Assembly Language Programming

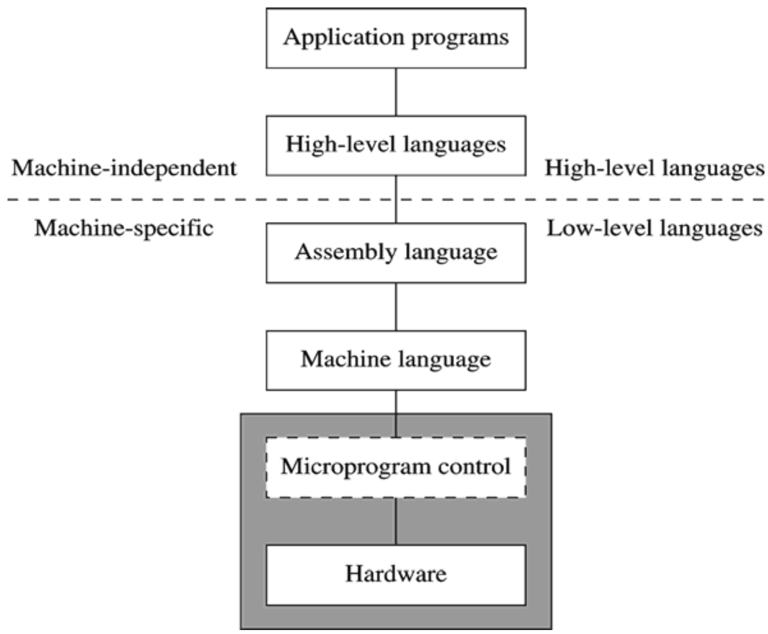
Introduction

Assembly language is a low level programming language. An Assembly Language program consists of a series of lines that are assembly language instructions. These instructions consist of a mnemonic, which is a command, and an operand, which is the data to be manipulated. The programs usually include comments which are written at the end of a line or in a separate line beginning with a ';' and are ignored by the assembler. Assembly Language uses two, three or 4 letter mnemonics to represent each instruction type.

Low level Assembly Language is designed for a specific family of Processors: the symbolic instruction directly relate to Machine Language instructions one for one and are assembled into machine language

To make programming easier than machine language, many programmers write programs in assembly language

They then translate Assembly Language program to machine language so that it can be loaded into memory and run.



Advantages of Assembly Language

- Assembly Language Program requires less Memory and execution time compared to High Level Language. Assembly Language is useful for implementing system software.
- Assembly Language gives a programmer the ability to program small embedded system applications. Firmware (that resides in memory while other programs execute) and Interrupt Service Routine (that handles input and output) are developed in Assembly Language.
- It helps to understand sources of program inefficiency and helps in tuning program performance by providing more control over handling particular hardware requirements.
- Helps to write smaller and compact executable codes.

Types of assembler

Assembler can be categorized according to passes.

One Pass Assembler:

A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references and assemble code in one pass. If each statement of an assembly language program is expressed only in terms of symbols previously defined in the program then the assembler can be implemented in one pass.

One pass assembler can be categorized as *load-and-go* type which produces executable modules and the other one which can produce load, or even objects that could be linked.

All the labels of the instructions are symbols and symbol tables has entry for symbol name, address value .Symbols that are defined in the later part of the program are called forward referencing. The main problem of one pass assembler is forward referencing.

Two pass assembler

A two pass assembler does two passes over the source file (the second pass can be over a file generated in the first pass). In the first pass all it does is looks for label definitions and introduces them in the symbol table. In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations and similar operations.

A two-pass assembler performs two sequential scans over the source code:
First Pass: symbols and literals are defined
Second Pass: object program is generated

First Pass
On the first pass, the assembler performs the following tasks:
•Checks to see if the instructions are legal in the current assembly mode.
•Allocates space for instructions and storage areas you request.
•Fills in the values of constants, where possible.
•Builds a symbol table, also called a cross-reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement

Second Pass
On the second pass, the assembler:
•Examines the operands for symbolic references to storage locations and resolves these symbolic references using information in the symbol table.
•Ensures that no instructions contain an invalid instruction form.

•Translates source statements into machine code and constants, thus filling the allocated space with object code.
•Produces a file containing error messages, if any have occurred.

Typical format of an assembly language instruction

Assembly language instructions have the format:

[label:]	mnemonic or opcode field	[operands field]	[;comment]
START :	MOV	AX, BX	;COPIES CONTENTS OF BX TO AX

Field inside [ ] is an optional field.

Assembly language statements are usually written in a standard form that has 4 fields. A **Label** is a symbolic name for an address. If an instruction is started with a label, then it can be addressed by this label in the program. We use a label to allow for branching, and naming memory location. They are followed by colon.

The **opcode field** contains reserved symbols that correspond to instructions which contain the mnemonics for the instruction to be performed. The instruction mnemonics are sometimes called as operation codes. Opcodes tell the CPU the following information:

- The operation
- Number of operands
- Data type upon which it will operate

Operands are either in memory or in registers. Programmers must specify the ways in which the instruction will find them. The different ways in specifying the location of the operands are called addressing modes. The operand field of the statement contains the data, the memory address, the port address or the name of the register on which the instruction is to be performed.

Comments start with semicolon. They are very important they, explain the program's purpose like when it was written, revised, and by whom, explain data used in the program, explain instruction sequences and algorithms used and application-specific explanations

Assembly Language Program Development Tools

1. **EDITOR:**

An Editor is a Program which allows user to create (write), modify and store a group of instructions or text under a filename containing the Assembly Language statements for your Program. The saved file must be in ASCII format for assembler to recognize. The file is called source file. It allows user to inputting or modifying text that is stored in mass storage device. Example: Notepad, MS DOS Edit.

2. **ASSEMBLER**

It translates a program written in assembly language into machine language or object code. An Assembler Program is used to translate the assembly language mnemonics for instruction to the corresponding binary codes.

3. **LINKER**

A linker is a program that combines object files to create an single “executable” file. Linking is the process of resolving symbols between independent object files of different modules. It resolves all references (ie. Program contains all parts needed to run). It is used to join several files into one large .obj file. It produces .exe file so that the program becomes executable.

4. **LOCATOR**

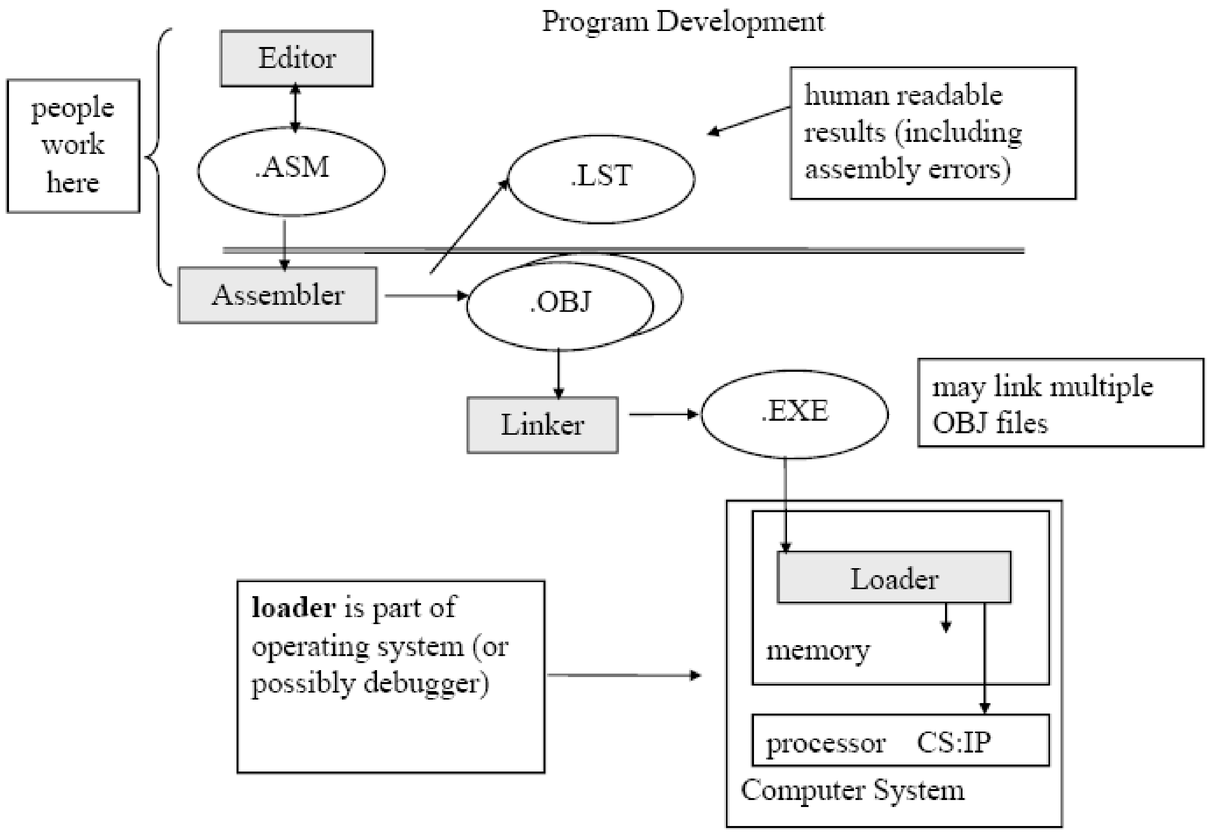
A Locator is a program used to assign the specific address of where the segment of object code is to be loaded into memory. It usually converts .exe file to .bin file which has physical address. A Locator program called EXE2BIN converts .exe files to .bin file.

5. **DEBUGGER**

A Debugger is a program which allows you to load your .obj code program into system memory, execute program and troubleshoot or debug it. Loader is the part of debugger that loads executable files into memory, and may initialize some registers (e.g. IP) and starts it going. Debugger loads but controls the execution of the program to start or stop execution, to view and modify state variables. It allows us to look at the content of registers and memory locations after we run program. It allows to set the breakpoint.

6. **EMULATOR**

An Emulator is a mixture of hardware and software. It is used to test and debug the hardware and software of an external system such as the prototype of a Microprocessor based instrument. It also allows us to look at the content of registers and memory locations after we run program and take snapshots. We can either use emulator or debugger to develop the program.



Instruction Sets

The instructions of 8086 are classified into SIX groups. They are:

1. DATA TRANSFER INSTRUCTIONS
2. ARITHMETIC INSTRUCTIONS
3. BIT MANIPULATION INSTRUCTIONS
4. STRING INSTRUCTIONS
5. PROGRAM EXECUTION TRANSFER INSTRUCTIONS
6. PROCESS CONTROL INSTRUCTIONS

1.DATA TRANSFER INSTRUCTIONS

The DATA TRANSFER INSTRUCTIONS are those, which transfers the DATA from any one source to any one destination.The datas may be of any type. They are again classified into four groups.They are:

GENERAL – PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS	SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTION	SPECIAL ADDRESS TRANSFER INSTRUCTION	FLAG TRANSFER INSTRUCTIONS
MOV PUSH POP XCHG XLAT	IN OUT	LEA LDS LES	LAHF SAHF PUSHF POPF

2.ARITHMETIC INSTRUCTIONS

These instructions are those which are useful to perform Arithmetic calculations, such as addition, subtraction, multiplication and division.They are again classified into four groups.They are:

ADDITION INSTRUCTIONS	SUBTRACTION INSTRUCTIONS	MULTIPLICATION INSTRUCTIONS	DIVISION INSTRUCTIONS
ADD ADC INC AAA DAA	SUB SBB DEC NEG CMP AAS DAS	MUL IMUL AAM	DIV IDIV AAD CBW CWD

3.BIT MANIPULATION INSTRUCTIONS

These instructions are used to perform Bit wise operations.

LOGICAL INSTRUCTIONS	SHIFT INSTRUCTIONS	ROTATE INSTRUCTIONS
NOT AND OR XOR TEST	SHL / SAL SHR SAR	ROL ROR RCL RCR

4. STRING INSTRUCTIONS

The string instructions function easily on blocks of memory.They are user friendly instructions, which help for easy program writing and execution. They can speed up the manipulating code.They are useful in array handling, tables and records.

STRING INSTRUCTIONS
REP REPE / REPZ REPNE / REPNZ MOVS / MOVSB / MOVSW COMPS / COMPSB / COMPSW SCAS / SCASB / SCASW LODS / LODSB / LODSW STOS / STOSB / STOSW

5.PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions transfer the program control from one address to other address. (Not in a sequence). They are again classified into four groups.They are:

UNCONDITIONAL TRANSFER INSTRUCTIONS	CONDITIONAL TRANSFER INSTRUCTIONS		ITERATION CONTROL INSTRUCTIONS	INTERRUPT INSTRUCTIONS
CALL	JA / JNBE	JLE / JNG	LOOP	INT
RET	JAE / JNB	JNC	LOOPE / LOOPZ	INTO
JMP	JB / JNAE	JNE / JNZ	LOOPNE / LOOPNZ	IRET
	JBE / JNA	JNO	JCXZ	
	JC	JNP / JPO		
	JE / JZ	JNS		
	JG / JNLE	JO		
	JGE / JNL			
	JL / JNGE	JP / JPE		
		JS		

6.PROCESS CONTROL INSTRUCTIONS

These instructions are used to change the process of the Microprocessor. They change the process with the stored information. They are again classified into Two groups.They are:

FLAG SET / CLEAR INSTRUCTIONS	EXTERNAL HARDWARE SYNCHRONIZATION INSTRUCTIONS
STC CLC CMC STD CLD STI CLI	HLT WAIT ESC LOCK NOP

1. DATA TRANSFER INSTRUCTIONS

1.1 GENERAL PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS

MOV Destination, Source
MOV
MOV CX,04H
	Copy byte or word from specified source to specified destination.

PUSH PUSH Source PUSH BX	Copy specified word to top of stack.
POP POP Destination POP AX	Copy word from top to stack to specified location.
XCHG XCHG Destination, Source XCHG AX,BX	Exchange word or byte.
XLAT	Translate a byte in AL using a table in memory. It first adds AL + BX to form memory address. It then copies the content into AL

1.2 SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
IN IN AX, Port_Addr IN AX,34H	Copy a byte or word from specified port to accumulator.
OUT OUT Port_Addr, AX OUT 2CH,AX	Copy a byte or word from accumulator to specified port.

1.3 SPECIAL ADDRESS TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LEA LEA Register, Source LEA BX,PRICE	Load effective address of operand into specified register.

LDS
LDS Register, Source
LDS BX,[4326H]
	Load DS register and other specified register from memory.

LES	Load ES register and other specified register from memory.

1.4 FLAG TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LAHF	Copy to AH with the low byte of the flag register.
SAHF	Stores AH register to low byte of flag register.
PUSHF	Copy flag register to top of stack.
POPF	Copy word at top of stack to flag register.

2. ARITHMETIC INSTRUCTIONS

2.1 ADDITION INSTRUCTIONS

INSTRUCTIONS	COMMENTS

ADD
ADD Destination, Source
ADD AL,74H
	Add specified byte to byte or word to word.

ADC ADC Destination, Source ADC CL,BL	Add byte + byte + carry flag Add word +word + carry flag
INC INC Register INC CX	Increment specified byte or word by 1.
AAA	ASCII adjust after addition.
DAA	Decimal adjust after addition.

2.2 SUBTRACTION INSTRUCTIONS

INSTRUCTIONS	COMMENTS

SUB
SUB Destination, Source
SUB CX,BX
	Subtract byte from byte or word from word.

SBB SBB Destination, Source SBB CH,AL	Subtract byte and carry flag from byte. Subtract word and carry flag from word.
DEC DEC Register DEC CX	Decrement specified byte or word by 1.
NEG NEG Register NEG AL	Form 2's complement.

CMP
CMP Destination, Source
CMP CX,BX
CF ZF SF
CX = BX 0 1 0
CX > BX 0 0 0
CX < BX 1 0 1
	Compare two specified bytes or words.

AAS	ASCII adjusts after subtraction.
DAS	Decimal adjusts after subtraction.

2.3 MULTIPLICATION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
MUL MUL Source MUL CX	Multiply unsigned byte by byte or unsigned word by word. When a byte is multiplied by the content of AL, the result is kept into AX. When a word is multiplied by the content of AX, MS Byte in DX and LS Byte in AX.
IMUL IMUL Source IMUL CX	Multiply signed byte by byte or signed word by word.
AAM	ASCII adjusts after multiplication. It converts packed BCD to unpacked BCD.

2.4 DIVISION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
DIV DIV Source DIV BL DIV CX	Divide unsigned word by byte Divide unsigned double word by byte. When a word is divided by byte, the word must be in AX register and the divisor can be in a register or a memory location. After division AL (quotient) AH (remainder) When a double word is divided by word, the double word must be in DX: AX pair and the divisor can be in a register or a memory location. After division AX (quotient) DX (remainder)
AAD	ASCII adjust before division BCD to binary convert before division.
CBW	Fill upper byte of word with copies of sign bit of lower byte.
CWD	Fill upper word of double word with sign bit of lower word.

3. BIT MANIPULATION INSTRUCTIONS

3.1 LOGICAL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
NOT NOT Destination NOT BX	Invert each bit of a byte or word.
AND AND Destination, Source AND BH,CL	AND each bit in a byte/word with the corresponding bit in another byte or word.
OR OR Destination, Source OR AH,CL	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR XOR Destination, Source XOR CL,BH	XOR each bit in a byte or word with the corresponding bit in another byte or word.
TEST TEST Destination, Source TEST AL,BH	AND operands to update flags, but don't change the operands.

3.2 SHIFT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SHL/SAL SAL Destination, Count SHL Destination, Count CF←MSB←LSB←0	Shift Bits of Word or Byte Left, Put Zero(s) in LSB.
SHR SHR Destination, Count 0→MSB→LSB→CF	Shift Bits of Word or Byte Right, Put Zero(s) in MSB.
SAR SAR Destination, Count MSB→MSB→LSB→CF	Shift Bits of Word or Byte Right, Copy Old MSB into New MSB.

3.3 ROTATE INSTRUCTIONS

INSTRUCTIONS	COMMENTS
ROL	Rotate Bits of Byte or Word Left, MSB to LS and to CF.
ROR	Rotate Bits of Byte or Word Right, LSB to MSB and to CF.
RCL	Rotate Bits of Byte or Word Left, MSB to CF and CF to LSB.
RCR	Rotate Bits of Byte or Word Right, LSB TO CF and CF TO MSB.

4. PROGRAM EXECUTION TRANSFER INSTRUCTIONS

4.1 UNCONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
CALL	Call a Subprogram/Procedure.
RET	Return From Procedure to Calling Program.
JMP	Go to Specified Address to Get Next Instruction (Unconditional Jump to Specified Destination).

4.2 CONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
JA/JNBE	Jump if Above/Jump if Not Below or Equal.
JAE/JNB	Jump if Above or Equal/Jump if Not Below.
JB/JNAE	Jump if Below/Jump if Not Above or Equal.
JBE/JNA	Jump if Below or Equal/Jump if Not Above.
JC	Jump if Carry Flag CF=1.
JE/JZ	Jump if Equal/Jump if Zero Flag (ZF=1).
JG/JNLE	Jump if Greater/Jump if Not Less than or Equal.
JGE/JNL	Jump if Greater than or Equal/Jump if Not Less than.
JL/JNGE	Jump if Less than/Jump if Not Greater than or Equal.
JLE/JNG	Jump if Less than or Equal/Jump if Not Greater than.
JNC	Jump if No Carry i.e. CF=0
JNE/JNZ	Jump if Not Equal/Jump if Not Zero(ZF=0)
JNO	Jump if No Overflow.
JNP/JPO	Jump if Not Parity/Jump if Parity Odd.
JNS	Jump if Not Sign(SF=0)
JP/JPE	Jump if Parity/Jump if Parity Even (PF=1)
JS	Jump if Sign (SF=1)

4.3 ITERATION CONTROL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LOOP	Loop Through a Sequence of Instructions Until CX=0.
LOOPE/LOOPZ	Loop Through a Sequence of Instructions While ZF=1 and CX!=0.
LOOPNE/LOOPNZ	Loop Through a Sequence of Instruction While ZF=0 & CX!=0.
JCXZ	Jump to Specified Address if CX=0.

4.4 INTERRUPT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
INT	
INT0	Interrupt Program Execution if OF=1
IRET	Return From Interrupt Service Procedure to Main Program.

5 PROCESSOR CONTROL INSTRUCTIONS

5.1 FLAG SET/CLEAR INSTRUCTION

INSTRUCTIONS	COMMENTS
STC	Set Carry Flag CF to 1.
CLC	Clear Carry Flag to 0.
CMC	Complement the State of CF.
STD	Set Direction Flag to 1.
CLD	Clear Direction Flag to 0.
STI	Set Interrupt Flag to 1. (Enable INTR Input).
CLI	Clear Interrupt Enable to 0

5.2 NO OPERATION INSTRUCTION

INSTRUCTIONS	COMMENTS
NOP	No Action Except Fetch and Decode.

5.3 EXTERNAL HARDWARE SYNCHRONIZATION INST.

INSTRUCTIONS	COMMENTS
HLT	Halt (Do Nothing) Until Interrupt or Reset.
WAIT	Wait Until Signal On the TEST Pin is Low.
ESC	Escape to External Coprocessor Such as 8087 or 8089.
LOCK	Prevents Another Processor From Taking the Bus While the Adjacent Instruction Executes.

6 STRING INSTRUCTIONS

INSTRUCTIONS	COMMENTS
REP	Repeat Instruction Until CX=0.
REPE/REPZ	Repeat if Equal/Repeat if Zero
REPNE/REPNZ	Repeat if Not Equal/Repeat if Not Zero.
MOVS/MOVSB/MOVSW	Move Byte or Word From One String to another.
COMPS/COMPSB/COMPSW	Compare Two String Bytes or Two String Words.
SCAS/SCASB/SCASW	Compares a Byte in AL or Word in AX With a Byte or Word Pointed By DI in ES.

Assembly Language Program Features

RESERVED WORDS :

A reserved word has a special meaning fixed by the language. You can use it only under certain conditions. Reserved words in MASM include:

Instructions, which correspond to operations the processor can execute. Ex . MOV, ADD
Directives, which give commands to the assembler. Ex . END,SEGMENT , .CODE
Attributes, which provide a value for a field, such as segment alignment. Ex. A LOD R1,54' assigns to label A the size of the LOD instruction (in words)
Operators, which are used in expressions. Ex. FAR,OFFSET
Predefined symbols, which return information to your program. Ex. @CODE

IDENTIFIERS

An identifier is a name that you invent and attach to a definition. Identifiers can be symbols representing variables, constants, procedure names, code labels, segment names, and user-defined data types such as structures, unions, records, and types defined with TYPEDEF. Identifiers longer than 247 characters generate an error. Certain restrictions limit the names you can use for identifiers. Follow these rules to define a name for an identifier:
The first character of the identifier can be an alphabetic character (A-Z) or any of these four characters: @ _ $? The other characters in the identifier can be any of the characters listed above or a decimal digit (0-9). Avoid starting an identifier with the at sign (@)

An Identifier (or symbol) can also be defined as name that you apply to an item in your program that you expect to reference. The two types of identifiers are NAME and LABEL.
NAME: Refers to the Address of a data item COUNTER DB 0
LABEL: Refer to the Address of an instruction, procedure, or segment.
Example: Start PROC
Next2: MOV AL, BL

PREDEFINED SYMBOLS

The assembler includes a number of predefined symbols (also called predefined equates). You can use these symbol names at any point in your code to represent the equate value. For example, the predefined equate @Data expands to data segment by pointing the start of the segment .The predefined symbols for segment information include:

@code	Returns the name of the current code segment.	Text value
@data	Returns the name of the current data segment.	Text value
@FarData?	Returns the name of the current far data segment.	Text value
@Word-Size	Returns two if this is a 16 bit segment, four if this is a 32 bit segment.	Numeric value
@Code-Size	Returns zero for Tiny, Small, Compact, and Flat models. Returns one for Medium, Large, and Huge models.	Numeric value
@DataSize	Returns zero for Tiny, Small, Medium, and Flat memory models. Returns one for Compact and Large models. Returns two for Huge model pro-grams.	Numeric value

@Model	Returns one for Tiny model, two for Small model, three for Compact model, four for Medium model, five for Large model, six for Huge model, and seven for Flag model.	Numeric value
@CurSeg	Returns the name of the current code segment.	Text value
@stack	The name of the current stack segment.	Text value

INTEGER CONSTANTS AND CONSTANT EXPRESSIONS

An integer constant is a series of one or more numerals followed by an optional radix specifier. For example, in these statements

mov ax, 45

mov bx, 0D3h

the numbers 45 and 0D3h are integer constants. The h appended to 0B3 is a radix specifier. The specifiers are:

b for binary

o or q for octal

d for decimal

h for hexadecimal

Radix specifiers can be either uppercase or lowercase letters. Hexadecimal numbers must always start with a decimal digit (0-9). For example, MASM interprets ABCh as an identifier.

Constant expressions contain integer constants and (optionally) operators such as shift, logical, and arithmetic operators. The assembler evaluates constant expressions at assembly time. (In addition to constants, expressions can contain labels, types, registers, and their attributes.) Constant expressions do not change value during program execution.

STATEMENTS

An Assembly Program consists of a set of statements. Statements are the line-by-line components of source files. The two types of statements are:

1.INSTRUCTION Instructions such as MOV & ADD which the Assembler translates to Object Code.

2.DIRECTIVES

Directives tell the Assembler to perform a specific action, such as define a data item etc.

DIRECTIVES

The directives are commands to the assembler, directing it to perform operations other than assembling instructions. The directives are thus executed by the assembler, not assembled by it. They may affect all the operations of the assembler. Assembly Language supports a number of statements that enable to control the way in which a source program assembles and lists. These Statements are called Directives. They act only during the assembly of a program and generate no machine executable code. The most common Directives are PAGE, TITLE, PROC, and END.

We will describe the following directives only .

PAGE DIRECTIVE

The PAGE Directive helps to control the format of a listing of an assembled program. It is optional Directive. At the start of program, the PAGE Directive designates the maximum number of lines to list on a page and the maximum number of characters on a line.

Its format is

PAGE [LENGTH], [WIDTH]

Omission of a PAGE Directive causes the assembler to set the default value to PAGE 50,80

TITLE DIRECTIVE

The TITLE Directive is used to define the title of a program to print on line 2 of each page of the program listing. It is also optional Directive.

Its format is TITLE [TEXT]

TITLE "PROGRAM TO PRINT FACTORIAL NO"

SEGMENT DIRECTIVE

The SEGMENT Directive defines the start of a segment. A Stack Segment defines stack storage, a data segment defines data items and a code segment provides executable code. MASM provides simplified Segment Directive. The format (including the leading dot) for the directives that defines the stack, data and code segment are

.STACK [SIZE]

.DATA

………………. ; Initialize Data Variables

.CODE

The Default Stack size is 1024 bytes. To use them as above, Memory Model initialization should be carried out.

MEMORY MODEL DEFINTION

The different models tell the assembler how to use segments to provide space and ensure optimum execution speed.

The format of Memory Model Definition is

.MODEL [MODEL NAME]

The Memory Model may be TINY, SMALL, MEDIUM, COMPACT, LARGE and HUGE

TINY :All DATA, CODE & STACK Segment must fit in one Segment of Size <=64K.

SMALL : One Code Segment of Size <=64K. One Data Segment of Size <=64 K.

MEDIUM: One Data Segment of Size <=64K. Any Number of Code Segments.

COMPACT: One Code Segment of Size < =64K. Any Number of Data Segments.

LARGE: Any Number of Code and Data Segments.
HUGE: Any Number of Code and Data Segments.

THE PROC DIRECTIVE

The Code Segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC Directive and ended with the ENDP Directive. Its Format is given as:

PROCEDURE NAME **PROC**
………………..
……………….
PROCEDURE NAME **ENDP**

ENDP DIRECTIVE

The ENDP Directive indicates the end of a procedure. It is used to" bracket a procedure".

END DIRECTIVE

An END Directive ends the entire Program and appears as the last statement. The assembler will ignore any statements after END . Its Format is
END [PROCEDURE NAME]

PROCESSOR DIRECTIVE

MASM supports a set of directives for selecting processors and coprocessors. Once you select a processor, you must use only the instruction set for that processor. The default is the 8086 processor. If you always want your code to run on this processor, you do not need to add any processor directives.To enable a different processor mode and the additional instructions available on that processor, use the directives .186, .286, .386, and .486.Most Assemblers assume that the source program is to run on a basic 8086 level. As a result, when you use instructions or features introduced by later processors, you have to notify the assemblers be means of a processor directive as .286,.386,.486 or.586 This directive may appear before the Code Segment.

THE EQU DIRECTIVE

It is used for redefining symbolic names
EXAMPLE
NUM DB 25
DATA EQU NUM

THE .STARTUP AND .EXIT DIRECTIVE

MASM 6.0 introduced the .STARTUP and .EXIT Directive to simplify program initialization and Termination. .STARTUP generates the instruction to initialize the Segment Registers. If you do not use .STARTUP, you must give the starting address as an argument to the END directive. .EXIT generates the INT 21H function 4ch instruction for exiting the Program. .EXIT generates executable code, while END does not.

DEFINING TYPES OF DATA

The Format of Data Definition is given as

[NAME] DV [EXPRESSION]   ; DV represents various data types.

EXAMPLES

STRING DB 'kathmandu$'
NO1 DB 5
NO2 DB 78

DEFINITION DIRECTIVE

DB Used to declare BYTE type variable
DW Used to declare WORD type variable
DD Used to declare DOUBLE WORD type variable
DF Used to declare FAR WORD type variable
DQ Used to declare QUAD WORD type variable
DT Used to declare TEN BYTES type variable

Duplication of Constants in a Statement is also possible and is given by

[NAME] DV [REPEATCOUNT[DUP (EXPRESSION)]
EXAMPLES
NUM DB 5 DUP(11) ; 5 Bytes containing hex 0b0b0b0b0b
VAL DW 10 DUP(?) ; 10 Words Uninitialized '?' represents uninitialized space
VALUE DB 3 DUP(5 DUP(4)) ; 44444 44444 44444

1. CHARACTER STRINGS

Character Strings are used for descriptive data which mainly contains ASCII characters. Consequently DB is the conventional format for defining character data of any length
Example
DB 'Management'
DB "Information"
DB "Star"

2. NUMERIC CONSTANTS

BINARY : NUM DB 10111010B
DECIMAL : DATA DB 478
HEXADECIMAL : VAL1 DB 78H

ASSEMBLY LANGUAGE PROGRAMMING USING MASM

GENERAL PATTERN FOR WRITING ALP IN MASM
[Page Directive]
[Title Directive]
[Memory Model Definition]
[Segment Directive]
[Proc Directive]
…………..; codes
…………..;
[End Directive]

Example Format
PAGE 60,80
TITLE "Program Name"
.MODEL [MODEL NAME]
.STACK
.DATA ……………………… ; INITIALIZE DATA VARIALBLES
.CODE
MAIN PROC
……………………………
; INSTRUCTION SETS
…………………………..
MAIN ENDP
END MAIN

Sample Program

```
 EXAMPLE 1
 ;Program to Print Hello World in ALP
  .MODEL SMALL
 .STACK
 .DATA
 STRING DB 'HELLO WORLD $'
 .CODE
 ;-----------------------------------------------------------
START PROC
MOV AX,@DATA
MOV DS,AX ; Initialize the DATA Segment
MOV DX,OFFSET STRING ; Load the Offset Address into DX
```

```
MOV AH,09H ; AH=09H For String Display until $
INT 21H ; DOS Interrupt Function
MOV AX,4C00H ; End Request with AH=4CH  or AX=4C00H
INT 21H
START ENDP ; End Procedure
END MAIN ; End Program
 ;-----------------------------------------------------------
```

The first three lines of the program are comments to give the name of the file containing the program, explain its purpose, give the name of the author and the date the program was written. The first two directives, **.model** and **.stack** are concerned with how your program will be stored in memory and how large a stack it requires. The third directive, **.code**, indicates where the program instructions (i.e. the program code) begin. For the moment, suffice it to say that you need to start all assembly languages programs in a particular format (not necessarily that given above. Your program must also finish in a particular format; the end directive indicates where your program finishes. In the middle comes the code that you write yourself. You must also specify where your program starts, i.e. which is the first instruction to be executed. This is the purpose of the label, **start**.(Note: We could use any label, e.g. begin or main in place of start).
This same label is also used by the end directive. When a program has finished, we return to the operating system. Like carrying out an I/O operation, this is also accomplished by using the **int** instruction. This time MS-DOS subprogram number **4c00h** is used.It is the subprogram to terminate a program and return to MSDOS.Hence, the instructions:
mov ax, 4c00h ; Code for return to MS-DOS
int 21H ; Terminates program
terminate a program and return you to MS-DOS.

First of all we will explanation some of Instruction Sets

AAA Instruction - AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand.Two operands of the addition must have its lower 4 bits contain a number in the range from 0-9.The AAA instruction then adjust AL so that it contains a correct BCD digit. If the addition produce carry (AF=1), the AH register is incremented and the carry CF and auxiliary carry AF flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases the higher 4 bits of AL are cleared to 0.

AAA will adjust the result of the two ASCII characters that were in the range from 30h (“0”) to 39h(“9”).This is because the lower 4 bits of those character fall in the range of 0-9.The result of addition is not a ASCII character but it is a BCD digit.
Example:

```
MOV AH,0       ; Clear AH for MSD
MOV AL,6       ; BCD 6 in AL
ADD AL,5       ; Add BCD 5 to digit in AL
```

AAA ; AH=1, AL=1 representing BCD 11.

AAD Instruction - ADD converts unpacked BCD digits in the AH and AL register into a single binary number in the AX register in preparation for a division operation.
Before executing AAD, place the Most significant BCD digit in the AH register and Last significant in the AL register. When AAD is executed, the two BCD digits are combined into a single binary number by setting AL=(AH*10)+AL and clearing AH to 0.

Example:
```
MOV AX,0205h    ;  The unpacked BCD number 25
AAD             ;  After AAD , AH=0 and;AL=19h (25)
```
After the division AL will then contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder.

Example:
```
;AX=0607 unpacked BCD for 67 decimal ;CH=09H
AAD        ;Adjust to binary before division ;AX=0043 = 43H =67 decimal
DIV CH     ;Divide AX by unpacked BCD in CH ;AL = quotient = 07 unpacked BCD ;AH = remainder = 04 unpacked BCD
```

AAM Instruction - AAM converts the result of the multiplication of two valid unpacked BCD digits into a valid 2-digit unpacked BCD number and takes AX as an implicit operand.
To give a valid result the digits that have been multiplied must be in the range of 0 – 9 and the result should have been placed in the AX register. Because both operands of multiply are required to be 9 or less, the result must be less than 81 and thus is completely contained in AL.
AAM unpacks the result by dividing AX by 10, placing the quotient (MSD) in AH and the remainder (LSD) in AL.
Example:
```
MOV AL, 5
MOV BL, 7
MUL BL     ;  Multiply AL by BL , result in AX
AAM        ;  After AAM, AX =0305h (BCD 35)
```

AAS Instruction - AAS converts the result of the subtraction of two valid unpacked BCD digits to a single valid BCD number and takes the AL register as an implicit operand. The two operands of the subtraction must have its lower 4 bit contain number in the range from 0 to 9 .The AAS instruction then adjust AL so that it contain a correct BCD digit.
```
MOV AX,0901H    ;  BCD 91
SUB AL, 9       ;  Minus 9
AAS             ;  Give AX =0802 h (BCD 82)
```

(a)
; AL =0011 1001 =ASCII 9 ;BL=0011 0101 =ASCII 5

```
SUB AL, BL               ;  (9 - 5) Result AL = 00000100 = BCD 04,CF = 0
AAS                      ;Result AL=00000100 =BCD 04 CF = 0 NO Borrow  required
```

(b)

```
;AL = 0011 0101 =ASCII 5 BL = 0011 1001 = ASCII 9
SUB AL, BL              ;( 5 - 9 ) Result AL = 1111 1100 = - 4   in 2's complement
  CF = 1
AAS                     ;Results AL = 0000 0100 =BCD 04 CF = 1 borrow needed .
```

Example 2

```
;Program to Print the Sum of Two 8 Bit Numbers
.MODEL SMALL
.STACK
.DATA
VAL1 DB 89
VAL2 DB 10
MSG DB 'SUM OF 2 NUMBERS: $'
.CODE
;-------------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX
 MOV AX, 0
MOV AL, VAL1
ADD AL, VAL2
AAM; AAM Converts Binary Value to Unpacked BCD.
ADD AX, 3030H ;Ax is Added with 3030H to Obtain ASCII Value
PUSH AX
 ;;;;;;;;;;;;DISPLAY MESSAGE
LEA DX, MSG
MOV AH, 09H
INT 21H ;;;;;;;;;;;;END DISPLAY MESSAGE
POP AX
MOV DL, AH
MOV DH, AL
MOV AH, 02H
INT 21H
MOV DL, DH
MOV AH, 02H
INT 21H
MOV AX, 4C00H
INT 21H
```

```
MAIN ENDP
END MAIN
;-----------------------------------------------------------
Example 3
Program to display from 0 to 9
.MODEL SMALL
.STACK
.DATA
VAL DB '0'
.CODE
;-----------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX
MOV CX,10
MOV DL,VAL
TOP:
MOV AH,02H
INT 21H  ; CALL DOS INTERRUPT FOR DISPLAY
INC DL ; INCREASE DL BY 1
PUSH DX ;PUSH DX TO STACK
; ; SPACE
MOV DL,' '
MOV AH,02H
INT 21H
;;;CODE FOR SPACE ENDS

POP DX  ; POP CONTENT OF DX FROM STACK
DEC CX  ; DECREASE CX BY 1
JZ LAST  ;JUMP TO LABEL LAST IF CX=0
JMP TOP ;JUMP TO LABEL TOP

LAST:
MOV  AH,4CH ; EXIT DOS INTERRUPT
INT 21H
MAIN ENDP
END MAIN
```

Learning any imperative programming language involves mastering a number of common concepts:

Variables: declaration/definition
Assignment: assigning values to variables

Input/Output: Displaying messages
Displaying variable values
Control flow: if-then
Loops
Subprograms: Definition and Usage
Programming in assembly language involves mastering the same concepts and a few other issues.

Variables
For the moment we will skip details of variable declaration and simply use the 8086 registers as the variables in our programs. Registers have predefined names and do not need to be declared. We have seen that the 8086 has 14 registers. Initially, we will use four of them – the so called the general purpose registers:

ax, bx, cx, dx
These four 16-bit registers can also be treated as eight 8-bit registers: **ah, al, bh, bl, ch, cl, dh, dl**

Assignment
In C, assignment takes the form:
x = 42 ;
y = 24;
z = x + y;
In assembly language we carry out the same operaion but we use an instruction to denote the assignment operator (“=” in C).
mov x, 42
mov y, 24
add z, x
add z, y

The **mov** instruction carries out assignment in 8086 assembly language.It which allows us place a number in a register or in a memory location (a variable) i.e. it assigns a value to a register or variable.

Example: Store the ASCII code for the letter A in register bx.
A has ASCII code 65D (01000001B, 41H)
The following mov instruction carries out the task:
mov bx, 65d
We could also write it as:
mov bx, 41h
or mov bx, 01000001b
or mov bx, ‘A’

All of the above are equivalent. They each carry out exactly the same task, namely the binary number representing the ASCII code of A is copied into the bx register.

Control Flow: Jump Instructions

Unconditional Jump Instruction

The 8086 unconditional **jmp** instruction causes control flow (i.e. which instruction is next executed) to transfer to the point indicated by the label given in the jmp instruction.

Example: This example illustrates the use of the **jmp** instruction to implement an endless loop – not something you would noramlly wish to do!

again:
Mov dx,06h ; read a character
Add bx,09h ; display character
jmp again ; jump to again

This is an example of a backward jump as control is transferred to an earlier place in the program. The code fragment causes the instructions between the label again and the jmp instruction to be repeated endlessly.

Conditional Jump Instructions

The 8086 provides a number of conditional jump instructions (e.g. je, jne, ja). These instructions will only cause a transfer of control if some condition is satisfied. For example, when an arithmetic operation such as add or subtract is carried out, the CPU sets or clears a flag (Z-flag) in the status register to record if the result of the operation was zero, or another flag if the result was negative and so on.

If the Z-flag has value 1, it means that the result of the last instruction which affected the Z-flag was 0. If the Z-flag has value 0, it means that the result of the last instruction which affected the Z-flag was not 0.

By testing these flags, either individually or a combination of them, the conditional jump instructions can handle the various conditions (==, !=, <, >, <=, >=) that arise when comparing values. In addition, there are conditional jump instructions to test for conditions such as the occurrence of overflow or a change of sign. The conditional jump instructions are sometimes called jump-on-condition instructions. They test the values of the flags in the status register. (The value of the cx register is used by some of them).

One conditional jump is the jz instruction which jumps to another location in a program just like the jmp instruction except that it only causes a jump if the Z-flag is set to 1, i.e. if the result of the last instruction was 0. (The jz instruction may be understood as standing for 'jump on condition zero' or 'jump on zero').

Example : Using the jz instruction.

```
mov ax, 2 ; ax = 2
sub ax, bx ; ax = 2 - bx
jz nextl ; jump if (ax-bx) == 0
```

```
inc ax ; ax = ax + 1
nextl:
inc bx
```

The above is equivalent to:

```
ax = 2;
if ( ax != bx )
{
ax = ax + 1 ;
}
bx = bx + 1 ;
```

Example 4

Program to display from A to Z

```
.MODEL SMALL
.STACK
.DATA
VAL DB 'A'
.CODE
;-------------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX
MOV CX,26
MOV DL,VAL
TOP:
MOV AH,02H
INT 21H  ; CALL DOS INTERRUPT FOR DISPLAY
INC DL ; INCREASE DL BY 1
PUSH DX ;PUSH DX TO STACK
; ; SPACE
MOV DL,' '
MOV AH,02H
INT 21H
;;;CODE FOR SPACE ENDS

POP DX  ; POP CONTENT OF DX FROM STACK
DEC CX  ; DECREASE CX BY 1
JZ LAST  ;JUMP TO LABEL LAST IF CX=0
JMP TOP ;JUMP TO LABEL TOP

LAST:
MOV  AH,4CH ; EXIT DOS INTERRUPT
```

```
INT 21H
MAIN ENDP
END MAIN
```

Example 5

Program to display sum of Natural numbers (1+2+…….+10)

```
.MODEL SMALL
.STACK
.DATA
VAL DB 1
.CODE
;-----------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX
MOV CX,10
MOV DL,0

UP:
ADD DL,AL ; DL=DL+AL
INC AL  ;INCREASE AL CONTENT
DEC CX  ;DECREASE CX CONTENT
JZ DOWN ; JUMP TO DOWN IF CX =0
JMP UP  ; UNCONDITIONAL JUMP TO UP.

DOWN:
MOV AL,DL  ;FINAL SUM IS PASSED TO AL SINCE AAM WORKS WITH AX REGISTER
AAM
;;;Display the content
ADD AX,3030H
MOV DL,AH
MOV DH,AL
MOV AH,02H
INT 21H
MOV DL,DH
MOV AH,02H
INT 21H
;;;;END DISPLAY
MOV  AH,4CH ; EXIT DOS INTERRUPT
INT 21H
MAIN ENDP
END MAIN
```

Looping

There are several ways to loop, but we will discuss the easiest way to do it. You can easily loop with LOOP instruction. This instruction uses one operand, the memory location to loop to. It also uses the CX register as a counter. Loop simply decreases CX and checks if CX!= 0 (NOT EQUALS TO), if so, a Jump to the specified memory location is issued. Example:

```
MOV CX,100
UP:
INC AX
LOOP UP
```

This will increase AX 100 times. There are two other types of Loops: LOOPZ / LOOPNZ
Sometimes these instruction are also called: LOOPE / LOOPNE

LOOPZ works like LOOP except that it only loops when the zero flag is set, LOOPNZ only loops when the zero flag is NOT set. Now before I can give an example you need to know how to compare. The CMP instruction is used for this. It compares the two operands given and sets/clears the appropriate flags. After a CMP conditional instructions can be used to act on the result of the compare. For example jump to a special routine when two registers have the same value.

Example:

```
MOV CX,10
UP:
DEC AX
CMP AX,3
LOOPNE UP
```

This code might look like you'll never use it, but in some programs it can be very useful. It decreases AX ten times, but when AX == 3 it stops. Note I used LOOPNE, but LOOPNZ is the same. Now let's look at the CMP a little closer. In fact it does SUB AX,3 but doesn't store the result in AX, just alters the flags. So if AX == 3 the result of the SUB will be 0 and the Zero flag will be set. For the CPU equal is the same as zero (with conditionals) and it will always set the zero flag when the result of a mathematical operation is zero. So if we wanted to stop when AX == 0, there's no need to do a CMP. Just DEC AX and LOOPNZ.

(Note: Don't use LOOP(N)Z, nor LOOP. These instructions are slow, and decrease performance. Instead of LOOP you better use the combination: DEC CX / JNZ UP when necessary)

EXAMPLE 6

Write an 8086 ALP to exchange the contents of the memory location 10400h and 10500h. Assume the word count is stored in count register.

```
.MODEL SMALL
.STACK
.DATA
VAL DB 1
.CODE
;------------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX
XOR AX,AX
MOV AX,1000H
XCHG DS,AX
MOV CX,100
MOV SI,0400H
MOV DI,0500H
UP: MOV AX,[SI]
XCHG [DI],AX
MOV [SI],AX
INC SI
INC SI
INC DI
INC DI
LOOP UP
END
```

Example 7

Program to print all ASCII character.

```
.MODEL SMALL
.STACK 100H
.DATA
   PROMPT  DB  'The 256 ASCII Characters are : $'
.CODE
  MAIN PROC
  MOV AX, @DATA                ; initialize DS
  MOV DS, AX
```

```
  LEA DX, PROMPT              ; load and print PROMPT
  MOV AH, 9
  INT 21H
  MOV CX, 256                ; initialize CX
  MOV AH, 2                  ; set output function
  MOV DL, 0                  ; initialize DL with 0
    REDO:                    ; loop label
      INT 21H                ; print ASCII character
      INC DL                 ; increment DL to next ASCII character
    LOOP  REDO               ; loop's condition

    MOV AH, 4CH                ; return control to DOS
    INT 21H
  MAIN ENDP
 END MAIN
```

EXAMPLE 8

PROGRAM THAT TAKES INPUT STRING FROM KEYBOARD & PRINT IT in Assembly Language

```
.MODEL SMALL
.STACK
.DATA
   MSG DB 80
          DB 0
          DB 80 DUP('$')
.CODE
;-------------------------------------------------------------
MAIN PROC
MOV AX,@DATA
MOV DS, AX

   MOV AH,0AH
   MOV AX,OFFSET MSG
   INT 21H

   MOV AH,09H
   MOV DX,OFFSET MSG+2
   INT 21H
   MOV AH,4CH
   INT 21H
MAIN  ENDP
```

END MAIN

SOME COMMON DOS FUNCTIONS

Software interrupts are used by programs to request system services. A software interrupt occurs when a program calls an interrupt routine using the INT instruction.The format of the INT instruction is
INT interrupt_number

The 8086 treats this interrupt number in the same way as the interrupt number generated by a hardware device. We have already seen a number of examples of this using **INT 21h** and **INT 10h**
We will first describe dos interrupt .(**INT 21 H**)

Function **01**- Character input with echo
Action: Reads a character from the standard input device and echoes it to the standard output device.
If no character is ready it waits until one is available.
I/O can be re-directed, but prevents detection of OEF.
On entry: AH = 01h
Returns: AL = 8 bit data input

Function **02** - Character output
Action: Outputs a character to the standard output device. I/O can be re-directed, but prevents detection of 'disc full'.
On entry: AH = 02h
DL = 8 bit data (usually ASCII character)
Returns: Nothing

Function **09**- Output character string
Action: Writes a string to the display.
On entry: AH = 09h
DS:DX = segment:offset of string
Returns: Nothing
Notes: The string must be terminated by the $ character (24h), which is not transmitted. Any ASCII codes can be embedded within the string.

Function **0Ah** - Buffered input
Action: Reads a string from the current input device up to and including an ASCII carriage return (0Dh), placing the received data in a user-defined buffer Input can be re directed, but this prevents detection of EOF
On entry: AH = 0Ah
DS:DX = segment:offset of string buffer

Returns:	Nothing
Notes:	The first byte of the buffer specifies the maximum number of characters it can hold (1 to 255). This value must be supplied by the user. The second byte of the buffer is set by DOS to the number of characters actually read, excluding the terminating RETURN. If the buffer fills to one less than its maximum size the bell is sounded and subsequent input is ignored.

Function **4Ch** - Terminate program with return code
Action:	Terminates execution of a program with return to COMMAND.COM or a calling routine, passing back a return code. Allocated memory is freed, vectors for interrupts 22h to 24h are restored from the PSP and all file buffers are flushed to media. All files are closed and directories are updated.
On entry:	AH = 4Ch
AL = Return code (Error level)
Returns:	Nothing
Notes: This is the approved method of terminating program execution. It is the only way that does not rely on the contents of any segment register and is thus the simplest exit, particularly for EXE files.

BIOS Interrupt (INT 10h)
This corresponds to the BIOS interrupt call for video services. Such services include setting the video mode, character and string output, and graphics primitives (reading and writing pixels in graphics mode). To use this call, load AH with the subfunction you want to use, load other parameters in the other registers, and make the call. INT 10h is fairly slow, so many programs bypass this BIOS routine and access the display hardware directly. Setting the video mode, which is done infrequently, can be accomplished by using the BIOS, while drawing graphics on the screen in a game needs to be done quickly, so direct access to video RAM is more appropriate than making a BIOS call for every pixel.

BIOS Video Functions (Partial List)

AH	**Input Parameters**	**Output Parameters**	**Description**
0	al=mode		Sets the video display mode.
1	ch- Starting line. cl- ending line		Sets the shape of the cursor. Line values are in the range 0..15. You can make the cursor disappear by loading ch with 20h.

2	`bh`- page `dh`- y coordinate `dl`- x coordinate		Position cursor to location (x,y) on the screen. Generally you would specify page zero. BIOS maintains a separate cursor for each page.
3	`bh`- page	`ch`- starting line `cl`- ending line `dl`- x coordinate `dh`- y coordinate	Get cursor position and shape.
4			Obsolete (Get Light Pen Position).
5	`al`- display page		Set display page. Switches the text display page to the specified page number. Page zero is the standard text page. Most color adapters support up to eight text pages (0..7).
6	`al`- Number of lines to scroll. `bh`- Screen attribute for cleared area. `cl`- x coordinate UL `ch`- y coordinate UL `dl`- x coordinate LR `dh`- y coordinate LR		Clear or scroll up. If `al` contains zero, this function clears the rectangular portion of the screen specified by `cl/ch` (the upper left hand corner) and `dl/dh` (the lower right hand corner). If `al` contains any other value, this service will scroll that rectangular window up the number of lines specified in `al`.
7	`al`- Number of lines to scroll. `bh`- Screen		Clear or scroll down. If `al` contains zero, this function clears the rectangular portion of the screen specified by `cl/ch` (the upper left hand corner) and `dl/dh` (the lower right hand corner). If `al` contains any other value, this

	attribute for cleared area. `cl`- x coordinate UL `ch`- y coordinate UL `dl`- x coordinate LR `dh`- y coordinate LR		service will scroll that rectangular window down the number of lines specified in `al`.
8	`bh`- display page	`al`- char read `ah`- char attribute	Read character's ASCII code and attribute byte from current screen position.
9	`al`- character `bh`- page `bl`- attribute `cx`- # of times to replicate character		This call writes cx copies of the character and attribute in `al/bl` starting at the current cursor position on the screen. It does not change the cursor's position.
0Ah	`al`- character `bh`- page		Writes character in al to the current screen position using the existing attribute. Does not change cursor position.
0Bh	`bh`- 0 `bl`- color		Sets the border color for the text display.
0Eh	`al`- character `bh`- page		Write a character to the screen. Uses existing attribute and repositions cursor after write.
0Fh		`ah`- # columns `al`- display mode	Get video mode

		bh- page	