], CL
```

Base-Index With Displacement Addressing – This addressing mode, a variation on base-index, combines a base register, an index register and a displacement to form an effective address

```
MOV AX, [BX+SI+10]            ; or 10[BX+DI]
MOV CL, DATA_TBL[BX+DI] ; or [BX+DI+DATA_TBL]
```

Lesson 2 Branching Instructions

Learning Outcomes

- LO 3.2.1 Explain the machine execution that would happened if an unconditional branch would take place.
- LO 3.2.2 Explain the needed parameters on how a loop or conditional branch would take place, and implement it as control flow statements in assembly code.

Short, Near, and Far Addresses

The assembler supports three types of addresses that are distinguished by their distance from the current address:
A short address, limited to a distance of -128 (80H) to 127 (7FH) bytes
A near address, limited to a distance of -32,768 (800H) to 32, 767 (7FFFH) bytes within the same segment.
A far address, which may be within the same segment at a distance over 32K, or in another segment.
A jump operation reaches a short address by a 1-byte offset and reaches a near address by a one-or-two- word offset. A far address is reached by a segment address and an offset; CALL is the normal instruction for this purpose because it facilitates linking to the requested address and the subsequent return.

The table below lists the rules on distances from JMP, LOOP, and CALL operation.

	Short	Near	Far
Instruction	-128 to 127 Same Segment	-32,768 to 32,767 Same segment	Over 32K or in Another segment
JMP	Yes	Yes	Yes
Jnnn	Yes	Yes (80386+)	No
LOOP	Yes	No	No
CALL	N/A	Yes	Yes

The JMP Instruction

A commonly-used instruction for transferring control is the JMP (Jump) instruction. A jump is unconditional because the operation transfer control under all circumstances. JMP also flushes the processor's prefetch instruction queue so that a program with many jump operations may lose some processing speed. The format for JMP is

[Label:]	JMP	Short/near/far address

Short and Near Jumps

A JMP operation within the same segment may be short or near (or even far if the destination is procedure with the FAR attribute). On its first pass through a source program, the assembler generates the length of each instruction. However, a JMP instruction may be two, three, or four bytes long. A JMP operation within -128 (80H) to +127 (7FH) bytes is a short jump. The assembler generates on byte for the operation (EB) and one byte for the operand. The operand acts as an offset value that the processor adds to the IP register when executing the program.

A JMP that exceeds -128 to +127 becomes a near jump (within 32K), for which the assembler generates different machine code (E9) and a 2 byte operand (8086/80286) or 4-byte operand (80386+).

Backward and forward Jumps

A jump may be backward or forward. The assembler may have already encountered the designated operand (a backward jump) within -128, as in

```
L10:        ; Jump address
…
JMP L10     ; Backward jump
```

In this case, the assembler generates a 2 byte machine instruction. In a forward jump, the assembler has not yet encountered the designated operand:

```
JMP L20     ; Forward jump
…
L20:        ; Jump address
```

Because the assembler doesn't know at this point whether the forward jump is short or near, some versions assume near and generated a 3 byte instruction. However, provided that the jump really is short, you can use the short operator to force a short jump and a 2 byte instruction.

The Loop Instruction

A standard practice to code a routine that loops a specified number of times or until it reaches a particular condition. The LOOP instruction, which serves this purpose, requires an initial value in CX. For each iteration, LOOP automatically deducts 1 from CX. Once CX reaches zero, control drops through to the following instruction; if CX is nonzero, control jumps to the operand address. The distance to the operand must be a short jump, within -128 to +127 bytes. For an operation that exceeds this limit, the assembler issues a message such as "relative jump out of range". The format for loop is:

[Label:]	LOOP	Short- address

The there are two variations on the LOOP instruction, both of which also decrement CX by 1. LOOPE/LOOPZ (loop while equal/zero) continues looping as long as CX is zero or the zero condition is set. LOOPNE/NZ (loop while not equal/zero) continues looping as long as CX is not zero condition is not set. Neither LOOP nor its LOOPxx variants changes the setting of any flags n the Flags register

The CMP Instruction

CMP is used to compare two numeric data fields, one or both of which are contained in a register. Its format is

[Label:]	CMP	Register/memory, register/memory/immediate

Technically, you can CMP to compare string (character) data, but CMPS is the appropriate instruction for this purpose. The result of CMP operation affects the AF, CF, OF, PF, SF, and ZF flags, although you do not have to test these flags individually. The following code test DX for a zero value.

```
CMP DX, 00  ; DX=Zero?
JE L10          ; If yes, jump to L10
… (action if non-zero)
L10…        ; Jump point if DX = zero
```

If DX contains zero, CMP sets the ZF to 1, and may not change the settings of other flags. The JE (Jump if Equal) instruction set tests only the ZF. Because ZF contains 1 (meaning a zero condition), JE transfers control (jumps) to the address indicated by operand L10.

In effect, a CMP operation compares the first to the second operand; for example, is the value of the first operand higher than, equal to, or lower that the value of the second

operand? (CMP acts like SUB without the additional storage cycle required for execution.) Conditional jump provides the various ways of transferring control based on tested conditions.

Conditional Jump Instruction (Jnnn Instruction)

The processor supports a variety of conditional jump instructions that transfer control depending on setting in the Flag register. For example, you can compare or add two fields and then jump conditionally according to flag values that the compare sets. The format for conditional jump is

[Label:]	Jnnn	Short-address

```
DEC CX              ; Equivalent to LOOP
JNZ A20     ;
…
```

As explained on section 4.6, the LOOP instruction decrements CX; if it is nonzero, control transfers to the operand address. The code above is an equivalent LOOP statement, using JNZ. JNZ tests the setting if the Zero flag, if CX us nonzero, control jumps to A20, and if CX is zero control drops through the next instruction. (A jump operation that branches also flushes the processor's prefetech instruction queue.) Although LOOP has limited uses, it executes faster and uses fewer bytes than does the use of the DEC and JNZ instructions.

Just as for JMP and LOOP, the machine code operand for JNZ contains the distance from the end of the instruction to the address of A20, which the operation adds the IP register. For 8086/80286, the distance for a conditional jump must be a short, within -128 to +127 bytes. If the operation exceeds this limit, the assembler issues a message "relative jump out of range." The 80386+ processor provide for 32-bit (near) offset that allow reaching any address within 32K.

Signed and Unsigned Data: Distinguishing the purpose of conditional jumps should clarify their use. The type of data (unsigned or signed) on which you are performing comparisons or arithmetic can determine which instruction to use. An unsigned numeric item (logical data) treats all bits and data bits; typical examples are numeric values such as customer numbers, phone numbers, and many rates and factors. A signed numeric item (arithmetic data) treats the leftmost bit as a sign where 0 means a positive and 1 means negative; typical examples are quantity, bank balance, temperature, which may be either positive or negative.

Jumps Based on Unsigned (Logical) Data

The following conditional jumps used for unsigned data:

Symbol	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump Not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/ Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF

JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

You can express each of these conditional jumps in one of the two symbolic operations; choose the one that is clearer and more descriptive.

Jumps Based on Signed (Logical) Data

The conditional jumps below are used for signed data:

Symbol	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump Not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/ Equal or Jump Not Below	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

The jumps for testing equal/zero (JE/JZ) and not equal/zero (JNE/JNZ) are included in the list for both unsigned and signed data because the condition exists regardless of the presence or absence of a sign.

Special Arithmetic Test

The following conditional jump instructions have special uses:

Symbol	Description	Flags tested
JCXZ	Jump if CX is zero	None
JC	Jump Carry	CF
JNC	Jump No Carry	CF
JO	Jump Overflow	OF
JNO	Jump No Overflow	OF
JP/JPE	Jump Parity/Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign	SF
JNS	Jump No Sign	SF

JCX test the contents of CX for zero. This instruction need not be placed immediately following an arithmetic or compare instruction. One use for JCXZ could be at the start of a loop to ensure that the routine is bypassed if CX is initially zero. JC and JNC are often used to test the success or failure of disk operations.

Now, don't expect to memorize the names of all these instructions or the flags they test. As a reminder, however, note that a jump for unsigned data is equal, above, or below, whereas jump for signed data is equal, greater, or less. The jumps for testing the Carry, Overflow, and Parity Flags have unique purposes. The assembler translates symbolic to object code, regardless of which instruction you use but, for example, JAE and JGE, although apparently similar, do not test the same flags.

Lesson 3 Calling Procedures and Passing Parameters

Learning Outcomes

- LO 3.3.1 Rationalize why procedures are important in organizing an assembly code.
- LO 3.3.2 Explain what would happen when the program would initiate Calling to and Returning from a specific procedure, then implement it in assembly coding.
- LO 3.3.3 Explain and implement the various passing parameter in assembly language.

Calling Procedures

A code segment, however, may contain any number of procedures, each distinguished by its own PROC and ENDP directives. A called procedure (or subroutine) is a section of code that performs a clearly defined task (such as set cursor position or get keyboard input). Organizing a program into procedures provides the following benefits:

- Reduces the amount of code because a common procedure can be called from any number of places in the code segment
- Encourages better program organization
- Facilitates debugging of a program because defects can be more clearly isolated.
- Helps in the ongoing maintenance of programs because procedure are readily indentified and modified.

A programming convention is to provide comments at the start of each procedure to identify its purpose and registers used and; if necessary, to push changed registers at the start and to pop them on exiting.

Call and Retn Operation

The purpose of the CALL instruction is to transfer control to a called procedure. The RETn instruction, effectively the counterpart of CALL, returns from the called procedure to the original calling procedure. RETn is normally the last instruction in the called procedure. The format of CALL and RETn are:

[Label:]	CALL	Procedure-name
[Label:]	RETn	[Immediate]

The assembler can tell from the procedure whether RET is near or far an generates the appropriate object code. However, for purposes of clarity, you may code RETN for near and RETF for far returns. The particular object code that CALL and RET generate depends on whether the operation involves a NEAR and FAR procedure.

Near Call and Return: A CALL to a procedure within the same segment is near and performs the following:

- By means of a push operation, decrements SP by 2 (one word) and transfers IP (containing the offset of the instruction following the CALL) onto the stack.
- Inserts the offset address of the called procedure into IP (and also flushes the processor's prefetch instruction queue)
- A RET (or RETN) that returns from a near procedure basically reverses the CALL's steps:
- Pops the old IP value from the stack back into the IP (and flushes the processor's prefetch instruction queue)
- Increments SP by 2.

Passing Parameters

A common practice when calling a procedure is to pass what are called parameters (or arguments) that the procedure is to use. A program may pass parameters by value (the actual data item) or by reference (the address of the data). As well, it may pass parameters in register or in the stack.

Passing Parameters by Value

The examples below illustrate passing parameters by value. The program passes multiplicand and multiplier as parameters to a procedure that simply multiplies them. This version of MUL assumes the multiplicand in AX and multiplier in the operand and develops the product in DX:AX pair.

Pass values in register

```
MOV AX, MULTICAND ; Moves MULTICAND value to AX
MOV BX, MULTIPLER ; Moves MULTIPLER value to BX
CALL MULT_FUN C
…
MULT_FUNC PROC NEAR
      MUL BX            ; Multiplies AX with BX and stores in AX
      RET
MULT_FUNC ENDP
```

Pass values in stack

```
PUSH MULTICAND
PUSH MULTIPLER
CALL MULT_FUNC


MULT_FUNC PROC NEAR
      PUSH BP     ; Preserve the value of BP
      MOV BP, SP  ; Since SP cannot indirectly access the stack BP
is used
      MOV AX, [BP+6]    ; BP indirectly addressing to access the
passed value
      MUL WORD PTR [BP+4]     ;at BP+6 and BP+4
```

```
      POP BP            ; POPS BP (the operation increments SP by
2)
      RET 4
;Ret 4 Executes the following: 1) Loads Return address IP and
increments SP ;by 2 (it now points to Multiplier in stack, 2) Adds
the RET immediate value ;(or POP-value) to SP, effectively
"removing" the two parameters from the ;stack.
MULT_FUNC ENDP
```

Passing Parameters by Reference

The examples below illustrate passing parameters by reference and are the methods used by high-level languages for calling subroutines. Passed parameters are the addresses of the multiplier and multiplicand

```
Addresses in Register
LEA BX, MULTICAND     ; Loads address of MULTICAND on BX
LEA SI, MULTIPLER     ; Loads address of MULTICAND on BX
CALL MULT_FUNC
…
MULT_FUNC PROC NEAR
      MOV AX, [BX]    ; Moves value pointed by BX to AX
      MUL WORD PTR [SI]; Multiplies AX with the value pointed
by SI
      RET
MULT_FUNC ENDP
```

Addresses in Stack

```
.386 ;Needed for next 2 pushes
PUSH OFFSET MULTICAND
PUSH OFFSET MULTIPLER
CALL MULT_FUNC
…
MULT_FUNC PROC NEAR
      PUSH BP
      MOV BP, SP
      MOV BX, [BP+6]    ; This is the same with the pass by value
but it
      MOV DI, [BP+4]    ; places address on the stack instead of
value.
      MOV AX, [BX]
      MUL WORD_PTR [DI]
      POP BP
      RET 4
MULT_FUNC ENDP
```

Lesson 4 Boolean Operations, Shifting Bits and Rotating Bits

Learning Outcomes

- LO 3.4.1 Realize how Boolean operation would manipulate the contents of the specific registers.
- LO 3.4.2 Explain how a data would change when it is shifted or rotated.

Boolean Operations

Boolean logic is important in circuitry design and has a parallel in programming logic. The instructions for Boolean logic are AND, OR, TEST, and NOT, all of which can be used to clear,
set, and test bits. The format for the Boolean operations is:

[Label:]	operation	Register/memory, register/memory/immediate

The first operand references on one byte, word, or doubleword in a register or memory and is the only value that is changed. The Second operand references a register, memory, or immediate value, but a memory-to-memory operation is invalid. The operation matches the bits of the two referenced operands and set the CF, OF, PF, SF, and ZF flags accordingly (AF is undefined)

The AND instruction: In the case of AND, if matched bits are both 1, the operation sets the result to 1, all other conditions result to 0.

Operand 1:	0101
And Operand 2:	0011
Result in operand 1	0001

The OR instruction: In the case of OR, if either (or both) of he matched bits is 1, the operation sets the result to 1; if both bits are 0, the result is 0.

Operand 1:	0101
OR Operand 2:	0011
Result in operand 1	0111

The XOR instruction: In the case of XOR, if the matched bits differ, the operation sets the result to 1. If matched bits are the same (both or both 1), the result is zero.

Operand 1:	0101
XOR Operand 2:	0011
Result in operand 1	0110

The TEST instruction: TEST sets the flags as AND does, but does not change the bits referenced in the target operand. If any matching bits are both 1, TEST clears the Zero Flag. See examples below:

```
; 1    TEST CX, 0FFH          ; Does CX contain
       JZ exit                ; a zero value?
; 2    TEST BL, 00000001B     ; Does BL contain
       JNZ exit               ; an odd number?
;3     TEST  CL, 11110000B    ; Are any of the 4 leftmost
       JNZ exit               ;  bits in CL nonzero?
```

The NOT instruction: NOT simply reverses the bits in a byte, word, or a double word in a register or memory so that 0s become 1s and 1s becomes 0s, in effect one's complement. The format is

[Label:]	NOT	Register/memory

NOT differs from NEG in this way: NEG performs two's complement, which changes a binary value from positive to negative and vice-versa by reversing the bits and adding 1. Typically NOT is used on unsigned data and NEG on signed data.

Shifting Bits

The shift instructions, which are part of the computer's logical capability, can perform the following actions:

- Reference a register or memory address
- Shift bits left or right
- Shift up to 8 bits in a byte, 16 bits in a word, and 32 bits in a doubleword
- Shift logically (unsigned) or shift arithmetically (unsigned)

SHR/SAR: Shifting Bits Right

The SHR (Shift Logical Right), SAR(Shift Arithmetic Right),operations shift bits to the right in the designated register or memory location. The format is:

[Label:]	SHR/SAR	Register/Memory, CL/Immediate

Each bit shifted off enters the Carry Flag. SHR (Shift Logical Right) provides for logical (unsigned) data and SAR (Shift Arithmetic Right) for arithmetic (signed) data.

SHR:

SAR:

The second operand contains the shift value, which is an immediate value or a reference to the CL register. For the 8088/8086 processors, the immediate shift value may be only 1; a greater value must be contained in CL, whereas later processors allow immediate shift values up to 31.

Example for SHR:

Instruction	**Comment**	**Binary**	**Decimal**	**CF**
MOV BH, 10110111B	;Initialize BH	10110111	183	
SHR BH, 01	;Shift right 1	01011011	91	1
MOV CL, 02	; Set shift value			
SHR BH, CL	;Shift right 2 more	00010110	22	1

SHR BH, 02	;Shift right 2 more	00000101	5	1

SAR differs from SHR in one important way: SAR uses the sign bit to fill leftmost vacated bits. In this way, positive and negative values retain their signs.

Example for SAR

Instruction	**Comment**	**Binary**	**Decimal**	**CF**
MOV BH, 10110111B	;Initialize BH	10110111	-73	
SAR BH, 01	;Shift right 1	11011011	-37	1
MOV CL, 02	; Set shift value			
SAR BH, CL	;Shift right 2 more	11110110	-10	1
SAR BH, 02	;Shift right 2 more	11111101	-3	1

SHL/SAL: Shifting Bits Left

The SHL (Shift Logical Left), SAL (Shift Arithmetic Left) operations shift bits to the left in the designated register or memory location. Their format is

[Label:]	SHR/SAR	Register/Memory, CL/Immediate

SHL and SAL has similar functionality in terms of shift value, which is an immediate or a reference to the CL. Each bits shifted off enters the Carry flag. SHL and SAL are identical in operation and both provide for logical (unsigned) and arithmetic (signed) data; that is, there is no difference between left shifting unsigned and signed data.

SHL and SAL

Example for SHL/SAL

Instruction	**Comment**	**Binary**	**Decimal**	**CF**
MOV BH, 00000101B	;Initialize BH	00000101	5	
SHR BH, 01	;Shift right 1	00001010	10	0
MOV CL, 02	; Set shift value			
SHR BH, CL	;Shift right 2 more	00101000	40	0
SHR BH, 02	;Shift right 2 more	10100000	160	0

Rotating Bits

The rotate instructions, which are part of the computer's logical capability, can perform the following actions:

- Reference a register or a memory
- Rotate right or left. The bit that is shifted off rotates to fill the vacated bit position in the register or memory location and is also copied into the Carry flag.

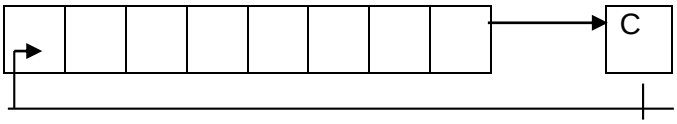
- Rotates up to 8 bits in a byte, 16 bits in a word, and 32 bits in a doubleword
- Rotates logically (unsigned) or Rotates arithmetically (unsigned).

ROR/RCR: Rotating Bits Right

The ROR and RCR operations rotate the bits to the right in the designated register or memory location. Each bit rotated off enters the Carry Flag. ROR (Rotate Logical Right) provides for logical (unsigned) and RCR (Rotate with Carry Right) for arithmetic (signed) data.

ROR

RCR

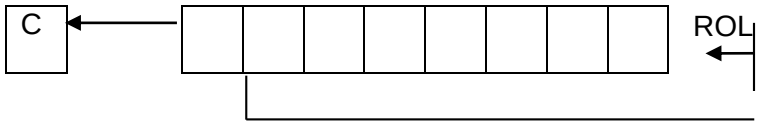


The examples below illustrate ROR

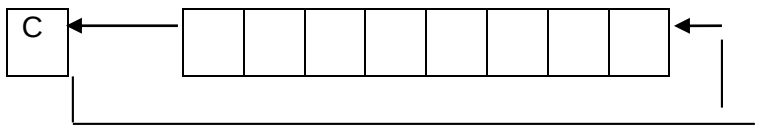
Instruction	**Comment**	**Binary**	**CF**
MOV BL, 10110111B	;Initialize BL	10110111	-
ROR BL, 01	;Rotate right 1	11011011	1
MOV CL, 03	; Set shift value		
ROR BL, CL	;Rotate right 3 more	01111011	0
ROR BL, 03	;Rotate right 3 more	01101111	0

ROL/RCL: Rotating Bits Right

The ROL and RCL operations rotate the bits to the left in the designated register or memory location. Each bit rotated off enters the Carry Flag. ROL (Rotate Logical Right) provides for logical (unsigned) and RCR (Rotate with Carry Right) for arithmetic (signed) data.



RCL



Examples of ROL

Instruction	**Comment**	**Binary**	**CF**
MOV BL, 10110111	;Initialize BL	10110111	-
ROL BL, 01	;Rotate right 1	01101111	1
MOV CL, 03	; Set shift value		
ROL BL, CL	;Rotate right 2 more	01111011	1
ROL BL, 03	;Rotate right 3 more	11011011	1