PREFACE

In this course, we shall deal with various aspects of assembly language. This SLM designed for those who want to learn the basics of assembly language programming .A basic understanding of any of the programming language will help you in understanding the assembly language programming concepts and faster on the learning.

The block consists of the following units:

- CPU organization Its Operation
- Introduction to Assembly Language

The first this unit might have studied various concepts of CPU Organization, Fetch-Execution cycle, Representing Programs, Read-Write timing diagram etc. Some details of general BUS operations are also discussed.

The second unit, we discuss about the 8086 microprocessor. We have discussed about the register set, instruction set and addressing modes for this microprocessor. In this unit, we will discuss the importance of assembly language, basic components of assembly program followed by the discussions on the program developmental tools available. We will then discuss about what has a microprocessor that manages the computer's arithmetical, logical, and control activities

CONTENTS

UNIT I: CPU ORGANISATION ITS OPERATION

Structure

1.1 Introduction

All the student reading this unit might have studied various concepts of CPU Organization, Fetch-Execution cycle, Representing Programs, Read-Write timing diagram etc. Some details of general BUS operations are also discussed.

1.2 Objectives

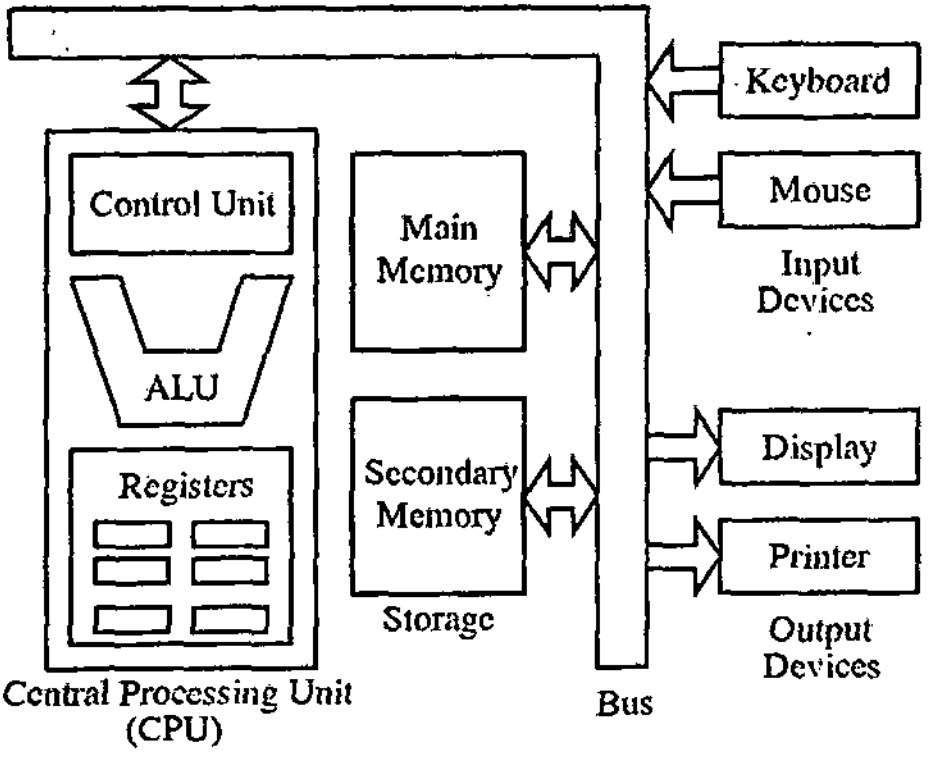
After going through this unit, you should be able to.

1. Various concepts of CPU Organization, Fetch-Execution cycle, Representing Programs, Read-Write timing diagram etc.
2. Some details of general BUS operations are also discussed

1.3 Von Neumann Architecture

The idea about how computers should be built was proposed by John von Neumann in 1945. This idea is called the von Neumann Architecture or Model. This is still the basis for computers today. Using these four components, a von Neumann computer will execute a series of instructions, called a program, which are stored in the computer's memory. This is called the "stored program concept". The components of von Neumann Architecture is:

1. Input/Output (I/O)
2. Memory
3. A Control Unit
4. An Arithmetic Logic Unit (ALU)
5. Input/Output (I/O) Devices



Von Neumann Architecture

The Input/Output (I/O) : components of a computer are hardware devices that are responsible for getting data from the computer to the user or from the user to the computer. Data going from the user to the computer is called "input." The two main input devices are the mouse and the keyboard. Output devices are used to transmit data from the computer's memory to the user. The two output devices almost every computer system has are the monitor and the printer.

Memory Unit: Computer has several types of memory.Memory unit in the Von Neumann model is the main memory, also called RAM or Random Access Memory. Main memory is used by the computer for storing a program and its data while the program is running. What distinguishes a computer from a calculator is the ability to run a stored program; main memory allows the computer to do that. RAM can be thought of as a sequence of boxes, called cells, each of which can hold a certain amount of data. The remaining three components of the von Neumann model of a computer are found inside the Processor.

Control Unit :The control unit controls the sequencing and timing of all operations. It contains a "clock," that is actually a quartz crystal that vibrates million times per second. The clock emits an electronic signal for each vibration. Each separate operation is synchronized to the clock signal. For example 1st pc operates at 4.7 MHz means 4.7 million instructions per second. The functions of CU are given below: Interprets and carries out instruction of program.

- Selects program statements from memory.
- Moves these instructions to instruction registers
- Carries out instructions
- Directs flow of data between components of CPU and to and from other devices.

Arithmetic & Logic Unit (ALU) : Arithmetic unit perform arithmetical operations like +, -, *, and / while logical unit are to compare two quantities. Logical operations are important in computer programming. ALU can be thought of as being similar to a calculator, except that, in addition to normal math, it can also do logical (true/false) operations. The functions of ALU are given below:

The arithmetic unit carries out arithmetic like addition, division.

The logic unit enables the processor to make comparison like =, and logical decisions like AND, OR, NOT.

The arithmetic logic unit carries out communication with peripheral devices. It also carries out bit shifting operation.

Register: Registers are Immediate Access Store (IAS) located on the CPU, and used temporarily for storing data. Because the registers are close to the ALU, they are made out of fast memory, efficiently speeding up calculations. There are 14 registers. Some examples are

a) MAR (Memory Address Register) holds the memory addresses of data and instructions.

b) Program Counter keeps track of the next memory address of the instruction that is to be executed once the execution of the current instruction is completed.

c) Accumulator Register is used for storing the Results those are produced by the System.

d) Memory Data Register (MDR) is the register of a computer's control unit that contains the data to be stored in the computer storage (e.g. RAM), or the data after a

fetch from the computer storage. It acts like a buffer and holds anything that is copied from the memory ready for the processor to use it. e) Data Register is used in microcomputers to temporarily store data being transmitted to or from a peripheral device.

Buses: "The set of wires used to travel signals to and from CPU and different components of computer is called Bus." Bus is a group of parallel wires that is used as a communication path. As a wire transmits a single bit so 8-bits bus can transfer 8 bits (1 byte) at a time and 16-bits bus can transfer 16 bits (2 bytes) and so on. There are three types of buses according to three types of signals, these are:

a) Data Bus: "The buses which are used to transmit data between CPU, memory and peripherals are called Data Bus."

b) Address Bus: "The buses which are connecting the CPU with main memory and used to identify particular locations (address) in main memory where data is stored are called Address Buses."

1.4 The Fetch-Execute Cycle

The operation of the CPU is usually described in terms of the Fetch-Execute cycle.

Fetch-Execute Cycle	**The cycle raises many interesting questions, e.g.**
Fetch the *Instruction*	What is an Instruction? Where is the Instruction? Why does it need to be fetched? Isn't it okay where it is? How does the computer keep track of instructions? Where does it put the instruction it has just fetched?
Increment the *Program Counter*	What is the Program Counter? What does the Program Counter count? Increment by how much? Where does the Program Counter point to after it is incremented?
Decode the Instruction	Why does the instruction need to be decoded? How does it get decoded?
Fetch the *Operands*	What are operands? What does it mean to fetch? Is this fetching distinct from the fetching in Step 1 above? Where are the operands? How many are there? Where do we put the operands after we fetch them?
Perform the Operation	Is this the main step? Couldn't the computer simply have done this part? What part of the CPU performs this operation?
Store the results	What results? Where from? Where to?
Repeat forever	Repeat what? Repeat from where? Is it really an infinite loop? Why? How do these steps execute any instructions at all?

In order to appreciate the operation of a computer we need to answer such questions and to consider in more detail the organisation of the CPU.

1.4 Representing Programs

Each complex task carried out by a computer needs to be broken down into a sequence of simpler tasks and a **binary machine instruction** is needed for the most primitive tasks. Consider a task that adds two numbers, held in memory locations designated by B and C and stores the result in memory location designated by A.

$$A = B + C$$

This assignment can be broken down (compiled) into a sequence of simpler tasks or **assembly instructions**, e.g:

Assembly Instruction	Effect
LOAD R2, B	Copy the contents of memory location designated by B into Register 2
ADD R2, C	Add the contents of the memory location designated by C to the contents of Register 2 and put the result back into Register 2
STORE R2, A	Copy the contents of Register 2 into the memory location designated by A.

Each of these assembly instructions needs to be encoded into binary for execution by the Central Processing Unit (CPU).

1.5 CPU Organisation

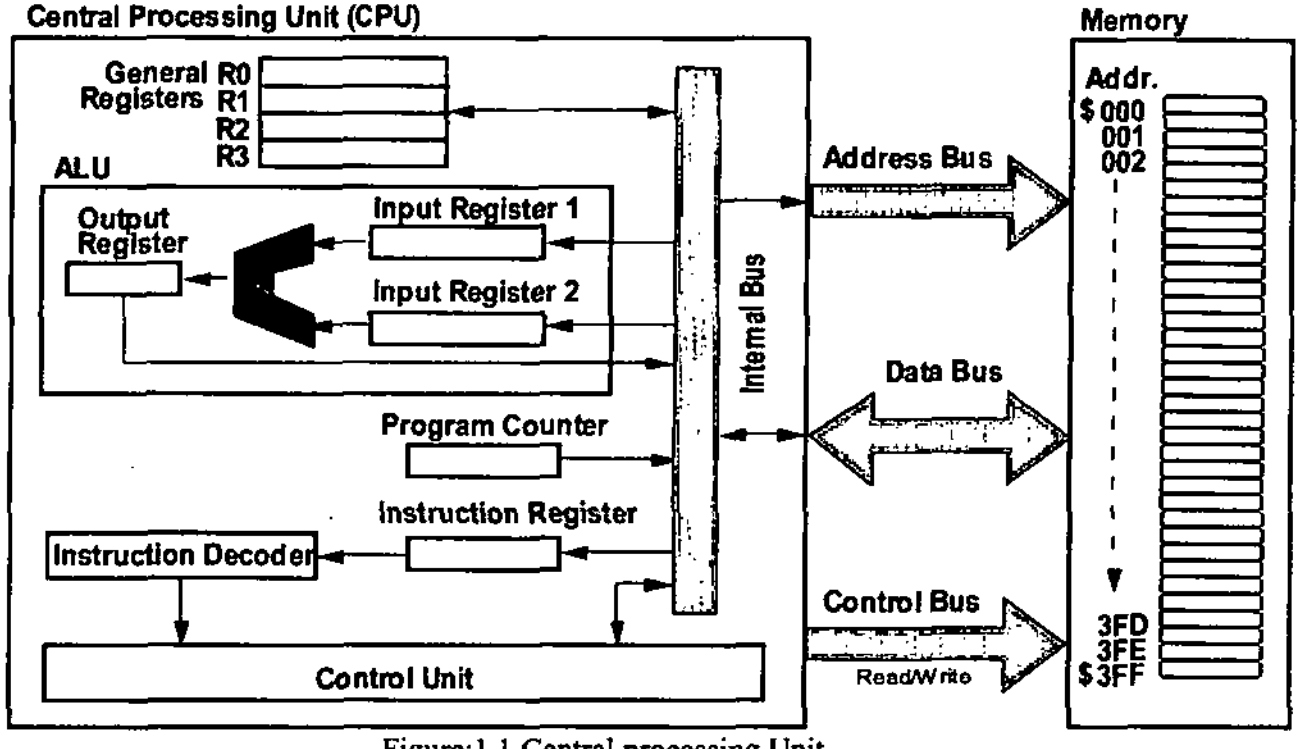


Figure:1.1 Central processing Unit

The **Program Counter (PC)** is a special register that holds the **address** of the next instruction to be fetched from Memory (for TOY1, the PC is 10-bits wide). The PC is *incremented to "point to" the next instruction while an instruction is being fetched from main memory.*

The **Instruction Register (IR)** is a special register that holds each instruction after it is fetched from main memory. For TOY1, the IR is 16-bits since instructions are 16-bit wide.

The **Instruction Decoder** is a CPU component that decodes and interprets the contents of the Instruction Register, i.e. its splits whole instruction into fields for the Control Unit to interpret. The Instruction decoder is often considered to be a part of the Control Unit.

The **Control Unit** is the CPU component that co-ordinates all activity within the CPU. It has connections to all parts of the CPU, and includes a sophisticated timing circuit.

The **Arithmetic & Logic Unit (ALU)** is the CPU component that carries out arithmetic and logical operations e.g. addition, comparison, Boolean AND/OR/NOT.

The **ALU Input Registers 1 & 2** are special registers that hold the input operands for the ALU.

The **ALU Output Register** is a special register that holds the result of an ALU operation. On completion of an ALU operation, the result is copied from the ALU Output register to its final destination, e.g. to a CPU register, or main-memory, or to an I/O device.

1.6 Read Write Timing Diagram

General Bus Operation

The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed using a few latches and transreceivers, whenever required.

Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, T4. The address is transmitted by the processor during T1, It is present on the bus only for one cycle. The negative edge of this ALE pulse is used to separate the address and the data or status information.

In maximum mode, the status lines S0, S1 and S2 are used to indicate the type of operation. Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.

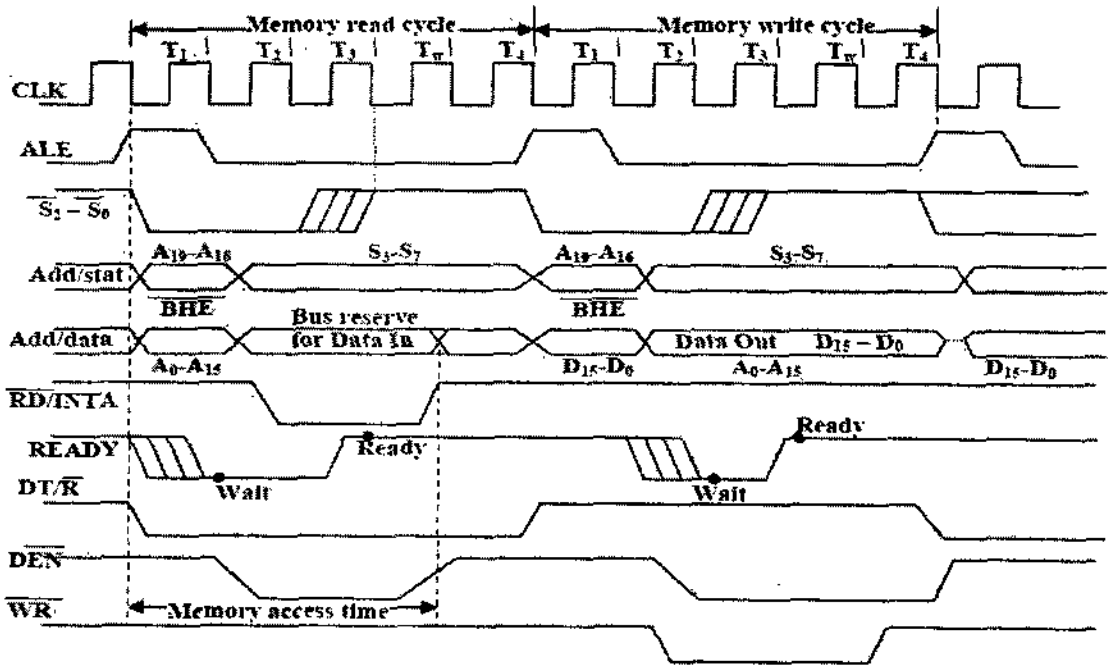


Fig. 1.2 General Bus Operation Cycle of 8086

Maximum Mode

i. In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.

ii. In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information.

iii. In the maximum mode, there may be more than onemicroprocessor in the system configuration.

Minimum Mode

i. In a minimum mode 8086 system, the microprocessor 8086is operated in minimum mode by strapping its MN/MX pin to logic 1.

ii. In this mode, all the control signals are given out by the microprocessor chip itself.

1.1 Summary

Computer Organization refers to the level of abstraction above the digital logic level, but below the operating system level. At this level, the major components are functional units or subsystems that correspond to specific pieces of hardware built from the lower level building blocks. This SLM gives a primarily understanding on Computer Organization starting from basic computer overview till its advanced architecture.

Exercise Questions

1. What is the role of the control unit?
2. Define Von Neumann Architecture **with diagram.**
3. What defines a CPU architecture? Describe one alternative architecture classification.
4. Describe each of the architecture classifications based on instruction operands.
5. Design an instruction code for instructions other than load and store in load-store architecture with a maximum of three operands per instruction. The architecture has 64 general purpose registers, and 40 instructions not counting load and store instructions, and uses byte-addressable RAM.
6. Design an instruction code for a memory-to-memory architecture with three operands per instruction (two sources and one destination). One possible assembly language instruction would be

Explain the difference between a hardwired and a micro-programmed control unit.
Describe each of the 4 phases of the Basic Computer instruction cycle.

References

- Carl Hammacher,"Computer Organization,"Fifth Edition,McGrawHill International Edition,2002
- P.Pal Chaudhuri,"Compter Organization and Design", 2nd Edition ,PHI,2003
- William Stallings,"Computer organization and ArchitectureDesigning for Performance",PHI,2004

UNIT II: INTRODUCTION TO ASSEMBLY LANGUAGE

Structure

2.1 INTRODUCTION

In the previous unit, we have discussed about the 8086 microprocessor. We have discussed about the register set, instruction set and addressing modes for this microprocessor. In this unit, we will discuss the importance of assembly language, basic components of assembly program followed by the discussions on the program developmental tools available. We will then discuss about what has a microprocessor that manages the computer's arithmetical, logical, and control activities.

2.2 OBJECTIVES

1. Define the need and importance of an assembly program.

2. Define the various directives used in assembly program.

3. Write a very simple assembly program with simple input –output services.

4. Define Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.

2.3 COMPUTER LANGUAGES

In order for a computer to be able to execute a program, the program must first be present in binary form within the computer's memory. However, writing a program directly in binary form is completely out of the question. The solution to this conflict is for the programmer to write a program in a more suitable language and then to use a special computer program to translate it into the binary machine code necessary for the target computer.

When preparing software for a microprocessor system, the designer must first select the particular language in which to write the programs. Three classes of computer languages exist: machine languages, assembly languages, and higher-level languages. Of these, only the last two are normally used by a programmer in preparing programs.

2.3.1 Machine Language

A computer is a binary machine. The signals used by the various parts of the machine to communicate with each other must be in binary form. A program shown in binary form (or in hex for ease of reading) so as to reflect these values is said to be in *machine language*. Machine language is the most basic and elementary form in which any program can exist.

2.3.2 Assembly Language

Before it can be executed a program must reside in memory in machine language.

One step away from using machine language would be to write the program using mnemonics rather than op-codes for the instructions.

A major improvement would be to include some convenient way to keep track of these elements. *Assembly language* provides this by allowing the user to refer to numerical addresses and constants by name, that is, to use mnemonics for addresses and data as well as for instructions.

Since every program must eventually be in machine language there must always be two versions of each program. *The original version prepared by the programmer is called the source program.* The machine language version which must be loaded into the computer memory is called the *object program.*

Table 2.1 the source program and part *b* is the object program. Observe the

word/symbol/mnemonic orientation of the source code as opposed to the strictly binary form of the object code (shown here in hex).

(a) source program		(b) object program	
ORG	5000H	*Address*	*Content*
LDB	#SIZE	5000	C614
LDX	#VEC1	5002	8E1000
LDY	#VEC2	5005	108E2000
LDS	#VEC3	5009	10CE3000
LOOP LDA	,X+	500D	A680
ADDA	,Y+	500F	ABAO
STA	,S+	5011	A7EO
DECB		5013	5A
BNE	LOOP	5014	26F7
SWI		5016	3F
FCB	8	5017	08
SIZE EQU	20		
VEC1 EQU	1000H		
VEC2 EQU	2000H		
VEC3 EQU	3000H		
END			

Table 2.1

The conversion from the source program into the object program is the sort of a program, which translates the source into the machine code object program. If the source program is written in some higher-level language such as FORTRAN or Pascal then the translation program is called a *compiler*. If it is written in assembly language then the translation program is called an *assembler*.

2.3.3 Assembly Language versus Higher-level Languages

Most programmers are capable of writing a relatively constant number of program statements per day regardless of the programming language used. An assembly language program requires on the order of ten times as many statements as an equivalent higher-level language program. However, assembly language may be a suitable choice in some cases.

A good assembly language: (i) occupies less space in memory, (ii) generally run faster than their higher-level counterparts.

Even in applications where a higher-level language is used for the bulk of the system programs, certain segments of the code may be written in assembly language. In many simple control applications it is not unusual for all of the programming to be done in assembly

language.

2.3.4 Characteristics of Assembly Languages

Although assembly languages are processor-dependent they do have some common characteristics. All assembly languages use mnemonics to specify the processor instructions and certain supporting operations required of the assembler itself.

All assembly languages include a set of directives, which are used by the programmer to issue commands to the assembler itself. These are called *pseudo-instructions* since they resemble instructions in format although they are not instructions. They are also known as *assembler directives,* since they are commands or directives issued to the assembler. For example, the *origin* directive, whose mnemonic is (universally) ORG, may be used to specify the origin or starting address of a section of the program.

All assemblers provide for the use of *symbolic labels* which may be affixed to constants and memory addresses. These labels may be used as substitutes for the numbers with which they are identified. The use of labels rather than absolute numbers is a powerful programming tool.

A branch target address, too, is easier to implement in the program when the programmer need not be concerned with its actual value or its distance but may refer to it by name.

Constants may be specified in one of several different bases with decimal, hexadecimal, octal, and binary being common options.

The set of rules regarding the proper form for statements in the assembly language is called the *syntax* of the assembler. The syntax for most assemblers divides each statement into four fields: from left to right, the *label, mnemonic* or *operation, operand,* and *comment* fields.

An assembler scans the source program twice while translating it into machine code. During the first pass it counts bytes and locates all of the label definitions, which it places into a table along with the corresponding numerical values. During the second pass it uses this label table or *symbol table* to generate the machine code. This feature is particularly important in branches where the target address may be defined by being a label on a later instruction.

2.4 ASSEMBLY LANGUAGES

2.4.1 The Character Set

The following characters are recognized by the assemblers:

1. The alphabet A through Z
2. The integers 0 through 9
3. Four arithmetic operators: + — * /
4. Characters used as special prefixes:

# specifies the immediate addressing mode
$ specifies a hexadecimal number
% specifies a binary number
@ specifies an octal number

5. Characters used as delimiters:
   a. * or ; in column 1 specifies a comment line
   b. * or ; or space separates comment from operand
   c. a space separates operation from label and operand from operation; this is also the standard delimiter between operand and comment in Motorola assemblers
   d. Comma separates multiple operands
6. Characters with special meanings in certain places:
   a. * in operand stands for present location in memory
   b. " in operand stands for beginning and end of a string of ASCII characters
   c. Letters used as suffixes in numbers:
      i. B denotes a binary number
      ii. D denotes a decimal number
      iii. H denotes a hexadecimal number
      iv. O or Q denotes an octal number
7. A comment may include any printable character.

2.4.2 Program Statements

A source program is composed of a sequence of statements, one per line. These statements must be written into an ASCII file with an editor program.

A source statement includes between one and four fields. From left to right these are:

(1) label (2) operation (3) operand (4) comment

A ***label*** is required with some statements involved in the definitions of symbols; otherwise it is optional. The use of labels on branch destinations, although optional, will greatly reduce the programming effort and opportunity for error. The ***operation*** or mnemonic must be included in every statement except those, which are purely comment. An operand may or may not be required, depending upon the nature of the operation. The ***comment*** is always optional and may be included in any statement. Comments are intended for the convenience of the programmer and to facilitate proper documentation of the program.

The standard delimiter or field separator in Motorola assemblers is the space, however also allow the use of the asterisk or the semicolon to separate the operand from a comment.

```
        ORG     1000H   6809 Program Starts In Location 1000
START   LDA     FIRST     Pick Up First Operand
        ADDA    SCND      Add Second Operand
        STA     SUM       Store The Result
        SWI               Return To Monitor
        FCB     8         Monitor Code
```

```
        BRA     START   Start Again
FRST    FCB     34H     Data Area
SCND    FCB     56H
SUM     FCB     0
        END
```

2.4.3 The Label Field

Labels always start in the first column of the statement line. If a label is not used in a particular statement, that statement must begin with a space. Except when it is used with the assembler directives EQU or SET (described below) to define a constant, a label corresponds to a numerical address in the computer. The label provides a convenient means for the programmer to refer to that address within the operand field of other statements in the program. The programmer need only use the identical symbols.

The following rules apply to labels:

1. A label consists of one or more alphanumerical characters. ASSYM09 will accept a maximum of six characters, ASSYM000 will accept a maximum of 16 characters.
2. The first character of a label must be alphabetic.
3. A label must begin in the first column of the statement.
4. Each label used in a program must be unique.

Although the above rules are quite unrestrictive, it is good practice to follow the following "unwritten rules" in devising labels:

1. Use labels which are meaningful in the context of the program, i.e., SPEED rather than X7.
2. Avoid using a label, which duplicates any mnemonic of the assembly language. Some assemblers specifically exclude these as labels and their use can be confusing.
3. Do not use labels which are too similar to each other. Although the assembler can distinguish between PARTI and PART1 or between IIIIII and IIIII, a human reader will have difficulty.

2.4.4 The Operation Mnemonic Field

The operation mnemonics recognized by an assembler include the executable instructions of the respective processor and a set of assembler directives. Each instruction mnemonic, together with its addressing mode and operands, is translated by the assembler into the appropriate machine code. Certain of the assembler directives result directly in the generation of binary code as part of the object program. Others simply control the assembly process and do not call for any binary code.

2.4.5 Assembler Directives

The following list describes some of the more common Motorola assembler directives and includes an example of each.

ORG Defines **ORIGIN** of the code. Tells the assembler where in memory to start assembling code ORG 1000H *sets the starting address of the subsequent code to be 1000H*

EQU Permanently defines the value of the label to be **EQUAL** to the value of the operand. A label is required. CR EQU ODH *makes the label "CR" equal ODH*

EXTERN Imports the value of a label from another module so that the current module may reference it. The other module must also declare the label(s) as global (see GLB).

EXTERN TIME,SIZE *import the values of TIME and SIZE from another module*

GLB Makes a label available to be exported to another module, which will be linked with the current one before they are executed

GLB TIME,SIZE *declares that the labels TIME AND SIZE may be used in another module and that their values are defined in the current module*

END The last statement in the source program. It tells the assembler to start the assembly process.

ABS-SHORT Changes the *default* absolute addressing mode in the **ABS_LONG** MC68000 to either short (16 bits) or long (32 bits).

DC.X Define Constant. Stores the operand list in memory using one byte per operand if *X is* B, two bytes per operand if *X is* W, and four bytes per operand if *X* is L. May have a label. Operands may be expressed as an ASCII string enclosed within quotes.

DC.B $4C,100,"Do it." *stores the hex values 4C, 64, 44, 6F, 20, 69, 74, 2E in memory starting at the current address*

DS.X Define Storage. Reserves memory locations without placing any values in them.

ARRAY DS.L 15 *reserves 60 consecutive bytes (15 long words of four bytes each) starting at the current address which will be named ARRAY*

EVEN Adjusts the assembler's location counter to the next even address if it is not already even.

2.4.6 The Operand Field

The form of the operand will vary depending upon the operation mnemonic in the statement. The operand may convey addressing mode information; it may list data to be stored in memory locations by the assembler; it may provide an address for an EQU or an ORG directive; or it may not be present because it is neither required nor allowed.

When any part of the operand conveys numerical information, it may be included in any one of several alternative ways. Numbers may be included directly in one of the supported number systems; they may be expressed in terms of the location of the current instruction plus an offset; they may be instruction labels or other address or constant labels; or they may be arithmetic expressions involving any of these. The following examples illustrate some of the possibilities:

Direct Numerical

a) Decimal: 1234 or 1234D Decimal is the default number system. Numbers with no other prefix or suffix are always interpreted as decimal.

b) Hex: $1234 or 1234H Numbers must start with digits 0-9. A234H is invalid; it is interpreted as a label! Use $A234 or 0A234H.

c) Octal: 1234Q or @1234 or 1234O (letter O not digit 0)

d) Binary: 01001001B or %01001001

Offset from Current Location

*+15 This number is the address of the current instruction op-word plus 15. *-35 This number is the address of the current instruction op-code minus 35.

Constant Labels

HNDRD EQU 100 Defines the label HNDRD as 64 hex.
LDA #HNDRD The immediate operand in the instruction is evaluated as 64 hex.

Address Labels

ORG $567A Defines the origin.
CMD FCC "Do it!" Defines the label CMD as the value 567AH.
CLR CMD,Y The operand is evaluated as $567A,Y.

Arithmetic Expressions (using defined labels from above)

LDA #2*HNDRD Coded as LDA #200 or LDA #$C8
LDX CMD-$0A Coded as LDX $5670

2.5 Assembly - Addressing Modes

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored. Some instructions do not require an operand, whereas some other instructions may require one, two, or three operands.

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Source contains either the data to be delivered (immediate addressing) or the address (in register or memory) of the data. Generally, the source data remains unaltered after the operation.

The three basic modes of addressing are –

- Register addressing
- Immediate addressing
- Memory addressing

2.5.1 Register Addressing

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

For example,

```
MOV DX, TAX_RATE   ; Register in first operand
MOV COUNT, CX      ; Register in second operand
MOV EAX, EBX       ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

2.5.2 Immediate Addressing

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For example,

```
BYTE_VALUE DB 150    ; A byte value is defined
WORD_VALUE DW 300    ; A word value is defined
ADD BYTE_VALUE, 65   ; An immediate operand 65 is added
MOV AX, 45H          ; Immediate constant 45H is transferred to AX
```

2.5.3 Direct Memory Addressing

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called **effective address**.

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.

For example,

```
ADD    BYTE_VALUE, DL      ; Adds the register in the memory location
MOV    BX, WORD_VALUE      ; Operand from the memory is added to register
```

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data –

```
BYTE_TABLE DB  14, 15, 22, 45     ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123  ; Tables of words
```

The following operations access data from the tables in the memory into registers –

```
MOV CL, BYTE_TABLE[2]        ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2       ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3]        ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3       ; Gets the 4th element of the WORD_TABLE
```

Indirect Memory Addressing

This addressing mode utilizes the computer's ability of *Segment:Offset* addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

The following code snippet shows how to access different elements of the variable.

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

The MOV Instruction

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands.

Syntax

The syntax of the MOV instruction is –

```
MOV  destination, source
```

The MOV instruction may have one of the following five forms –

```
MOV  register, register
```

```
MOV register, immediate
MOV memory, immediate
MOV register, memory
MOV memory, register
```

Please note that –

- Both the operands in MOV operation should be of same size
- The value of source operand remains unchanged

The MOV instruction causes ambiguity at times. For example, look at the statements –

```
MOV EBX, [MY_TABLE]  ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110       ; MY_TABLE[0] = 110
```

It is not clear whether you want to move a byte equivalent or word equivalent of the number 110. In such cases, it is wise to use a **type specifier**.

Following table shows some of the common type specifiers –

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

2.6 Variables

NASM provides various **define directives** for reserving storage space for variables. The define assembler directive is used for allocation of storage space. It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

The syntax for storage allocation statement for initialized data is –

```
[variable-name]  define-directive  initial-value  [,initial-value]...
```

Where, *variable-name* is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.

There are five basic forms of the define directive –

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Following are some examples of using define directives –

```
choice      DB    'y'
number      DW    12345
neg_number  DW    -12345
big_number  DQ    123456789
real_number1  DD    1.234
real_number2  DQ    123.456
```

Please note that –

- Each byte of character is stored as its ASCII value in hexadecimal.
- Each decimal value is automatically converted to its 16-bit binary equivalent and stored as a hexadecimal number.
- Processor uses the little-endian byte ordering.

- Negative numbers are converted to its 2's complement representation.
- Short and long floating-point numbers are represented using 32 or 64 bits, respectively.

The following program shows the use of define directive –

```
section .text
  global _start        ;must be declared for linker (gcc)

_start:                ;tell linker entry point
  mov   edx,1                ;message length
  mov   ecx,choice     ;message to write
  mov   ebx,1                ;file descriptor (stdout)
  mov   eax,4                ;system call number (svs_write)
  int   0x80                 ;call kernel
  mov   eax,1                ;system call number (sys_exit)
  int   0x80                 ;call kernel
section .data
choice DB 'y'
```

When the above code is compiled and executed, it produces the following result –

```
y
```

Allocating Storage Space for Uninitialized Data

The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved. Each define directive has a related reserve directive.

There are five basic forms of the reserve directive –

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word

RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Multiple Definitions

You can have multiple data definition statements in a program. For example –

```
choice    DB     'Y'              ;ASCII of y = 79H
number1   DW     12345            ;12345D = 3039H
number2   DD  12345679            ;123456789D = 75BCD15H
```

The assembler allocates contiguous memory for multiple variable definitions.

Multiple Initializations

The TIMES directive allows multiple initializations to the same value. For example, an array named marks of size 9 can be defined and initialized to zero using the following statement –

```
marks TIMES 9 DW 0
```

The TIMES directive is useful in defining arrays and tables. The following program displays 9 asterisks on the screen –

```
section   .text
  global _start      ;must be declared for linker (ld)
_start:              ;tell linker entry point
  mov   edx,9                ;message length
  mov   ecx, stars ;message to write
  mov   ebx,1                ;file descriptor (stdout)
  mov   eax,4                ;system call number (sys_write)
  int   0x80                 ;call kernel
  mov   eax,1                ;system call number (sys_exit)
  int   0x80                 ;call kernel
section   .data
stars  times 9 db '*'
```

When the above code is compiled and executed, it produces the following result –

```
*********
```

2.7 Constants

There are several directives provided by NASM that define constants. We have already used the EQU directive in previous chapters. We will particularly discuss three directives –

- EQU
- %assign
- %define

The EQU Directive

The EQU directive is used for defining constants. The syntax of the EQU directive is as follows –

```
CONSTANT_NAME EQU expression
```

For example,

```
TOTAL_STUDENTS equ 50
```

You can then use this constant value in your code, like –

```
mov ecx, TOTAL_STUDENTS
cmp eax, TOTAL_STUDENTS
```

The operand of an EQU statement can be an expression –

```
LENGTH equ 20
WIDTH  equ 10
AREA   equ length * width
```

Above code segment would define AREA as 200.

Example

The following example illustrates the use of the EQU directive –

```
SYS_EXIT  equ 1
SYS_WRITE equ 4
STDIN     equ 0
STDOUT    equ 1
```

```
section    .text
   global _start    ;must be declared for using gcc

_start:            ;tell linker entry point
   mov eax, SYS_WRITE
   mov ebx, STDOUT
   mov ecx, msg1
   mov edx, len1
   int 0x80
   mov eax, SYS_WRITE
   mov ebx, STDOUT
   mov ecx, msg2
   mov edx, len2
   int 0x80
   mov eax, SYS_WRITE
   mov ebx, STDOUT
   mov ecx, msg3
   mov edx, len3
   int 0x80
   mov eax,SYS_EXIT    ;system call number (sys_exit)
   int 0x80            ;call kernel
section    .data
msg1 db 'Hello, programmers!',0xA,0xD
len1 equ $ - msg1
msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $ - msg2
msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

When the above code is compiled and executed, it produces the following result –

```
Hello, programmers!
Welcome to the world of,
Linux assembly programming!
```

The %assign Directive

The **%assign** directive can be used to define numeric constants like the EQU directive. This directive allows redefinition. For example, you may define the constant TOTAL as –

```
%assign TOTAL 10
```

Later in the code, you can redefine it as –

```
%assign  TOTAL  20
```

This directive is case-sensitive.

The %define Directive

The **%define** directive allows defining both numeric and string constants. This directive is similar to the #define in C. For example, you may define the constant PTR as –

```
%define PTR [EBP+4]
```

The above code replaces *PTR* by [EBP+4].

This directive also allows redefinition and it is case-sensitive

2.8 Arithmetic Instructions

The INC Instruction

The INC instruction is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.

Syntax

The INC instruction has the following syntax –

```
INC destination
```

The operand *destination* could be an 8-bit, 16-bit or 32-bit operand.

Example

```
INC EBX          ; Increments 32-bit register
INC DL     ; Increments 8-bit register
INC [count]  ; Increments the count variable
```

The DEC Instruction

The DEC instruction is used for decrementing an operand by one. It works on a single operand that can be either in a register or in memory.

Syntax

The DEC instruction has the following syntax –

```
DEC destination
```

The operand *destination* could be an 8-bit, 16-bit or 32-bit operand.

Example

```
segment .data
  count dw  0
  value db  15

segment .text
  inc [count]
  dec [value]

  mov ebx, count
  inc word [ebx]

  mov esi, value
  dec byte [esi]
```

The ADD and SUB Instructions

The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16-bit or 32-bit operands, respectively.

Syntax

The ADD and SUB instructions have the following syntax –

```
ADD/SUB        destination, source
```

The ADD/SUB instruction can take place between –

- Register to register
- Memory to register
- Register to memory
- Register to constant data
- Memory to constant data

However, like other instructions, memory-to-memory operations are not possible using ADD/SUB instructions. An ADD or SUB operation sets or clears the overflow and carry flags.

The MUL/IMUL Instruction

There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

Syntax

The syntax for the MUL/IMUL instructions is as follows –

MUL/IMUL multiplier

Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases –

SN	Scenarios
1	**When two bytes are multiplied -** The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High-order 8 bits of the product is stored in AH and the low-order 8 bits are stored in AL.
2	**When two one-word values are multiplied -** The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX. The resultant product is a doubleword, which will need two registers. The high-order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX.

3	**When two doubleword values are multiplied -** When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register. 

Example

```
MOV AL, 10
MOV DL, 25
MUL DL
...
MOV DL, 0FFH   ; DL= -1
MOV AL, 0BEH   ; AL = -66
IMUL DL
```

The DIV/IDIV Instructions

The division operation generates two elements - a **quotient** and a **remainder**. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.

The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.

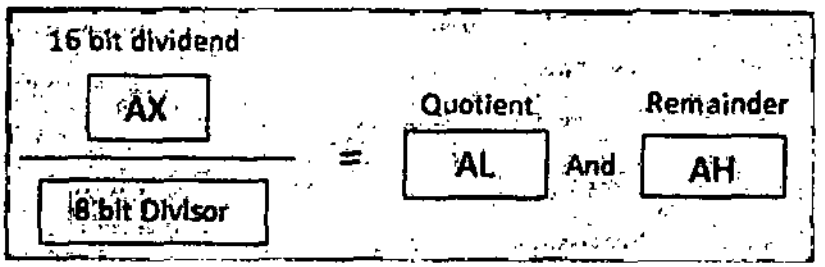
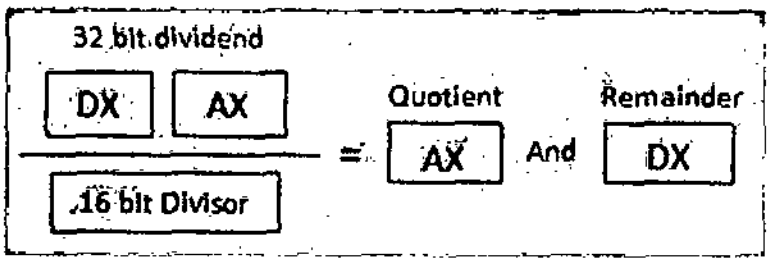
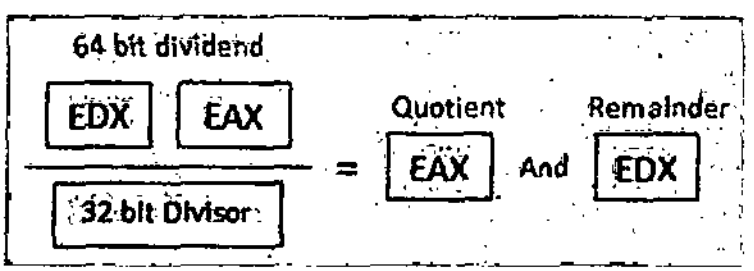
Syntax

The format for the DIV/IDIV instruction –

```
DIV/IDIV        divisor
```

The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size –

SN	Scenarios

1	**When the divisor is 1 byte -** The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register. 
2	**When the divisor is 1 word -** The dividend is assumed to be 32 bits long and in the DX:AX registers. The high-order 16 bits are in DX and the low-order 16 bits are in AX. After division, the 16-bit quotient goes to the AX register and the 16-bit remainder goes to the DX register. 
3	**When the divisor is double word -** The dividend is assumed to be 64 bits long and in the EDX:EAX registers. The high-order 32 bits are in EDX and the low-order 32 bits are in EAX. After division, the 32-bit quotient goes to the EAX register and the 32-bit remainder goes to the EDX register. 

Example

The following example divides 8 with 2. The **dividend 8** is stored in the **16-bit AX register** and the ***divisor 2*** *is stored in the* ***8-bit BL register***.

```
section   .text
  global _start   ;must be declared for using gcc

_start:           ;tell linker entry point
  mov    ax,'8'
  sub    ax, '0'

  mov    bl, '2'
  sub    bl, '0'
  div    bl
  add    ax, '0'

  mov    [res], ax
  mov    ecx,msg
  mov    edx, len
  mov    ebx,1     ;file descriptor (stdout)
  mov    eax,4     ;system call number (sys_write)
  int    0x80      ;call kernel

  mov    ecx,res
  mov    edx, 1
  mov    ebx,1     ;file descriptor (stdout)
  mov    eax,4     ;system call number (sys_write)
  int    0x80      ;call kernel

  mov    eax,1     ;system call number (sys_exit)
  int    0x80      ;call kernel

section .data
```

```
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

When the above code is compiled and executed, it produces the following result –

```
The result is:
4
```

2.9 Logical Instructions

The processor instruction set provides the instructions AND, OR, XOR, TEST, and NOT Boolean logic, which tests, sets, and clears the bits according to the need of the program.

The format for these instructions –

SN	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

The first operand in all the cases could be either in register or in memory. The second operand could be either in register/memory or an immediate (constant) value. However, memory-to-memory operations are not possible. These instructions compare or match bits of the operands and set the CF, OF, PF, SF and ZF flags.

The AND Instruction

The AND instruction is used for supporting logical expressions by performing bitwise AND operation. The bitwise AND operation returns 1, if the matching bits from both the operands are 1, otherwise it returns 0. For example –

```
        Operand1:          0101
        Operand2:          0011
-------------------------
After AND -> Operand1:     0001
```

The AND operation can be used for clearing one or more bits. For example, say the BL register contains 0011 1010. If you need to clear the high-order bits to zero, you AND it with 0FH.

```
AND     BL,  0FH  ; This sets BL to 0000 1010
```

Let's take up another example. If you want to check whether a given number is odd or even, a simple test would be to check the least significant bit of the number. If this is 1, the number is odd, else the number is even.

Assuming the number is in AL register, we can write –

```
AND     AL, 01H    ; ANDing with 0000 0001
JZ   EVEN_NUMBER
```

The following program illustrates this –

Example

```
section .text
  global _start         ;must be declared for using gcc

_start:                 ;tell linker entry point
  mov  ax,  8h          ;getting 8 in the ax
  and  ax, 1            ;and ax with 1
  jz   evnn
  mov  eax, 4           ;system call number (sys_write)
  mov  ebx, 1           ;file descriptor (stdout)
  mov  ecx, odd_msg     ;message to write
  mov  edx, len2        ;length of message
  int  0x80             ;call kernel
  jmp  outprog

evnn:
```

```
   mov  ah, 09h
   mov  eax, 4            ;system call number (sys_write)
   mov  ebx, 1            ;file descriptor (stdout)
   mov  ecx, even_msg     ;message to write
   mov  edx, len1         ;length of message
   int  0x80              ;call kernel

outprog:

   mov  eax,1             ;system call number (sys_exit)
   int  0x80              ;call kernel

section  .data
even_msg  db  'Even Number!' ;message showing even number
len1  equ  $ - even_msg

odd_msg db  'Odd Number!'   ;message showing odd number
len2  equ  $ - odd_msg
```

When the above code is compiled and executed, it produces the following result –

```
Even Number!
```

Change the value in the ax register with an odd digit, like –

```
mov  ax, 9h                ; getting 9 in the ax
```

The program would display:

```
Odd Number!
```

Similarly to clear the entire register you can AND it with 00H.

The OR Instruction

The OR instruction is used for supporting logical expression by performing bitwise OR operation. The bitwise OR operator returns 1, if the matching bits from either or both operands are one. It returns 0, if both the bits are zero.

For example,

```
Operand1:   0101
Operand2:   0011
-------------------
After OR -> Operand1:   0111
```

The OR operation can be used for setting one or more bits. For example, let us assume the AL register contains 0011 1010, you need to set the four low-order bits, you can OR it with a value 0000 1111, i.e., FH.

```
OR BL, 0FH          ; This sets BL to 0011 1111
```

Example

The following example demonstrates the OR instruction. Let us store the value 5 and 3 in the AL and the BL registers, respectively, then the instruction,

```
OR AL, BL
```

should store 7 in the AL register –

```
section .text
  global _start         ;must be declared for using gcc

_start:                 ;tell linker entry point
  mov   al, 5           ;getting 5 in the al
  mov   bl, 3           ;getting 3 in the bl
  or    al, bl          ;or al and bl registers, result should be 7
  add   al, byte '0'    ;converting decimal to ascii

  mov   [result], al
  mov   eax, 4
  mov   ebx, 1
  mov   ecx, result
  mov   edx, 1
  int   0x80

outprog:
  mov   eax,1           ;system call number (sys_exit)
  int   0x80            ;call kernel
```

```
section .bss
result resb 1
```

When the above code is compiled and executed, it produces the following result –

```
7
```

The XOR Instruction

The XOR instruction implements the bitwise XOR operation. The XOR operation sets the resultant bit to 1, if and only if the bits from the operands are different. If the bits from the operands are same (both 0 or both 1), the resultant bit is cleared to 0.

For example,

```
Operand1:   0101
Operand2:   0011
-------------------------
After XOR -> Operand1:   0110
```

XORing an operand with itself changes the operand to 0. This is used to clear a register.

```
XOR   EAX, EAX
```

The TEST Instruction

The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand. So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

```
TEST   AL, 01H
JZ   EVEN_NUMBER
```

The NOT Instruction

The NOT instruction implements the bitwise NOT operation. NOT operation reverses the bits in an operand. The operand could be either in a register or in the memory.

For example,

```
Operand1:   0101 0011
After NOT -> Operand1:   1010 1100
```

2.10 Condition

Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios –

SN	*Conditional Instructions*
1	**Unconditional jump** This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.
2	***Conditional jump*** This is performed by a set of jump instructions j<condition> depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

Let us discuss the CMP instruction before discussing the conditional instructions.

CMP Instruction

The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.

Syntax

CMP destination, source

CMP compares two numeric data fields. The destination operand could be either in register or in memory. The source operand could be a constant (immediate) data, register or memory.

Example

```
CMP DX,        00  ; Compare the DX value with zero
JE  L7     ; If yes, then jump to label L7
```

```
L7: ...
```

CMP is often used for comparing whether a counter value has reached the number of times a loop needs to be run. Consider the following typical condition –

```
INC     EDX
CMP     EDX, 10 ; Compares whether the counter has reached 10
JLE     LP1     ; If it is less than or equal to 10, then jump to LP1
```

Unconditional Jump

As mentioned earlier, this is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.

Syntax

The JMP instruction provides a label name where the flow of control is transferred immediately. The syntax of the JMP instruction is –

```
JMP     label
```

Example

The following code snippet illustrates the JMP instruction –

```
MOV  AX, 00   ; Initializing AX to 0
MOV  BX, 00   ; Initializing BX to 0
MOV  CX, 01   ; Initializing CX to 1
L20:
ADD  AX, 01   ; Increment AX
ADD  BX, AX   ; Add AX to BX
SHL  CX, 1    ; shift left CX, this in turn doubles the CX value
JMP  L20      ; repeats the statements
```

Conditional Jump

If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction. There are numerous conditional jump instructions depending upon the condition and data.

Following are the conditional jump instructions used on signed data used for arithmetic operations –

Instruction	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 or Jump Not Less	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

Following are the conditional jump instructions used on unsigned data used for logical operations –

Instruction	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

The following conditional jump instructions have special uses and check the value of flags –

Instruction	Description	Flags tested

JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

The syntax for the J<condition> set of instructions –

Example,

```
CMP     AL, BL
JE      EQUAL
CMP     AL, BH
JE      EQUAL
CMP     AL, CL
JE      EQUAL
NON_EQUAL: ...
EQUAL: ...
```

Example

The following program displays the largest of three variables. The variables are double-digit variables. The three variables num1, num2 and num3 have values 47, 72 and 31, respectively –

```
section   .text
  global _start       ;must be declared for using gcc

_start:               ;tell linker entry point
  mov   ecx, [num1]
  cmp   ecx, [num2]
  jg    check_third_num
  mov   ecx, [num3]

        check_third_num:

  cmp   ecx, [num3]
  jg    _exit
  mov   ecx, [num3]

        _exit:

  mov   [largest], ecx
  mov   ecx,msg
  mov   edx, len
  mov   ebx,1       ;file descriptor (stdout)
  mov   eax,4       ;system call number (sys_write)
  int   0x80        ;call kernel

  mov   ecx,largest
  mov   edx, 2
  mov   ebx,1       ;file descriptor (stdout)
  mov   eax,4       ;system call number (sys_write)
  int   0x80        ;call kernel

  mov   eax, 1
  int   80h
```

```
section   .data

  msg db "The largest digit is: ", 0xA,0xD
  len equ $- msg
  num1 dd '47'
  num2 dd '22'
  num3 dd '31'

segment .bss
  largest resb 2
```

When the above code is compiled and executed, it produces the following result –

```
The largest digit is:
47.
```

2.11 Loops

The JMP instruction can be used for implementing loops. For example, the following code snippet can be used for executing the loop-body 10 times.

```
MOV    CL, 10
L1:
<LOOP-BODY>
DEC    CL
JNZ    L1
```

The processor instruction set, however, includes a group of loop instructions for implementing iteration. The basic LOOP instruction has the following syntax –

```
LOOP   label
```

Where, *label* is the target label that identifies the target instruction as in the jump instructions. The LOOP instruction assumes that the **ECX register contains the loop count**. When the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.

The above code snippet could be written as –

```
mov ECX,10
l1:
<loop body>
loop l1
```

Example

The following program prints the number 1 to 9 on the screen –

```
section   .text
  global _start       ;must be declared for using gcc

_start:               ;tell linker entry point
  mov ecx,10
  mov eax, '1'

l1:
  mov [num], eax
  mov eax, 4
  mov ebx, 1
  push ecx

  mov ecx, num
  mov edx, 1
  int 0x80

  mov eax, [num]
  sub eax, '0'
  inc eax
  add eax, '0'
  pop ecx
  loop l1
```

```
   mov eax,1          ;system call number (sys_exit)
   int 0x80           ;call kernel
section    .bss
num resb 1
```

When the above code is compiled and executed, it produces the following result –

```
123456789:
```

2.12 Strings

We have already used variable length strings in our previous examples. The variable length strings can have as many characters as required. Generally, we specify the length of the string by either of the two ways –

- Explicitly storing string length
- Using a sentinel character

We can store the string length explicitly by using the $ location counter symbol that represents the current value of the location counter. In the following example –

```
msg  db  'Hello, world!',0xa ;our dear string
len  equ  $ - msg          ;length of our dear string
```

$ points to the byte after the last character of the string variable *msg*. Therefore, ***$-msg*** gives the length of the string. We can also write

```
msg db 'Hello, world!',0xa ;our dear string
len equ 13                 ;length of our dear string
```

Alternatively, you can store strings with a trailing sentinel character to delimit a string instead of storing the string length explicitly. The sentinel character should be a special character that does not appear within a string.

For example –

```
message DB 'I am loving it!', 0
```

String Instructions

Each string instruction may require a source operand, a destination operand or both. For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination operands, respectively.

For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination, respectively.

There are five basic instructions for processing strings. They are –

- **MOVS** – This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
- **LODS** – This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
- **STOS** – This instruction stores data from register (AL, AX, or EAX) to memory.
- **CMPS** – This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
- **SCAS** – This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.

Each of the above instruction has a byte, word, and doubleword version, and string instructions can be repeated by using a repetition prefix.

These instructions use the ES:DI and DS:SI pair of registers, where DI and SI registers contain valid offset addresses that refers to bytes stored in memory. SI is normally associated with DS (data segment) and DI is always associated with ES (extra segment).

The DS:SI (or ESI) and ES:DI (or EDI) registers point to the source and destination operands, respectively. The source operand is assumed to be at DS:SI (or ESI) and the destination operand at ES:DI (or EDI) in memory.

For 16-bit addresses, the SI and DI registers are used, and for 32-bit addresses, the ESI and EDI registers are used.

The following table provides various versions of string instructions and the assumed space of the operands.

Basic Instruction	Operands at	Byte Operation	Word Operation	Double word Operation
MOVS	ES:DI, DS:EI	MOVSB	MOVSW	MOVSD

LODS	AX, DS:SI	LODSB	LODSW	LODSD
STOS	ES:DI, AX	STOSB	STOSW	STOSD
CMPS	DS:SI, ES: DI	CMPSB	CMPSW	CMPSD
SCAS	ES:DI, AX	SCASB	SCASW	SCASD

Repetition Prefixes

The REP prefix, when set before a string instruction, for example - REP MOVSB, causes repetition of the instruction based on a counter placed at the CX register. REP executes the instruction, decreases CX by 1, and checks whether CX is zero. It repeats the instruction processing until CX is zero.

The Direction Flag (DF) determines the direction of the operation.

- Use CLD (Clear Direction Flag, DF = 0) to make the operation left to right.
- Use STD (Set Direction Flag, DF = 1) to make the operation right to left.

The REP prefix also has the following variations:

- REP: It is the unconditional repeat. It repeats the operation until CX is zero.
- REPE or REPZ: It is conditional repeat. It repeats the operation while the zero flag indicates equal/zero. It stops when the ZF indicates not equal/zero or when CX is zero.
- REPNE or REPNZ: It is also conditional repeat. It repeats the operation while the zero flag indicates not equal/zero. It stops when the ZF indicates equal/zero or when CX is decremented to zero.

2.13 Arrays

We have already discussed that the data definition directives to the assembler are used for allocating storage for variables. The variable could also be initialized with some specific value. The initialized value could be specified in hexadecimal, decimal or binary form.

For example, we can define a word variable 'months' in either of the following way –

```
MONTHS      DW      12
```

```
MONTHS      DW    0CH
MONTHS      DW    0110B
```

The data definition directives can also be used for defining a one-dimensional array. Let us define a one-dimensional array of numbers.

```
NUMBERS     DW 34, 45, 56, 67, 75, 89
```

The above definition declares an array of six words each initialized with the numbers 34, 45, 56, 67, 75, 89. This allocates 2x6 = 12 bytes of consecutive memory space. The symbolic address of the first number will be NUMBERS and that of the second number will be NUMBERS + 2 and so on.

Let us take up another example. You can define an array named inventory of size 8, and initialize all the values with zero, as –

```
INVENTORY  DW 0
           DW 0
           DW 0
           DW 0
           DW 0
           DW 0
           DW 0
           DW 0
```

Which can be abbreviated as –

```
INVENTORY  DW 0,0,0,0,0,0,0,0
```

The TIMES directive can also be used for multiple initializations to the same value. Using TIMES, the INVENTORY array can be defined as:

```
INVENTORY TIMES 8 DW 0
```

2.14 Recursion

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number. Factorial of a number is given by the equation –

```
Fact (n) = n * fact (n-1) for n > 0
```

For example: factorial of 5 is 1 x 2 x 3 x 4 x 5 = 5 x factorial of 4 and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when n is 0.

2.15 Macros

Writing a macro is another way of ensuring modular programming in assembly language.

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with **%macro** and **%endmacro**directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition –

```
%macro macro_name  number_of_params
<macro body>
%endmacro
```

Where, *number_of_params* specifies the number parameters, *macro_name*specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions –

```
mov     edx,len     ;message length
mov     ecx,msg     ;message to write
mov     ebx,1       ;file descriptor (stdout)
mov     eax,4       ;system call number (sys_write)
int     0x80        ;call kernel
```

In the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s). If the program was already using those registers for keeping important data, then the existing data from these registers should be saved in the stack and restored after the instruction is executed.

2.16 File management

The system considers any input or output data as stream of bytes. There are three standard file streams –

- Standard input (stdin),
- Standard output (stdout), and
- Standard error (stderr).

File Descriptor

A **file descriptor** is a 16-bit integer assigned to a file as a file id. When a new file is created or an existing file is opened, the file descriptor is used for accessing the file.

File descriptor of the standard file streams - **stdin, stdout** and **stderr** are 0, 1 and 2, respectively.

File Pointer

A **file pointer** specifies the location for a subsequent read/write operation in the file in terms of bytes. Each file is considered as a sequence of bytes. Each open file is associated with a file pointer that specifies an offset in bytes, relative to the beginning of the file. When a file is opened, the file pointer is set to zero.

File Handling System Calls

The following table briefly describes the system calls related to file handling –

%eax	Name	%ebx	%ecx	%edx
2	sys_fork	struct pt_regs	-	-

3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-
8	sys_creat	const char *	int	-
19	sys_lseek	unsigned int	off_t	unsigned int

The steps required for using the system calls are same, as we discussed earlier –

- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h).
- The result is usually returned in the EAX register.

Creating and Opening a File

For creating and opening a file, perform the following tasks –

- Put the system call sys_creat() number 8, in the EAX register.
- Put the filename in the EBX register.
- Put the file permissions in the ECX register.

The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

Opening an Existing File

For opening an existing file, perform the following tasks –

- Put the system call sys_open() number 5, in the EAX register.
- Put the filename in the EBX register.
- Put the file access mode in the ECX register.
- Put the file permissions in the EDX register.

The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

Among the file access modes, most commonly used are: read-only (0), write-only (1), and read-write (2).

Reading from a File

For reading from a file, perform the following tasks –

- Put the system call sys_read() number 3, in the EAX register.
- Put the file descriptor in the EBX register.
- Put the pointer to the input buffer in the ECX register.
- Put the buffer size, i.e., the number of bytes to read, in the EDX register.

The system call returns the number of bytes read in the EAX register, in case of error, the error code is in the EAX register.

Writing to a File

For writing to a file, perform the following tasks –

- Put the system call sys_write() number 4, in the EAX register.
- Put the file descriptor in the EBX register.
- Put the pointer to the output buffer in the ECX register.
- Put the buffer size, i.e., the number of bytes to write, in the EDX register.

The system call returns the actual number of bytes written in the EAX register, in case of error, the error code is in the EAX register.

Closing a File

For closing a file, perform the following tasks –

- Put the system call sys_close() number 6, in the EAX register.
- Put the file descriptor in the EBX register.

The system call returns, in case of error, the error code in the EAX register.

Updating a File

For updating a file, perform the following tasks –

- Put the system call sys_lseek () number 19, in the EAX register.
- Put the file descriptor in the EBX register.
- Put the offset value in the ECX register.

- Put the reference position for the offset in the EDX register.

The reference position could be:

- Beginning of file - value 0
- Current position - value 1
- End of file - value 2

The system call returns, in case of error, the error code in the EAX register.

2.17 Memory Management

The **sys_brk()** system call is provided by the kernel, to allocate memory without the need of moving it later. This call allocates memory right behind the application image in the memory. This system function allows you to set the highest available address in the data section.

This system call takes one parameter, which is the highest memory address needed to be set. This value is stored in the EBX register.

In case of any error, sys_brk() returns -1 or returns the negative error code itself. The following example demonstrates dynamic memory allocation.

Example

The following program allocates 16kb of memory using the sys_brk() system call –

```
section   .text
  global _start       ;must be declared for using gcc

_start:               ;tell linker entry point

  mov     eax, 45             ;sys_brk
  xor     ebx, ebx
  int     80h

  add     eax, 16384          ;number of bytes to be reserved
  mov     ebx, eax
  mov     eax, 45             ;sys_brk
  int     80h
```

effort to convert an arithmetic formula to assembly language than it does to, say, Pascal, as long as you follow some very simple rules the conversion is not hard. For a step-by-step description, see Arithmetic Expressions, Simple assignments, Simple Expressions, Complex Expressions, Commutative Operators, Logical (Boolean) Expressions, One big advantage to assembly language is that it is easy to perform nearly unlimited precision arithmetic and logical operations. This chapter describes how to do extended precision operations for most of the common operations.

Exercise

1. Write the control functions and micro-operations that implement the fetch phase of the Basic Computer instruction cycle.
2. Write a Basic Computer assembly language loop that uses software polling to write a character to the output device as soon as it is ready to receive one.
3. List the steps necessary to enable interrupts to be used for output on the Basic Computer.

References

- Michael Singer, *PDP-11. Assembler Language Programming and Machine Organization*, John Wiley & Sons, NY: 1980.
- Peter Norton, John Socha, *Peter Norton's Assembly Language Book for the IBM PC*, Brady Books, NY: 1986.
- Dominic Sweetman: *See MIPS Run*. Morgan Kaufmann Publishers, 1999. ISBN 1-55860-410-3
- John Waldron: *Introduction to RISC Assembly Language Programming*. Addison Wesley, 1998. ISBN 0-201-39828-1
- Jeff Duntemann: *Assembly Language Step-by-Step*. Wiley, 2000. ISBN 0-471-37523-3
- Paul Carter: *PC Assembly Language*. 2001.

```
   cmp     eax, 0
   jl      exit        ;exit, if error
   mov     edi, eax    ;EDI = highest available address
   sub     edi, 4                ;pointing to the last DWORD
   mov     ecx, 4096             ;number of DWORDs allocated
   xor     eax, eax    ;clear eax
   std                           ;backward
   rep     stosd         ;repete for entire allocated area
   cld                           ;put DF flag to normal state

   mov     eax, 4
   mov     ebx, 1
   mov     ecx, msg
   mov     edx, len
   int     80h                   ;print a message

exit:
   mov     eax, 1
   xor     ebx, ebx
   int     80h

section    .data
msg        db        "Allocated 16 kb of memory!", 10
len    equ           $ - msg
```

When the above code is compiled and executed, it produces the following result –

```
Allocated 16 kb of memory!
```

2.18 Summary

To conclude this chapter I would like to point out that not all possible problems related to using Assembly language with high level language have been covered here. In this chapter, you will be able to solve most such problems on your own. Arithmetic expressions are much simpler in a high-level language than in assembly language Although it takes a little more