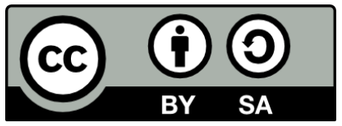
# CSC 216: ASSEMBLY LANGUAGE

## (2 Credits)

Published by the Centre for Open and Distance Learning,
University of Ilorin, Nigeria

E-mail: codl@unilorin.edu.ng
Website: https://codl.unilorin.edu.ng





# Course Development Team


**Subject Matter Expert**

**Rasheed Gbenga JIMOH, Ph.D.**
Department of Computer Science
University of Ilorin, Nigeria

**Instructional Designers**

**Olawale Koledafe**
Center for Open and Distance (CODL)
University of Ilorin, Nigeria

**Miss. Damilola Adesodun**
Department of Educational Technology,
University of Ilorin, Nigeria

**Mr. Jibril Mohammed**
Department of Educational Technology,
University of Ilorin, Nigeria

**Hassan Selim Olarewaju**
Department of Educational Technology,
University of Ilorin, Nigeria

**Language Editors**

**Bankole Ogechi Ijeoma**
Center for Open and Distance (CODL)
University of Ilorin, Nigeria




# From the Vice Chancellor

**C**ourseware development for instructional use by the Centre for Open and Distance Learning (CODL) has been achieved through the dedication of authors and the team involved in quality assurance based on the core values of the University of Ilorin. The availability, relevance and use of the courseware cannot be timelier than now that the whole world has to bring online education to the front burner. A necessary equipping for addressing some of the weaknesses of regular classroom teaching and learning has thus been achieved in this effort.

This basic course material is available in different electronic modes to ease access and use for the students. They are available on the University's website for download to students and others who have interest in learning from the contents. This is UNILORIN CODL's way of extending knowledge and promoting skills acquisition as open source to those who are interested. As expected, graduates of the University of Ilorin are equipped with requisite skills and competencies for excellence in life. That same expectation applies to all users of these learning materials.

Needless to say, that availability and delivery of the courseware to achieve expected CODL goals are of essence. Ultimate attention is paid to quality and excellence in these complementary processes of teaching and learning. Students are confident that they have the best available to them in every sense.

It is hoped that students will make the best use of these valuable course materials.

**Professor S. A. Abdulkareem**
**Vice Chancellor**

# Foreword

Courseware remains the nerve centre of Open and Distance Learning. Whereas some institutions and tutors depend entirely on Open Educational Resources (OER), CODL at the University of Ilorin considers it necessary to develop its own materials. Rich as OERs are and widely as they are deployed for supporting online education, adding to them in content and quality by individuals and institutions guarantees progress. Doing it in-house as we have done at the University of Ilorin has brought the best out of the Course Development Team across Faculties in the University. Credit must be given to the team for prompt completion and delivery of assigned tasks in spite of their very busy schedules.The development of the courseware is similar in many ways to the experience of a pregnant woman eagerly looking forward to the D-day when she will put to bed. It is customary that families waiting for the arrival of a new baby usually do so with high hopes. This is the apt description of the eagerness of the University of Ilorin in seeing that the centre for open and distance learning [CODL] takes off.

The Vice-Chancellor, Prof. Sulyman Age Abdulkareem, deserves every accolade for committing huge financial and material resources to the centre. This commitment, no doubt, boosted the efforts of the team. Careful attention to quality standards, ODL compliance and UNILORIN CODL House Style brought the best out from the course development team. Responses to quality assurance with respect to writing, subject matter content, language and instructional design by authors, reviewers, editors and designers, though painstaking, have yielded the course materials now made available primarily to CODL students as open resources.

Aiming at a parity of standards and esteem with regular university programmes is usually an expectation from students on open and distance education programmes. The reason being that stakeholders hold the view that graduates of face-to-face teaching and learning are superior to those exposed to online education. CODL has the dual-mode mandate. This implies a combination of face-to-face with open and distance education. It is in the light of this that our centre has developed its courseware to combine the strength of both modes to bring out the best from the students. CODL students, other categories of students of the University of Ilorin and similar institutions will find the courseware to be their most dependable companion for the acquisition of knowledge, skills and competences in their respective courses and programmes.

Activities, assessments, assignments, exercises, reports, discussions and projects amongst others at various points in the courseware are targeted at achieving the objectives of teaching and learning. The courseware is interactive and directly points the attention of students and users to key issues helpful to their particular learning. Students' understanding has been viewed as a necessary ingredient at every point. Each course has also been broken into modules and their component units in sequential order.

At this juncture, I must commend past directors of this great centre for their painstaking efforts at ensuring that it sees the light of the day. Prof. M. O. Yusuf, Prof. A. A. Fajonyomi and Prof. H. O. Owolabi shall always be remembered for doing their best during their respective tenures. May God continually be pleased with them, Aameen.

**Bashiru, A. Omipidan**
**Director, CODL**



# INTRODUCTION

I welcome you to Internet Technology I, a second-semester course. Internet Technology I is a two (2) unit course that provides a general introduction to Internet Technology, covering a brief history of the Internet, how it grew from its humble origins into the worldwide network that is available today, identifying the most popular Internet services such as information retrieval, WWW and communication services.

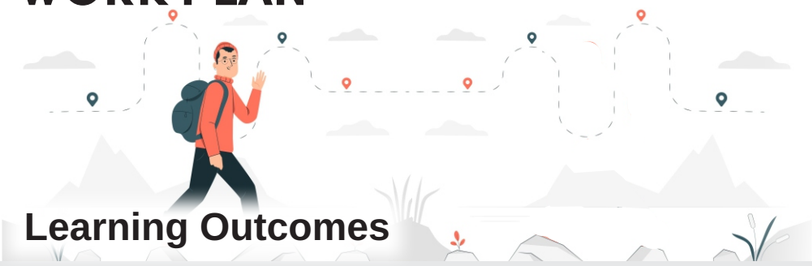
The relationship between the Internet and the World Wide Web is discussed. The course provides you with comprehensive knowledge on the concepts of Internet Technology, which include its internet architecture and internet protocol. The two most important protocols that allow networks to communicate with one another and exchange information, that is the TCP (Transmission Control Protocol) and IP (Internet Protocol), are also discussed.

Also, the functions of each layer at the TCP/IP networking model are covered. The brief history of HTML, XML, XHTML and DHTML is addressed. The course also covers in depth, HTML5, CSS and Javascript. The course also discusses the concept of a markup language and how to create web pages using HTML5 elements, CSS and Javascript. the course also discusses other equally important topics like WYSIWYG, Test Editors, including notepad, notepad++ and others.

## Course Goal

The major goal of this course, CSC 224,is to introduce you to the concept of Internet technology and teach you how to develop websites using available technologies.

# WORK PLAN



## Learning Outcomes

At the end of this course, you should be able to:

**Week 01**

- Define programming language and all levels of programming language
- Draw a block diagram of digital computer and state the functions of each unit
- Identify numbering system and convert from one number system to another

**Week 02**

- Know floating point system and its various types
- Define register and state the functions of each register
- Define addressing modes and recognize an operand in a line of instruction

**Week 03**

- Know all the types of addressing modes
- Differentiate between each type of addressing modes in an instruction
- Identify and calculate the physical and effective address of 8086 processor
- Know an instruction set and its functions
- Identify the formats of Assembly language and write a program in Assembly language

## Course Infomation

This is a compulsory course for students in the Departments of Chemistry, Industrial Chemistry, Geology, Mathematics, Computer Science. You are expected to participate in all the course activities and have minimum of 75% attendance to be able to write the final examination

Pre-requisite

**CSC 319**
Introduction to Digital Design and Microprocessors

## Course Guide

### Module 1

Introduction to Programming Language

**Unit 1** - Definition and History of programming language

**Unit 2** - Block diagram of digital computer and functions of each unit

**Unit3** - Numbering system

**Unit4** - Intel 8086/ 8087 Floating Point Software Architecture

### Module 2

Registers

**Unit 1** - Registers

**Unit 2** - Different categories of registers and their functions

### Module 3

Addressing modes

**Unit 1** - Addressing modes

**Unit 2** - Types of addressing modes

**Unit3** - Sample calculations 8086 physical memory, logic address and effective address

### Module 4

Instruction Sets Intel 8086

**Unit 1** - Instruction sets

**Unit 2** - Syntax and functions of Instruction sets

### Module 5

Assembly Language Program

**Unit 1** - Introduction to MASM Assembler

**Unit 2** - Format of an Assembly language program

**Unit3** - Sample of programs in Assembly language

## Course Requirements

Requirements for success

**The CODL Programme** is designed for learners who are absent from the lecturer in time and space. Therefore, you should refer to your Student Handbook, available on the website and in hard copy form, to get information on the procedure of distance/e-learning. You can contact the CODL helpdesk which is available 24/7 for every of your enquiry.

Visit CODL virtual classroom on http://codllms.unilorin.edu.ng. Then, log in with your credentials and click on CSC 111. Download and read through the unit of instruction for each week before the scheduled time of interaction with the course tutor/facilitator. You should also download and watch the relevant video and listen to the podcast so that you will understand and follow the course facilitator.

At the scheduled time, you are expected to log in to the classroom for interaction.

Self-assessment component of the courseware is available as exercises to help you learn and master the content you have gone through.

You are to answer the Tutor Marked Assignment (TMA) for each unit and submit for assessment.

## Embedded Support Devices

Support menus for guide and references

Throughout your interaction with this course material, you will notice some set of icons used for easier navigation of this course materials. We advise that you familiarize yourself with each of these icons as they will help you in no small ways in achieving success and easy completion of this course. Find in the table below, the complete icon set and their meaning.

| | | |
|---|---|---|
|  **Introduction** |  **Learning Outcomes** |  **Main Content** |



| | | |
|---|---|---|
|  **Summary** |  **Tutor Marked Assignment** |  **Self Assessment** |
|  **Web Resources** |  **Downloadable Resources** |  **Discuss with Colleagues** |
|  **References** |  **Futher Reading** |  **Self Exploration** |

## Grading and Assessment


TMA


CA


Exam


Total


**01** | **Picture:** Introduction to programming language

**Photo**: Unsplash.com

Module 1

# Introduction to Programming Language

## Units

**Unit 1** - Definition and History of programming language

**Unit 2** - Block diagram of digital computer and functions of each unit

**Unit3 -** Numbering system

02 | **Picture:** history of programming

**Photo**: Unsplash.com

UNIT 1

# Definition and History of Programming Language

## Introduction

In this unit you will be introduced to the importance of programming language and various levels of programming languages. You will also learn the various advantages and problems encountered in using each of this programming language.

## Learning Outcomes

**At the end of this unit, you should be able to:**

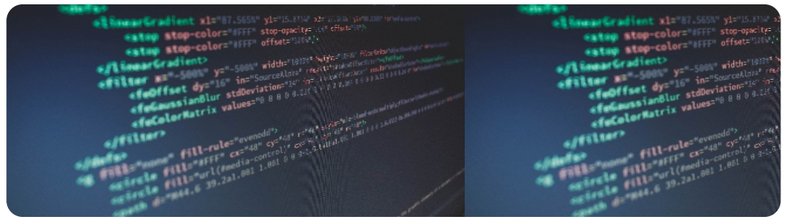
1. Define programming language
2. Mention two advantages of each levels of programming language
3. List two disadvantages of each levels of programming language

# Main Content

## Definition and History of Programming Languages

1 min

SAQ 2



1. You will agree with me that, in order for computers to accept commands from humans and perform tasks vital to productivity, a means of communication must exist. Programming languages provide this necessary link between man and machine. Therefore, one can say Programming language is the medium of communication between Man and Machine. It is a step by step ways of telling the computer what to do. Computers require a series of instructions (programs) before they can do any processing. It is these programs that will direct the computer to carry out the required task. The programs have to be written out comprehensively, to cover all possibilities and in the right order before the control unit of the Central Processing Unit (CPU) can use them effectively. Just as human languages too, we have many spoken languages, the same thing is also applicable to programming languages, and there are several types of programming languages that can be used to communicate with the computer. That is Programs can be written in several languages. These languages are categorized into four levels, they are:

1. Machine Language
2. Low level Language,
3. High Level language and
4. Fourth generation language.





## Machine Language

4 mins

SAQ 3,4, 5,6,7,8

I want you to note that a Machine Language consists of zeros and ones and it is the only language that a computer understands. It is the numeric codes for the operations that a particular machine can execute directly. The codes are strings of 0s and 1s, also known as binary digits (“bits”), which are often converted both from and to hexadecimal (base 16) for human viewing and modification. Machine language instructions typically use some bits to represent operations, such as addition, and some to represent operands, or perhaps the location of the next instruction.

You should be aware that machine Language has the advantages of producing less code, Storage is saved, User has direct control of machine instruction and Execution is faster as no translation is needed. However, Machine language is difficult to read and write; since it does not resemble conventional mathematical notation or human language, it is time consuming, it is difficult to locate and correct errors, it is not relocatable and its codes vary from computer to computer (i.e. it is machine dependent, a particular machine language can be used on only one type of computer).

In other to provide solutions to some of the problems encountered in machine language, another language was developed called low level language (LLL), which will be discussed after this. Example of machine language to add two numbers together is shown below:

Machine Language program to add two numbers:

| location Hex | Instruction Code Binary | Instruction Code |
|---|---|---|
| 100 | 0010 0001 0000 0100 | 2104 |
| 101 | 0001 0001 0000 0101 | 1105 |
| 102 | 0011 0001 0000 0110 | 3106 |
| 103 | 0111 0000 0000 0001 | 7001 |
| 104 | 0000 0000 0101 0011 | 0053 |
| 105 | 1111 1111 1111 1110 | FFFE |
| 106 | 0000 0000 0000 0000 | 0000 |

## Low Level Language (LLL)

LLL is a language in which Mnemonics or abbreviations are used to represent computer operation and memory addresses. It is called LLL because of its closeness to machine language. It uses short mnemonic codes for instructions and allows the programmer to introduce names for blocks of memory that holds data. One might thus write "ADD B, SUM" instead of "0110101100101000" for an instruction that adds two numbers in machine language.

A typical example of LLL is Assembly Language, in which mnemonics or symbolic codes are used to represent computer operations.

It is one of the programming languages that relieve programmers from coding in ones and zeros. Assembly language is much easier to write compare with Machine language, it is less cumbersome, debugging and maintenance is very easy. However, AL requires a translator to translate it to Machine language for computer to understand, Program development in Assembly language is slow as the programmer must have detailed knowledge of the hardware structure, Program maintenance is slow and error prone (not as much as ML) and it is equally not portable (i.e. it is defined for a particular processor).

It is easy to read and write since it is very close to human language compare to ML. Analysis of assembly code is a pertinent skill for C and C++ programmers, who may occasionally have to diagnose a fault by looking at the contents of CPU registers and single-stepping through machine instructions. One of the major interesting things about Assembly language is that it serves as an important part to the path of understanding how the machine works. Without a programmer learning assembly language first, it is not possible to learn advanced concepts such as microcode, pipelining, instruction scheduling, out-of-order execution, threading, branch prediction, and speculative execution, concepts that are very essential when dealing with operating systems and computer architecture. All these require some understanding of assembly language.

Some examples of assembly language to add two numbers together is shown below:





Assembly Language program to subtract two numbers:

```
   ORG 100/    Origin of program is location 100
   LDA A/      Load operand from location A
   SUB  B/     Add operation form location B
   STA C/      Store sum in location C
   HLT/        Halt computer
A, DEC 83/     Decimal operand
B, DEC –2/     Decimal operand
C, DEC 0/      Sum stored in location C
   END
```

## High Level Language (HLL)

You should know that computers usage increased rapidly with the advent of assembly languages, but programmers still find it very hard to use many instructions to accomplish even the simplest tasks. To speed up the programming process, high level language were developed in which simple statements could be written to accomplish substantial tasks. Compilers as a translator convert high level language programs into machine language. Some high level languages also use a translator called interpreter. A compiler is a program that translates a source program written in some high-level programming language (such as Java) into object code for some computer architecture (such as the Intel 8086 architecture) all at once. An interpreter translate source program written in a high-level programming language to machine language one line at a time. High level language allows programmers to write instructions that look almost like every day English and contain commonly used mathematical notations.

In HLL, Programs are much easier and faster to write, debug and maintain than in both machine and low level language, Compilers are easy to get, It is easier to use than any other programming languages, use of data-names carefully chosen by programmers make the program self-documenting and it is machine independent. However, HLL also needs a translator and Object programs produced by a compiler are not generally as efficient in terms of main memory usage and speed of execution compared with machine code or object program produced by an assembler.

Example of HLL program written in BASIC to add two numbers together:

```
10 Let Length = 83
20 Let Breadth = 2
30 Area  =Length * Breadth
40 Print Area
50 End
```

## Fourth Generation Language (4GL)

It may interest you to know that all the above three levels of programming languages are usually referred to as procedural languages. This implies that, the programmer have to tell the computer how to do what you want it to do step by step. In non-procedural or declarative language, you only tell the computer what you want it to do and not how it is to be done. This is the main characteristic of the fourth generation language. They are sometime called program generators. Structured Query Language (SQL) used with a relational database is one of the earliest forms of the 4GL.

## Summary

Communication is an aspect in our way of life that it's important and cannot be over emphasized. Communicating with human require learning a particular language, this is also applicable to Computer, without learning any programming language, communicating with computer cannot be possible. There are several programming languages which include Machine Language, Low level language and High level language, of which Machine language is the only one understands by the computer, others require translators to translate it into Machine Language. All these are what we have learnt in this unit.





## Self-Assessment Questions

1. _____ is the medium of communication between Man and Machine.
a. Programming Language b. Generation Language c. Procedural
2.There are _____ levels of programming language a.Three b. Five c. Four
3. Machine Language consists of _____ a. zeros and ones b. ones and twos c. twos and threes
4. Machine Language has the advantages of producing less code and Storage is saved. a.True b. False c. None
5. _____is the first programming language that relieves programmers from writing in zeros and ones. a.Machine Language b. Assembly Language c. High Level Language
6. Program development in Assembly language is slow as the programmer must have detailed _____a. knowledge of the hardware structure b. knowledge of the binary structure c. knowledge of the binary structure
7. An example of High Level Language is _____ a. Java b. Assembly Language c. Machine Language
8. HLL Programs are much _____ to write. a. Easier and faster b. difficult c. not easy

## Tutor Marked Assessment

- What is a program?
- State the difference between programming language and program
- List the four levels of programming language
- State three advantages of High level language over Assembly language
- In your own view, what is the unique difference of Machine language with other programming language?
- Mention four examples of High level language

## References


- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Kann, Charles W., "Introduction To MIPS Assembly Language Programming" (2015). Gettysburg College Open Educational Resources
- Carter P.A. (2003). PC Assembly Language. Paul carter, USA. 21-53
- https://www.britannica.com/technology/computer-programming-language


## Further Reading

- http://cupola.gettysburg.edu/oer

**03** | **Picture:** Diagram of digital computer

**Photo**: Unsplash.com

UNIT 2

# Block Diagram of Digital Computer and Functions of each unit

## Introduction

In unit 1 you have learnt the definition and history of programming language. In this unit you will be introduced to the diagrammatic representation of digital computer, its various components as well as its functions.

## Learning Outcomes

**At the end of this unit, you should be able to:**

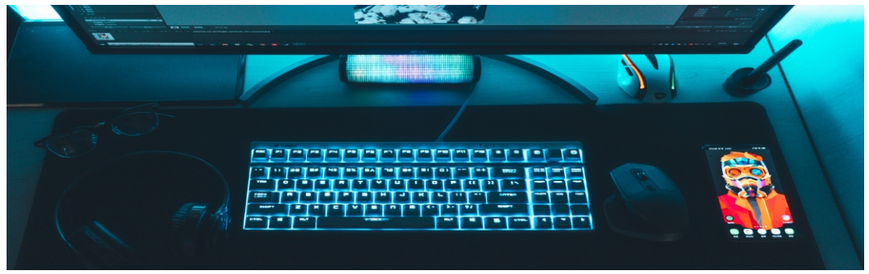
1. Identify the block diagram of a digital computer
2. State at least two functions of each unit

## Main Content

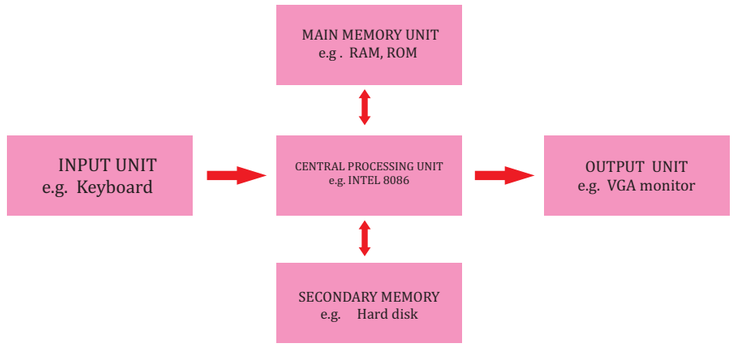
SAQ 1

# Block Diagram of Digital Computer and Functions of each unit

1 min



You are already familiar with the meaning of Digital Computer System as a type of computer that performs calculations, logical operations and other process using binary numbering system. Digital computer is the most popular type of computer in contrast with analogue computer. This computer is comprises of various components or units each with its own unique functions. A typical digital computer is represented with the block diagram in figure 1.2.1.




**Block diagram of Digital Computer Units**

Digital Computer has four main circuits, namely, central processing unit (CPU), Input unit, Output unit and memory (Main memory and Secondary memory)





SAQ 1,2 3,4,5&6

# Functions of each Unit in a Digital Computer

2 mins

**Central Processing Unit (CPU):** The CPU is usually what microprocessor used for fetching and executing instructions or programs. Intel 8086 microprocessor is an example of a CPU. This processor can be made to perform two functions on data. It can read one instruction at a time, match the instruction with its set and performs a specific manipulation on them and finally the result can be stored in the memory or displayed in any of the output devices such as monitor or printer. Some of them use three different busses to achieve most of their operations, for instance 8085/8086 processors uses Address bus, Data bus and Control bus.

The Address bus allows the processor to access locations with numbers ranging from 0 to 65,536 or access a total of 64K and 16 bit wide. If a peripheral needs from this processor, it will place the 16-bit address on the address bus and then sends the necessary control signals. On the other hand the Data bus is 8-bit wide and helps the processor in transferring binary information. Lastly the control bus is only made up of numbers of single bit control signals.

**Main Memory Unit:** The main function of the memory unit is that all programs needed for execution by the CPU is loaded into main memory, the CPU fetches the program from the main memory, instruction by instruction, and then execute the instruction in its internal registers, results of such execution can then be stored back in the main memory. It consists of RAM (Random Access Memory) and ROM (Read Only Memory). RAM is volatile; hence all program and data stored in it disappear when power is turned off. In other to solve the problem of volatility and low storage capacity of main memory, secondary memory is used.

Secondary Memory is also known as back up memory; mass memory or external memory. In case of secondary memory, It is non- volatile and of very large capacity. The RAM is made up of registers that contains flip flops. The number of flip flops in memory register determines the size of the memory word while ROM uses diodes to permanently hold the information. In other for microprocessor to access information either from RAM or ROM, it needs to perform three actions using the bus (Address, data and control busses) explained under Microprocessor. Firstly, it will select the right memory chip using a part of the address bus, identify the memory location using the rest of the address bus and finally access the data using the data bus.

**Input and Output Unit:** Digital computer provides facility for the attachment of input and output devices. Input unit provides a platform for attachment of input devices such as keyboard, mouse, barcode reader and scanner while Output Unit provides a platform for attachment of output devices such as computer monitor, printer or LED. They are sometimes refer to as medium of communicating with the outside world and also called the computer peripherals.

Communication path between the processor and peripherals is achieved by a group of wires carrying the voltages and currents representing the different bit values and can only communicate with only one peripheral at a time.

## Summary

You have been able to learnt that Digital Computer which requires binary in performing its operations. Majority of the circuits in a digital computer have their major functions that make the computer works perfectly.





## Self-Assessment Questions

1. _____ is a type of computer that performs calculations, logical operations and other process using binary numbering system. a.Digital Computer System b. Analogue Computer system c. None
2. _____, _____, _____, and _____ are the four main circuits of a digital computer system. a.Central Processing Unit (CPU), Input unit, Output unit and Memory b. CPU, Secondary, Keyboard and Monitor c. CPU, Monitor, Printer and Keyboard
3. Secondary Memory is also known as _____ a. Back end memory b. Back up memory c. Back front Memory
4. An example of an input device is _____ a. Printer b. Keyboard c. Monitor
5. Main memory consists of _____ and _____ a.RAM and ROM b. RAM and MOR c. MAR and MOR
6. _____ and _____ are the functions of Central processing unit a. fetches the program from the main memory and execute the program fetched b.Store and retrieve the instructions c. Consists of RAM and ROM

## Tutor Marked Assessment

- Draw a block diagram of a digital computer
- What are the differences between RAM and ROM
- Mention two examples of Input and Output devices

## References


- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
https://www.coursehero.com/file/block

## Further Reading

- http://cupola.gettysburg.edu/oer

| Numbering System | | |
|---|---|---|
| System | Base | Digits |
| Binary | 2 | 0, 1 |
| Octal | 8 | 0,1,2,3,4,5,6,7 |
| Decimal | 10 | 0,1,2,3,4,5,6,7,8,9 |
| Hexadecimal | 16 | 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F |

**04** | **Picture:** Numbering system

**Photo**: Wikipedi.com

UNIT 3

# Numbering System

## Introduction

This unit introduces the students to the numbering system, the important of numbering system in a digital computer, types of numbering system and methods of converting from one numbering system to another.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Define numbering system
2. State at least two important of numbering system
3. Convert form one numbering system to another

## Main Content

SAQ 1, 3,4

### Numbering System

2 mins

You should note that the basic unit of data in a digital computer is the binary digit also known as bit. A bit consists of zero or one. In order to store numbers larger than 1, bits are combined into larger units. The numbering system that is being taught in school is called decimal or base 10. This numbering system is called decimal because it has 10 digits, and the numbers is from 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Any values up to 9 can be easily referenced in this system by a single number. Computers use switches that can either be on (1) or off (0), therefore its major operations is in binary or base 2 numbering system. In binary, there are only two digits, 0 and 1. So values up to 1 can be easily represented by a single digit. Humans are more convenient with decimal numbering system than the binary numbering system. Hence, there is need to convert from decimal to binary number. Numbering system is the art or ways of expressing or writing numbers. Every number system uses positional notation i.e., each position that contains a digit is written has a different positional value. Each position is power of the base, which is 2 for binary number system, and these powers begin at 0 and increase by 1.

### Types of Numbering System

It will interest you to know that there are four major types of numbering system, which are:

- Binary
- Decimal
- Octal
- Hexadecimal

Note that the Binary numbering system consists of 0 and 1, decimal consists of 0 to 9, Octal is from 0 to 8 and hexadecimal is from 0 to 15. The real numbering system to reckon with in the computer world is base 16, which is normally refer to as hexadecimal or simply "hex."





Table 1.3.1 shows the numbering system and their representations.

**Table 1.3.1: Numbering system**

| Type | Base | Numbers |
|---|---|---|
| Binary | 2 digits | {0,1} |
| Octal | 8 digits | {0,1,2,3,4,5,6,7} |
| Decimal | 10 digits | {0,1,2,3,4,5,6,7,8,9} |
| Hexadecimal | 16 digits | {0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F} ·From 10 to 15 is represented as A-F in Hexadecimal |

SAQ 5, 7&8

### Conversions in Numbering System

5 mins

Programmers mostly work with base two (binary), base eight (octal), base ten (decimal) and base sixteen (hexadecimal). Base eight and ten are often use because they are easily translated quickly to and from base two, and are often easier for humans to work with than base two.

Before teaching assembly language, it is always essential to know how to convert from one base to any other base. This unit explains how to convert from any arbitrary numbers to base ten and base ten to other bases.

### Arbitrary Numbers to Base Ten Conversion

Converting from an arbitrary base n (where n denote the base number) to base ten simply involves multiplying each digit in base n by base n with a superscript of the significant number and summing all of the results. For example, converting the base five numbers 4215 to base ten is performed as follows

**Example 1:** Convert 4215 to base ten

$4215 = 4 \times 5^2 + 2 \times 5^1 + 1 \times 5^0$

$= 10010 + 1010 + 110$

$= 111_{10}$

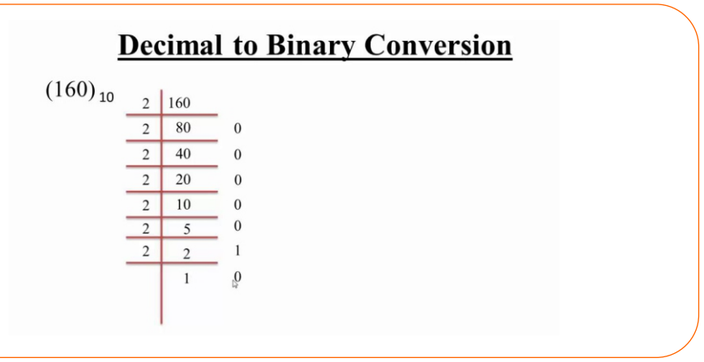
**Example 2:** Convert 2156 to base ten

$2155 = 2 \times 6^2 + 1 \times 6^1 + 5 \times 6^0$

$= 7210 + 610 + 510$

$= 83_{10}$

These examples above are applicable for converting from any arbitrary numbers to base ten.




## Decimal to Binary Conversion

Decimal to binary conversion is by a process of repeated division of the decimal number by 2 until the quotient is zero and keeping the remainder at each division as the equivalent binary.

**Example 3:** Convert $42_{10}$ to binary

| | | |
|---|---|---|
| 2 | 42 | R |
| 2 | 21 | 0 |
| 2 | 10 | 1 |
| 2 | 5 | 0 |
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| | 1 | 1 |

**$= 101010_2$**





**Example 4:** Convert $195_{10}$ to binary

| | | |
|---|---|---|
| 2 | 195 | R |
| 2 | 97 | 1 |
| 2 | 48 | 1 |
| 2 | 24 | 0 |
| 2 | 12 | 0 |
| 2 | 6 | 0 |
| 2 | 3 | 0 |
| 2 | 1 | 1 |
| | 0 | 1 |

**$= 11000011_2$**

## Decimal to Hexadecimal Conversion

It is very compact using base of 16 meaning that the number of digits used to represent a given number is usually fewer than in binary or denary. It is easy to convert between hex and binary and also easy to go between hex and denary. Remember that the microprocessor only works in binary, all the conversions between hex and binary are carried out in other circuits

Decimal to hexadecimal conversion is by a process of repeated division of the decimal number by 16 until the quotient is zero and keeping the remainder at each division as the equivalent binary.

**Example 5:** Convert $759_{10}$ to hexadecimal (base 16)

759 /16 = 47 rem 7
47/16 = 2 rem 15
15/16 = 0 rem 2
$2F7_{16}$ (10-15 in hexadecimal is represented by A-F)

**Example 6:** Convert $66_{10}$ to base 16

66/16 = 4 rem 2
4/16 = 0 rem 4
$42_{16}$

## Addition of Binary Numbers

Here, you will learn how addition and subtraction is perform on binary numbers, this is essential to discuss because it will aid us in learning how integer values are stored and used in a computer for calculations. An illustration of binary addition is shown in example 7 and 8.

**Example 7:** Add 0011 and 0111

$$\begin{array}{r} 0\ 0\ 1\ 1 \\ +\ 0\ 1\ 1\ 1 \\ \hline 1\ 0\ 1\ 0 \end{array}$$

In this example, first bit adds $1_2 + 1_2$, (2-2) which produces a $0_2$ in this bit and a carry bit of $1_2$. The next bit now has to add $1_2 + 1_2 + 1_2$ (3-2,the extra one is the carry bit), which produces a $1_2$ for this bit and a carry bit of $1_2$, the next is adding of $1_2 + 0_2 + 1_2$, which yields a $0_2$ and a carry bit of $1_2$ and the last bit which produces a $1_2$ ($0_2 + 0_2 + 0_2$). If you follow the arithmetic properly, you will have 00112 (310) + 01112 (710) = 10102 (1010).

**Example 8:** Add 1001 from 0101

$$\begin{array}{r} 1\ 0\ 0\ 1 \\ +\ 0\ 1\ 0\ 1 \\ \hline 1\ 1\ 1\ 0 \end{array}$$

In this example, first bit adds $1_2 + 1_2$, (2-2) which produces a $0_2$ in this bit and a carry bit of $1_2$. The next bit now has to add $0_2 + 0_2 + 1_2$, which produces a $1_2$ for this bit and no carry, the next is adding of $0_2 + 1_2$, which yields a $1_2$ and no carry bit and the last bit which produces a $1_2$ ($1_2 + 0_2$). If you follow the arithmetic properly, you should be able to do it perfectly





## Subtraction of Binary Numbers

Subtraction operation in binary is effected by first taking the 2's complement of the subtracted (i.e number to be subtracted) number and then adding it to the subtractor (i.e. number from which subtraction is to be made). Example 9 and 10 shows how to subtract numbers in binary

**Example 9:** $19_{10}$-$13_{10}$

The first step is to obtain the binary number value of the numbers.

| | | |
|---|---|---|
| 2 | 19 | R |
| 2 | 9 | 1 |
| 2 | 4 | 1 |
| 2 | 2 | 0 |
| 2 | 1 | 0 |
| | 0 | 1 |

**= 10011**

| | | |
|---|---|---|
| 2 | 13 | R |
| 2 | 6 | 1 |
| 2 | 3 | 0 |
| 2 | 1 | 1 |
| | 0 | 1 |

**= 1101**

The next stage is to take the compliment -13 = $0010_2$ (taking 1's compliment)
= 100112 (taking 2's complement and 1 to the msb since it is negative)

**$10011_2$**

The next stage is to take the compliment -13 = $0010_2$ (taking 1's compliment)

= 100112 (taking 2's complement and 1 to the msb since it is negative)

$10011_2$

Add the two numbers 19 and 13 with sign bits:

| | | |
|---|---|---|
| 0 | 10011 | (since the first number is positive) |
| 1 | 10011 | (since the second number is negative) |
| 0 | 00110 | =+6 |

**00110**

**Example 10:** $13_{10}$-$19_{10}$

- obtain binary number
  $13_{10} = 1101_2$
  $19_{10} = 10011_2$s
- Take 2's complement of $19_{10}$ in binary and add 1 to the msb since it is negative
  2's complement of $10011_2$ is 01101
  With 1 as its msb, then we have 101101
- Add the two numbers 13 and 19 with sign bits:

| | |
|---|---|
| 0 | 001101 |
| 1 | 101101 |
| 1 | 111010 |

- if the sign bit of result is 1, i.e negative, then take 2's complement of the result again (2's complement of 111010 is 000110) and then insert '-' symbol in the decimal value obtained.

  Decimal value of 000110 is 6
  Result = $-6_{10}$

## Multiplication of Binary Numbers

In multiplication of binary numbers, write down the multiplicand (i.e number to be multiplied) in binary then write under it, right justified, the multiplier (i.e number to be used for multiplication) in binary. Then carry out these steps:

Step 1 Carry out the multiplication operation

Step 2 Add all the partial products





Normally, Computer performs this by repeated addition of multiplicand by multiplier number of times. Example 11 shows an example of multiplication of numbers.

For instance, 5 x 3. Here, 5 is multiplicand and 3 is multiplier. From this example, computer does $101_2 + 101_2 + 101_2 = 1111_2 = 15_{10}$

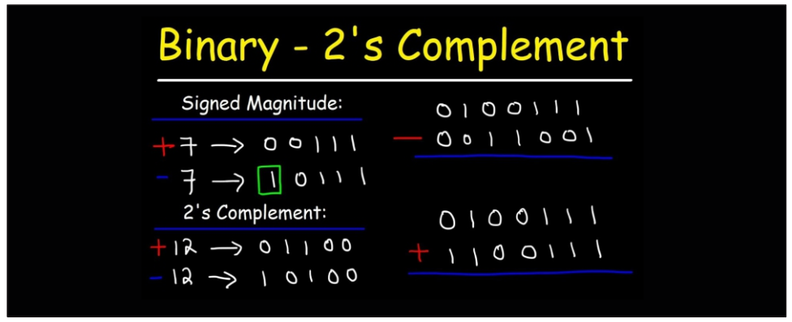
Convert 5 into binary and add it with the number of multiplier

| | |
|---|---|
| 1012 | (510) |
| x 11 | (310) |
| 101 | |
| 101 | |
| 11112 | = (1510) |

SAQ 6

## Compliment Numbers and their Operations

2 mins




The complement is the amount that must be added to something to make it "whole." For instance, in geometry, two angles are complementary if they add to 90◦. For example, in base ten, the complement of 4 is 10-4 = 6. In complement representation, the most significant digit of a number is reserved to indicate whether the number is negative or not.

There are two types of compliment representation: the 1's compliment and 2's compliment. These compliments have their operations.

## 1's Complement Operation:

1's complement of a binary number is obtained by changing every bit value to its complement.

**Example 1:** Obtain the 1's complement of $11011001_2$

Change the bit: $00100110_2$

Therefore the 1's complement = $\mathbf{00100110_2}$

**Example 2:** Obtain the 1's complement of $01011101_2$

Change the bit: 101000102

Therefore the 1's complement = $\mathbf{10100010_2}$

## 2's Complement Operation:

2's complement of a binary number is obtained by adding 1 to the 1's complement of that number. In other words, 2's complement is achieved by first getting the 1's complement and add 1 to the value of 1's complement.

**Example 1:** Obtain the 2's complement of $11011001_2$

Find the 1's complement by changing the bit: $00100110_2$

Add 1 to the value of the 1's complement

```
00100110
+        1
--------
00100111
```

Therefore 2's complement = $\mathbf{00100111_2}$

One very useful feature of complement notation is that it can be used to perform conversion of negative decimal number to binary. Examples of this are as follows:





**Example 1:** Convert $-5_{10}$ to binary

Convert the decimal number to binary

| | | |
|---|---|---|
| 2 | 5 | R |
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| | 0 | 1 |

$= 101_2$

- Take the 2's complement of the binary number
  = 0112
- Insert 1 as the msb (most significant bit) to represent the negative sign
  $-5_{10} = 1011_2$

**Example 2:** Convert -123 decimal to binary

Obtain binary number

| | | |
|---|---|---|
| 2 | 123 | R |
| 2 | 61 | 1 |
| 2 | 30 | 1 |
| 2 | 15 | 0 |
| 2 | 7 | 1 |
| 2 | 3 | 1 |
| 2 | 1 | 1 |
| | 0 | 1 |

$= 1111011_2$

- Take the 2's complement
  = 0000101
- Add 1 as msb
  -12310= $10000101_2$

## Summary

You have learnt that computer systems deal with binary that consists of zeros and ones; however, there are some other numbering system recognized by the Assembly language that must be known both in representations and conversion. All these were explained in this unit.

## Self-Assessment Questions

SAQ

1. Numbering system is the art or ways of expressing _____
a.Numbers b. Binary Numbers c. Digital
2. The basic unit of data in a digital computer is the _____ digit
a. Denary b. Binary c. Unary
3. Humans are more convenient with decimal numbering system than the binary numbering system. a.Yes b. No c. None
4. Decimal numbering system consists of_____
5. Convert 125 to binary
6. Find the two's compliment of $11011001_2$
7. Convert -2510 to binary
8. Convert 75910 to hexadecimal

## Tutor Marked Assessment

- What is numbering system?
- Briefly explain the four types of numbering system
- Find the addition of this number: 0011 and 0110
- Subtract 5 from 3 in binary number



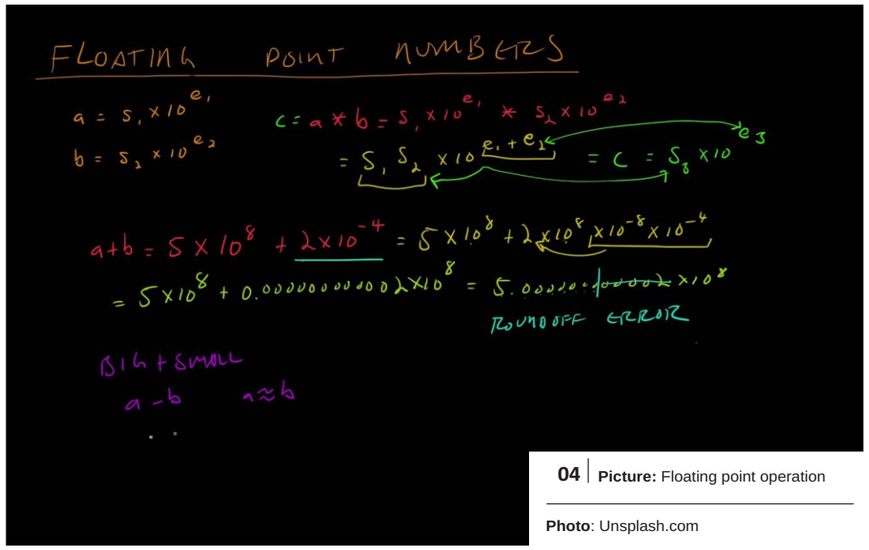


## References


- Jeff Duntemann Assembly Language Step-by-Step: Programming with DOS and Linux, (2000). Second Edition. John Wiley & Sons. ISBN:0471375233
- John Crisp. 'Introduction to Microprocessors and Microcontrollers' (2014). Newnes An imprint of Elsevier. 27-50
- Kann, Charles W., "Introduction To MIPS Assembly Language Programming" (2015). Gettysburg College Open Educational Resources. 16-35
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.


## Further Reading

- http://cupola.gettysburg.edu/oer
http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2




**04** | **Picture:** Floating point operation

**Photo**: Unsplash.com

UNIT 4

# Intel 8086/8087 Floating Point Operation

## Introduction

In this unit, you will be introduced to another representation of numbering system known as floating point operations, the three major ways of converting numbers the important of numbering system in a digital computer, types of numbering system and methods of converting from one numbering system to another.

## Learning Outcomes

**At the end of this unit, you should be able to:**

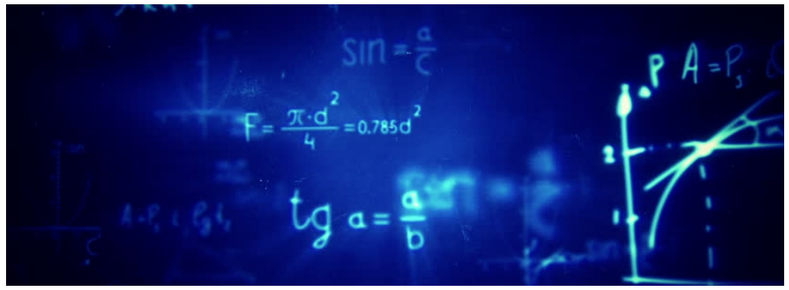
1. Define Floating point system
2. Mention the three types of floating point format
3. Convert number to any of the types of floating point format

## Main Content

SAQ 2&3

# Intel 8086/8087 Floating Point Operation

4 mins




It will interest you to note that the normal numbering system discussed in unit 3 is not the only way of representing numbers in computing. A floating point system can be used in representing together with fixed number of digits, the numbers of different orders of magnitude. There are several ways of representing floating point operations in computer system. One of them is the IEEE 754 Standard for Floating-Point Arithmetic. The floating point unit is a part of computer system responsible for carrying out operations in floating point. The IEEE 754 standard helps to define floating point in three different formats, namely: Short Real, Long Real and Temporary Real.

Short real: It has 32 bits, 1 bit for the sign, 8 bits for the exponent and 23 bits for the fractional part of the significant or mantissa.

Long real: It has 64 bits, 1 bit for the sign, 11 bits for the exponent and 52 bits for the fractional part of the significant or mantissa.

Temporary Real: It has 80 bits, 1 bit for the sign, 16 bits for the exponent and 63 bits for the fractional part of the significant or mantissa.

Table 1.4.1 shows the structures of the three (3) floating point formats:





## Table 1.4.1: The Floating Point Representation

| Data format | Word Length | No. of Sign | No. of Exponent | Exponent Bias | No. of mantissa/ significant | Description |
|---|---|---|---|---|---|---|
| **SHORT REAL** | 32 | 1 | 8 | 7F | 23 | **The first 1 in mantissa is suppressed i.e. not part of the 23 bits** |
| **LONG REAL** | 64 | 1 | 11 | 3FF | 52 | **The first 1 in mantissa is suppressed i.e. not part of the 52 bits** |
| **TEMPOR-ARY REAL** | 80 | 1 | 15 | 3FFF | 54 | **No suppressed bit i.e. all the 64 bits are explicit including the 1 preceding the binary point** |

From table 1.4.1, the Sign bit is represented as 0 for positive constant value and 1 for negative constant value. The Exponent bias is calculated as $(2^{n-1} - 1)$, where n = no of exponent bits. Converting numbers to any floating point system is calculated in the examples that follow:

**Example 1:**

Convert $5.375_{10}$ to short real format

**Solution:**

Step 1 The value will first be expressed in binary

Step 2 Convert $5.375_{10}$ in binary

$5.375_{10} = 101.011_2$

Step 3 Adjust the binary point to place the first 1 on the left of the binary point and put the exponent accordingly to reflect the binary point movement.

$101.011^2 = 1.01011 \times 2^{+2}$

Step 4 Put the sign bit value. Since the constant value is positive, hence sign bit is 0

Step 5 calculate the exponent:

= bias + 2
= 7F + 2
= 81 (Hex)
= 10000001
(express in 8 bits binary equivalent)

Step 6 calculate the mantissa: 23 bits with 1 before binary point suppressed

= 0101 1000 0000 0000 0000 000

Step 7 Add all the values together:

1 SignBit + 8 BiasedExponentBits + 23 MantissaBits = 32 bits altogether

= 0 10000001 0101 1000 0000 0000 0000 000
5.37510 = 40AC000016

**Example 2:**

Convert $1234.5625_{10}$ to long real format

**Solution:**

Step 1 The value will first be expressed in binary

Step 2 Convert $1234.5625_{10}$ to binary

$1234.5625_{10} = 10011010010.1101_2$

Step 3 Adjust the binary point to place the first 1 on the left of the binary point and put the exponent accordingly to reflect the binary point movement.

$10011010010.1101_2 = 1.00110100101101 \times 2^{+10}$

Step 4 Put the sign bit value. Since the constant value is positive, hence sign bit is 0

Step 5 calculate the exponent
Exponent: 11 bits
exponent is $10_{10} = 0A_{16}$
= bias + 0A
= 3FF + 0A
= 409 (Hex)
= 10000001001 (express in 11 bit binary equivalent)





Step 6 calculate the mantissa: 52 bits with 1 before binary point suppressed

= 0011 0100 1011 0100 0000 0000 0000 0000 0000 0
000 0000 0000 0000

Step 7 Add all the values together:

1 SignBit + 11 BiasedExponentBits + 52 Mantissa Bits = 64 bits altogether

= 0 100 0000 1001 0011 0100 1011 0100 0000 0000 0000
0000 0000 0000 0000 0000 0000

$1234.5625_{10} = 40934B4000000000_{16}$

**Example 3:**

Convert $5.375_{10}$ to temporary real format

Step 1 The value will first be expressed in binary

Step 2 Convert $5.375_{10}$ in binary

$5.375_{10} = 101.011_2$

Step 3 Adjust the binary point to place the first 1 on the left of the binary point and put the exponent accordingly to reflect the binary point movement.

$101.011_2 = 1.01011 \times 2^{+2}$

Step 4 Put the sign bit value. Since the constant value is positive, hence sign bit is 0

Step 5 calculate the exponent:

= bias + 2
= 3FFF + 2
= 4001 (Hex)
= 100 0000 0000 0001
(express in 15 bit binary equivalent)

Step 6 calculate the mantissa: 64 bits including the 1 before the binary point

= 1010 1100 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000

Now put everything together as follow

1 SignBit + 15 BiasedExponentBits + 64 MantissaBits = 80bits altogether

= 0 100 0000 0000 0001 1010 1100 0000 0000 0000 0000 000
0 0000 0000 0000 0000 0000 0000 0000 0000 0000

$5.375_{10} = 4001AC00000000000000_{16}$

## Summary

Method of floating point conversion were discussed in this unit. The three format as recognized by IEEE754 standard were also discussed. Several examples of how to do the conversion to short, long and temporary real were explained.

## Self-Assessment Questions

SAQ

1. Define floating point
2. Convert 5.37510 to short real format
3. State the three types of floating system format

## Tutor Marked Assessment

- Convert 4.37510 to long real format
- What is the difference between short and long real format
- Discuss the IEEE 754 Standard for floating point system

## References


- Kann, Charles W., "Introduction To MIPS Assembly Language Programming" (2015). Gettysburg College Open Educational Resources. 16-35
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall. 5-25
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.






## Further Reading

- http://cupola.gettysburg.edu/oer
  http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2


05 | **Picture:** Register

**Photo**: Wikipedi.com


## Module 2

# Registers

## Units

**Unit 1-** Registers

**Unit 2-** Different categories of registers and their functions

06 | **Picture:** Floating point operation

**Photo**: Unsplash.com

UNIT 1

# Registers

## Introduction

Assembly Language programming involves the use of registers for most of their variable declaration. In this unit you will learn all the types of registers, their specific functions and how it can be used in assembly language.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Define a register
2. State importance of registers in Assembly language

## Main Content

SAQ 1,2,3&4

### Registers

3 mins


You should note that a processor or microprocessor does two operations or functions. It processes information stored in RAM and also manages external devices such as the keyboard, disks and other peripheral devices. Majorly, the processor tasks mostly involve processing of data. These data can be stored in the memory and accessed from it. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the data through the same channel. To accelerate the processor operations, the processor includes some internal memory storage locations called Registers. Registers are memory locations inside a microprocessor which have particular storage capacity.

Registers are like special storage containers located in a processor. The registers form a storage system which is completely different from the RAM or memory. Access to registers is much faster than the RAM, it takes less time and their organization is also simpler. In other words, instructions which use registers demand less running time and less memory both of these are major virtues for a programmer. They store data elements for processing without having to access the memory. Therefore, variables are often kept in registers, especially when writing programs in Assembly language.





Values that exist directly in the CPU in order to do anything useful with data values in memory most especially in assembly language, they must first be loaded into registers. Registers are necessary for the CPU to operate on data, and that there are a limited number of them. Number of registers is very limited and is carefully allocated and controlled. There are some registers that are mainly used for a specific purpose, and the rules governing them must be adhere to.

These registers are at once the foreman's pockets and the foreman's workbench. When the CPU needs a place to tuck something away for a while, an empty register is just the place. The CPU could always store the data out in memory, but that takes considerably more time than tucking it in a register. Because the registers are actually inside the CPU, placing data in a register or reading it back again from a register is fast. But more important, registers are the foreman's workbench. When the CPU needs to add two numbers, the easiest and fastest way is to place the numbers in two registers and add the two registers together. For instance, it can make the sum (in usual CPU practice) replaces one of the two original numbers that were added, but after that the sum could then be placed in yet another register or added to another number or register, or stored out in memory, or any of a multitude of other operations.

The CPU's immediate work-in-progress is held in temporary storage containers called registers. Work involving registers is always fast, because the registers are within the CPU and very little movement of data is necessary.

Like memory cells and, indeed, the entire CPU, registers are made out of transistors. But rather than having numeric addresses, registers have names such as AX or DI. To make matters even more complicated, while all CPU registers have certain common properties, some registers have unique special powers not shared by other registers. Most peripherals also have registers, and peripheral registers are even more limited in scope than CPU registers.

## Summary

Registers plays an important role in writing an Assembly language. Majority of the variables used are been stored in the registers. All these registers have their own specific functions in terms of usage in writing the program.

## Self-Assessment Questions

1. _____ are like special storage containers located in a processor a. Registers b. RAM c. ROM
2. Access to registers is much _____ than the RAM
a. Slower b. Faster c.
3. _____are often kept in registers, especially when writing programs in Assembly language. a. RAM b. Variables c. ROM
4. _____ and _____ are both major virtues for a programmer
a. Less running time and memory b. less memory and registers
c. less organization and running time.

## Tutor Marked Assessment

- What are the important of registers in writing Assembly language
- Define Registers

## References


- Detmer, Richard C. (2001). 'Introduction to 80x86 Assembly Language and Computer Architecture' Jones and Bartlett 0-7637-1773-8. 50-92
- Jeff Duntemann (2000). Assembly Language Step-by-Step: Programming with DOS and Linux, Second Edition. John Wiley & Sons. ISBN:0471375233







- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Vitally Maljugin, Jacov Izrailevich, Semyon Lavin and Aleksandr Sopin (1993). The Revolutionary Guide to Assembly Language, 1st edition, Wrox Press.

https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm


## Further Reading

- http://cupola.gettysburg.edu/oer
http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2

**07** | **Picture:** Different categories of register

**Photo**: Wikipedi.com

UNIT 2

# Categories of Registers and their Functions

## Introduction

In this unit, various types of registers and their functions will be discussed. It also entails what type of register can be used for a particular operation. Registers are a limited number of memories

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. List the three categories of Registers
2. States the types of register
3. Mention the functions of each register

## Main Content

SAQ 1,2,3,4&5

# Different Categories of Registers

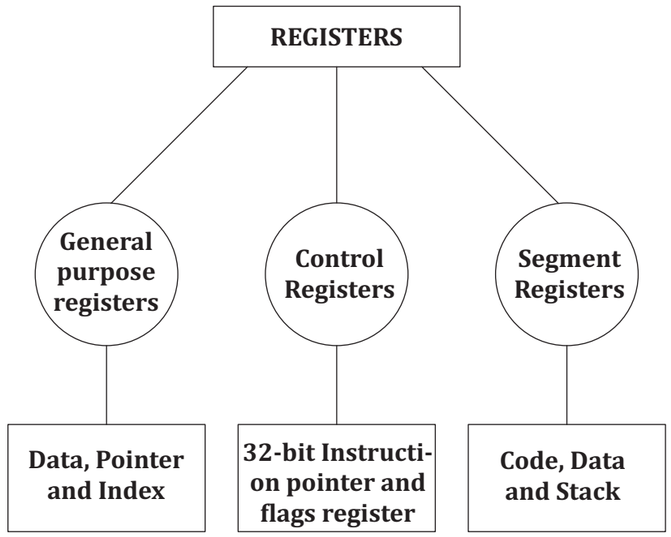
6 mins

Registers are to be used for certain purposes, and the rules governing the role of the register should be followed strictly. Registers are grouped into three major categories.

These are:

1. General Purpose Registers
2. Control Registers
3. Segment Registers

Each of these categories will be discussed in details. Figure 2.2.1 shows a diagrammatic representation of these registers.








## Figure 2.2.1: Categories of Registers

### General Purpose Registers

The general purpose registers is further divided into the following groups as shown in figure 2.2.1

a. Data Register
b. Pointer Register
c. Index Register

### Data Register

The main function of registers in this data group is to contain the initial data for the result of arithmetic or logical instructions. The registers in this group can be used as follows:

- As a complete 32-bit data registers which is written as: EAX, EBX, ECX, EDX.
- As a Lower half of the 32-bit registers which can be used as four 16-bit data registers and is written as AX, BX, CX and DX.
- As a Lower and higher halves of the above-mentioned four 16-bit registers which can be used as eight 8-bit data registers and is written as: AH, AL, BH, BL, CH, CL, DH, and DL.

Each of these data group registers can be treated as consisting of two 8-bit registers. A lower part is from 0 to 7 bit and a higher part is from 8 to 15 bit. The rule for naming 8-bit registers is to add letter H to the higher part or letter L to the lower part of the first letter of the corresponding 16-bit register name. Thus, AX 16 bit has AH and AL has the 8-bit register where AL denotes the lower part of 0-7 bit and AH denotes the higher part of 8 to 15 bit.

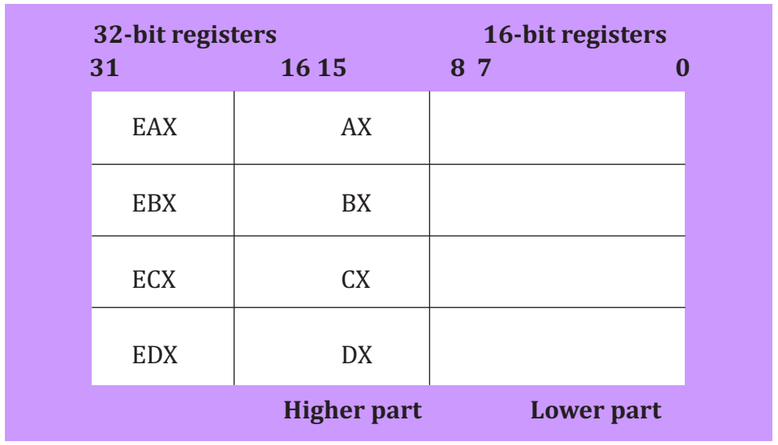
Intel 8086 processor has 16 bit data registers; each register has its own name as shown in figure 3:

**AX-** the accumulator register (divided into AH and AL which are 8bit each)

**BX-** the base register (divided into BH and BL which is 8bit each)

**CX-**the count register (divided into CH and CL which is 8bit each)

**DX-** the data register (divided into DH and DL which is 8bit each)




## Figure 2.2.2: Data registers

AX is the primary accumulator; it is used majorly for input/output operation and most arithmetic instructions. It is more efficient to use AX register for arithmetic and logical computations. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

**BX is known as the base register:** It is used in indexed and indirect addressing.

**CX is known as the count register:** ECX, CX registers store the loop count in iterative operations. That is, it is used to count off the number of iterations in a loop.

**DX is known as the data register:** It is also used in input/output operations. It is also used along with AX register for multiplication and divides operations involving large values. It can also be used to holds the overflow from certain arithmetic operations.





# Pointer Register

There are three categories of the pointer register. These are:

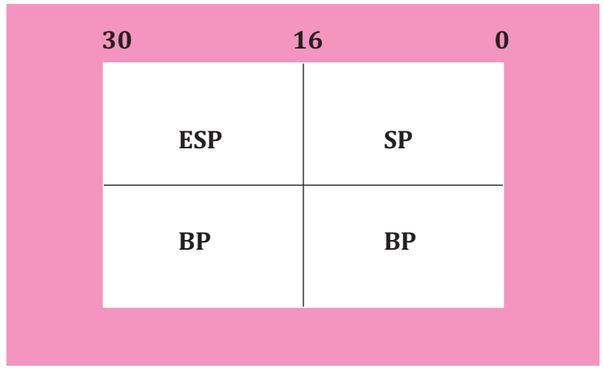
1. Instruction Pointer (IP, 16 bit)
2. Stack Pointer (SP, 16bit)
3. Base Pointer (BP, 16bits)

There are also 32-bit pointer register which is depicted as EIP, ESP and EBP

**Instruction Pointer (IP) -** The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) provides the complete address of the current instruction in the code segment.

**Stack Pointer (SP) -** The 16-bit SP register keeps the offset value within the program stack. SP in association with the SS register (SS:SP) refers to the current position of data or address within the program stack.

**Base Pointer (BP) -** The 16-bit BP register generally helps in referencing the parameter variables passed to a subroutine, in other words the BP determines the value of parameter variable passed to a subroutine. The address in SS register is merged with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.




## Figure 2.2.3: Pointer registers

## Index Register

The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index registers:

**Source Index (SI) -** It is used as source index for string operations

**Destination Index (DI) -** It is used as destination index for string operations

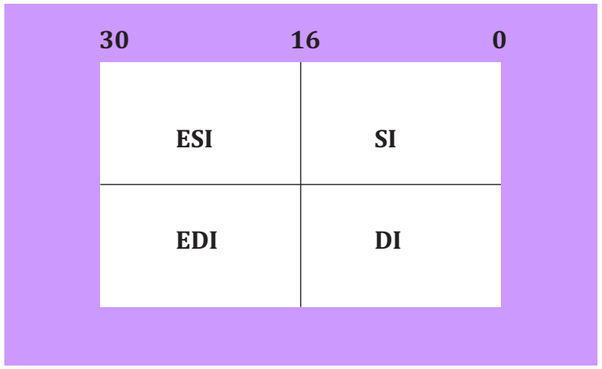



**Figure 2.2.4: Index registers**

## Control Registers

The 32-bit instruction pointer register and 32-bit flags register are combined to form the control registers. Several instructions and operations involve comparisons and mathematical calculations, which often change the status of the flags and some other conditional instructions that test the value of the status of these flags to take the control flow to other location.

Some of the common flag bits are:

**Overflow Flag (OF):** It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.





**Direction Flag (DF):** This is use to determine the left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

**Interrupt Flag (IF):** This determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.

**Trap Flag (TF):** It allows the setting of operation of the processor in single-step mode. The DEBUG program, used sets the trap flag, so we could step through the execution, one instruction at a time.

**Sign Flag (SF):** It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

**Zero Flag (ZF):** This is used to indicate the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.

**Auxiliary Carry Flag (AF):** It contains the carry from bit 3 to bit 4 following an arithmetic operation and it is used for specialized arithmetic operation. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

**Parity Flag (PF):** It shows the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits set the parity flag to 1.

**Carry Flag (CF):** This contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotates operation.

Table 2.2.1 indicates the position of flag bits in the 16-bit Flags register:

## Table 2.2.1: Control Registers

| Flag: | - | - | - | - | OF | DF | IF | TF | SF | ZF | - | AF | - | PF | - | CF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit no: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**Note:** CF=Carry Flag, PF=Parity Flag, AF= Auxillary Flag, ZF=Zero Flag, SF=Sign Flag, TF=Trap Flag, IF=Interrupt Flag, DF= Direction Flag, OF=Overflow Flag. The following illustrates the function of these flags in terms of on (1) or off (0).

- **CF: Carry Flag** = 1, If last operation resulted in a carryout of msb
  = 0, If last operation does not resulted in a carryout of msb

- **PF: Parity Flag** = 1, If lower 8-bit of result of last operation has even numbers of 1's
  = 0, If lower 8-bit of result of last operation has odd numbers of 1's

- **AF: Auxiliary Flag** = 1, If there is a carry from the third bit to the fourth bit in a Binary coded operation. For example Arithmetic operation.
  = 0, If there is no carry from the third bit to the fourth bit in a Binary coded operation

- **ZF: Zero Flag** = 1, if the result of last operation is 0
  = 0, if the result of last operation is not zero

- **SF: Sign Flag** = 1, if the result of last signed operation is negative number
  = 0, if the result of last signed operation is positive number

- **TF: Trap Flag** = 1, causes an interrupt after a single instruction is executed. When it is 0 a continuous execution of instructions takes place.

- **IF: Interrupt Flag** = 1, Enables hardware interrupt, when it is 0 interrupt is disabled.

- **DF: Direction Flag** = 1, Registers SI and DI are automatically decremented in a string operation. If DF=0, SI and DI are automatically incremented.
  OF, Overflow Flag =1, if the result of a 2's complement operation is out of range otherwise is 0.





## Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:

**Code Segment (CS):** It comprises of all the instructions to be executed. A 16-bit Code Segment register or CS register stocks the starting address of the code segment.

**Data Segment (DS):** It comprises of data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.

**Stack Segment (SS):** It comprises of data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Apart from the DS, CS and SS registers, there are other extra segment registers with 32-bits value - ES (extra segment), which provide additional segments for storing data.

In an assembly language program, each program needs to access the memory locations. All memory locations within a segment are relative to the starting address of the segment. Therefore, segment registers stores the starting addresses of a segment. To get the exact location of data or instruction within a segment, an offset value (or displacement) is required. To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location. Table 2.2.2 represents the segment registers while figure 2.2.5 depicts the architectural layout of Intel 8086 registers.

## Table 2.2.2: Segment registers

| 15 0 | Description |
|---|---|
| CS | Code Segment |
| DS | Data Segment |
| ES | Extra Segment |
| SS | Stack Segment |

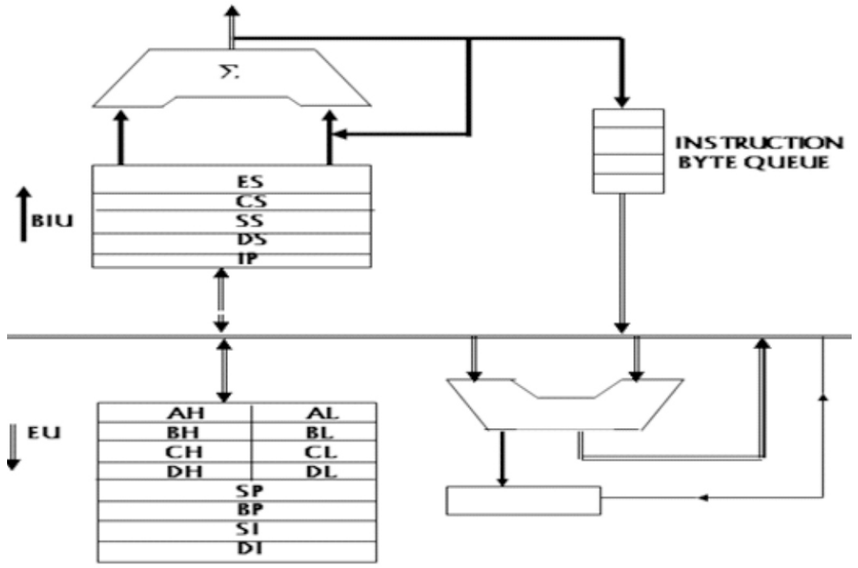



**Figure 2.2.5: CISC based INTEL 8086 Architectural Layout**

## Summary

Registers are classified based on their functions into three major categories. Each of these categories has their functions that are peculiar to them. For instance, data registers are known for their usefulness in arithmetic operations, such as addition, subtraction and multiplication. All of these were discussed in this unit





## Self-Assessment Questions

SAQ

1. _____, _____ and _____ are the three categories of registers
a. General purpose, control and segment registers b. General purpose, data and pointer registers c. General purpose, control and data registers
2. _____ is an example of a data register with 16 bits
a. EAX b. IP c. AX
3. _____ is a register that shows the total number of 1-bits in the result obtained from an arithmetic operation.
a. Parity Flag b. Overflow flag c. Directional Flag
4. BX is a base register used in_____ and _____ addressing.
a. Indexed and Indirect b. Indexed and direct c. Direct and Indirect
5. Trap flag set to 1 will causes an interrupt after a single _____ is executed
a. Program b. Instruction c. variable

## Tutor Marked Assessment

- Draw a diagrammatic representation of types of registers
- List three examples of pointer registers and control registers
- Explain the major functions of Accumulator register

## USE THIS TO ANSWER QUESTIONS 4 and 5

```
section.text
global start
start:
mov edx, len
mov ecx, msg
mov ebx, 1
mov eax, 4
```

```
int 0x80
mov eax, 1
int 0x80
section.data
msg db 'This is my first Assembly language program',  0xa
len equ $-msg
```

- How many registers are used in the program
- What happens to the value of len in line 4 after its execution?
- What type of addressing modes is used in the program

## References


- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.
- Vitally Maljugin, Jacov Izrailevich, Semyon Lavin and Aleksandr Sopin (1993) The Revolutionary Guide to Assembly Language, 1st edition, Wrox Press.

http://unaab.edu.ng/wp-content/upload/2009

https://www.tutorialspoint.com/assembly_programming/

https://www.coursehero.com/file/p64


## Further Reading

- https://www.tutorialspoint.com/assembly_programming/


08 | **Picture:** Addressing mode

**Photo**: Wikipedi.com

Module 3

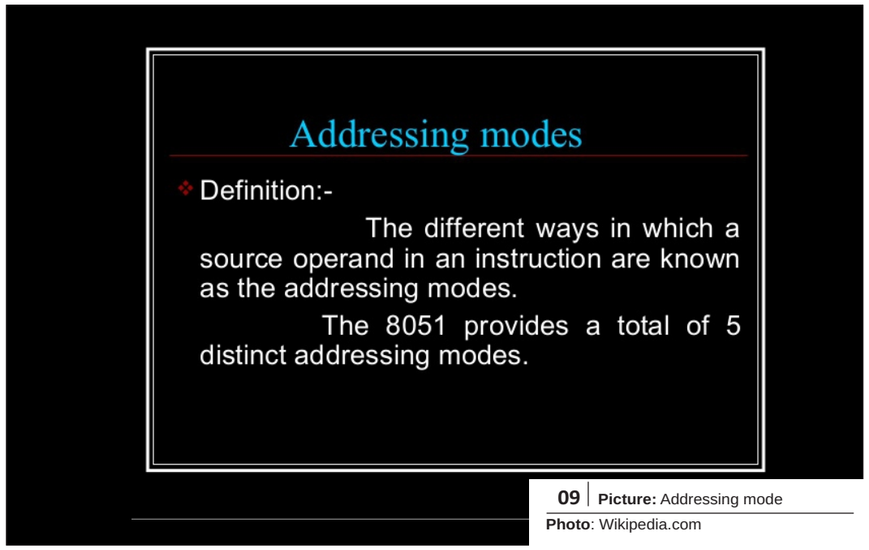
# Addressing Modes

## Units

**Unit 1-** Addressing Modes

**Unit 2-** Types of addressing modes

**Unit 3-** Intel 8086 physical memory addressing




09 | **Picture:** Addressing mode
**Photo**: Wikipedia.com

UNIT 1

# Addressing Modes

## Introduction

In this unit, all addressing modes in Intel 8086 will be discussed as well as the reasons why addressing modes are important in Assembly language.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Define Addressing modes
2. Recognize operands in an instruction
3. State the reasons why Addressing modes are important in Assembly Language

## Main Content

SAQ 1,2,3,4&5

# Addressing Modes

3 mins

Most Assembly Language instruction demands that our operands should be processed (operand specifies which objects are processed by instruction). An operand address provides the location where the data to be processed is stored. Some instructions do not involve the use of an operand, whereas some may require one, two or three operands. When an instruction requires two operands, the first operand is generally the destination which contains the data in a register or memory while the second operand is the source which contains either the data to be delivered or the address of the data in register or memory. Generally, the instructions of your program will be accessing registers and memory, and the mechanisms by which this is done are called the addressing modes.

Addressing modes are principles used in AL program to specify how certain values are read within a given set of assembly instruction. In other words, an operation may require zero, one, or two operands but the various ways of specifying the required operand(s) is known as the Addressing modes.

Generally, addressing modes differ from one processor to another, however, for the purpose of this course; we will discuss the addressing modes of INTEL 8085/8086 processor.

Addressing modes in INTEL 8086 is categorized into two, which are:

1. Memory Referencing Addressing modes and
2. Non-memory referencing addressing modes.

Operands in memory or I/O ports can only be accessed through memory referencing addressing modes while those in registers are access through non-memory addressing modes. All these types will be discussed in the next unit.





## Summary

You should note that, in writing an Assembly language, the arrangement of a line of instruction refers to addressing modes. It is through this arrangement that we are able to distinguish the types of addressing modes used in an instruction. Every instruction consists of an operand, which specified the address of where to store the data to be processed.

## Self-Assessment

SAQ

1. _____ are principles used in AL program to specify how certain values are read within a given set of assembly instruction. a. Addressing Mode b. Addressing Modes c. Mode of Addressing
2. An operand address provides the location where _____ to be processed is stored a. Data b. Information c. operand
3. INTEL 8086 addressing mode is categorized into _____ ___a. Three b. Two c. Four
4. In a line of instruction, the first operand is called _____ a. Source b. Data c. Destination
5. In a line of instruction, the first operand is called _____ a. Source b. Data c. Destination

## Tutor Marked Assessment

- List the two categories of Intel 8086 Addressing Modes.
- Arrangement of an instruction in Assembly language is achieved through addressing modes. Discuss.

## References

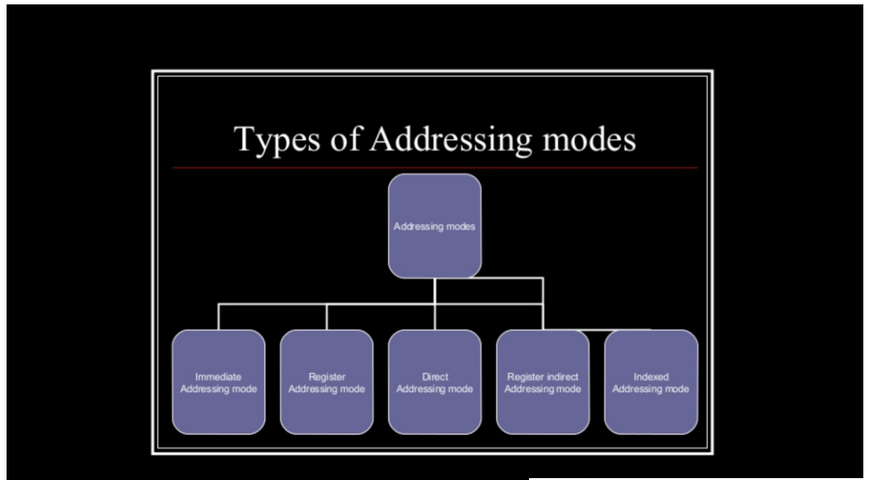

- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Vitally Maljugin, Jacov Izrailevich, Semyon Lavin and Aleksandr Sopin (1993) The Revolutionary Guide to Assembly Language, 1st edition, Wrox Press.

https://www.tutorialspoint.com/assembly_programming/assembly_quick_guide.htm

https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm


## Further Reading


- http://cupola.gettysburg.edu/oer
http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2




10 | **Picture:** Type of addressing mode
**Photo**: Wikipedia.com

UNIT 2

# Types of Addressing Modes

## Introduction

In this unit you will be introduced to the different types of addressing modes, their examples and how it can be used in a line of instruction.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Know all the types of Addressing modes
2. State the reasons why Addressing modes are important in Assembly Language

## Main Content

SAQ 1,2,3,4&5

# Types of Addressing Modes

3 mins

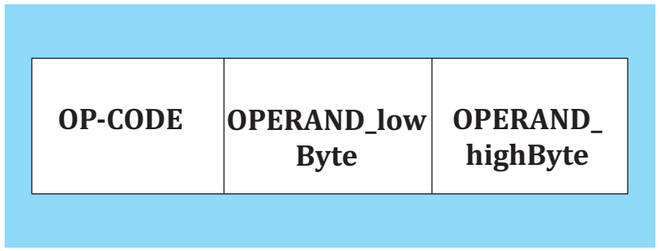
As earlier stated, only the various types of addressing modes in INTEL 8086 processor will be discussed. There are two categories, namely

I. Memory referencing and
ii. Non-memory referencing. Each of these categories is further sub-divid -ed as follows:

## Non-Memory Referencing Addressing Mode

In this type of addressing mode, Operands are available within the microprocessor's registers. Hence, no extra bus cycles are needed to retrieve the data. This is also divided into two:

(1) **Immediate addressing:** In immediate addressing mode, the 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. That is, the operand needed is contained in the instruction. The memory image of an immediate addressing instruction is shown in figure 3.








## Examples of Immediate Addressing modes are:

- MOV AX, 3FF2(the source operand (3FF2) is immediate addressing)
- MOV AX, 45d (Immediate constant 45d is transferred to AX)
- ADD bytevalue, 65 (Operand 65 is added)

**In this example, the constant values in them make it to be immediate addressing**

Note that immediate addressing cannot be used as destination (they are always the source operand).

(2) **Register Addressing:** In this type of addressing mode, a register contains the operands. Depending on the instruction, the register may be the first operand (destination), second operand (source) or both.

## Examples of such addressing modes are:

- MOV DX, tax (Register is the first operand or destination operand)
- ADD Sum, AX (Register is the second operand or source operand)
- MOV EAX, EBX (Both the destination and source operands are registers)

**Memory Reference Addressing Modes:** In this addressing mode, operand used in the instruction is in the main memory. This is divided into six, namely:

(I) Direct Memory Addressing
(ii) Register indirect Addressing
(iii) Base relative Addressing
(iv) Index relative addressing
(v) Base Index Addressing
(vi) Base index relative Addressing

### (1) Direct Memory Addressing:

In this type of memory reference addressing mode, the number given in the operand part of an instruction is the actual address of the operand to be used. Direct addressing is simple, fast and effective.

**Examples:** MOV cx, [34E2] (the number represents the actual address of the operand)

MOV bl, [3E4F]

MOV ax, [500F]

## (2) Register Indirect Addressing:

In this type of addressing mode, it is the register that contains the memory address of an operand. In other words, register indirect addressing mode is the opposite of direct addressing mode since the actual address is in the register. It is inside the register, that the address is stored. The required address is not often specified in register indirect addressing. The registers that can be used for this are BX, BP, DI or SI.

**Examples:** MOV [BX], al

MOV [DI], ah

| OP-CODE | MOD R/M<br>Where:<br>EA = ( BX , BP ) or (SI, DI) |
|---|---|

**Where R denotes Register, M denotes Memory and EA denotes Effective Address**

## (3) Base Relative Addressing:

The base relative addressing uses the same rules with register indirect only that a 16bit or 8bit displacement sometimes call value will be added to the register. Note that, It is only BX or BP register that can be used for base relative addressing.

| OP-CODE | MOD R/M<br><br>BX, or BP reg. | Displacement |
|---|---|---|

**Example:** EA = BX + disp or EA = BP + disp.
MOV ax, [BX+10] ; EA=BX+10
MOV ah, [BP+10]





## (4) Index relative addressing:

In this addressing mode, the register SI or DI is added with the displacement. Index addressing is the same with base relative addressing only that the registers used are different

| OP-CODE | MOD R/M<br><br>SI, or DI reg. | Displacement |
|---|---|---|

EA = SI + disp. or EA = DI + disp.
MOV ax, [SI+10] ; EA=BX+10
MOV ah, [DI+10]

## (5) Base index addressing:

In base index, the base and index registers can be used for the address but the displacement will not be there. It is a combination of only base and index register without displacement.

| OP-CODE | MOD R/M<br><br>BX, or BP |
|---|---|

| OP-CODE | +<br><br>SI, or DI reg. |
|---|---|

**Example:** MOV ax, [BX + SI] ; EA = BX + SI
MOV ax, [BP + DI]

## (6) Base index relative addressing:

The displacement or relative will be added to the base index addressing.

| OP-CODE | MOD R/M<br><br>BX, or BP<br>+<br>SI, or DI reg. | Displacement |
|---|---|---|

EA = BX or BP + SI or DI + disp
**Example:** MOV ax, [BX + SI+10] ; EA = BX + SI
MOV ax, [BP + DI+10]

## Summary

In this unit, we have discussed the different types of addressing modes in Intel 8086 and how you can differentiate them. Several examples of it were also given.

## Self-Assessment Modes

1. In immediate addressing mode, the immediate operand has a _____ value or an expression a. Constant b. Immediate c. Variable
2. Memory reference addressing mode is divided into _____ a. Seven b. Six c. Four
3. MOV ax, [BX + SI+10] is an example of _____ a. Base Index Relative b. Index relative c. Base relative
4. MOV DX, tax is an example of _____ a. Base relative b. Register c. Index relative
5. An example of an immediate addressing mode is _____ a. ADD ax, bx b. ADD bytevalue, 65 c. ADD bx, 65





## Tutor Marked Assessment

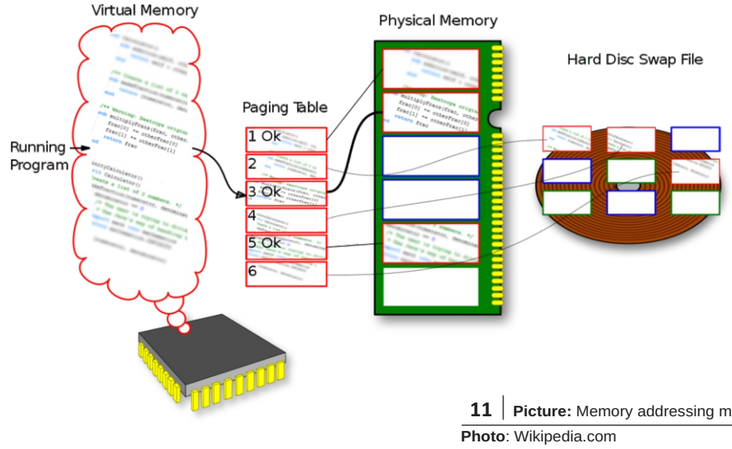
- With the aid of example, explain three of the memory referencing addressing modes
- What are the advantages of direct addressing modes over others?
- Differentiate between immediate addressing and register addressing.

## References


- Barry, B. B. (1997). The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Architecture, Programming and Interfacing, 4th Edition, Prentice-Hall Inc.
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.


## Further Reading

- http://cupola.gettysburg.edu/oer
  http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2




**11** | **Picture:** Memory addressing mode
**Photo**: Wikipedia.com

UNIT 3

# Intel 8086 Physical Memory Addressing

## Introduction

In unit 2 you have learnt the various types of addressing modes, their examples and uses in line of instruction. In this unit you will be introduced to the physical memory of the 8086 microprocessor, how to calculate both the logical and effective address.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify the diagrammatic representation of 8086 physical memory
2. Calculate the physical and effective address of 8086 processor

## Main Content

SAQ 1,2,3&4

# Intel 8086 Physical Memory Addressing

3 mins

Intel 8086/8088 is one of the processor that operates in real mode i.e. the processor can directly access maximum of 1Mbytes of main memory (i.e. 1 Mbyte = 220 ). The memory in 8086 is organized as segmented memory. In this way, the complete physical memory is divided into a number of logical segments. Each segment is 64kilobytes in size and is addressed by one of the segment registers. The Operand physical memory address is computed automatically by the Bus Interface Unit (BIU) of the processor from two sources, namely, segment register and offset address. The segment registers points to a particular segment in the memory while the offset address indicates the location in the segment. The Offset address is also known as effective address (EA).

The two sources contribute 16 bit address each called logical address. Physical address of 20 bits is obtained by multiplying the given segment address by 16 (in decimal) and adding the given offset address to it. -i.e. logical address contains 16bit segment register and 16 bit offset address or Effective address while Physical address = [(segment address) * 1610) + offset address]

The offset address values are from 0000H to FFFFH, so that the physical address ranges from 0000H to FFFFH. 20 bit physical address generation

Figure 3.3.1 shows the diagrammatic representation of 20 bit physical address of Intel 8086 processor.





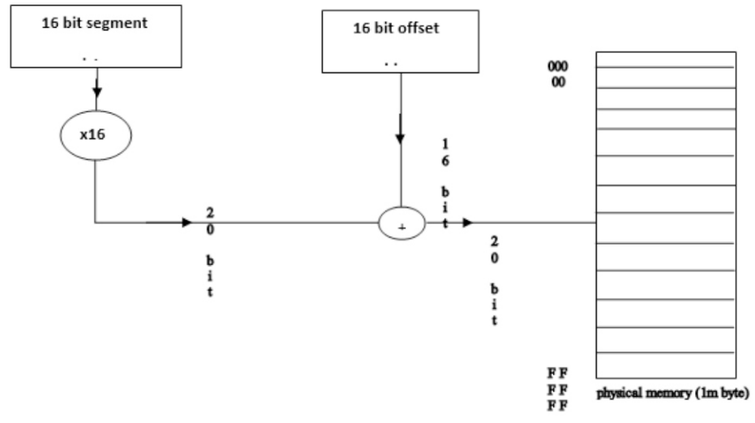



**Figure 3.3.1: Intel 8086 Physical Memory**

SAQ 5

## Examples on How to Calculate Physical Address Mode

(1) Find the physical memory address, given DS = $2E5F_{16}$ , and EA = $2382C_{16}$

**Solution:**

Physical Address: = [(segment address x $16_{10}$) + offset address (EA)]

= (2E5F16 x 1610 ) + ( 282C16 )

= ($2E5F_{16}$ x $10_{16}$ ) + ( $382C_{16}$ ) ($16_{10}$ is converted to base 16, so that all the parameters will be in the same base)

$$\begin{array}{rr} & = \ 2E5F \\ \times & \underline{\quad\quad 10} \\ & \quad 0000 \\ & \underline{\quad 2E5F\quad} \\ & 2E5F0 \end{array}$$

This will now be added to the offset Address

$2E5F0 + 282C = 30E1C_{16}$

The final answer is, which is the physical memory address is $30E1C_{16}$

(2) Find the physical memory address, given DS = $4FB6_{16}$ , and EA = $A82C_{16}$

**Solution:**
Physical Address: = [(segment address x $16_{10}$) + offset address (EA)]

= ($4FB6_{16}$ x 1610 ) + ( $A82C_{16}$ ) ($16_{10}$ is converted to base 16, so that all the parameters will be in the same base)

= ($4FB6_{16}$ x $10_{16}$ ) + ( $A82C_{16}$ )
= 4FB6
x 10
0000
4FB6
4FB60

This will now be added to the offset Address
4FB60 + A82C = $5A38C_{16}$
The final answer is, which is the physical memory address is **$5A38C_{16}$**

(4) Find the physical memory address, given DS = 8E3D, Offset address = B502, determine the corresponding physical address

**Solution:**
Physical Address: = [(segment address x $16_{10}$) + offset address (EA)]
= ($8E3D_{16}$ x $16_{10}$ ) + ( $B502_{16}$ )
= ($8E3D16$ x $10_{16}$ ) + ( $B502_{16}$ ) ($16_{10}$ is converted to base 16, so that all the parameters will be in the same base)
= 8E3D
x 10
0000
8E3D
8E3D0

This will now be added to the offset Address
8E3D0 + B502 = 898D216
The final answer is, which is the physical memory address is 898D216





## Summary

In this unit, we explained in details what physical memory and logical addresses are. This is also buttress with a diagram and calculations showing how physical and logic address are been calculated.

## Self-Assessment Questions

SAQ

1. The segment registers points to a particular _____ in the memory a. Segment b. Logic c. Physical
2. Another name for Offset address is _____ a. Logical address b. Effective address c. Physical address
3. Draw the diagrammatic representation of Intel 8086 physical memory
4. Physical memory address is obtained by _____ the given segment address by 16 (in decimal) and adding the given offset address to it. a. Multiplying b. Adding c. Dividing
5. Given DS = 2E5F16 , and EA = 582C16, . Calculate the physical memory address.

## Tutor Marked Assessment

- Write short notes on the following terms:
  a. Physical Addressb. Logical Addressc. Segment Address
- Given DS = 2E5F16 , and EA = 582C16, Find the physical memory address.
- Given DS = 8E2D, Offset address = B502, determine the corresponding physical address.

## References


- Barry, B. B. (1997). The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Architecture, Programming and Interfacing, 4th Edition, Prentice-Hall Inc.
- Carter P.A. (2003). PC Assembly Language. Paul carter, USA. 21-53
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.


## Further Reading

- http://cupola.gettysburg.edu/oer
  http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2

Module 4

# Instruction Sets Intel 8086

## Units

**Unit 1-** Instruction sets and its types

**Unit 2-** Syntax and functions of Instruction sets



Picture:

Photo:


**12** | **Picture:** Instruction sets
**Photo**: Unsplash.com

UNIT 1

# Instruction Sets and its Types

## Introduction

In this unit, we will be studying the various instruction sets for Intel 8086 from their categorization up to the various types under each category.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify all the instruction sets in Intel 8086
2. State the functions of an instruction set
3. Identify an instruction set in a line of instruction

## Main Content

SAQ 1,2,3&4

# Instruction Sets and its Types

5 mins

You should note that the language command of any computer architecture comprises of instruction and the instruction set refers to the vocabulary of that language. In other words, we can say an instruction set is a list of all the instructions, and all their variations, that a processor can execute.

There are different ways of grouping instruction depending on the type of machine in use. In Intel 8086, the following groupings are mostly used.

(I) Arithmetic Instructions
(ii) Data transfer Instructions
(iii) Bit manipulation Instructions
(iv) Program transfer Instructions
(v) String Instructions
(vi) Branch Instruction

**Figure 4.1.1** shows the diagrammatic illustrations of this instruction set

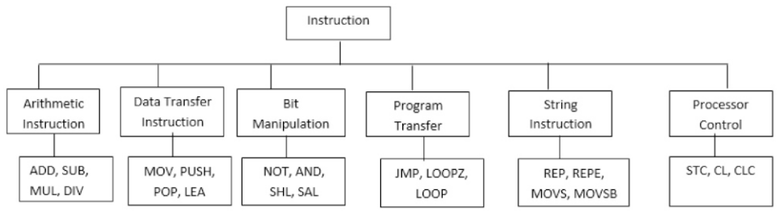



**Figure 4.1.1: Instruction set in 8086**

## Arithmetic Instructions

You should be aware that the arithmetic instructions consists of Addition, subtraction, Multiplication and Division. The "ADD" symbol is used for Addition, "SUB" is used for Subtraction, "MUL" is used for Multiplication and "DIV" is used for Division.





The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte (8bits), half word (16bits), word (32bits), and double word (64bits).

Syntax: The ADD and SUB instructions have the following syntax

**ADD destination, source**

**SUB destination, source**

The ADD and SUB instruction can take place between register to register, memory to register, register to memory, register to constant data and memory to constant data. However, memory to memory are not allowed in ADD and SUB instruction. In other words, an operation involving Addition and subtraction should not have the source and destination operands to be memory.

**Examples:**

- ADD usword, bx
- SUB al, prf

Note that, The ADD instruction can also add the source and destination operands and places the sum in the destination operand. i.e destination= destination + source

## MUL and DIV instruction

The MUL instruction is used for multiplication and DIV for division. Both the multiplicand and multiplier are stored in registers depending on their size and the product generated is also stored in a register. If it is byte, register AL can be used but if it is word then AX will be used.

The DIV operation will generates two elements- Quotient and Remainder.

## 3.1.2 Data Transfer Instructions

## MOV

The MOV instruction is one of the data transfer instruction that is used to copy data from the source to the destination operand. It's usually uses two operands.

Syntax: MOV destination, source

The MOV instruction can have the following formats:

MOV register, register
MOV register, memory
MOV register, immediate
MOV Memory, register
MOV memory, immediate

**Examples:** MOV AX, BX
MOV ax, sum

## PUSH

It is used to copy the operand into the stack. The stack pointer is decreased and the source value is copied to the top of the new stack. The operand may be a segment register, a memory location or a general purpose 16-bit register. A PUSH instruction will always have a single operand.

**Examples:**

PUSH DX PUSH EDX PUSH temp

## POP

The POP instruction pops the top of the stack into the destination operand. The Stack Pointer is incremented by 2 to point to the new top of the stack. The destination operand of the POP instruction can be any 16-bit registers, memory location or segment register, except CS.

e.g: POP SI POP BP

LEA (Loads Effective Address)

The LEA instruction loads the offset address of the source operand into the destination operand. However, In LEA, the destination operand cannot be a segment register.

LEA destination, source.

LEA DX, Alrmsg





## LAHF

It load register AH from the processor's flags i.e the flags SF,ZF,AF,PF,CF are copied to bits 7,6,4,2,0 of register AH

SAHF

It store register AH into flags i.e bits 7,6,4,2,0 of register AH are copied into SF,ZF,AF,PF,CF of the processors flag but DF, TF, IF are not affected.

INC and DEC 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.

It's syntax: INC destination

The destination operand can be 8-bit, 16-bit and 32-bit registers.

**Example:**

INC BX ; Increments 16-bit register

INC DL ; Increments 8-bit register

The DEC instruction is the opposite of INC instruction. It is used to decrement an operand by one.

**Example:**

DEC CX ; Decrements 16-bit register

DEC [count] ; Decrements count variable

## ORG Directive (ORIGIN)

This instruction ORG directive causes the object codes of the program to be loaded into memory starting from the location address specified in the ORG instruction.

Syntax: ORG expression ; where expression is a constant or evaluate to a constant.
**Example:** ORG 5000h; this will cause the object program to be loaded into address 5000h
CMP Instruction (Compare)

It will interest you to note that CMP Instruction is usually used when comparing two operands and used in conditional execution. It will subtract the source from the destination and adjust the flag accordingly. The CMP instruction normally has their destination to be register and source can either be register, memory or Immediate. It is used to compare a byte or word source operand with that of the destination.

## Bit Manipulation Instruction

Instructions in this group are used to manipulate bits within a byte or word for operand individually. This group is further divided into Logical, Shift and Rotate. The Logical instruction is used as a Boolean operator and consists of NOT, AND, OR, XOR, TEST. The shift is used to move the registers in different direction and it's consist of SHL, SAL, SHR, SAR etc and the Rotate is for rotation, examples of this are: ROL, ROR, RCL, RCR.

Examples of Bit manipulation instruction in a line of instruction are:

```
NOT   dst   ; takes 1's complement of a byte or word, dst= reg. or mem.
        NOT    ax
        NOT    table
AND  dst , src   ; logical AND of a byte or word operand
       Example:   AND   al, bl
                  OR    al, bl
```

## String instructions

Instructions in this category are: REP, REPE, REPNZ, MOVS, MOVSB, MOVSW, CMPS, SCAS, LODS, and STOS. Table 4.1.1 show a list of some of this instruction.





## Table 4.1.1: String instructions

| Mnemonic | Description |
|---|---|
| REP | Repeat for a specified number of iteration value in register CX. |
| REPE<br>REPZ | REPE and REPZ are the same. Repeat while the ZF =0. The maximum number of iterations should be specified in register CX. |
| REPNE<br>REPNZ | REPNE and REPNZ are the same. Repeat while ZF ≠ 0. The maximum number of iterations should be specified in register CX. |
| MOVS | Moves a string of bytes or words from location to another in memory. |
| SCAS | Scans string for specified values. |
| CMPS | Compare values in source string with values in destination string. |
| LODS | Loads values from a string to accumulator register |
| STOS | Store values from accumulator register to a string |

## Table 4.1.2: String Instruction Registers & Use

| Register/Flag | Use |
|---|---|
| SI | Source string offset |
| DI | Destination string offset |
| CX | Repetition Counter |
| AL<br>AX | Scan value register, Source for STOS, Destination for LODS. |
| DF | DF=0 means autoincrement of SI, DI<br>DF=1 means autodecrement of SI, DI |
| ZF | Scan, and Compare terminator. |

## 3.1.5 Program Transfer Instructions

There are four subgroups of instructions that can be used to facilitate this program transfer of control. These are unconditional transfer, conditional transfer, iteration control, and interrupt instructions.

## Table 4.1.3: Unconditional Transfer Instruction Table

| Mnemonic & Operand | Description |
|---|---|
| CALL target | Call a procedure.<br>target = near_proc, far_proc, memptr16, regptr16, memptr32.<br>e.g. CALL near_proc<br>CALL [BX+SI+task] |
| RET<br>optional_pop_value | optional_pop_value = no_pop, intra_segment_pop, inter_seg._pop<br>e.g. RET<br>RET 4<br>RET 2 |
| JMP target | Jump uncoditionally to target location.<br>target = short_label, near_label, far_label, memptr16, regptr16, memptr32.<br>e.g. JMP short<br>JMP [BX+ label]<br>JMP far_label |

## 3.1.6 Processor Control Instructions

They are instructions that control CPU functions. Some of them are listed in the table 6





## Table 4.1.4: Processor control instructions with flag

| Mnemonic | Flag Operation |
|---|---|
| STC | CF = 1 ;set carry flag. |
| CLC | CF = 0 ; clear carry flag. |
| CMC | CF = ! CF; complement carry flag. |
| STD | DF=1 ; set direction flag. |
| CLD | DF=0 ; clear direction flag. |
| STI | IF =1 ; set interrupt enable flag. |
| CLI | IF =0 ; clear interrupt enable flag. |

## Table 4.1.5: Processor control instruction without flag

| Mnemonic | External Synchronization |
|---|---|
| HLT | Halt CPU until interrupt or reset. |
| WAIT | Wait for TEST pin to be active low. |
| ESC | Escape to external processor. |
| LOCK | Lock bus during next instruction. |
| | |
| NOP | No Operation causes the CPU to nothing. |

## Summary

In this unit, we explained in details the instructions sets and their examples in a line of instruction.

## Self-Assessment Questions

1. What is the function of the MOV instruction a. Copying from source to destination operand b. Copying from destination to source destination c. Copying from source to destination and store the result in source operand
2. Operation involving Addition and subtraction instruction should not have the source and destination operands to be _____
a. Register b. Memory c. Constant
3. A PUSH instruction will always have _____ operand
a. Double b. None c. Single
4. What is the function of LEA instruction

## Tutor Marked Assessment

- In a tabular form, list three data transfer instructions and state their functions
- What is the function of ORG instruction? Illustrate with an example.

## References


- Barry, B. B. (1997). The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Architecture, Programming and Interfacing, 4th Edition, Prentice-Hall Inc.
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall. 5-25







- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.


## Further Reading

- http://cupola.gettysburg.edu/oer
http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2

13 | **Picture:** Syntax and function
**Photo**: Unsplash.com

UNIT 2

# Syntax and Functions of Instruction Sets

## Introduction

In this unit, we will be studying the various ways of using instruction sets for Intel 8086 discussed in the previous unit and specify how it can be used in a program.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify the syntax for writing an instruction sets in Intel 8086
2. State the functions of an instruction set
3. Recognize an instruction set in a line of instruction

## Main Content

# Syntax and Functions of Instruction Sets

3 mins

Syntax is the rules guiding the writing of any programming language. They help to specify how certain instruction set can be written and used in Assembly language. Any violation of any of these rules might render the program to be error prone and malfunctioning. In this unit, some of the instruction sets identified in the previous unit will be taken one after the other, state their syntax and how it is being used in assembly language program.

SAQ: 1

# Allocating Storage Space for Data Initialization

1 mins

Assembly language allows the declaration of storage space for variables to be used in a program. This is achieved with the use of directives meant for storage space allocation. There a quite a number of this directive but for this course only five that are peculiar to 8086 will be discussed. Allocating storage is achieved with the use of 'define' directive depending on the amount of storage space needed. The five directives and their space are explained in table 4.3.1 as follows:

## Table 4.2.1: Directives and their Storage Space

| Directives | Storage Space | Function |
|---|---|---|
| DB | Define Byte | It will allocates 1 byte = 8 bits |
| DW | Define Word | It will allocates 2 byte = 16 bits |
| DD | Define Double | It will allocates 4 byte = 32 bits |
| DQ | Define Quadword | It will allocates 8 byte = 64 |





| | | |
|---|---|---|
| | | bits |
| DT | Define Ten bytes | It will allocates 10 byte = 800 bits |

Define directive syntax is as follows:
[variable-name] define directiveinitial valueWhere [variable-name] is the identifier for each storage to be allocated, define directive specifies the type of directive to be used and initial value is the value to be in the storage space.
Examples of assembly language program with define directive:

**1.sum DB 'a'**

In this example, sum is the identifier, the define directive DB will allocate a space of 1 byte and 'a' is the value.

**2.Chose number DW 1457**

Choose number is the identifier, the define directive DW will allocate a space of 2byte and 1457 is the initial value.

**3.Float numbers DD 1.2788**

Float numbers is the identifier, the define directive DD will allocate a space of 4 byte and 1.2788 is the initial value.

SAQ: 1, 3&4

# Instructions for Constant values

5 mins

There are three ways of normally declaring constant values in Assembly program. It can be declared with the use of EQU, %assign and %design.

The EQU is used for assigning constant and its syntax is: CONSTANT_NAME EQU expression where CONSTANT_NAME is the name of the constant to be declared Expression is the value to be in the constant. Examples of how EQU can be used are as follow:

- Number STUDENTS EQU 70. In this example, 70 will be the constant value stored in Number STUDENTS.
- Length rec EQU 7
- Breadth recEQU 6
- Area recLength rec * Breadth rec

In example 2 above, values 7 and 6 are constant values allocated to Length rec and Breadth rec in the program. Therefore, the value of Area rec will be 42 when the program is run.

The %assign directive can be used to define numeric constants like the EQU directive. The only difference is that the %assign directive allows redefinition. For example, you may define the constant Number student as follow:

%assign Number student 20

Later in the program, it can be redefined as

%assign Number student 30

The %define Directive

The %define directive allows defining both numeric and string constants to be declared in a program. This directive works similarly with #define in C. For example, you may define the constant SUM as:

%define SUM [EIP+4]; SUM will have the value [EIP+4]

## Arithmetic Instructions

Some of the arithmetic instructions in assembly language are ADD, SUB, INC, DEC, MUL and DIV.
ADD and SUB: The two instructions used the same rules in writing them.
ADD Num1, Num2
SUB Num1, Num2. The Num1 and Num2 are the values to be added or subtracted.

1. ADD Sum, ax: This example will add register ax and Sum and put the value in Sum.
2. SUB alr, bx: subtract the register from alr and store the result in alr





Note that ADD/SUB arithmetic instructions do not bother whether your operands are 2's complement or unsigned numbers - that interpretation is up to the programmer to decide. It is only the size of the operands that are important in these calculations, and the instruction will determine that from the operands supplied.

INC and DEC: The two instructions are used to increase and decrease a value. It normally takes a single operand.
INC destination_Operand: The operand to be incremented is destination_Operand
DEC destination_Operand: The operand to be decremented is destination_Operand

1. INC bx: Increment the value of 16-bit register with 1
2. DEC bl:: Increment the value of –bit register with 1

## Logical Instructions

The logical instruction set provides the instructions AND, OR, XOR, TEST and NOT Boolean logic, this tests, sets and clears the bits according to the need of the program. The format for these instructions is:

```
AND:    AND operand1, operand2
OR:     OR operand1, operand2
XOR:    XOR operand1, operand2
NOT:    NOT operand1, operand2
```

Operand1: 0101Operand2: 0011
AND operand1. This will result in operand1 having 0001 has the value after the AND operation
OR operand1. This will result in operand1 having 0111 has the value after the OR operation
XOR operand1: This will result in operand1 having 0110 has the value after the XOR operation
NOT operand1: This will result in operand1 having 1010 has the value after the XOR operation

## Conditional Instructions

You should note that the conditional instruction in assembly language is usually accomplished by several looping and branching instructions. The conditional instruction normally changes the flow of control in a program. It is either an unconditional or conditional jump. The unconditional jump will use the JMP instruction to transfer control to the address specified. The Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps while the conditional jump is achieved by using a set of jump instructions depending upon the condition. Its normally break the sequential flow of program execution.

JMP label: JMP is the instruction for unconditional while label is the label number where the control will transferred to.

```
A20:
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 A20 ; repeats the statements
```

This will transfer the control of program back to the label named A20

## String Instructions

This group of instructions are designed for moving blocks of data from one place in memory to another, and some of them are for searching through and comparing blocks of data.

The string instruction requires a source operand, a destination operand, or both. It normally uses the source and destination index register, 32-bit segments, string instructions will 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 Double word of data from memory location to another.

**CMP –** This will compare one operand from another depending on the condition set

The CMP instruction can be followed by a conditional jump can be made to jump or not depending upon the flags. Although CMP does the comparison by subtracting the two values hypothetically with their values left unchanged. CMP can be used as unsigned number or 2's complement.

**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 double word 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.

## Shift Instructions

The shift instructions help in the movement of strings. It consists of SHL (SHift Left), SHR (SHift Right), ROL (Rotate Left), ROR (Rotate Right), SAR (Shift Arithmetic Right).

SHL (SHift Left) and SHR (SHift Right) helps in moving the bit left or right. The SHL moves all bits in AL one place to the left while SHR moves all bits in AL one place to the right while 0 is into the most significant bit (MSB). This instruction is sometimes used to test individual bits. The shift operations can also be on 16-bit or 32-bit registers. The two instructions works in opposite way, such that SHL does exactly the opposite of SHR, moving zeros into the significant bit LSB (least significant bit) and the MSB out to the carry flag.

**Examples:** mov bx, 01010011 shl 3' will moves the binary number '01010011000' (LSB) with three 0s added into register bx

mov bx, 01010011 shr 3' will moves the binary number '00001010011' (MSB) with three 0s added into register bx.

ROL (Rotate Left) and ROR (Rotate Right) work the same way with shift instructions, except what falls out is rotated around back in the other end. Thus the contents are never lost, but circulate around the register. ROL is the mirror-image of ROR, sending the MSB to the carry flag and back around to the LSB.

## Summary

In this unit, the rules for writing each of the instruction were discussed. We also studied how some of them can be used in a program through several examples. explained in details the instructions sets and their examples in a line of instruction.

## Self-Assessment Questions

1. DB can be used to _____
2. The LODS Operation will _____
3. Operand1: 0001Operand2: 0111

From the values giving above, if AND operand1 instruction is used. What will be the value?

4. What is the difference between EQU and %assign?





## Tutor Marked Assessment

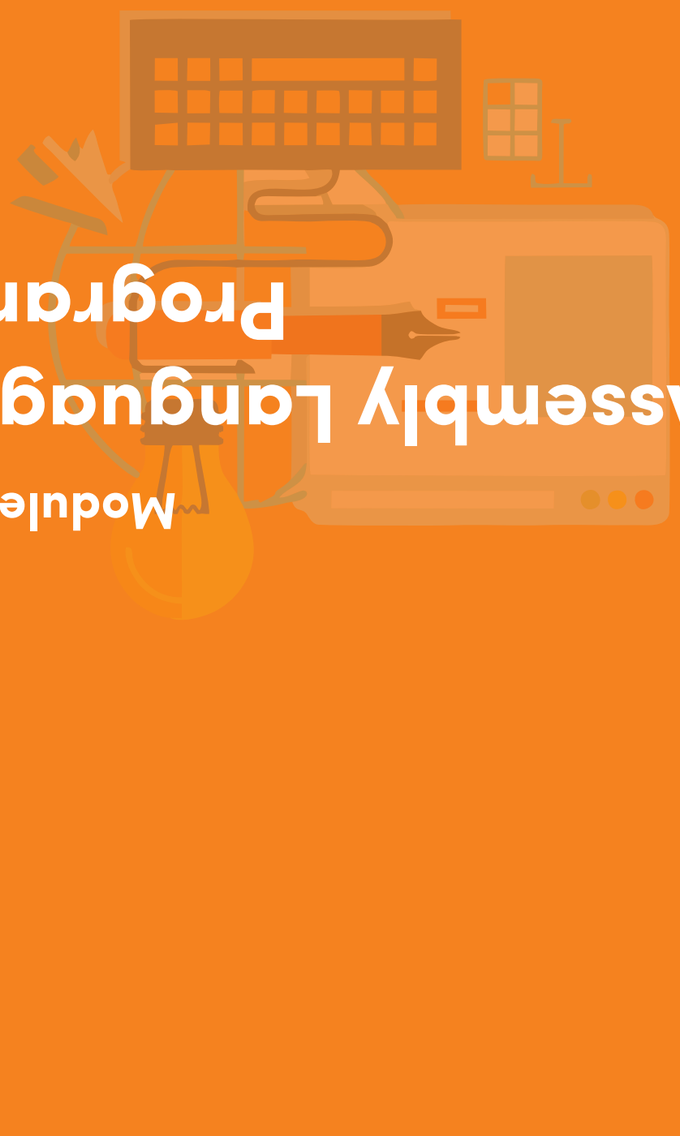
- Write a line of instruction using the following instruction set
  a. EQU b. %define c. CMPd. JMP
- Operand1: 0001 Operand2: 0111
  What will be the value of XOR operand1 and OR Operand1
- Using SHR and SHL operation, what will be the value of i. mov, ax, 00010111 shr 2, ii. mov, ax, 00010111 shl 2

## References


- Barry Kauler. (1997). Windows Assembly Language & Systems. 2nd edition, R&D Books. 25-56
- Barry, B. B. (1997). The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Architecture, Programming and Interfacing, 4th Edition, Prentice-Hall Inc.
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.


## Further Reading

- http://cupola.gettysburg.edu/oer
  http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2



Module 5

# Assembly language Program

## Units

**Unit 1**-Introduction to MASM Assembler

**Unit 2**-Format of an Assembly Language program

**Unit 3**-Sample of programs in Assembly language

**14** | **Picture:** Initial saver setup

**Photo**: Wikipedi.com

15 | **Picture:** Introduction to Microsoft

**Photo**: Wikipedi.com

UNIT 1

# Introduction to Microsoft Macro Assembler (MASM)

## Introduction

In this unit we will discuss the main features of MASM assembler, its format and how to recognize a line of instruction in MASM.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify the features of MASM Assembler
2. Recognize a line of Instruction in MASM Assembler

## Main Content

SAQ 1,2,3&4

# Introduction to MASM Assembler

3 mins

Try to know that assembler is the standard by which all other assemblers for the IBM PC are measured. It has gone through several versions, through adding of features and removing of bugs in the process. It is part of a sophisticated development system that consists of an editor, a cross-reference generator, a librarian, a linker and a debugger (CodeView).

MASM which has its full meaning as Microsoft Macro Assembler is actually an assembler for the Intel 80x86 and 80x88 families of microprocessors, including the 80x87 coprocessors. An important MASM feature is the ability to maintain a library of pre-assembled programs. These can easily be located by the librarian, and loaded and linked with any new program. Some of its unique features are discussed below.

### 3.1.1 Features of MASM

### MASM Option

MASM is invoked by the MASM command, which specifies the names of all files involved (source, object, listing, and cross reference) and can select many options. The following is a list of some interesting options. Many options are available for the command line, so the /H option helps the user by displaying all the options on the screen. Therefore, the result of the command

'masm /h ...' is:
Usage: masm /options source(.asm),[out(.obj)],[list(.lst)],[cref(.crf)][;]
/a Alphabetize segments
/c Generate cross-reference
/d Generate pass 1 listing
/D<sym>[=<val>] Define symbol
/e Emulate floating point instructions and IEEE format
/I<path> Search directory for include files
/l[a] Generate listing, a-list all





/M{ lxu}Preserve case of labels: l-All, x-Globals, u-Uppercase Globals
/n Suppress symbol tables in listing
/p Check for pure code
s Order segments sequentially
/t Suppress messages for successful assembly
/v Display extra source statistics
/w{ 012}Set warning level: 0-None, 1-Serious, 2-Advisory
/X List false conditionals
/z Display source line for each error message
/Zi Generate symbolic information for CodeView
/Zd Generate line-number information

These are all options and it will be displayed on the screen. For instance, any information needed for A pass-1 listing can be created by the /d option.

## MASM Listing

Some of the listing produced by MASM contains the source and object codes and the source file line numbers. The line numbers are used by the debugger to direct the user to next lines. In addition, certain tables can optionally be listed, giving information about the macros, the structures and records, and the groups and segments used in the program. In addition to that, things such as the symbol table, pass-1 listing, and assembly statistics, can also be listed. A special utility, CREF, can be used to generate the cross reference information.

## MASM Source Line Format

In MASM source line, only the labels of instructions are required to terminate with a ':' (full colon). Labels of directives don't require a ':'. Lines are written in free format and may start in column 1 even if there is no label.

## MASM Directives

MASM supports more than 50 directives. Some directive names must start with a period, but there does not seem to be any consistency. A few of the more interesting ones are described here.

The DF (Define Far) directive allocates 6 bytes in memory. It is normally used on the 80X86 microprocessor to store a pointer.

A question mark '?' is used to indicate a zero-value, and the notation '(?)' indicates undefined value. Thus 'x db ?,?,?' allocates 3 bytes with zeros value.

## MASM Expressions

MASM can handle expressions with many operators. Operands must be absolute, except that the binary '+' operator can operate on one absolute and one relative operand, and the binary '-' operator can, in addition to that, be used to subtract two relative operands (but only if they are located in the same segment). Indexing is denoted by square brackets, and there can be several of them in the same expression. Thus 'movah,array[bx][di] ' is valid.

Expressions can be shifted by the shl & shr operators. They are not the 80x86 shift instructions (even though they have the same name), and are executed at assembly time. A typical example is 'mov bx, 01010011 shl 3' which moves the binary number '01010011000' into register bx, while mov bx, 01010011 shr 3' will moves the binary number '00001010011' into register bx.

The not, and, or & xor logical operators can be used in expressions, and are thus executed at assembly time.

## MASM Macros

Assembly language allows macros. A macro can be defined as a set of pre-defined set of code. It instruction requests the assembler to process a specific named macro. As a result of this, the assembler generates machine and assembler instructions which are then processed as if they were part of the original source module. As can be expected, MASM supports an extensive macro facility. Macros can have many parameters (as many as will fit on one line), both macro expansions and macro definitions can be nested, and an extensive set of conditional assembly directives permits recursive macros. Macros can be redefined and even deleted to make room for new definitions. An interesting point is that a macro can purge itself, but the purge command must be the last line of the macro.





## Summary

In this unit, we have discussed majority of the features of MASM as an assembler were learnt. We discussed about the syntax of writing a line of instruction in MASM as well as some of its unique features among other Assembler.

## Self-Assessment Questions

SAQ

1. State two features of MASM assembler
2. MASM is invoked by the MASM _____
3. A question mark '?' is used to indicate a _____ in MASM assembler a. Zero-value b. One-value c. Two-value
4. MASM support more than _____ directives a. 80 b. 60 c. 50

## Tutor Marked Assessment

Complete this table based on the features of MASM

| S/N | FEATURES | MASM |
|---|---|---|
| 1. | Developer | |
| 2. | Option | |
| 3. | Listing | |
| 4. | Source Line Format | |
| 5. | Directives | |
| 6. | Expressions | |

## References


- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.
- SYS-ED/ Computer Education Techniques, Inc. 7-27

  https://vdocuments.mx/assembly-language-5584561cd9278.html

  https://vdocuments.mx/microprocessor


## Further Reading

- https://vdocuments.mx/assembly-language-5584561cd9278.html

## Format of Assembly language

- General format for an assembly language statement
- Label Instruction Comment
- Start: Mov AX, BX ; copy BX into AX

Start is a user defined name and you only put in a ***label*** in your statement when necessary!!!!

The symbol : is used to indicate that it is a label

**16** | **Picture:** Format of an assembly language

**Photo**: Wikipedi.com

UNIT 2

# Format of an Assembly Language Program

## Introduction

In this unit we will discusses how Assembly language is being written, the format that must be followed as well as its syntax.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify all the fields required in writing an assembly language
2. Recognize the fields in a line of an instruction

## Main Content

SAQ 1,2,3,4&5

### Formats of an AL program

3 mins

Did you know that AL was the first programming language that relieved programmers from writing in zeros and ones and enabled them to use meaningful names for instructions and operands? In fact, the first assembler was simply a system for representing machine instructions with simple mnemonics or symbols. Its main advantages were in automatically assigning values to instruction addresses and operation codes. Other advantages are:

1. AL require less memory and execution time
2. It allows hardware specific complex jobs in an easier way
3. It is suitable for time critical jobs
4. It is suitable for writing interrupt service routines and other memory resident programs.

Instructions and directives are the main element of AL program statements. Directives are mostly used to control the assembly process while instructions are actually executed by the processor when the program runs. (i.e. runtime).

You should note that each line of the instruction in AL consists of four fields namely:

1. **LABEL**
2. **OPCODE**
3. **OPERAND**
4. **COMMENT**

**A label:** is the first element in a line of an instruction. In other words, it is the name given to a line containing an instruction. It is optional and is for reference purpose. To create labels, the characters A-Z can be used. The rules for creating labels are slightly different depending on the type of assemblers you are using. However, most PC assemblers, allows label to be separated from an instruction code by a colon ":" and labels cannot start with numbers or special characters. E.g.





**A:**
**Start:**
**Income:**

**The opcode:** (Operation code) specifies the operation to be performed. It defines which machine command is generated by assembler when it process this statement. The opcode is divided into two: Machine Opcode (Mnemonics) and Pseudo opcode (Assembler directives). The machine opcode is the one that requires generation of machine codes at assembly time e.g LDA, JMP, MOV, ADD, MUL etc while pseudo opcode or assembler directives does not generate machine codes at assembly time. Examples of assembler directives are END, HLT.

**The operand:** field indicates the address of the memory location in which the operand on which the operation to be performed is stored. In other words, the operand specifies which objects are processed by this instruction. Some AL instructions do not have operand while those that have can accept a variety of different kinds. Some operands can refer to the registers or memory address, some may contain initial data for the instruction being performed and some may indicate where a result should be stored. Examples:

ADD ax, sum (ax and sum are the operands in this line of instruction, while ax is register, sum is memory address which indicate where the result will be stored)

**Comments:** are used in AL programs to aid users in reading through the program. Comments in AL program begin with a semi colon ";", it is usually at the end of a line and can be put either after the last operand of an instruction or on a separate line. It is important to add comment in an AL program for it to be easily understood even though it is optional.

It is worthy to note that, not all the four fields are required in an AL instruction. Label, Operands and comment are optional because there are some lines of instructions which require only opcode. Each line of an AL program must be separated by at least one blank character or a space.

Examples of standard line of instruction in AL program are as follows:

1. A: Push ax; Place content of ax onto a stack (there is label, opcode and operand in this example)
2. section.data: Mov dx, len ; Message length (there is label, opcode and two operands in this example)
3. Mov bx, sum (there is no label and comment in this example)

## Summary

You have been able to learn that for every programming language, there are usually format that needs to follow and this is also applicable to Assembly language. The format for AL requires four major fields, some are compulsory while others are optional. Examples of all these are shown in the contents of the unit.

## Self-Assessment Questions

1. State the four fields required in writing Assembly language
2. _____ is the compulsory field required in a line of instruction
a. Operand b. Opcode c. Comment
3. Each line of instruction in AL consists of _____ fields
a. Five b. Six c. Four
4. Label is optional and is use for _____ purpose. a. reference
b. labeling c. Content
5. The machine opcode is the one that requires generation of _____ at assembly time a. machine codes b. operation codes
c. opcodes

## Tutor Marked Assessment

- Briefly explain the four fields require in an Assembly language
- Give three examples of AL where the four fields are used





## References


- https://www.tutorialspoint.com/assembly_programming/assembly_quick_guide.htm


## Further Reading

- https://www.tutorialspoint.com/assembly_programming/assembly_quick_guide.htm

```
/*      BIN1.C          M Bates        Version 1.0

        Program to output a binary count to Port B LEDs

*****************************************************************************/

#include <p18f4580.h>                   /* Include port labels for this chip */
#include <delays.h>

int counter                             /* Label a 16-bit variable location  */

void main(void)                         /* Start main program sequence       */
{
        counter = 0;                    /* Initialise variable value         */
        TRISB = 0;                      /* Configure Port B for output       */

        while (1)                       /* Start an endless loop             */
        {
                PORTB = counter;        /* Output value of the variable      */
                counter++;              /* Increment the variable value      */
                Delay10KTCY(100);       /* Wait for 100 x 10,000 cycles      */
        }
}                                       /* End of program                    */
```

**17** | **Picture:** Assembly language program

**Photo**: Wikipedi.com

UNIT 3

# Sample of Programs in Assembly Language

## Introduction

In this unit you will be discussing the intrigues of writing assembly language. Some examples will be used to expatiate this.

## Learning Outcomes

**At the end of this unit, you should be able to:**

1. Identify the two sections in an assembly language program
2. Recognize the fields in a line of an instruction
3. Write a simple Assembly Language program

## Main Content

SAQ 1,2&3

# Sample of Programs in Assembly Language

4 mins


It will interest you to know that writing Assembly Language involves the division of the program in to two sections, namely Data and Code sections. Each line of instruction will now contain all the fields required in writing the program. To start with, we need to download MASM V6.11 and its programming manual. After that, it will be installed on the system. Then, we can open it and start writing our assembly language.

**Example 1:** Write an assembly language program for addition of two 8-bit numbers

**Solution:**
Since it is an 8-bit numbers then, Registers AL and BL will be used
Description: The Two 8-bits numbers are added and the result is stored in memory location Sum.
Below is the program

```
Data segment
First DB?
Second DB?
Sum DB ?
Data ends
Code segment
 Assume cs:code, ds:data
start:
      Mov AL, first
Mov BL, second
Add AL, BL
Mov Sum, AL
hlt
code ends
end start
```

The result in Sum will be the value provided for AL and BL.

**Example 2:** Write an assembly language program to add two 16-bit numbers

**Solution:**
This addition involves 16-bit numbers, therefore, a 16-bit registers will be used.
Registers: AX, BX
Description: Two 16-bits numbers are added and the result is stored in memory location Add.

```
data segment
first dw ?
second dw ?
AddVal dw 0
data ends
code segment
Assume cs:code, ds:data
_start:
mov ax, first
mov bx, second
add ax, bx
mov AddVal, ax
hlt
code ends
end _start
```

the result above will display addition of any two numbers entered.

**Example 3:** Write an assembly language program to find the factorial of N number

**Solution:**
Registers: AL, BL will be used
Description: Factorial of N number is computed and the result is stored in a pointer

```
data segment
```

n DB ? ; n is the number that will want to find its factorial
fact DB 2 dup(0) ; Variable fact is declare of byte type in two times and initialize them to zero
data ends
code segment
Assume cs: code, ds: data, es: data
_start:
mov al, 01 ; immediate addressing mode, with loading of 01 into register al.
mov bl, n
look: mul bl ; multiply the content of memory n with accumulator
dec bl ; decrement the content of register bl
jnz look ; goto look if register content of bl is not equal to zero
mov fact, al ; move the value of register al to memory location fact
hlt
code ends
end _start
The result above will display the factorial of any number entered

**Example 4:** Write an assembly language program to find the sum of squares of numbers from "N" to 1

**Hint:** $\sum_{\square=1}^{\square} \square^2$

**Solution:**
Registers: AL, CL
data segment
num db ?
sum db 1 dup(0) ; this is also the same as sum db 0
data ends
code segment
Assume cs: code, ds: data
start:
mov cl, num
mov sum, 0
L1: mov al, cl
mul al

add al, sum
mov sum, al
loop L1 ; the content of cl is automatically decremented by 1 due





to loop
hlt
code ends
end _start
This program if compiled and executed, for instance if N value is 3, then $3^2 + 2^2 + 1^2 = 14$

**Example 5:** Write an AL to declare whether a number is Odd or Even

**Solution:**
section .text
global main ;must be declared for using gcc
main: ;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 run, for instance if 5 is entered it will display 'Odd Number' and if 8 is entered it will display 'Even Number'.

**Example 6:** Write an AL to print numbers from 1-10

**Solution:**

```
section .text
mov ecx,10
mov eax, '1'
repeat_code:
mov [numbers], eax
mov eax, 4
mov ebx, 1
push ecx
mov ecx, numbers
mov edx, 1
int 0x80
mov eax, [numbers]
sub eax, '0'
inc eax
add eax, '0'
pop ecx
loop repeat_code
mov eax,1
int 0x80
section .bss
num resb 1
```

When the above code is compiled and run, it produces this result:
123456789

## Summary

This unit illustrates samples on how to write an assembly language program. Starting from how to download, install and run it on the system. Samples program showing various examples were equally explained.





## Self-Assessment Questions

1. _____ and _____ are the sections involved in writing an Assembly Language a. Data and Code segment b. Data and Stack segment c. Code and Stack segment
2. In writing an Assembly language, involving 16-bit registers, what register can be used a. AX b. AL c. BH
3. Write assembly program to add two numbers together

## Tutor Marked Assessment

- Write an Assembly language program to add two 32-bit numbers and store the result in AddNum
- Write an Assembly language to print “Hello ! World”

## References


- Barry, B. B. (1997). The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Architecture, Programming and Interfacing, 4th Edition, Prentice-Hall Inc.
- Kip, R. I. (2007). Assembly Language for Intel-based Computers, 5th Edition, Pearson Prentice Hall.
- Richard, E. H. (1993). Assembly Language Tutor for IBM PC and Compatibles, Regents/ Prentice Hall.

## Further Reading

- http://cupola.gettysburg.edu/oer
  http://dx.doi.org/10.1016/B978-0-12-803698-3.00001-2