M.Sc. (IT) Final Year

MIT-11

MICROPROCESSOR AND ASSEMBLY LANGUAGE PROGRAMMING

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

COURSE WRITER

Vikas Kumar, Assistant Professor (IT), Asia Pacific Institute of Management, New Delhi

Units (1-5)

Published by Registrar, MP Bhoj (open) University, Bhopal in 2020

VIKAS®

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Microprocessor and Assembly Language Programming

Syllabi	Mapping in Book
UNIT - I **Introduction to Microprocessor:** Evolution of Microprocessor, Overview of Intel Pro-Pentium, Motorola 68000 Series, Introduction to DEC Alpha, PowerPC, RISC & CISC Architecture.	**Unit-1:** Introduction to Microprocessor **(Pages 3-31)**
UNIT - II **Basic Microprocessor Architecture and Interface:** Internal Architecture, External System, Bus Architecture, Memory and Input/Output Interface.	**Unit-2:** Basic Microprocessor Architecture and Interface **(Pages 33-88)**
UNIT - III **Programming Mode:** Register Organization of 8086, Memory Addressing and Instruction Formats, Memory Interfacing, Cache Memory and Cache Controllers.	**Unit-3:** Programming Mode **(Pages 89-142)**
UNIT - IV **Basic I/O Interface:** I/O Interface, 8255 Programmable Interface, 8254 Programmable Timer, 8251 Programmable/Communication Interface, Interrupts, 8259 Programmable Interrupts Controller.	**Unit-4:** Basic I/O Interface **(Pages 143-175)**
UNIT - V **8086 Assembly Language Programming:** Instruction Set of 8086, Assembler Directives and Operators, A Few Machine Level Programs, Machine Coding and Programs, Programming with an Assembler, Assembly Language Example Programs.	**Unit-5:** 8086 Assembly Language Programming **(Pages 177-212)**

CONTENTS

INTRODUCTION

NOTES

A microprocessor is a computer processor where the data processing logic and control is included on a single integrated circuit, or a small number of integrated circuits. The microprocessor contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's Central Processing Unit (CPU). The integrated circuit is capable of interpreting and executing program instructions and performing arithmetic operations. The microprocessor is a multipurpose, clock-driven, register-based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results (also in binary form) as output. Microprocessors contain both combinational logic and sequential digital logic, and operate on numbers and symbols represented in the binary number system. The integration of a whole CPU onto a single or a few integrated circuits using Very-Large-Scale Integration (VLSI) greatly reduced the cost of processing power. Integrated circuit processors are produced in large numbers by highly automated Metal-Oxide-Semiconductor (MOS) fabrication processes, resulting in a relatively low unit price.

In computer programming, assembly language (or assembler language), is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code. Assembly code is converted into executable machine code by a utility program referred to as an assembler. The computational step when an assembler is processing a program is called assembly time.

This book is designed to be a comprehensive and easily accessible book covering the basic concepts of microprocessor and assembly language programming. It will help readers to understand the basics of microprocessor, Intel Pro-Pentium, Motorola 68000 series, DEC Alpha, PowerPC, RISC & CISC architecture, basic microprocessor architecture and interface, bus architecture, memory and Input/Output interface, register organization of 8086, memory addressing and instruction formats, memory interfacing, cache memory and cache controllers, 8255 programmable interface, 8254 programmable timer, 8251 programmable/communication interface, interrupts, 8259 programmable interrupts controller, 8086 assembly language programming, and machine coding and programs.

The book is divided into five units that follow the Self-Instruction Mode (SIM) with each unit beginning with an Introduction to the unit, followed by an outline of the Objectives. The detailed content is then presented in a simple but structured manner interspersed with Check Your Progress to test the student's understanding of the topic. A Summary along with a list of Key Terms and a set of Self-Assessment Questions and Exercises is also provided at the end of each unit for understanding, revision and recapitulation.

UNIT 1 INTRODUCTION TO MICROPROCESSOR

Structure

1.0 INTRODUCTION

The microprocessor is the biggest invention of the 20th century, which has changed the strengths and capabilities of the modern computer and, in particular, the personal computer. The microprocessor is the brain of any computer. The Central Processing Unit (CPU) is an electronic device which can fetch and execute a set of instructions, carry out arithmetic and logical operations and can control the input/output devices. A central processing unit, fabricated on a single chip of semiconductor is called microprocessor.

The microprocessor has travelled a long journey from the 4-bit device to the present-day 64-bit architecture. There are a number of companies manufacturing microprocessors, with the Intel Corporation governing the world. Most of the people today are more familiar with the Intel microprocessors rather than other microprocessors. The DEC Alpha, also known as the Alpha AXP, was a 64-bit RISC microprocessor originally developed and fabricated by digital equipment corp. PowerPC is a microprocessor architecture that was developed jointly by Apple, IBM, and Motorola in early 1990. The PowerPC employs Reduced Instruction Set Computing (RISC). Two paradigms of the microprocessor design are very common depending upon the complexity of the instructions and these are referred to as the CISC and RISC paradigms.

In this unit, you will study about the introduction to microprocessor, overview of Intel Pro-Pentium, Motorola 68000 series, and significance of DEC alpha, PowerPC, RISC and CISC architecture.

1.1 OBJECTIVES

After going through this unit, you will be able to:

NOTES

- Understand the basics of microprocessor
- Explain the overview of Intel Pro-Pentium
- Discuss about the Motorola 68000 series
- Explain the significance of DEC alpha
- Analyse the PowerPC
- Define the RISC and CISC architecture

1.2 INTRODUCTION TO MICROPROCESSOR

A microprocessor is a computer processor where in the data processing logic and control is included on a single integrated circuit, or a small number of integrated circuits. The microprocessor contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's central processing unit. The integrated circuit is capable of interpreting and executing program instructions and performing arithmetic operations. The microprocessor is a multipurpose, clock-driven, register-based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results (also in binary form) as output.

What is a Computer?

A computer is an electronic device, operating under the control of instructions (software) stored in its own memory unit, which can accept data (input), manipulate data (process) and produce information (output) from the processing. Generally, the term is used to describe a collection of devices that function together as a system. The essential components of a computer (Refer Figure 1.1) are:

1. System Bus
2. Microprocessor
3. Memory Unit
4. Input/Output Unit

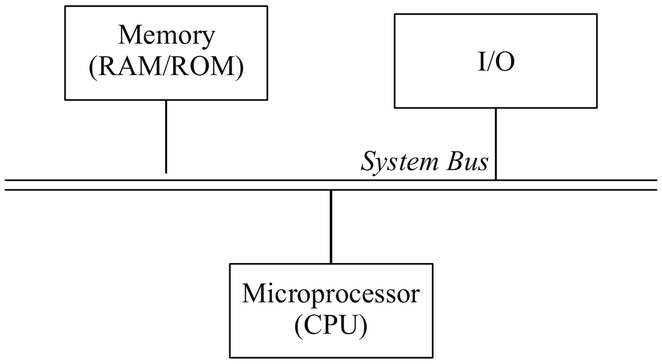


Fig. 1.1 *Components of a Computer*

The System Bus

The system bus is basically a collection of signal paths which are used to carry the electrical signals across different parts of the computer. This is further divided into,

- Address Bus
- Data Bus
- Control Bus

They together connect the microprocessor to each of memory and I/O (Input/ Output) elements, which facilitates the information transformation between them.

Microprocessor

The microprocessor is the brain of any computer. Technically defining, a CPU (Central Processing Unit) fabricated on a single chip is termed as microprocessor. The CPU is an electronic device which can fetch and execute a set of instructions, carry out arithmetic and logical operations and can control the input/output devices. The microprocessor is fabricated on a single chip using MOS (Metal–Oxide–Semiconductor) technology. It comprises of the following components:

(i) Register Unit
(ii) Arithmetic and Logical Unit
(iii) Timing and Control Unit

Memory Organization

The memory unit is an integral part of any microcomputer system and its primary purpose is to hold program and data. The major goal of the memory unit is to allow it to operate at a speed close to that of the processor. The cost factor inhibits the design of the entire memory unit with single technology that guarantees high speed. In order to seek a trade-off between the cost and operating speed, the memory system is usually designed with different technologies, such as solid state, magnetic and optical. In a broad sense, a microcomputer memory can be logically divided into following three groups:

(i) Processor Memory
(ii) Primary or Main Memory
(iii) Secondary Memory

Input/Output (I/O)

The microcomputer system communicates with the outside world via the I/O devices interfaced to it. The user can enter the program and data using the keyboard on the terminal and execute the program to obtain the results. Thus, I/O devices provide the efficient means of communication between the computer and the outside world. I/O devices are commonly called peripherals and include keyboard, CRT (Cathode Ray Tube) display, printers, disks, etc. The characteristics of I/O devices are normally different from those of the microcomputer. For example, the speed of operation of peripherals is usually slower than that of the microcomputer.

Digital Computer vs Microcomputer

The organization of I/O devices create a difference between a digital computer and a microcomputer. A computer which has the microprocessor as a CPU is called as a microcomputer. The analogy is illustrated in Figure 1.2.

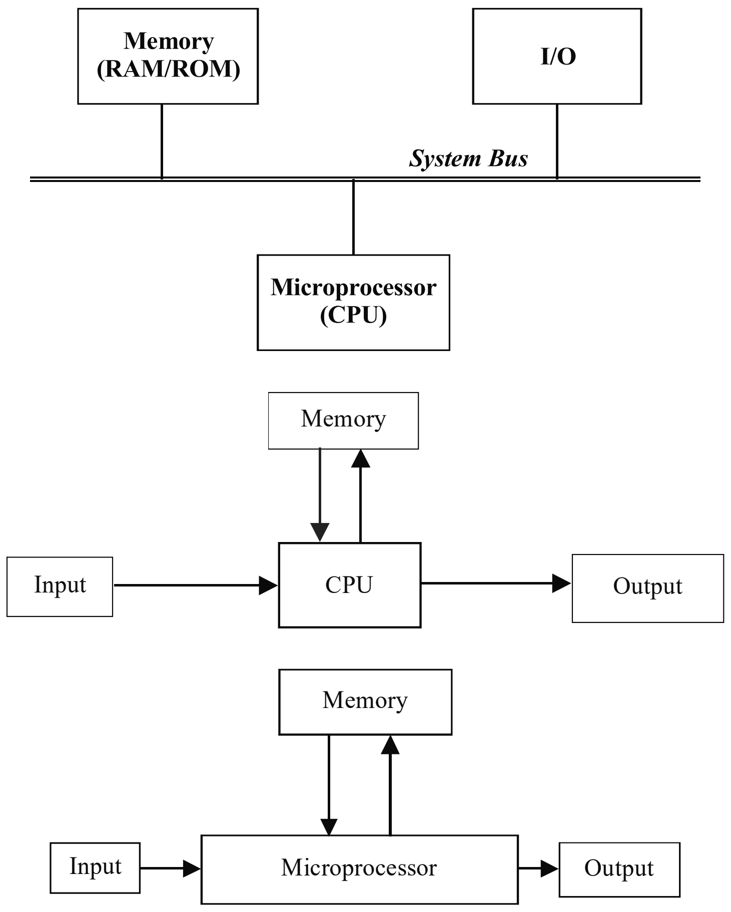


Fig. 1.2 *Microcomputer Versus Digital Computer*

1.2.1 Evolution of Microprocessors

Similar to the five generations of computers, the years of development of microprocessors can also be divided into five generations.

First Generation (1971–1973)

Intel Corporation introduced 4004, the first microprocessor, in 1971. It is evolved from the development effort while designing a calculator chip. The block diagram of Intel 4004 is shown in Figure 1.3. The layout of 4004 is shown in Figure 1.4.

NOTES

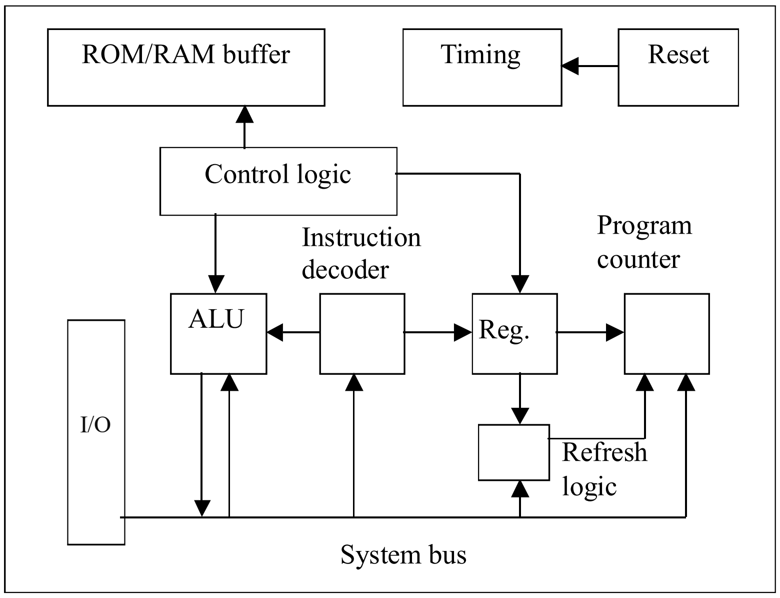


Fig. 1.3 *Block Diagram of Intel 4004*

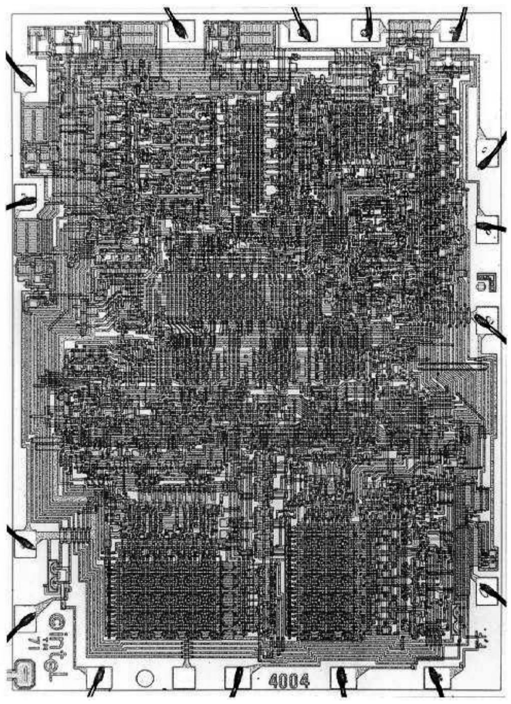


Fig. 1.4 *Layout of 4004*

The microprocessor 4004 was very successful in the calculator market at that time. However, there were three other microprocessors in the market during the same period:

- Rockwell International's PPS-4 (4 bits)
- Intel's 8008 (8 bits)
- National Semiconductor's IMP-16 (16 bits)

They were fabricated using PMOS (P-Channel Metal–Oxide–Semiconductor) technology, which provided low-cost, slow-speed and low-output currents. However, because of the technology, they were not compatible with TTL (Time to Live).

Second Generation (1974–1978)

The second generation, in fact, marked the beginning of very efficient 8-bit microprocessors. Some of the popular processors were:

- Motorola's 6800 and 6809
- Intel's 8085
- Zilog's Z80

They were manufactured using NMOS (N-Channel Metal–Oxide–Semicondcutor) technology. This technology offered faster speed and higher density than PMOS and also they were made to be TTL compatible. TTL was more common at that time.

Third Generation (1979–1980)

This age of microprocessors is dominated by 16-bit microprocessors. Some of them were:

- Intel's 8086/80186/80286
- Motorola's 68000/68010

They were designed using HMOS technology. HMOS (High–Performance Metal–Oxide–Semiconductor) provides some advantages over NMOS as Speed-power–product of HMOS is four times better than that of NMOS. This can also accommodate twice the circuit density compared to NMOS. Intel used HMOS technology to recreate 8085A and named it as 8085AH with a higher price tag.

Fourth Generation (1981–1995)

The fourth generation of microprocessors came really as a boon to the computing environment. This era marked the beginning of 32-bit microprocessors:

- Intel introduced 432, which was a tittle problematic.
- Then, a clean 80386 was launched.
- Motorola introduced 68020/68030.

They were fabricated using a low-power version of the HMOS technology called HCMOS.

Motorola introduced 32-bit RISC processors called MC88100, during the same time.

Fifth Generation (1995–Till Date)

The fifth generation includes chips that carry on-chip functionalities and improvements in the speed of memory and I/O devices along with introduction of 64-bit microprocessors. Intel leads the show here with Pentium, Celeron and very recently dual and quad core processors working at up to 3.5 GHz speed. A number of processors have been specifically launched for the wireless systems, with advances in the wireless technology. The growth of microprocessors is basically the result of VLSI (Very Large Scale Integration) Technology, because of which the actual size of the chip has shrunk with the integration of so many numbers of transistors on the same chip. The growth of VLSI technology has been listed as follows:

- Vacuum tube: 1946–1957
- Transistor: 1958–1964
- Small-scale integration: 1965 onwards
  – Up to 100 devices on a chip
- Medium–scale integration: till 1971
  – 100–3000 devices on a chip
- Large scale integration: 1971–1977
  – 3000–100,000 devices on a chip
- Very large scale integration: 1978–1991
  – 100,000–100,000,000 devices on a chip
- Ultra large scale integration: 1991–present
  – Over 100,000,000 devices on a chip

Transistors

Transistors are the fundamental components from which **logic circuits** are constructed. These logic circuits are, in turn, the basic building blocks of the CPU. They include devices called **gates** and **flip-flops**. (A single gate may use up to six transistors). Gates and flip-flops are used to construct more complex circuits, such as **adders**, **decoders**, **registers** and **counters**. These circuits, in turn, are used to build **ALUs** and **control units**, i.e., CPUs, as illustrated in Figure 1.5.

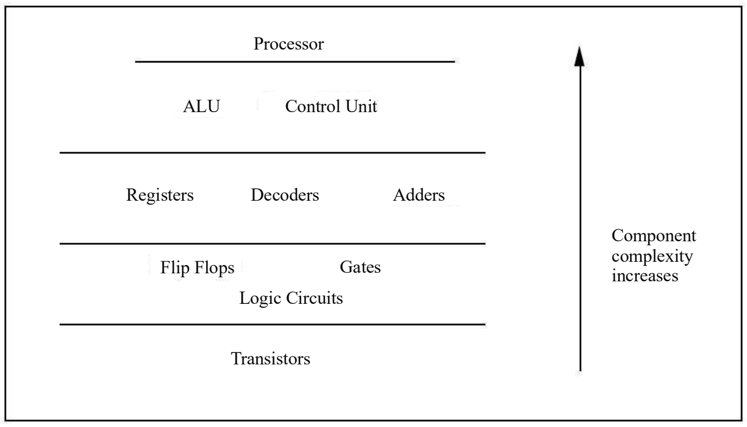


Fig. 1.5 *Circuits Buildings ALU and Control Units*

Let us understand the circuits and components by way of analogy. Houses are made up of rooms, rooms are made up of walls, walls are made up of bricks and bricks are made up of sand and cement. If, by analogy, houses correspond to CPUs, then the sand and cement correspond to transistors. In one sense, this analogy is quite appropriate since transistors are developed on silicon chips and silicon is essential element of computer chip.

Different names are used for the basic logic circuits, such as binary circuits, digital circuits, Boolean circuits and gates. They are called logic circuits because they perform the logic operations (e.g. AND, OR, etc.). There are a number of logic gates, such as AND, OR, XOR, NAND and NOR. The term gate refers to the fact that they act like gates, letting some signals through and blocking others, depending on their inputs. The term Boolean comes from George Boole, the originator of Boolean Algebra. Logic circuits fall into two classes: sequential logic circuits and combinatorial logic circuits. Combinatorial circuits are those where the output is at all times a function of the current inputs to the circuits (no feedback is allowed from the outputs to the inputs). Decoders and adders are important examples of such circuits, which are used in the construction of digital systems such as CPUs. Sequential circuits are those where the outputs depend on past inputs as well as current inputs (they allow feedback). They are sequential in that the output depends on the sequence leading to the present situation. As a consequence, such circuits exhibit memory, i.e., they can retain information. The flip-flop is one of the best known sequential circuits. There are various of types of flip-flops such as the RS flip-flop, JK flip-flop and D-type flip-flops. Registers can be constructed from D-type flip-flops and so D-type flip-flops are commonly used in computers. Memory may also be implemented using flip-flops, as may be shift registers and counters, which are also important computer components. In short, gates and flip-flops, which are constructed out of transistors, are the basic building blocks of computers. If you see advances in the digital electronics, it is because the density of the transistors on the chips has increased drastically as shown in Figure 1.6. Similarly, the other advancements have been illustrated in Figures 1.7, 1.8, 1.9 and 1.10.

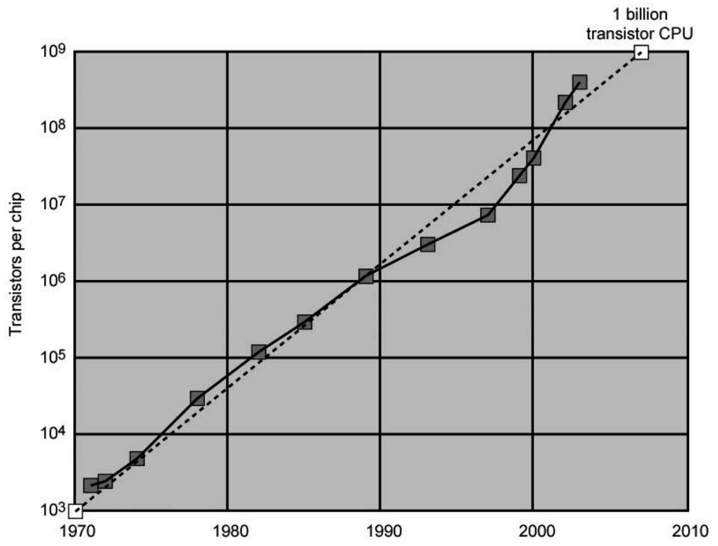


Fig. 1.6 *Density of Transistors on Chips*

NOTES

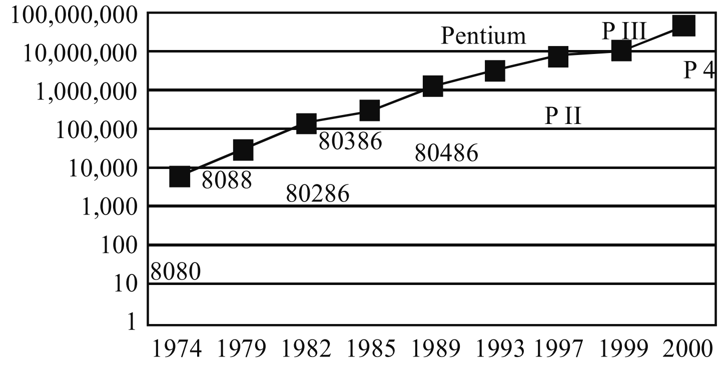


Fig. 1.7 *Number of Transistors on Intel Microprocessors*

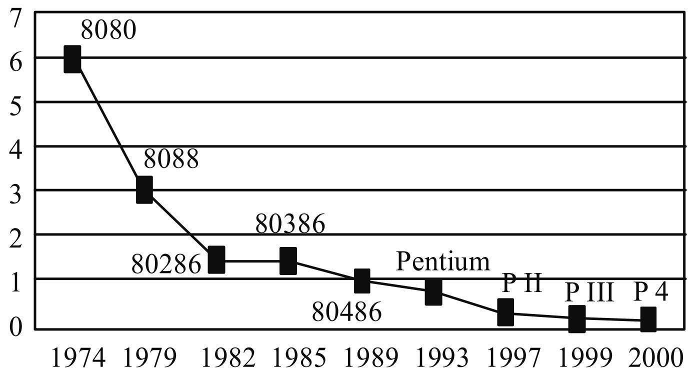


Fig. 1.8 *Minimum Size of a Transistor*

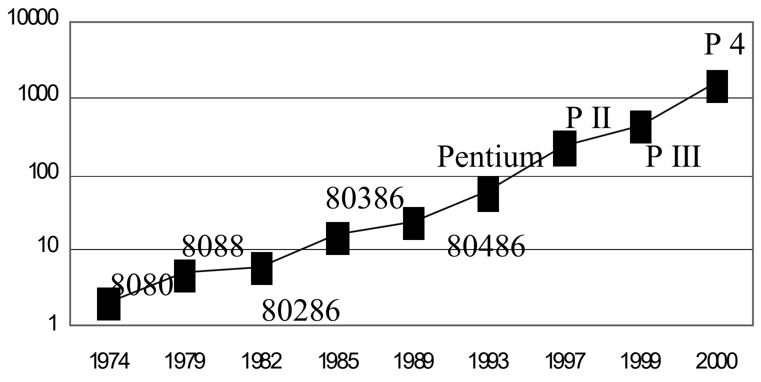


Fig. 1.9 *Clock Frequencies (MHz)*

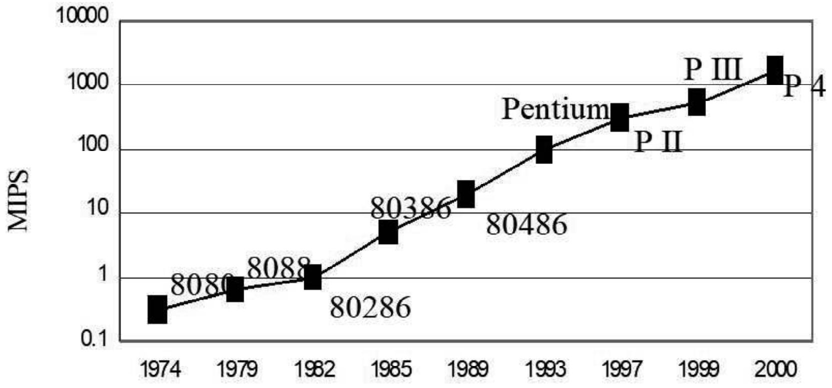


Fig. 1.10 *MIPS (Millions of Instructions Per Second)*

1.2.2 Performance of a Microprocessor

The performance of a microprocessor is most commonly measured in terms of MIPS previously, i.e. Million Instructions Per Second. The range of this rating for different microprocessors of Intel have been shown; however, technically, the MIPS is defined as follows:

$$\text{MIPS} = \frac{1}{\underbrace{[[\text{Circuit switch time} \times \text{levels of logic}] + \text{Package delays}]}_{\text{Tech}} \times \underbrace{\text{Clock cycles per instruction}}_{\text{Machine Organization}}}$$

How Can We Make Computers Work Faster?

The Fetch-Execute Cycle and Pipelining

The fetch–execute cycle represents the fundamental process in the operation of the CPU; attention has been focused on ways for making it more efficient. One possibility is to improve the speed at which instructions and data may be retrieved from memory, since the CPU can process information at a much faster rate than it can retrieve it from memory. Another way of improving the efficiency of the fetch–execute cycle is to use a system known as **pipelining**.

The basic idea here is to break the fetch–execute cycle into a number of separate stages, so that when one stage is being carried out for a particular instruction, the CPU can carry out another stage for a second instruction, and so on. This idea originates from the **assembly line** concept used in manufacturing industry. Consider a simplified car manufacturer's assembly line as shown in Figure 1.11.

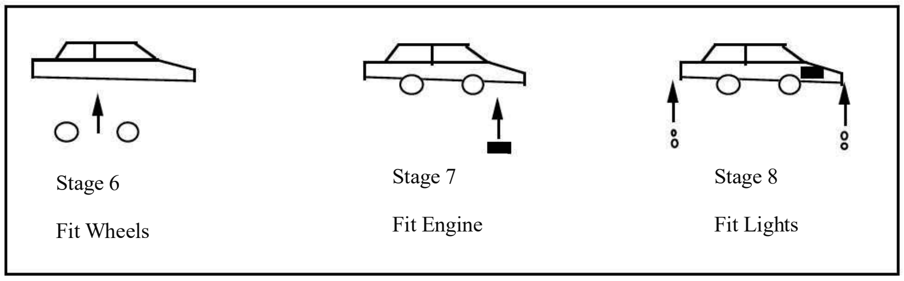


Fig. 1.11 *A Simplified Car Manufacturer's Assembly Line*

The production of a car involves a number of stages, three of which are illustrated. So, for example, if a car goes through 10 stages before being completed, then you can have up to ten cars being operated on at the same time on the assembly line. If, for the sake of simplicity, you assume that each stage takes one hour to complete, then it will take 10 hours to complete the first car since it will be processed for 1 hour at every stage on the assembly line. However, when the first car has moved to the second stage of the assembly line, you can start work on a second car at the first stage of the assembly line. When the first car moves on to the third stage, the second car can move on to the second stage and a third car can be started on the first stage of the assembly line. This process continues, so that when the first car reaches the 10th and final stage, there are 9 other cars in the nine stages of production. This means that when the first car is finished after 10 hours, then another car will be completed every hour thereafter.

The great advantage of assembly line production is the increase in throughput that is achieved. After the first car is completed, you continue production with a throughput of one car per hour. If you did not use an assembly line and worked on one car at a time, each car would take 10 hours to be completed and the throughput would be one car per 10 hours. For example, the time taken to complete 20 cars on the assembly line is 29 hours, while without using the assembly line, the time taken would be 200 hours. It should be noted that in the assembly line, each car still requires 10 hours of processing, i.e., it still takes 10 hours of work to produce a car, the point is that because you are doing the work in stages, you can work on 10 cars at the same time, i.e., in parallel.

Flowthrough Time

The time taken for all stages of the assembly line to become active is called the flowthrough time, i.e., the time for the first car to reach the last stage. Once all the assembly line stages are busy, you achieve maximum throughput. You have simplified the analysis of the assembly line and, in particular, the assumption that all stages take the same amount of time is not likely to be true. The stage that takes the longest time to complete creates a bottleneck in an assembly line. For example, if you assume that stage 5 in our car assembly line takes 3 hours then the throughput decreases to 1 car per 3 hours. This is because stage 6 must wait for 3 hours before it can begin and this delay is passed on to the remaining stages, slowing the time to complete each car to 3 hours.

Clock Period

You can express this by saying the **clock period** of the assembly line (time between completed cars) is 3 hours. The clock period, denoted by T_p, of an assembly line is given by the formula:

$$T_p = \max (t_1, t_2, t_3, t_i, \dots , t_n) \quad \text{for } i = 1, \dots n$$

where t_i is the time taken for the ith stage and there are n stages in the assembly line. This means that the clock period is determined by the time taken by the stage that requires the most processing time. In a non-assembly line system, the total time T, taken to complete a car, is the sum of the time for the individual stages, i.e.,

$$T = t_1 + t_2 + t_3 + \dots\dots + t_n$$

In our example, if all stages take 1 hour to complete, then $T = 10$ hours; it takes 10 hours to complete every car. If stage 5 takes 3 hours and the other stages take 1 hour to complete, then T rises to 12 hours and it will take 12 hours to complete every car.

Throughput

You can define the **throughput** of an assembly line to be $1/T_p$. Using this definition, the throughput for our assembly line where all stages take 1 hour is 1/1, i.e., 1 car/hour. If you assume that stage 5 takes 3 hours to complete, the throughput falls to 1/3 or .333 cars/hour. For non-assembly line production, the respective throughputs are 1/10 or cars/hour and 1/12 or .083 cars/hour.

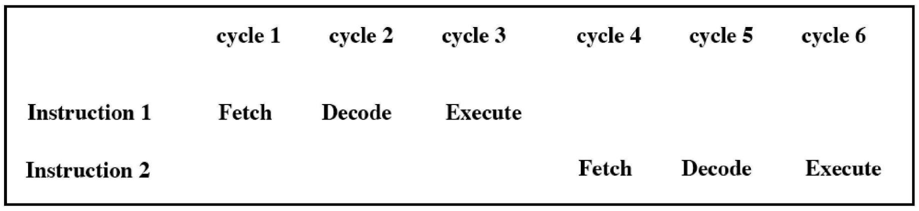
Pipelining

NOTES

The same principle as that of the assembly line can be applied to the fetch–execute cycle of a processor, where you refer to it as pipelining. Earlier, the fetch–execute cycle was described as consisting of three stages, which are repeated continuously:

1. Fetch an Instruction
2. Decode the Instruction
3. Execute the Instruction

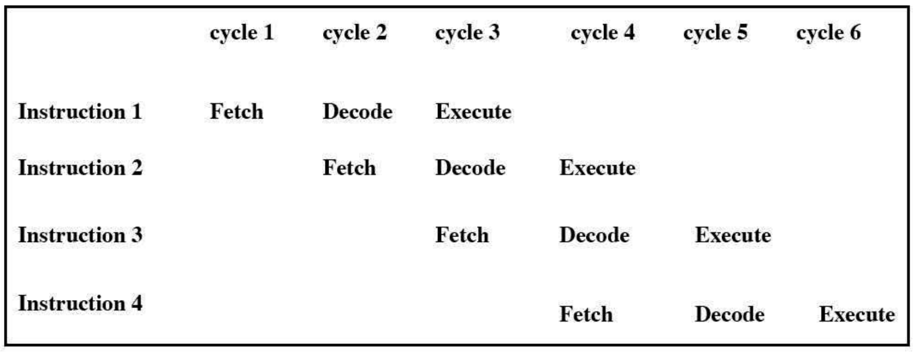
Assuming each stage takes one clock cycle, then in a non-pipelined system, for the first instruction 3 cycles are used, followed by 3 cycles for the second instruction, and so on, (as illustrated in Figure 1.12).



	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2				Fetch	Decode	Execute

Fig. 1.12 *Non-Pipelined System*

The throughput for such a system would be 1 instruction per 3 cycles. If you adopt the assembly line principle, then you can improve the throughput dramatically. Figure 1.13 illustrates the fetch–execute cycle employing pipelining.



	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		
Instruction 3			Fetch	Decode	Execute	
Instruction 4				Fetch	Decode	Execute

Fig. 1.13 *Fetch–Execute Cycle Enforcing Pipelining*

Using pipelining, the processing of instructions is as follows: while the first instruction is in the execution stage, the second instruction is in the decode stage and the third instruction is being fetched.

After 3 cycles the first instruction is completed and thereafter an instruction is completed on every cycle as opposed to a throughput of 3 cycles per instruction in a non-pipelined system. Again, as in the assembly line example, each instruction still takes the same number of cycles to complete, the gain comes from the fact that the CPU can operate on instructions in the different stages in parallel. The clock period and throughput of a pipeline are then defined for the assembly line.

$$\text{Clock period } T_p = \max(t_1, t_2, t_3, t_i \quad \text{for} \quad i = 1, ..., n..., t_n)$$

(for an *n*-stage pipeline)

$$\text{Throughput} = 1/T_p$$

The description is quite simplified, ignoring the fact, for example, that all stages may not be completed in a single cycle. It also omits stages that arise in practice such as an operand fetch stage, which is required to fetch an operand from memory, or a write back stage to store the result of an ALU operation in a register or in memory. In practice, pipelined systems range from 3 to 10 stages; for example, Intel's Pentium microprocessor uses a 5-stage pipeline for integer instructions. There are difficulties in pipelining that would not arise on a factory assembly line, because of the nature of computer programs. Consider the following three instructions in a pipeline:

```
jg label
move y, 0
move x, 3
   ......
   ......
label:
```

When the `jg` instruction is being executed, the following two instructions will be at earlier stages, one being fetched and the other being decoded. However, if the `jg` instruction evaluates the condition to be true, it means that the two `move` instructions will not be executed and new instructions have to be loaded, starting at the instruction indicated by `label`.

This means that the pipeline has to be flushed and reloaded with new instructions. The time taken to reload the pipeline is called the branch penalty and may take several clock cycles. Branch instructions occur very frequently in programs and so it is important to process them as efficiently as possible.

A technique known as branch prediction can be used to alleviate the problem of conditional branch instructions, whereby the system 'guesses' the outcome of a conditional branch evaluation before the instruction is evaluated and loads the pipeline appropriately. Depending on how successfully the guess is made, the need for flushing the pipeline can be reduced. When the branch has been evaluated, the processor can take appropriate action if a wrong guess was made. In the event of an incorrect guess, the pipeline will have to be flushed and new instructions loaded. Branch prediction is used on a number of microprocessors, such as the Pentium and PowerPC. Successful guesses ranging from 80% to 85% of the time are cited for the Pentium microprocessor. Another technique is to use delayed branching. In this case, the instruction following the conditional jump instruction is always executed. For example, if the conditional jump instruction is implementing a loop by jumping backwards, it may be possible to place one of the loop body instructions after the conditional jump instruction. If a useful instruction cannot be placed here, then a `nop` instruction can be used.

1.3 OVERVIEW OF INTEL PRO-PENTIUM

NOTES

The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor—a programmable controller on a chip. It addressed a mere 4096 4-bit wide memory locations. The 4004 instruction set contained only 45 instructions. It was fabricated with the then state-of-the-art p-channel MOSFET (Metal–Oxide–Semiconductor Field-Effect Transistor) technology that only allowed it to execute instructions at the slow rate of 50 kips (kilo-instructions per second). This was slow when compared to the 100,000 instructions executed per second by the 30-ton ENIAC (Electronic Numerical Integrator and Computer) computer in 1946. The main difference was that the 4004 weighed much less than an ounce. The 4-bit microprocessor debuted in early video game systems and small microprocessor-based control systems. The main problems with this early microprocessor were its speed, word width and memory size. The evolution of the 4-bit microprocessor ended when Intel released the 4040, an updated version of the earlier 4004. The 4040 operated at a higher speed, although it lacked improvements in word width and memory size. Other companies, particularly Texas Instruments (TMS-1000), also produced 4-bit microprocessors. The 4-bit microprocessor still survives in low-end applications, such as microwave ovens and small control systems, and is still available from some microprocessor manufacturers. Most calculators are still based on 4-bit microprocessors that process 4-bit BCD (Binary-Coded Decimal) codes.

Later in 1971, realizing that the microprocessor was a commercially viable product, Intel Corporation released the 8008—an extended 8-bit version of the 4004 microprocessor. The 8008 addressed an expanded memory size (16k bytes) and contained additional instructions (a total of 48) that provided an opportunity for its application in more advanced systems. As engineers developed more demanding uses for the 8008 microprocessor, they discovered that its somewhat small in size, it has slow speed and its instruction set limited its usefulness. Intel recognized these limitations and introduced the 8080 microprocessor in 1973—the first of the modern 8-bit microprocessors. About six months after Intel released the 8080 microprocessor, Motorola Corporation introduced its MC6800 microprocessor. The floodgates opened and the 8080 – and, to a lesser degree the MC6800, ushered in the age of 8-bit microprocessors. Soon, other companies began to introduce their own versions of the 8-bit microprocessors. Table 1.1 lists several of these early microprocessors and their manufacturers. Of these early microprocessor producers, only Intel and Motorola continue successfully to create newer and improved versions of the microprocessor. Motorola has declined from having nearly 50 % share of the microprocessor market to a much smaller share.

Table 1.1 *Early Microprocessors and Then Manufacturers*

Company	Microprocessor
Fairchild	F-8
Intel	8080
Motorola	MC6800
National Semiconductor	IMP-8
Rockwell International	RPS-8
Zilog	Z-8

In 1977, Intel Corporation introduced an updated version of the 8080—the **8085**. This was the first 8-bit, general-purpose microprocessor developed by Intel. Although only slightly more advanced than 8080 microprocessor, the 8085 executed instruction at an even higher speed. An addition that took 2.0 μs (500,000 instructions per second) on the 8080 required only 1.3 μs (769,230 instructions per second) on the 8085. The main advantages of the 8085 were its internal clock generator, internal system controller and higher clock frequency. This higher level of component integration reduced the 8085's cost and increased its usefulness. Intel has managed to sell well over 100 million units of the 8085 microprocessor, its most successful 8-bit, general-purpose microprocessor.

In 1978, Intel released the 8086 microprocessor and a year or so later, it released the 8088. Both devices are 16-bit microprocessors, which executed instructions in as little as 400 ns (2.5 MIPS, or 2.5 Million Instructions Per Second). This represented a major improvement over the execution speed of the 8085 processor. In addition, the 8086 and 8088 addressed 1 MB (Mega Byte) of memory, which was 16 times more memory than the 8085. This higher execution speed and larger memory size allowed the 8086 and 8088 to replace smaller minicomputers in many applications. The increased memory size and additional instructions in the 8086 and 8088 led to many sophisticated applications for microprocessors. Improvements to the instruction set included a multiply-and-divide instruction, which was missing on eartier microprocessors. In addition, the number of instructions increased from 45 on the 4004, to 246 on the 8085, to well over 20,000 variations on the 8086 and 8088 microprocessors. The 16-bit microprocessor evolved mainly because of the need for larger memory systems. The popularity of the Intel family was ensured in 1981, when IBM Corporation decided to use the 8088 microprocessor in its personal computer. Applications such as spreadsheets, word processors, spelling checkers and computer based thesauruses were memory intensive and required more than the 64 KB of memory found in 8-bit microprocessors to execute efficiently. The 16-bit 8086 and 8088 provided 1 MB of memory for these applications. Soon, even the 1 MB memory system proved limiting for large databases and other applications. This led Intel to introduce the 80286 microprocessor, an updated 8086, in 1983.

The 80286 Microprocessor

The 80286 microprocessor (also a 16-bit architecture microprocessor) was almost identical to the 8086 and 8088, except it addressed a 16 MB memory system instead of a 1 MB system. The instruction set of the 80286 was almost identical to that of the 8086 and 8088, except for a few additional instructions that managed the extra 15 MB of memory. The clock speed of the 80286 was increased, so it executed some instructions in as little as 250 ns (4.0 MIPS) with the originaly released 8.0 MHz version.

The 32-Bit Microprocessor

Applications began to demand faster microprocessor speeds, more memory and wider data paths. This led to the arrival of the 80386 in 1986, by Intel Corporation. The 80386 represented a major overhaul of the 16-bit 8086—80286 architecture. The 80386 was Intel's first practical 32-bit microprocessor that contained a

NOTES

32-bit data bus and a 32-bit memory address. Through these 32-bit buses, the 80386 addressed up to 4 GB (Giga Byte)) of memory. It can well be imagined that a 4GB memory can store 1000,000 typewritten, double-spaced pages of ASCII (American Standard Code of Information Interchange) text data. However, applications require higher microprocessor speeds and large memory systems including software systems that use a GUI (Graphical User Interface). Modern graphical displays often contain 256,000 or more picture elements (pixels, or Pels). The least sophisticated VGA (Variable Graphics Array) video display has a resolution of 640 pixels per scanning line with 480 scanning lines. To display one screen of information, each picture element must be changed, which requires a high-speed microprocessor. Many new software packages use this type of video interface.These GUI-based packages require high microprocessor speeds and accelerated video adapters for quick and efficient manipulation of video text and graphical data. The most striking system, which requires high-speed computing for its graphical display interface, is Microsoft Corporation's Windows.

The 32-bit microprocessor was in fact needed because of the size of its data bus, which transfers real (single-precision floating-point) numbers that require 32-bit wide memory. In order to efficiently process 32-bit real numbers, the microprocessor must efficiently pass them between itself and memory. If the numbers pass through an 8-bit data bus, it takes four read or write cycles; when passed through a 32-bit data bus, however, only one read or write cycle is required. This significantly increases the speed of any program that manipulates real numbers. Most High-Level Languages, spreadsheets and database management systems use real numbers for data storage. Real numbers are also used in graphical design packages that use vectors to plot images on the video screen. These include, such CAD (Computer Aided Drafting/Design**)** systems such as, AUTOCAD, ORCAD, and so on. Besides providing higher clocking speeds, the 80386 included a memory management unit that allowed memory resources to be allocated and managed by the operating system. Earlier microprocessors left memory management completely to the software. The 80386 included hardware circuitry for memory management and memory assignment, which improved its efficiency and reduced software overhead. The instruction set of the 80386 microprocessor was compatible with the earlier 8086, 8088 and 80286 microprocessors.

The 80486 Microprocessor

In 1989, Intel released the 80486 microprocessor, which incorporated an 80386-like microprocessor, an 80387-like numeric coprocessor (for real mathematical calculations), and an 8 kB cache memory system into one integrated package. Although the 80486 microprocessor was not radically different from the 80386, it did include one substantial change—the internal structure of the 80486 was modified from the 80386 so that about half of its instructions were executed in one. The architectural concept is illustrated in Figure 1.14.

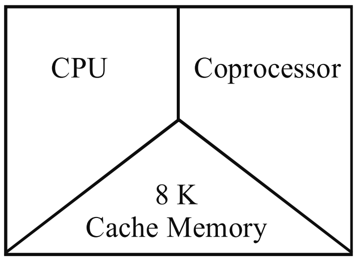


Fig. 1.14 *Architectural Concept of 80486 Microprocessor*

The Pentium Microprocessor

The Pentium, introduced in 1993, was similar to the 80386 and 80486 microprocessors. This microprocessor was originally labeled P5 or 80586, but Intel decided not to use a number because it appeared to be impossible to copyright a number. The two introductory versions of the Pentium operated with a clocking frequency of 60 MHz and 66 MHz. The double-clocked Pentium, operating at 120 MHz and 133 MHz, was also available, as were higher-speed versions. (The fastest version produced by Intel is the 233 MHz Pentium, which is a three and a half clocked version.) Another difference was that the cache memory size was increased to 16 kB from the 8 kB cache found in the basic version of the 80486. The Pentium contained an 8 kB instruction cache and an 8 kB data cache, which allowed a program that transfers a large amount of memory data to still benefit from a cache. The memory system contained up to 4 GB, with the data bus width increased from the 32 bits found in the 80386 and 80486 to a full 64 bits. This wider data bus width accommodated double-precision floating-point numbers used for modern high-speed, vector-generated graphical displays. These higher bus speeds allowed virtual reality software to operate at more realistic rates on current and future Pentium-based platforms. The widened data bus and higher execution speed of the Pentium allow full-frame video displays to operate at scan rates of 30 Hz or higher—comparable to commercial television. Recent versions of the Pentium also included additional instructions, called multimedia extensions, or MMX instructions. Although Intel hoped that the MMX instructions would be widely used, it appears that few software companies have used them.

Pentium Pro Processor

A recent entry from Intel is the Pentium Pro processor, formerly codenamed the P6 microprocessor. The Pentium Pro processor contains 21 million transistors, 3 integer units, as well as a floating-point unit to increase the performance of most software. The basic clock frequency was 150 MHz and 166 MHz in the initial offering made available in late 1995. In addition to the internal 16K Level-one (L 1) cache (8k for data and 8k for instructions), the Pentium Pro processor also contains a 256k Level-two (L2) cache. One other significant change is that the Pentium Pro processor uses three execution engines, so it can execute up to three instructions at a time, which can conflict and still execute in parallel. This represents a change from the Pentium, which executes two instructions simultaneously as long as they do not conflict. The Pentium Pro microprocessor has been optimized to efficiently execute 32-bit code; for this reason, it is often bundled with Windows NT rather than with normal versions of Windows 95. Intel launched the Pentium

Pro processor for the server market. Still another change is that the Pentium Pro can address either a 4 GB memory system or a 64 GB memory system. The Pentium Pro has a 36-bit address bus if configured for a 64 GB memory system.

Pentium II and Pentium II Xeon Microprocessors

The Pentium II microprocessor (released in 1997) represents a new direction for Intel. Instead of being an integrated circuit as with prior versions of the microprocessor, Intel has placed the Pentium II on a small circuit board. The main reason for the change is that the L2 cache found on the main circuit board of the Pentium was not fast enough to justify a new microprocessor. On the Pentium system, the L2 cache operates at the system bus speed of 60 MHz or 66 MHz. The L2 cache and microprocessor are on a circuit board called the Pentium II module. This on-board, L2 cache operates at a speed of 133 MHz and stores 512K bytes of information. The microprocessor on the Pentium II module is actually a Pentium Pro with MMX extensions, which has no internal L2 cache.

In 1998, Intel changed the bus speed of the Pentium II. Because the 266 MHz through the 333 MHz Pentium II microprocessors used an external bus speed of 66 MHz, there was a bottleneck, so the newer Pentium II microprocessors use a 100 MHz bus speed. The Pentium II microprocessors rated at 350 MHz, 400 MHz and 450 MHz all use this higher 100 MHz memory bus speed. The higher speed memory bus requires the use of 8 ns SDRAM in place of the 10 ns SDRAM found in use with the 66 MHz bus speed.

In mid-1998 Intel announced a new version of the Pentium II called the Xeon, which was specifically designed for high-end workstation and server applications. The main difference between the Pentium II and the Pentium II Xeon is that the Xeon is available with a L1 cache size of 32K bytes and a L2 cache size of either 512K, 1M, or 2M bytes. This newer product represents a change in Intel's strategy: Intel now produces a professional version and a home/business version of the Pentium II microprocessor.

Pentium III Microprocessor

The Pentium III microprocessor uses a faster core than the Pentium II, but it is still a P6 or Pentium Pro processor. It is also available in the slot 1 version mounted on a plastic cartridge and a socket 370 version called a flip-chip, which looks like the older Pentium package. Intel claims the flip-chip version costs less. Another difference is that the Pentium III is available to clock frequencies of 1 GHz. The slot 1 version contains a 512k cache and the flip-chip version contains a 256k cache. The speeds are comparable because the cache in the slot 1 version runs at one-half the clock speed, while the cache in the flip-chip version runs at the clock speed. Both versions use a memory bus speed of 100 MHz, while the Celeron uses a memory bus clock speed of 66 MHz.

Pentium 4 Microprocessor

The Pentium 4 microprocessor, the latest Intel release, was first available in late 2000. The Pentium 4, like the Pentium Pro through the Pentium III (PIII), uses

NOTES

the Intel P-6 architecture. The main difference is that the Pentium 4 is available in 1.3, 1.4 and 1.5 GHz-speed version, and the chipset that supports Pentium 4 uses the RAMBUS memory technology in place of SDRAM technology. These higher microprocessor speeds are made available by an improvement in the size of the internal integration. It is also interesting to note that Intel has changed the level 1 cache size from 32 kB to 8 kB. Research must have shown that this size is large enough for the initial release version of the microprocessor, with future versions possibly containing a 32k L1 cache. The level 2 cache remains at 256 kB as in the PIII coppermine version. Another change you are likely to see is a shift from aluminium to copper interconnections. Because copper is a better conductor, it should increase the clock frequencies for the microprocessor. This is especially true now that a method for using copper has surfaced. You may also see the front side bus speed increase from the current maximum of 133 MHz to 200 MHz or higher. Finally the dual core processors are now available in the market with doubling the accumulator units and hence the processing capacity with the 32 bit microprocessors. At the same time 64 bit microprocessors are also available in the market. A complete evolution of the Intel Microprocessors can be understood by the following data:

- **8080**
  - First general purpose microprocessor
  - 8 bit data path
  - Used in first personal computer—Altair
- **8086**
  - Much more powerful
  - 16 bit
  - Instruction cache, prefetch few instructions
  - 8088 (8 bit external bus) used in first IBM PC
- **80286**
  - 16 MB memory addressable
  - Up from 1MB
- **80386**
  - 32 bit
- **80486**
  - Sophisticated powerful cache and instruction pipelining
  - Built in maths co-processor
- **Pentium**
  - Superscalar
  - Multiple instructions executed in parallel
- **Pentium Pro**
  - Increased superscalar organization
  - Aggressive register renaming
  - Branch prediction

- Data flow analysis
- Speculative execution
- Support for multitasking

- **Pentium II**
  - MMX technology
  - Graphics, video & audio processing
- **Pentium III**
  - Additional floating point instructions for 3D graphics
- **Pentium 4**
  - Note Arabic rather than Roman numerals
  - Further floating point and multimedia enhancements
- **Itanium**
  - 64-bit Microprocessor
- **Itanium 2**
  - Hardware enhancements to increase speed

1.4 MOTOROLA 68000 SERIES

About six months after Intel released the 8080 microprocessor, Motorola Corporation introduced its MC6800 microprocessor. The floodgates opened and the 8080—and, to a lesser degree the MC6800—ushered in the age of the microprocessor. This was the first microprocessors from Motorola with all 8-bit devices (8-bit data bus). The 6800 appeared in 1975 with a clock frequency of 2 MHz and capable of addressing 64 kB of memory with a 16-bit address bus. It was somewhat similar to Intel 8080. The main architectural features of 6800 are:

1. The 6800 is a standard '8-bit' 2's complement microprocessor, like the Intel 8080.
2. The 6800 had a 16-bit stack pointer, so the stack can be located anywhere and can grow to any length up to the size of the memory.
3. Like the 8080, the 6800 supports multi-precision arithmetic using a carry flag and the ADC (Add with Carry) and SBC (Subtract with Carry) instructions.
4. Arithmetic instructions set the usual 2's complement flags: sign, zero, overflow and carry.
5. Unlike the 8080, the 6800 had no special purpose I/O instructions.
6. The 6800 had indexing, which allowed it to directly support data structures.
7. The 6800 had four addressing modes: immediate, indexed, extended and direct (or zero-page).

Then, there were different variations of 6800 with little modifications in memory. Finally the era of Motorola 68000 came, which is the first 16/32-bit CISC microprocessor from Motorola. Introduced in 1979 with HMOS technology as the first member of the successful 32-bit 68k family of microprocessors, it is generally software forward compatible with the rest of the line despite being limited to a 16-bit wide external bus. After three decades in production, the 68000 architecture is still in us. The complete 6800 family shown in Table 1.2.

NOTES

Table 1.2 *Complete 6800 Family*

Processor	Data Bus Width	Memory Size
6800	8	64 kB
6805	8	2 kB
6809	8	64 kB
68000	16	16 MB
6808Q	8	1 MB
68008D	8	4 MB
68010	16	16 MB
68020	32	4GB
68030	32	4 GB + 256 cache
68040	32	4 GB + 8 kB cache
68050	32	Proposed, but never released
68060	64	4 GB + 16 kB cache
PowerPC	64	4 GB + 32 kB cache

Check Your Progress

1. What is a CPU?
2. Write about the different categories of microcomputer memory.
3. Which was the first microprocessor from Intel?
4. When did Intel launched Pentium?
5. What do you understand by MMX?

1.5 INTRODUCTION TO DEC ALPHA

The **DEC Alpha**, also known as the **Alpha AXP**, was a 64-bit RISC microprocessor originally developed and fabricated by Digital Equipment Corporation (DEC), Group of Pictures (GOP) who used it in their own line of workstations and servers. Alpha, originally known as Alpha AXP, was a 64-bit Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA) developed by Digital Equipment Corporation (DEC). It was originally designed to replace the 32-bit VAX (CISC) ISA. Digital Equipment Corporation was a pioneering American company in the computer industry. It was often referred within the computing industry as DEC (this acronym was frequently officially used by

Digital itself but the trademark was always DIGITAL). Its PDP and VAX products were the most popular minicomputers for the scientific and engineering communities during 1970s and 1980s. The highlights of the Alpha processor were:

- Typical 64-bit RISC architecture.
- Fixed instruction length.
- Five Instruction format.
- Seven-stage split integer/floating-point pipeline.

The first few generations of the Alpha chips were some of the most innovative of their time. The first version, 21064 or EV4, was the first CMOS microprocessor with high The second, 21,164 or EV5, was the first microprocessor to place a large secondary cache on chip. The third, 21,264 or EV6, was the first microprocessor to combine both high operating frequency and the more complicated out-of-order execution microarchitecture. Alpha was implemented in microprocessors originally developed and fabricated by DEC. It was used in a variety of workstations and servers, eventually forming the basis for almost all of their mid-to-upper-scale processors. Alpha supports both the OpenVMS (previously known as OpenVMS AXP) operating system and Tru64 UNIX (previously known as DEC OSF/1 AXP and Digital UNIX). Microsoft supported the processor in Windows NT until NT 4.0 came. Open source operating systems that run on the Alpha are Linux (Debian, Gentoo Linux and Red Hat Linux), BSD UNIX (NetBSD, OpenBSD and FreeBSD up to 6.x).The Alpha architecture was sold, along with most parts of DEC, to Compaq in 1998. Compaq, already an Intel customer, decided to phase out Alpha in favour of the forthcoming Intel IA-64 'Itanium' architecture, and sold all Alpha intellectual property to Intel in 2001, effectively 'killing' the product. Hewlett-Packard purchased Compaq later that same year, continuing development of the existing product line until 2004, and promising to continue selling Alpha-based systems, largely to the existing customer base, until October 2006 (later extended to April 2007).

1.6 POWERPC

PowerPC is a microprocessor architecture that was developed jointly by Apple, IBM, and Motorola in early 1990. The PowerPC Employs Reduced Instruction-Set Computing (RISC). The three developing companies have made the PowerPC architecture an open standard, inviting other companies to build on it. PowerPC is an acronym, which stands for Performance Optimization with Enhanced RISC. PowerPC is mainly based upon the Power architecture of IBM. It was first used in IBM's RS/6000 workstation with its UNIX-based operating system, AIX, and in Apple Computer's Macintosh personal computers. Today, PowerPC chips are also used in diverse applications including internetworking equipment, routers, telecom switches, interactive multimedia, automotive control, and industrial robotics. PowerPC CPUs have since become popular embedded and high-performance processors. The main architectural features of the PowerPC are:

- Designed along RISC principles.
- Allows for a superscalar implementation.

- Versions of the design exist in both 32-bit and 64-bit implementations.
- Additional floating point instructions.
- A paged memory management architecture.

Developed at IBM, the Reduced Instruction-Set Computing (RISC) specifies that these simplest computer instructions are most frequently performed. Traditionally, processors have been designed to accommodate the more complex instructions. Typically, RISC performs the more complex instructions using combinations of simple instructions. The timing for the processor can then be based on simpler and faster operations, enabling the microprocessor to perform more instructions for a given clock speed.

The architecture of PowerPC provides an alternative to the popular processor architectures from Intel, including the Pentium. Microsoft has specifically build its Windows operating system which can efficiently run on Intel processors, and this widely-sold combination is sometimes termed as 'Wintel'.

The architecture of PowerPC machine includes the following components:

Memory

Memory consists of 8-bit bytes. Two consecutive bytes form a halfword, four bytes form a word, eight bytes form a doubleword, and sixteen bytes form a quadword. PowerPC programs can be written using a 264 bytes Virtual Address Space (VAS). Address spaces are divided into fixed-length segments which are further divided into pages.

Registers

There are 32 General-Purpose Registers (GPR) from GPR0 to GPR31. Length of each register is 64-bit. The general purpose register are used to store and manipulate data and addresses. As PowerPC machine support floating point data format so it have Floating-Point Unit (FPU) for computation.

Following Table 1.3 illustrates the register's supported by PowerPC architecture.

***Table** 1.3 Register's Supported by PowerPC Architecture*

Register	**Operations**
Link Register (LR)	Contain address to return at the end of the function call.
Condition Register (CR)	Signifies the result of an instruction.
Count Register (CTR)	Used for Loop count.

Data Formats

Integers are stored as 8-, 16-, 32-, or 64-bit Binary numbers.

Characters are represented using 8-bit ASCII (American Standard Code for Information Interchange) codes.

Floating points are represented using two different floating-point formats namely Single-precision format and double-precision format.

Instruction Formats

PowerPC support seven basic instruction formats. All of these instruction formats are 32-bits long. PowerPC architecture instruction format have more variety and complexity as compared to other RISC systems, such as SPARC (Scalable Processor ARChitecture). Bit numbering for PowerPC is the opposite of most other definitions:

- Bit 0 is the Most Significant Bit (MSB).
- Bit 31 is the Least Significant Bit (LSB).

Instructions are first decoded by the upper 6 bits in a field, called the primary opcode. The remaining 26 bits contain fields for operand specifiers, immediate operands, and extended opcodes, and these may be reserved bits or fields.

Instruction Set

PowerPC architecture is more complex than the other RISC systems. Thus PowerPC architecture has approximately 200 machine instructions. This architecture follow pipeline execution of instructions which means while one instruction is executed next one is being fetched from memory and decoded.

Input and Output

PowerPC architecture follow two different methods for performing I/O (Input/ Output) operations. In one approach Virtual Address Space (VAS) are used while in other approach I/O is performed using Virtual memory management.

Principally, the PowerPC is based on IBM's earlier POWER Instruction Set Architecture (ISA), and retains a high level of compatibility with it; the architectures have remained close enough that the same programs and operating systems will run on both if some care is taken in preparation; newer chips in the POWER series use the Power ISA.

1.7 RISC & CISC ARCHITECTURE

RISC (Redced Instruction Set Computer) and CISC (Complex Instruction Set Computer)) are two architectural paradigms in the microprocessors. Each one has it own popularity and applications, obviously for different reasons. In case of mainframes, RISC has been very popular since long back. However in case of the Personal Computers, CISC is very common. Through this discussion, we will try to main aspects of both of them and then try to make a comparison.

CISC

Today, the CISC can be thought as a microprocessor of the common man. However there were obvious technical goals behind the development of CISC:

1. Before the advent of compilers, all programming was done in machine code or assembly because of the following reasons:
   (i) To make programming easier, more complex instructions were created.
   (ii) These instructions were direct representations of high level functions in high level programming languages.

   (iii) This was supposed to be a good idea, since at the time, hardware design was easier than compiler design.

2. Computers had very little memory at that time and hence:
   (i) This promoted a high density of information in programs.
   (ii) Instructions of variable size, instructions which perform multiple operations, and instructions that both moved data and performed data computations.
   (iii) The ability to pack instructions densely was considered more important than instruction decidability.
3. Register count in the processors was small, hence
   (i) Bits in registers were more expensive than external memory, and would have been difficult to include in large numbers due to technology limitations.
   (ii) More registers require more instruction bits (so all registers can be addressed)—this would take up expensive RAM.

Because of these reasons, instructions were designed to do as much work as possible.

1. One instruction could load up two numbers, add them, and store the result back to memory.
2. Another version of that instruction would do the same, but store the result in a register.
3. Yet another version would read one number from memory, the second from register, and write the result back to memory.

This design philosophy became known as CISC (Complex Instruction Set Computer). Pronounced *sisk,* and stands for **C**omplex **I**nstruction **S**et **C**omputer. Most PC's use CPU based on this architecture. For instance Intel and AMD CPU's are based on CISC architectures. Typically CISC chips have a large amount of different and complex instructions. The philosophy behind it is that hardware is always faster than software, therefore one should make a powerful instruction set, which provides programmers with assembly instructions to do a lot with short programs. In common CISC chips are relatively slow (compared to RISC chips) per instruction, but use little (less than RISC) instructions.

RISC

With the advances in technology, CPUs started to run faster than the memory in the late 1970s. it was already apparent that CPU and memory speed would grow further apart. This development led to complete change in the architectural philosophy of microprocessors:

1. To support the higher CPU speeds, more registers were needed:
   (i) Additional registers would require more space on the chip.
   (ii) This space could be created by reducing the complexity of the CPU.
2. Real world examples showed that most processors were over-designed:
   (i) On average, 98% of the constants in a program will fit into 13 bits, while almost every CPU stored them in individual words.

NOTES

(ii) This suggests that the CPU should store the constants in unused bits of the instruction itself, decreasing the number of memory accesses.

(iii) If this scheme is used, the operation needs to be small, so that there is enough room left in the 32 bit instruction to hold larger constants.

3. Real world programs spend most of their time executing simple operations:

   (i) Focus was put on making these common operations as simple and fast as possible.

   (ii) The goal was to make instructions so simple, each could be fully completed in a single clock cycle.

Thus, instead of a single complex instruction, code was implemented as a series of smaller instructions. This left more space in the instruction for data. The number one rule for a RISC implementation was to 'make the common case fast' Unfortunately, this also meant that the total number of instructions that needed to be read from memory for any single program is larger, and takes longer.This focus on 'Reduced Instructions' led to the result being called 'Reduced Instruction Set Computer'(RISC) pronounced *risk,* and stands for **R**educed **I**nstruction **S**et **C**omputer. RISC chips evolved around mid-1980 as a reaction at CISC chips. The philosophy behind it is that almost no one uses complex assembly language instructions as used by CISC, and people mostly use compilers which never use complex instructions. Apple for instance uses RISC chips. Therefore fewer, simpler and faster instructions would be better, rather than the large, complex and slower instructions of CISC. However, more instructions are needed to accomplish a task. Another advantage is that RISC chips require fewer transistors, which makes them easier to design and cheaper to produce. Finally, it's easier to write powerful compilers as only few instructions are available.

Check Your Progress

6. Which is the latest 64-bit microprocessor from Intel?
7. Who developed PowerPC?
8. Define the term RISC.

1.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. CPU is an electronic device which can fetch and execute a set of instructions, carry out arithmetic and logical operations and can control the input /output devices.
2. There are three categories of memories:
   - Processor Memory
   - Primary or Main Memory
   - Secondary Memory
3. 4004 was the first microprocessor from Intel.
4. Intel launched Pentium in 1993.

5. MMX means MultiMedia Extensions; these are the instructions to support the multimedia capabilities.
6. Itanimum is the latest 64-bit microprocessor from Intel.
7. PowerPC is a microprocessor architecture that was developed jointly by Apple, IBM, and Motorola in early 1990. The Power PC employs Reduced Instruction Set Computing (RISC).
8. Instead of a single complex instruction, code was implemented as a series of smaller instructions. This left more space in the instruction for data. The number one rule for a RISC implementation was to 'Make the common case fast' Unfortunately, this also meant that the total number of instructions that needed to be read from memory for any single program is larger, and takes longer. This focus on 'Reduced Instructions' led to the result being called 'Reduced Instruction Set Computer' (RISC) pronounced risk, and stands for Reduced Instruction Set Computer.

1.9 SUMMARY

- A computer is an electronic device, operating under the control of instructions (software) stored in its own memory unit, which can accept data (input), manipulate data (process) and produce information (output) from the processing.
- The CPU is an electronic device which can fetch and execute a set of instructions, carry out arithmetic and logical operations and can control the input/output devices.
- The memory unit is an integral part of any microcomputer system and its primary purpose is to hold program and data.
- The major goal of the memory unit is to allow it to operate at a speed close to that of the processor. The cost factor inhibits the design of the entire memory unit with single technology that guarantees high speed.
- The microcomputer system communicates with the outside world via the I/O devices interfaced to it. The user can enter the program and data using the keyboard on the terminal and execute the program to obtain the results.
- Transistors are the fundamental components from which logic circuits are constructed. These logic circuits are, in turn, the basic building blocks of the CPU.
- The fetch–execute cycle represents the fundamental process in the operation of the CPU; attention has been focused on ways for making it more efficient. One possibility is to improve the speed at which instructions and data may be retrieved from memory, since the CPU can process information at a much faster rate than it can retrieve it from memory.
- The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor – a programmable controller on a chip. It addressed a mere 4096 4-bit wide memory locations.

- The 80286 microprocessor (also a 16-bit architecture microprocessor) was almost identical to the 8086 and 8088, except it addressed a 16 MB memory system instead of a 1 M B system.
- In 1989, Intel released the 80486 microprocessor, which incorporated an 80386- like microprocessor, an 80387-like numeric coprocessor (for real mathematical calculations), and an 8 KB cache memory system into one integrated package.
- The Pentium, introduced in 1993, was similar to the 80386 and 80486 microprocessors.
- PowerPC is a microprocessor architecture that was developed jointly by Apple, IBM, and Motorola in early 1990. The Power PC employs Reduced Instruction Set Computing (RISC).
- The DEC Alpha, also known as the Alpha AXP, was a 64-bit RISC microprocessor originally developed and fabricated by digital equipment corp.
- Alpha was implemented in microprocessors originally developed and fabricated by DEC. It was used in a variety of workstations and servers, eventually forming the basis for almost all of their mid-to-upper-scale processors.
- Alpha supports both the OpenVMS (previously known as OpenVMS AXP) operating system and Tru64 UNIX (previously known as DEC OSF/1 AXP and Digital UNIX).
- RISC and CISC are two architectural paradigms in the microprocessors. Each one has it own popularity and applications, obviously for different reasons. In case of mainframes, RISC has been very popular since long back. However in case of the Personal Computers, CISC is very common.
- This design philosophy became known as CISC (Complex Instruction Set Computer). Pronounced sisk, and stands for Complex Instruction Set Computer.

1.10 KEY TERMS

- **Computer:** A computer is an electronic device, operating under the control of its instructions stored in its own memory unit.
- **System bus:** The system bus is basically a collection of signal paths, which are used to carry the electrical signals across different parts of the computer.
- **CPU:** CPU is an electronic device, which can fetch and execute certain set of instructions, carry out arithmetic and logical operations, and can control the input/output devices.
- **PowerPC:** PowerPC is a microprocessor architecture that employs reduced instruction-set computing.
- **DEC Alpha:** DEC Alpha is a 64-bit RISC microprocessor originally developed and fabricated by Digital Equipment Corporation (DEC).

1.11 SELF-ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short-Answer Questions

1. What do you understand by the fifth-generation microprocessors?
2. Who developed PowerPC and why?
3. Define the main features of DEC Alpha.
4. Compare the Motorola processors with the Intel processors.
5. What is the technical goals behind the development of CISC?
6. State the need for RISC processors.

Long-Answer Questions

1. Briefly explain the evolution of microprocessor with the help of diagram and examples.
2. Discuss about the Intel Pro-Pentium giving appropriate examples.
3. Describe the Motorola 68000 series giving examples
4. Briefly explain about the PowerPC and DEC Alpha.
5. Compare the characteristics of CISC and RISC. Which is better in terms of speed and why?

1.12 FURTHER READING

Goankar, Ramesh. 2002. *Microprocessor Architecture, Programming and Application with the 8085*. Mumbai: Penram International Publishing.

Ram, B. 2005. *Fundamentals of Microprocessor and Microcomputers*. New Delhi: Dhanpat Rai Publications.

Lotia, Manahar and Pradeep Nair. 2003. *Modern All About Motherboard*. New Delhi: BPB Publications.

Mueller, Scott, and Craig Zacker. 2002. *Upgrading and Repairing PCs*. New Jersey: Pearson Education (Que Publishing).

Godse, D.A. and A.P. Godse. 2007. *Microprocessor and Assembly Language*

Programming. Pune: Technical Publications.

Kleitz, William. 2009. *Digital and Microprocessor Fundamentals: Theory and*

Applications. New Jersey: Prentice Hall.

Ray, A.K. and K.M. Bhurchandi. 2000. *Advanced Microprocessors and Peripherals*. New Delhi: Tata McGraw-Hill.

UNIT 2 BASIC MICROPROCESSOR ARCHITECTURE AND INTERFACE

Structure

2.0 INTRODUCTION

The microprocessor is a Central Processing Unit (CPU) that is the 'Brain' of the computer. It processes the information as per a program and provides output in the form of digital signals. The microprocessor is fabricated on a single chip by using the Metal Oxide Semiconductor (MOS) technology. The basic building blocks of most electronic devices consist of transistors and diodes. However, the organization differentiates the systems based on their performance. Primarily, there are a few standard microprocessors and architectures that are followed around the globe by different companies with some modifications. Each microprocessor has an address bus, a data bus and a control bus to carry the signals inside and outside the microprocessors.

Memory is required to store data and instructions. Data and instructions which can be lost after the power supply is stopped are stored in the Random-Access Memory (RAM). Data and instructions which should never be lost, even after the power is turned off, are stored in Read-Only Memory (ROM). ROM is a type of memory whose contents cannot be changed once the ROM chip is manufactured. The most common operations with the memory and the I/O devices are the read and write operations to receive and send data. Both of these operations use different techniques for working with the memory and the I/O devices and employ stringent procedures.

In this unit, you will study about the internal architecture of microprocessor, external system and bus architecture, memory, and input/output interface.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basics of internal architecture of microprocessor
- Explain the external system and bus architecture
- Discuss about the memory
- Explain the input/output interface

2.2 INTERNAL ARCHITECTURE OF MICROPROCESSOR

The microprocessor is a Central Processing Unit (CPU) that is the 'Brain' of the computer. It processes the information as per a program and provides output in the form of digital signals. The microprocessor is fabricated on a single chip by using the Metal Oxide Semiconductor (MOS) technology.

First, you will study those features, that are common to all microprocessors. Then, you will study each of the microprocessor families and their specific features.

A microprocessor comprises of:

(i) Register Section

(ii) One or more Arithmetic Logic Unit (ALU), and

(iii) A Control Unit

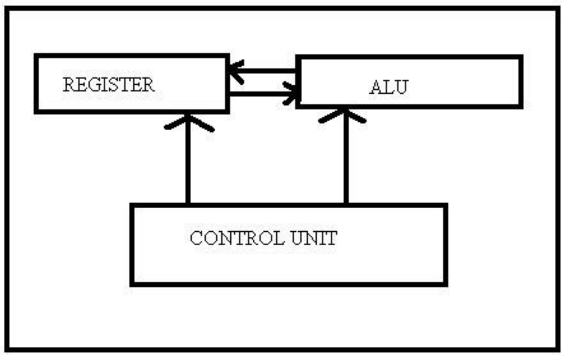


***Fig. 2.1** Components of a Microprocessor*

Now, let us discuss each of these components in detail.

Register Section

The register section (also known as register set) is related to the main memory of the computer system. During the processing of instructions, it stores the data.

Classification of Processors Based on Register Section

1. *Accumulator-Based Microprocessors*

One of the most often used parts of a microprocessor is the accumulator. In microprocessors, one of the operands holds a special register called 'Accumulator'.

The arithmetic and logical operations performed by using this accumulator alter the value that it contains. Usually, the result of an operation is also placed in the accumulator. For example, you can add the contents of the accumulator to the contents of a memory location. Figure 2.2 illustrates this operation.

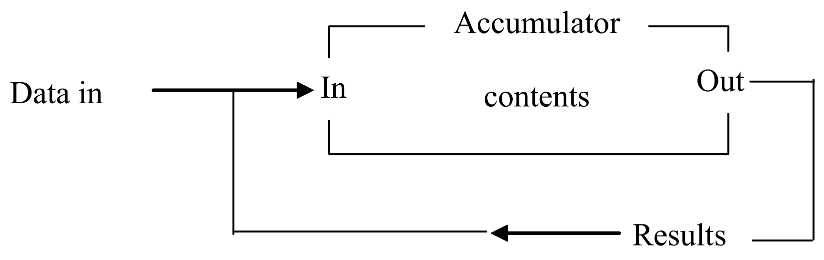


Fig. 2.2 *Accumulator Operation*

The microprocessor can take the contents of the accumulator as the data coming in, perform some operation and place the result back in the accumulator. Occasionally, there is no data coming in but some operation is still performed on the contents of the accumulator only. For example, the microprocessor might find the 1's complement of the contents of the accumulator and place the result in the accumulator in place of the original number. Some microprocessors have only one accumulator while others have more than one.

Such one-operand instructions are predominant in several organizations. Examples of accumulator-based microprocessors are Intel's 8085 and Motorola's 6809.

2. *The General-Purpose Register-Based Microprocessors*

These processors have a set of registers, which contain data, memory addresses and the results of an arithmetic or logic operations for an indefinite time. The number and size of these registers vary from processor to processor

There are primarily two types of registers:

(i) General Purpose Registers

(ii) Dedicated Registers

General-purpose registers are temporary storage locations. They store addresses or data and are capable of manipulating data by shift or rotate operations. General-purpose registers are similar to the accumulator. However, they differ from the accumulator in that operations involve two pieces of data that are usually not performed in them with the results going back into the register itself, as in the case of the accumulator. The microprocessor will often alter the contents of a register. Examples of such microprocessors are Intel's 8086/386, Motorola's 68000/020. Figure 2.3 shows the operations of a general-purpose register.

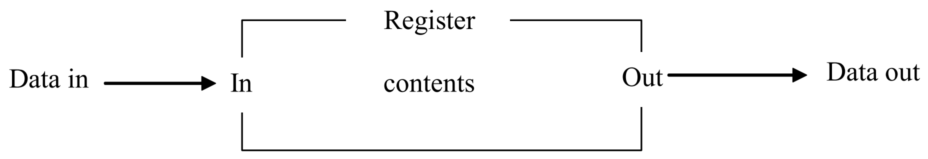


Fig. 2.3 *General-Purpose Register Operations*

One might wonder why a microprocessor needs general-purpose registers when it has RAM to temporarily store information. The answer is speed. Data in registers can be accessed and moved much more quickly than data in RAM.

NOTES

Dedicated Registers

On the other hand, typical dedicated registers include:

a. Program Counter (PC)
b. Instruction Register (IR)
c. Status Register or Flag Register
d. Barrel Shifter
e. Stack Pointer (SP)

The Program Counter

The PC stores the address (location) of the next instruction that needs to be executed. Its contents are automatically updated by the ALU. The microcomputer will process the instructions of a program successively until it encounters a jump/ branch/call instruction, when it will be loaded with the address present in the instruction. The size of the PC itself varies from processor to processor. For example, the 8085 has a 16-bit PC, while 68029 have 32-bits PC.

As mentioned earlier, instructions are stored in memory. Considering the fact that there can be tens of thousands, hundreds of thousands, or even millions of memory locations, it is obvious that the microprocessor must keep track of the location from which it will be receiving the next instruction. This is the function of the program counter.

The program counter is a dedicated register whose job is to keep track of the location of the next instruction, which the microprocessor will use. The PC 'Points' to the address of the next instruction to be retrieved and used by the microprocessor. The act of acquiring an instruction is referred as the **fetching** the instruction. The time period needed for this is called the **fetch cycle**. Figure 2.4 illustrates the operation of a program counter.

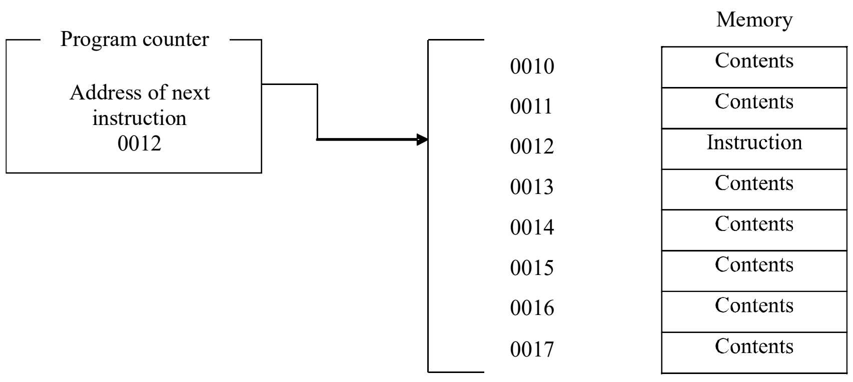


Fig. 2.4 *Program Counter Operation*

Instruction Register

An instruction on our simple computer consists of one 12-bit word. The leading four bits form the operation code (opcode) which specifies the action to be taken, and the remaining 8 bits, when used; indicate the memory address of one of the instruction's operands. For those instructions that have two operands, the other operand is always contained within the accumulator. The instructions stored in memory constitute a program. A single address computer instruction usually has two parts:

- The Operation Code
- The Operand

For e.g., ADD B, which when executed, leaves the sum of the contents of the accumulator and the contents of memory location B in the accumulator.

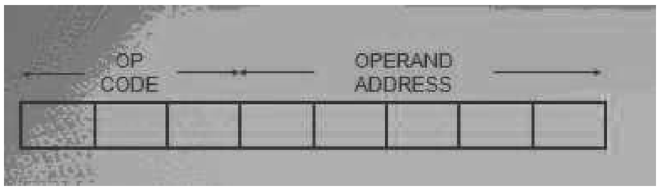


Fig. 2.5 *Instruction Format*

Table 2.1 gives eight instructions that form the instruction set we have chosen for our machine. Also shown in the table is the sequence of control signals necessary for execution of each of the instructions in the machine's instruction set and for fetching the next instruction. In each case the register transfers required for execution of each step are shown. For example, in the case of the LDA (Load Accumulator) instruction, the first step consists of copying the address of the operand, contained in the least significant 8 bits of the instruction register, to the memory address register. Thus the EI (Enable IR) and LM (Load MAR) control signals are active. The next step is to read the operand from memory into the memory data register. An active R (Memory Read) signal performs that task. The last step required to execute the LDA instruction is to copy the contents of the memory data register to the accumulator. Active ED (Enable MDR) and LA (Load Accumulator) do the trick. The Instruction register contains the actual instruction to be executed. After fetching the instruction from memory, the microprocessor places it in IR for translation.

NOTES

An Instruction Set for the Basic Computer

Table 2.1 *Instruction Set for Basic Computer*

Instruction Mnemonic	Op-Code	Execution Action	Register Transfers	Ring Pulse	Active Control Signals
LDA	1	ACC<--(RAM)	1. MAR <-- IR	3	EI, LM
(Load ACC)			2. MDR <-- RAM(MAR)	4	R
			3. ACC <-- MDR	5	ED, LA
STA	2	(RAM) <--ACC	1. MAR <-- IR	3	EI, LM
(Store ACC)			2. MDR <-- ACC	4	EA, LD
			3. RAM(MAR) <-- MDR	5	W
ADD	3	ACC <-- ACC + B	1. ALU <-- ACC + B	3	A
(Add B to ACC)			2. ACC <-- ALU	4	EU, LA
SUB	4	ACC <-- ACC - B	1. ALU <-- ACC - B	3	S
(Sub. B from ACC)			2. ACC <-- ALU	4	EU, LA
MBA	5	B <-- ACC	1. B <-- A	3	EA, LB
(Move ACC to B)					
JMP	6	PC <-- RAM	1. PC <-- IR	3	EI, LP
(Jump to Address)					
JN	7	PC <-- RAM if negative flag is set	1. PC <-- IR if NF set	3	NF: EI, LP
(Jump if Negative)					
HLT	8-15	Stop clock			
"Fetch"		IR <-- Next Instruction	1. MAR <-- PC	0	EP, LM
			2. MDR <--RAM(MAR)	1	R
			3. IR <-- MDR	2	ED, LI, IP

1	1	1	5
CD	MAP	HLT	CRJA

Fig. 2.6 *Next Address Field of the Microinstruction Register*

In Figure 2.6, CD is the condition bit MAP causes the address of the next microinstruction to be obtained from the address ROM, HLT stops the clock, and CRJA is the control ROM jump address field.

The Instruction Register (IR) stores the instruction currently being executed. In simple processors, each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and ultimately executed.

Status Register

The status register contains individual bits, each holding a special meaning. These bits are termed as flags. Each flag is either set or reset by an ALU operation. These flags are used by conditional branch instructions. Examples of typical flags are carry, sign, zero and overflow.

- The Carry (C) flag reflects whether an arithmetic operation, such as ADD generates a carry or not. If carry is generated, then CF = 1 else CF = 0. The carry is generated by the 8th bit for byte operations, 16th bit for word operations, etc. In addition, carry is used as a Borrow flag for subtraction.
- The Sign(S) flag indicates whether the result is positive or negative. SF = 1 indicates negative result which means that the most significant bit of the

result is 1. If SF = 0, the result is a positive number. This result is observed only for signed operations.

- The Zero (Z) flag indicates whether the result of an arithmetic or logic operation is zero. ZF = 1 for zero result and ZF = 0 for a non-zero result.
- The Overflow (O) flag is set if the result of an arithmetic and logical operation on signed numbers is too large for the microprocessor's maximum word size. OF can be shown as OF = C7 Å C8 where C7 is the final carry and C6 is the previous carry. Once again, this applies to signed numbers only.

NOTES

The status register is sometimes also called the *condition code register, or flag register.* This is because it also functions as a special register which keeps track of certain facts about the outcome of arithmetic, logical and other operations. This register enables the microprocessor to test for certain conditions and then to perform alternate functions based on those conditions. These actions are performed by using ***flags***.

Now, let us take an overall look at flags. The status register is divided into individual bits, each having their own unique functions. Each bit is called a *flag*, which either keeps track of, or 'Flags,' certain conditions. However, not every operation or instruction affects every flag. Some instructions affect several flags, and some don't affect any. When referring to flags, the following logic is used. 'If some condition has come to be, or is true, the flag uses a 1 to say, "Yes" this is true or has happened.' Causing a flag to become 1 is **setting a flag.** Causing a flag to become 0 is called **clearing a flag**. Figure 2.7 shows a model of a typical status register.

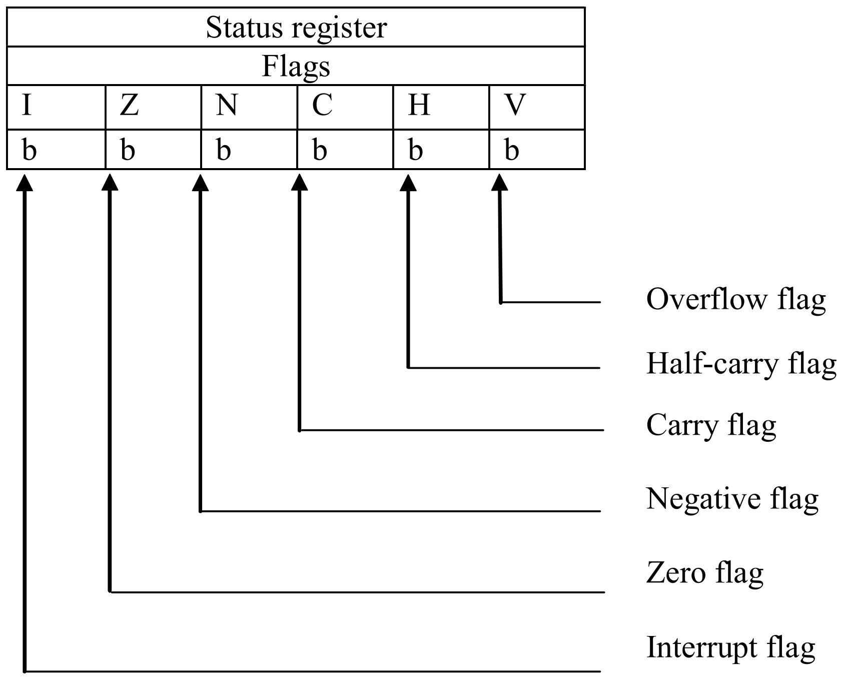


Fig. 2.7 *Model of a Typical Status Register. (b's represent bits.)*

NOTES

The zero flag keeps track of whether the last operation which affects this flag produced a result of zero. This flag is set or 1 if a zero has been produced as a result, and is cleared or 0 if a nonzero result has been produced.

The negative flag tells us if the last operation which affects this flag produced a negative number. When 8-bit signed binary numbers are used, if bit 7 (the eighth bit) of the number is 1, then the number is negative and the N flag will be set; if bit 7 of the number is 0, then the number is positive and the N flag will be either cleared or 0. (This negative flag is sometimes called a sign flag and is indicated with an 'S')

The carry flag tells us if the last operation which affects this flag produced a carry from bit 7 (in 8-bit systems) of the accumulator (bit 7 is the left-most or most significant bit) into the carry bit. The carry flag also tells us if, during subtraction, a borrow flag was required to be used into bit 7. To understand how a borrow flag is indicated depends on the microprocessor used, Figure 2.8 shows how to carry form bit 7 into a carry flag.

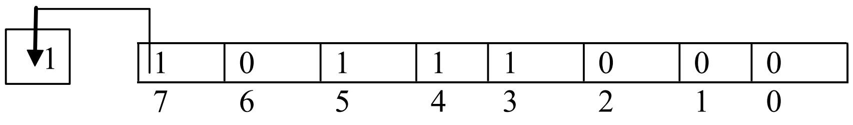


Fig. 2.8 *'Carry' from Bit 7 into the Carry Flag*

The half-carry flag tells us if the last operation that affects this flag was an arithmetic operation that produced a carry from bit 3 to bit 4. This feature is mainly used with BCD (Binary-Coded-Decimal) numbers.

The overflow flag tells us if the last operation which affects this flag caused a result that is outside the range of signed binary numbers for the word size being used at the time. In the case of 8-bit microprocessors, this is + 127 or – 128. If this range is exceeded, the overflow flag is set (1) to warn the programmer.

The interrupt flag (interrupt mask, interrupt flag, interrupt enable bit) prevents maskable interrupts from occurring when it is set and allows them when cleared.

Index Register

Another type of register is the index register. The way the index of a book helps a person locate information, similarly, the index register is used to locate data. The index register is generally used as an aid in accessing data from a table stored in the memory. The index register(s) can be either incremented by 1 or decremented by 1; but normally it does not have other arithmetic or logical capabilities.

The access data of tables or arrays is possible with the use of the index register. In addition, index register can be used to manipulate the address/location of the instruction to facilitate accessing the appropriate data in the table. The actual address called physical address of the data is calculated by adding the address with the instruction including the contents of the index register.

In 8086: MOV AL, 200[SI] refers to one byte present at an address. (DS) + 200 + (SI) will be moved into AL register. (DS – Data Segment register).

Barrel Shifter

The barrel shifter is a very important part of a combinational logic block. It was incorporated in the 386 processor and is also used in microcontrollers. Intel has since moved on to software implemented barrel shifters in their Pentium 4s but AMD still uses it to this day. The designed circuit should shift a data word by any number of bits in a single operation. An N-bit shifter would require $\log_2 N$ number of levels to implement. For an 8 bit barrel shifter, it would require 3 logic levels. Barrel shifters have the ability to shift data words in a single operation over standard shift left or shift right registers that utilize more than one clock cycle.

Barrel shifters will continue to be used in smaller devices because it has a speed advantage over software implemented ones. The 8-bit Barrel shifter is shown in the Figure 2.9.

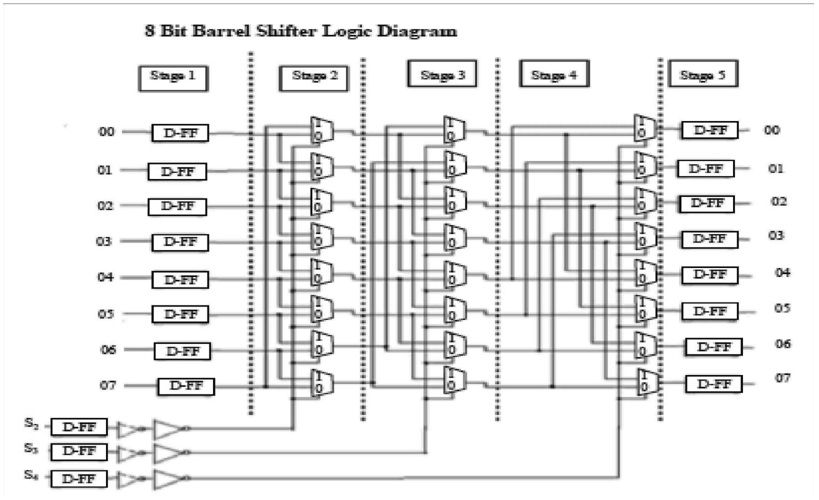


Fig. 2.9 8-Bit Barrel Shifter

In the Figure 2.9, an 8 bit Barrel Shifter is designed that shifts data from an input to an output depending on the combination of three select lines. The above design of the barrel shifter allows for an increase in the number of input bits without having to modify the existing design.

32-bit processors include a special type of register called barrel shifter. This register provides faster shift operations. For example, Intel's 80386 barrel shifter can shift a number from 0 to 64 positions in one clock period.

The registers are sometimes classified as the Control and Status Register and User Visible Register. This classification has been illustrated in Figure 2.10:

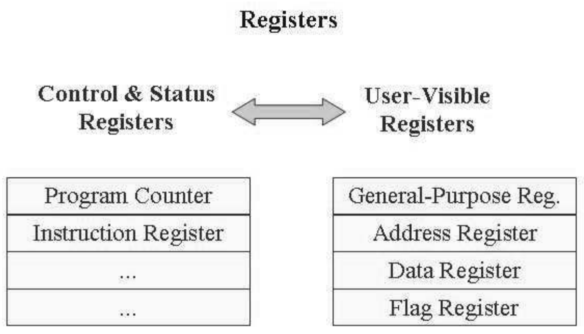


Fig. 2.10 Controls and Status Register and User Visible Register

Stack and Stack Pointer

The stack is a special place in memory, which is often used to store critical pieces of data during subroutines and interrupts. In other words, a **stack** is a part of the memory, which temporarily stores data. In a typical computer system, memory is logically divided into distinct areas. For example, your program code may be stored in one area; the variables may be in another and another area for the stack. Figure 2.11 shows a basic illustration of how memory is allocated to a user program running.

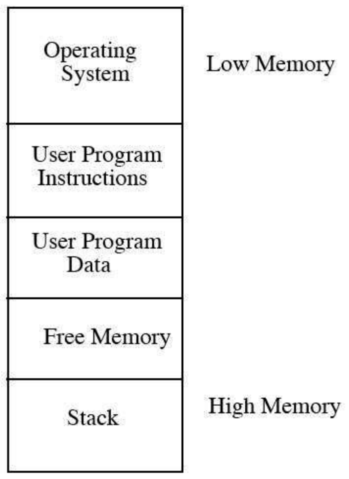


Fig. 2.11 *Memory Allocations to Programs*

The area of memory with addresses near zero, are called low memory whereas, high memory refers to the area of memory that is near the highest address. The area of memory used for your program code is fixed, i.e., once the code is loaded into memory it neither grows nor shrinks.

The stack on the other hand, requires varying amounts of memory. The actual memory required is based on how the stack is used in the program. This implies, that the size of the stack varies during program execution. We can store information on the stack and retrieve it later. the implementation of the subprogram facility is the most common usage of stack. This usage is transparent to the programmer, i.e., the programmer need not explicitly access the stack. The instructions to call a subprogram and to return from a subprogram automatically access the stack. This facilitates in returning the subprogram to the correct place in program when it has completed. The point where control returns after a subprogram is completed is known as the return address. The call instruction places the return address of a subprogram on the stack. Once the subprogram finishes, the return instruction recovers/retrieves the return address from the stack and transfers the control to that location. In addition, the stack is used to pass information to subprograms and to return information from the subprograms, i.e., as a mechanism for handling high-level language parameters.

Conceptually, a stack as the name implies, is a stack of data elements. The size of the elements depends on the processor for example, 1 byte, 2 bytes or 4 bytes. Still, a stack is illustrated in Figure 2.12.

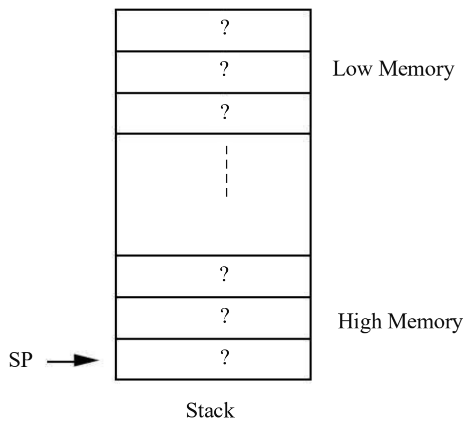


Fig. 2.12 *A Stack*

The processor uses the stack to keep track of where the items are stored on it. It does this by using the Stack Pointer (SP) register.

The SP register is one of the processor's special registers, which points to the top of the stack, i.e., it contains the address of the stack memory element containing the value that was last placed on the stack. When an element is placed on the stack, the stack pointer contains the address of that element on the stack. If a number of elements is placed on the stack, then the stack will always point to the last element placed on the stack. However, while retrieving elements from the stack, they are retrieved in the reverse order. This will become clearer when you write some stack manipulation programs. This concept is explained by a small example. Suppose you have a stack that can hold letters. Call it stack. What would a particular sequence of Push and Pops do to this stack? Start the steps as follows:

You can begin with an empty stack:

```
——
stack
```

Now, let us perform Push(stack, A), giving:

```
——
| A |   <— top
——
stack
```

Again, another push operation, Push(stack, B), giving:

```
——
| B |   <— top
——
| A |
——
stack
```

Now, let us remove an item, letter = Pop(stack), giving:

```
——                ——
| A |  <— top      | B |
——                ——
stack                letter
```

And finally, one more addition, Push(stack, C), giving:

```
——
| C |   <— top
——
| A |
——
stack
```

You would notice that the stack enforces a certain order to the use of its contents, i.e., the Last thing In is the First thing Out. Thus, we say that a stack enforces the **LIFO** order.

The two basic stack operations that manipulate the stack are known as PUSH and POP. The 8086 push instruction places (pushes) a value on the stack. The stack pointer still keeps pointing to the value pushed on the stack. For example, if ax contains the number 123, then the instruction: **`PUSH AX`** results in the value of ax being stored on the stack. In this case, number 123 is stored on the stack and SP points to the location on the stack where 123 is stored. The 8086 POP instruction retrieves a value that was previously placed on the stack. The stack pointer now, still points to the next element on the stack. Theoretically, POP removes the value from the stack. The value that was stored on the stack, can be retrieved by: **`POP AX.`** This transfers the data from the top of the stack to ax, (or any register). In this case, the number 123 is transferred. The data is stored on the stack starting from high memory locations. The stack pointer will successively point to the lower memory locations as data is placed on the stack. In other words, the stack grows downwards. For example, if you assume that the top of the stack is location 1000 (SP contains 1000) then the operation of push ax is as follows.

Firstly, SP is decremented by the size of the element (2 bytes for the 8086) to be pushed on the stack. Then, the value of ax is copied to the location pointed to by SP, in this case, it is 998. Next, in case you assign the value 212 to bx, and carry out a PUSH bx operation, then the SP is again decremented by two, giving it the value 996 and 212 is stored at this location on the stack. Now, there are two values on the stack. As mentioned earlier, in case you want to retrieve these values, you discover and understand the fundamental feature of any stack mechanism. The values are retrieved in reverse order. This means that the last item placed on the stack, is the first item to be retrieved. Such a process is called a Last-In-First-Out process or a LIFO process.

Subsequently, if you carry out a POP ax operation, ax gets value 212, i.e. the last value pushed on the stack. Now, in case you carry out a POP bx operation, then bx gets 123 as its value; this is the second last value pushed on the stack. Therefore, the operation of POP is to copy a value from the top of the stack, as pointed to by the SP and increment SP by 2, so that now it points to the previous value on the stack. It is possible to push the value of any register or memory variable on the stack. The value from the stack can be retrieved and stored in any register or memory variable.

Note: For the 8086, you can only push 16-bit items onto the stack e.g. any register.

The following are ILLEGAL: PUSH AL, POP BH

The preceding example is illustrated in Figure 2.13. Steps (1) to (4) correspond to the states of the stack and stack pointer after each instruction.

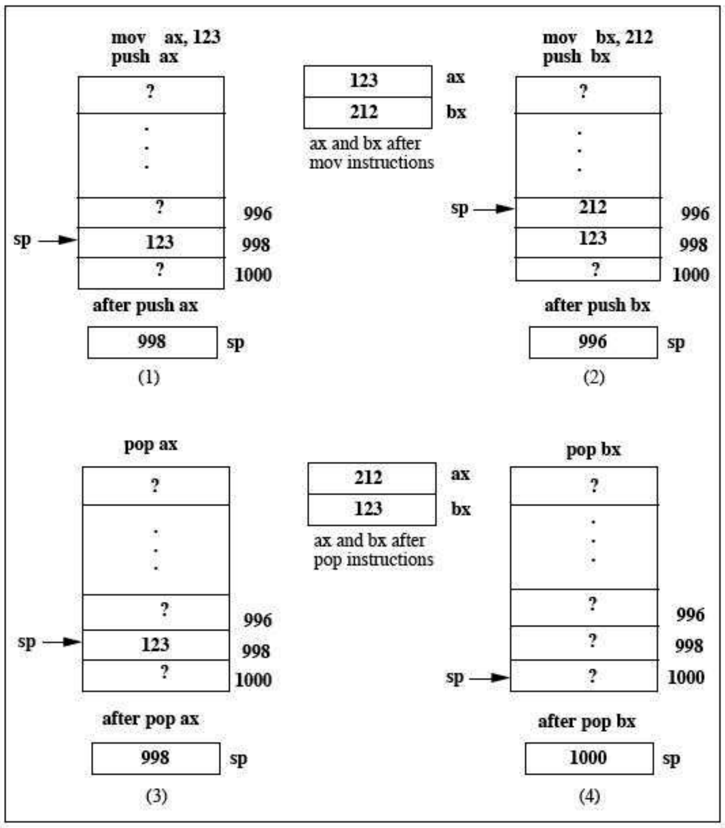


Fig. 2.13 *Stack and Stack Pointer Operations*

For a clearer understanding of the stack, study the structure of a stack as shown in Figure 2.14:

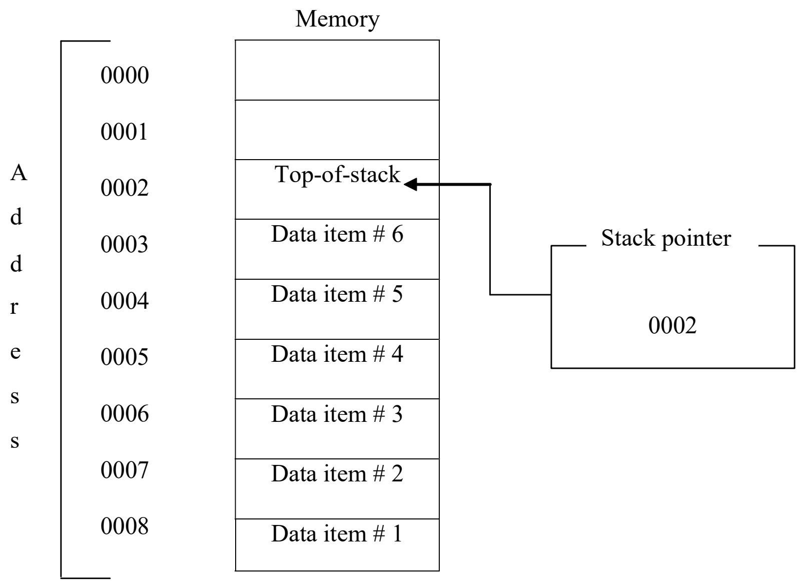


Fig. 2.14 *Typical Stack and Stack Pointer*

NOTES

The structure of the stack is *First-In-Last-Out* (FILO) type structure. Unlike the main memory, where you can access any data item in any order, the stack is designed in such a way that you can access only the top of the stack. The data placed in the stack, goes on top of the stack. In case you want to remove data from the stack, then it must be on the top before it is removed. Let us see how the stack has come to be. To do so, refer to Figure 2.15. Data item # 1 is the first item placed on the stack.

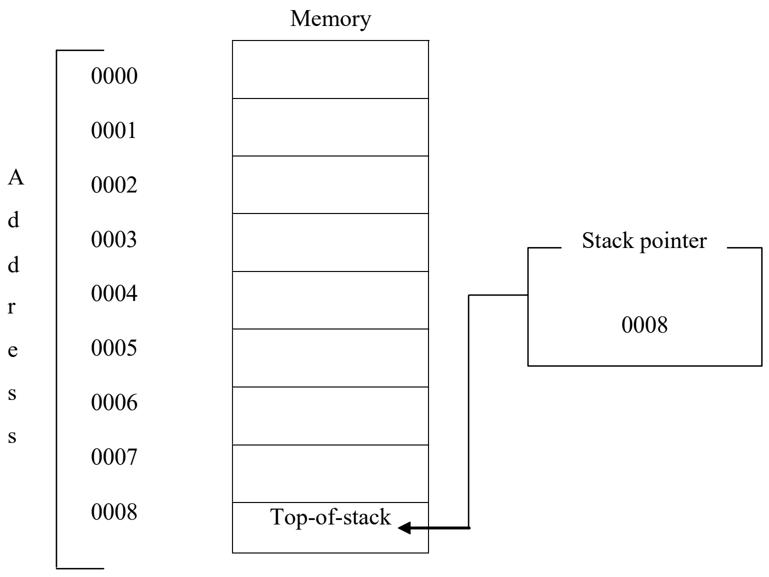


Fig. 2.15 *Typical Stack with Stack Pointer*

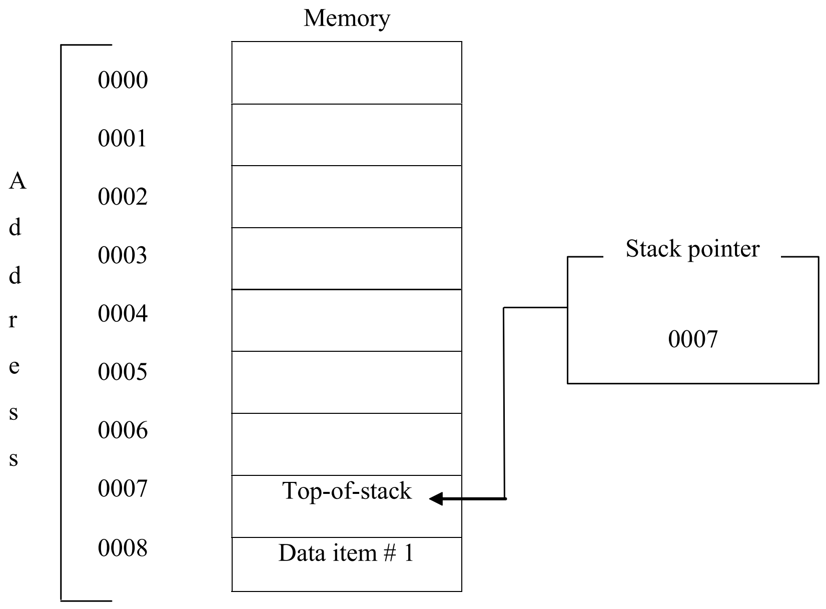


Fig. 2.16 *Operation on Stack with Stack Pointer*

The stack pointer in Figure 2.15 is 'Pointing' to memory location 0008; therefore, data item # 1 is placed in the stack at that memory location. This act of putting a part of the data in the stack is called ***pushing*** data onto the stack. It is as though the data is being pushed in from the top. Now study Figure 2.16.

Data item # 1 has been pushed onto the stack and the stack pointer has been decremented or decreased by one, which means that it is now pointing to the memory location 0007. Location 0007 is now the top-of-the-stack. Push data item # 2 onto the stack. The stack will appear as shown in Figure 2.17.

When data item # 2 is ***pushed*** onto the stack, it is placed in the location the stack pointer was pointing to –0007. The stack pointer is now decremented to 0006. At any point of time, this data will be required in the stack; and it will be retrieved from the top-of-the-stack. This action is called ***popping*** or ***pulling*** data from the stack. Now, let's reverse the entire process. As each data item is removed, the stack pointer drops, which in this case, means that it will point to the next-greater memory address.

Example 2.1

Study Figure 2.17. If data item # 2 is pulled from the stack, will the stack pointer increment or decrement? What hexadecimal value will appear in the stack pointer?

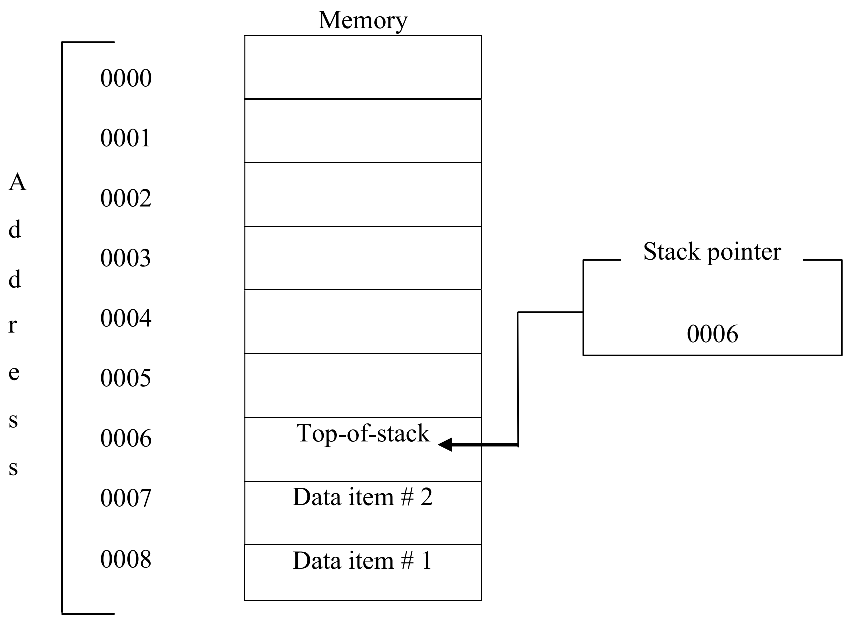


Fig. 2.17 *A Stack and Stack Pointer*

Solution

The stack pointer will be incremented when data item # 2 is pulled from the stack. The hexadecimal value 0007 will appear in the stack pointer. In fact, the stack will appear as it appeared in Figure 2.16.

The important points to note about the stack are:

- A stack is a Last-In-First-Out (LIFO) read/write memory. The items that go in last come out first. This is because all read (POP) and write (PUSH) operations take place from one end, that is, the Top Of the Stack (TOS).
- Stack is implemented by using hardware or software.

NOTES

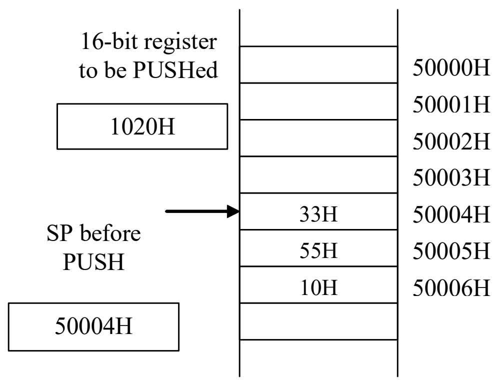
- The hardware stack is designed by using a set of high-speed registers to provide a fast response. The disadvantage is that the stack size is limited. But push and pop operations are fast. Intel's 4040, an 8 – bit processor, uses hardware stack.
- The software stack on the other hand, is implemented by using a portion of the memory. Some RAM locations are demarcated as stack. The advantage is that they provide unlimited space for stack, depending on the amount of memory allocated to the microprocessor, though. However, it is slower than hardware stack.
- The SP always contains the memory address of the last byte of the currently pushed item on TOS i.e. it always points to the TOS. A stack is generally used by subroutines or interrupts for saving certain registers such as the program counter and status register.

PUSH and POP Operations

- If the stack is accessed from the top, the stack pointer is decremented after a PUSH operation and incremented before a POP operation.
- On the other hand, if the stack is accessed from the bottom, SP is incremented after a PUSH and decremented after a POP.
- Typical microprocessors access stack from the top.
- Depending upon the microprocessor, 8-, 16- or 32-bits can be either pushed onto or popped, from the stack.
- The value by which the SP is incremented or decremented after PUSH or POP operations depends on the register size. For example, in 8086 microprocessor, PUSH and POP operations can be done only on 16-bit data. Hence, SP always is incremented or decremented by a value of two always.

The PUSH and POP operations are illustrated in the Figure 2.18.

Before PUSH



After PUSH

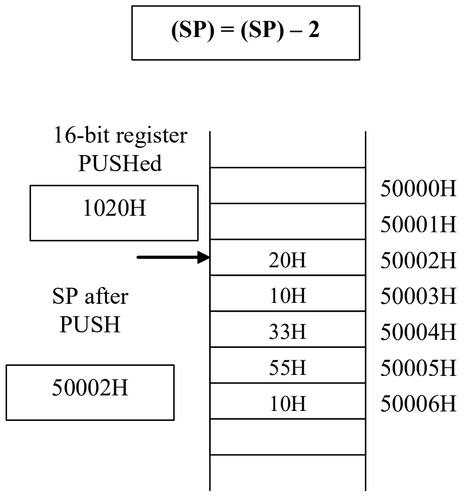
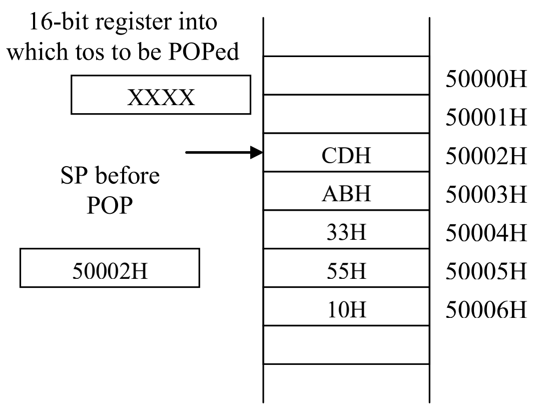


Fig. 2.18(a) *PUSH and POP Operations*

Before POP



After POP

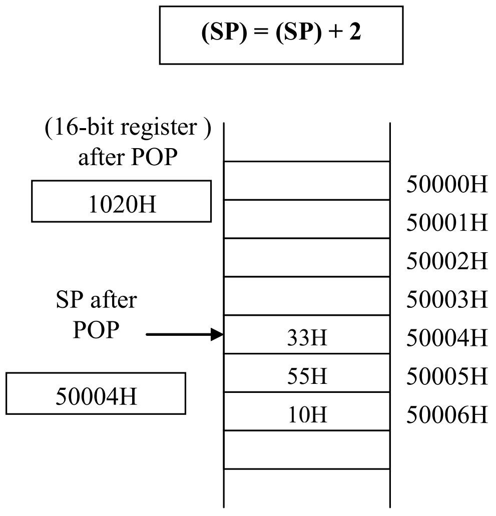


Fig. 2.18(b) *PUSH and POP Operations*

The PUSH operation as shown in Figure 2.18(a) of the stack is accessed from the top. SP is decremented by 2 after PUSH. Similarly, after POP operation, SP is incremented by 2, as the stack is accessed from the top.

Example 2.2

Using the stack, swap the values of the ax and bx registers, so that now ax contains what bx contained and bx contains what ax had contained.

Solutions

To carry out this operation, at least one temporary variable is required.

Solution 1:

```
PUSH AX  ; STORE AX ON STACK
PUSH BX  ; STORE BX ON STACK
POP AX   ; COPY LAST VALUE ON STACK TO AX
POP BX   ; COPY FIRST VALUE TO BX
```

The preceding solution stores both ax and bx on the stack and utilizes the LIFO nature of the stack to retrieve the values in reverse order, thus swapping them. All that is actually required is to store one of the values on the stack; therefore the following is a more efficient solution.

Solution 2:

```
    PUSH AX        ; STORE AX ON STACK
    MOV AX, BX     ; COPY BX TO AX
    POP BX         ; COPY OLD AX FROM STACK
```

When using the stack, the number of items pushed should be equal to the number of items popped. This is crucial if the stack is being used inside a subprogram. This is because when a subprogram is called its return address is pushed on the stack. Incase something is pushed inside the subprogram, do not remove it, and then the return instruction will retrieve the item you left on the stack instead of the return address. This means that your subprogram is not able to return to where it was called from and it will most likely crash!

ALU (Arithmetic and Logical Unit)

ALU (Arithmetic and Ligical Unit) performs arithmetic and logic operations on the data. The size of ALU defines the size of the microprocessor. For example, Intel 8086 is a 16-bit microprocessor since its ALU is 16-bits wide. Intel 8088 is also a 16-bit microprocessor even though its data bus is 8-bits wide. That is because of its 16-bit ALU. Some 32-bit microprocessors like Motorola 68030 include multiple ALUs for parallel operations to achieve faster speed.

Table 2.2. *ALU Function Table*

S_3	S_2	S_1	S_0	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X$ and Y
1	x	0	1	$G = X$ or Y
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

In Table 2.2 a sample function table for an ALU is shown. All of the arithmetic operations have S_3=0, and all of the logical operations have S_3=1. These are the same functions we saw when we built our arithmetic and logic units a few minutes ago. Since our ALU only has 4 logical operations, we don't need S_2. The operation done by the logic unit depends only on S_1 and S_0.

The Control Unit (CU)

The CU (Control Unit) primarily executes basically two tasks:

Instruction Interpretation

(i) CU reads instructions from memory using PC.

(ii) It recognizes the instruction type, obtains the necessary operands, and then routes to the appropriate functional units of the execution unit.

(iii) It issues necessary signals to perform the desired operation.

(iv) The results are routed to the specific destination.

Instruction Sequencing

The CU determines the address of the next instruction to be executed and loads it to PC. The CU is designed by using one of the three techniques:

(i) Hardwired Control: Designed by physically connecting typical components such as gates and flip-flops. For example, Zilog's 16-bit Z8000

(ii) Microprogramming: This type of CUs include a control ROM for translating the instructions. Intel's 8086 is a microprogrammed microprocessor.

(iii) Nanoprogramming: It includes two ROMs inside the CU. The first ROM, which is called microROM, stores all the addresses of the second ROM, which is called the nanoROM. If the microinstructions repeat many times in a microprogram, the use of two-level ROMs provides tremendous memory savings. Motorola's 68000, 68020 and 68030 are nanoprogrammed.

Width of Registers in a Microprocessor

All registers have a maximum capacity. They hold only a certain number of bits where the width is generally 8, 16, or 32 bits.

NOTES

8-Bit Registers

An 8-bit register is 8 bits wide, i.e., it can hold 1 byte as shown in Figure 2.19. Most computers and trainers you will be using will not display an 8-bit register in binary. Instead, they will have a hexadecimal display.

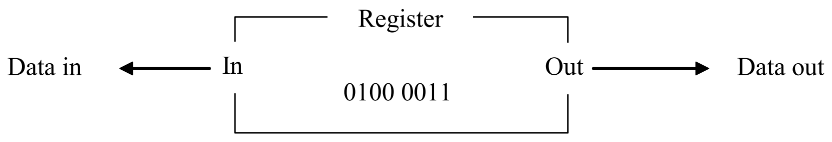


Fig. 2.19 8-bit Register Model

However, it is useful to separate the 8 bits into two groups of 4 bits each. The left group of 4 bits is called the upper nibble, and the right group of 4 bits is called the lower nibble. This is illustrated in Figure 2.20:

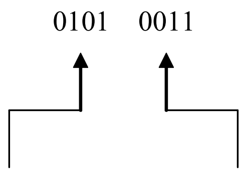


Fig. 2.20 *Upper- and Lower-Nibble Positions*

Example 2.3

If a register contains the binary number shown in Figure 2.21 what would appear in the hexadecimal display for that register?

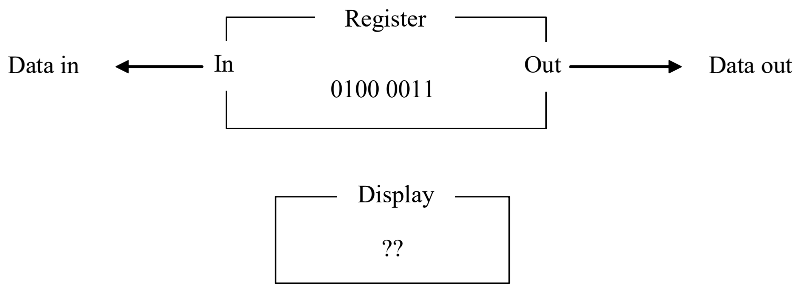


Fig. 2.21 *A Binary Number*

Solution

The upper nibble, 1100, is the same as the hexadecimal digit C. The lower nibble, 1011, is the same as the hexadecimal digit B. Therefore, the hexadecimal display will show CB.

16-Bit Registers

A 16-bit register is 16 bits wide, which is illustrated in Figure 2.22. As you can see, the 16 bits are again separated into groups of 4 bits. Each nibble, or group of 4 bits, is represented in the display as 1 hexadecimal digit.

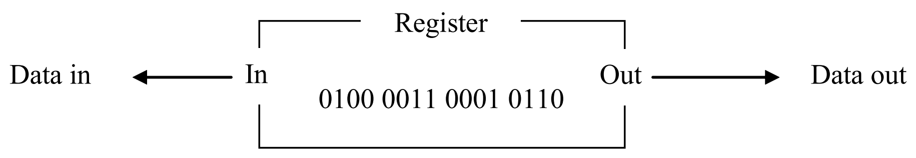


Fig. 2.22 16-bit register model

Example 2.4

What are the binary contents of the register when the output displayed is as shown in Figure 2.23?

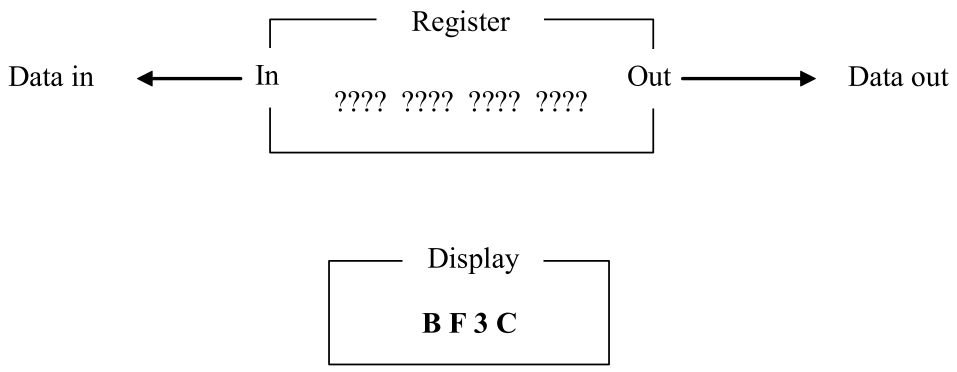


Fig. 2.23 Register

Solution

The far left digit (called the *most significant digit*) B, has a binary equivalent of 1011. The F would be 1111. The 3 would be 0011 and the hexadecimal digit C would be represented by 1100 in binary. Putting the four nibbles together produces 1011 1111 0011 1100, which constitutes the binary contents of this register.

2.3 EXTERNAL SYSTEM AND BUS ARCHITECTURE

A bus is a subsystem that transfers data between computer components inside a computer or between computers. Bus-based computers are structured in terms of processors and memory, and are connected to a backbone bus that acts as a 'Superhighway' for data or instructions to move between processors and memory. In practice, the external system bus architecture has the same components as the Von Neumann architecture. However they are arranged along a bus, as shown in Figure 2.24.

NOTES

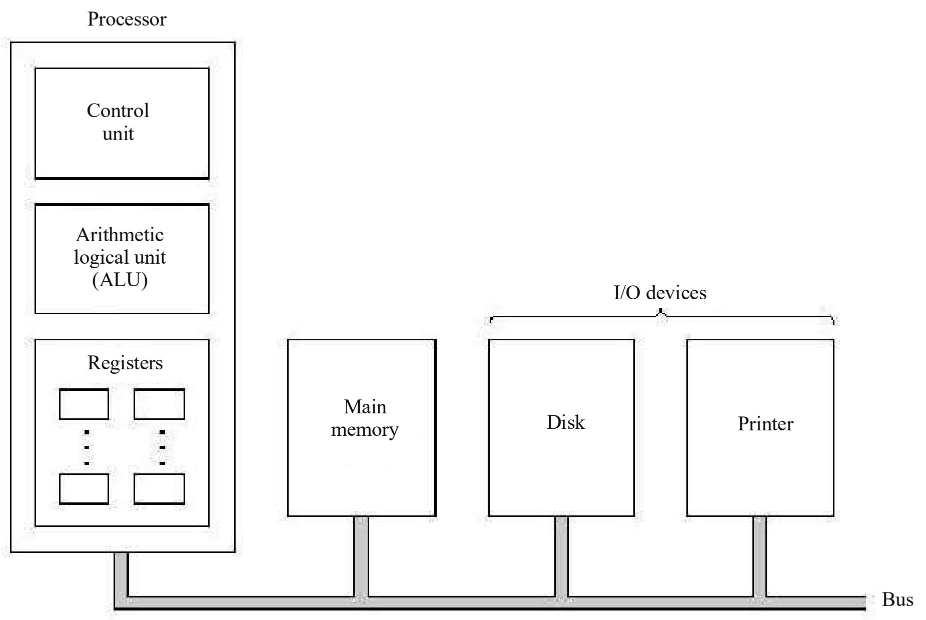


Fig. 2.24 *External System Bus Architecture*

The following Figure 2.25 shows the layout of components in a modern PC, which is a useful example of bus architecture.

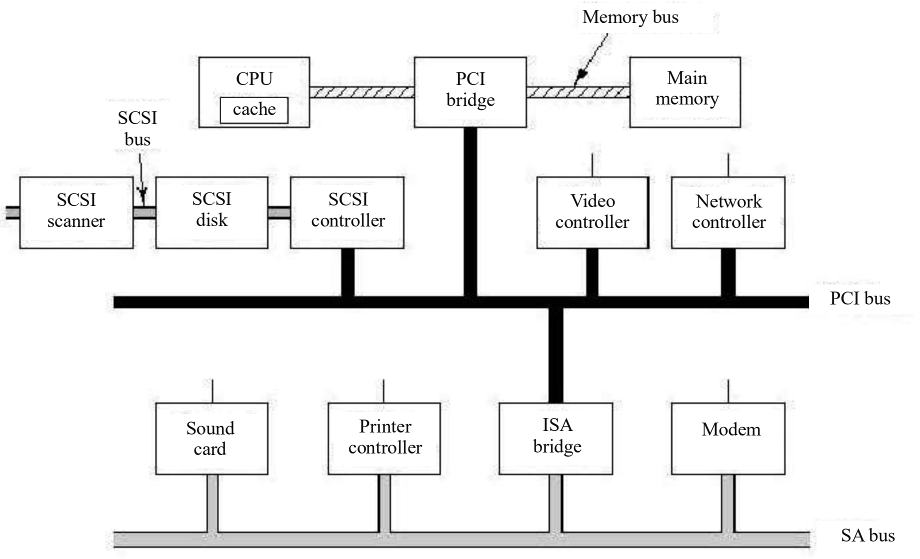


Fig. 2.25 *Schematic Diagram of PC Bus Architecture*

In Figure 2.25, there are multiple buses. The PCI bus mediates large-scale data transfer between external components, as does the ISA bus. The SCSI bus is used to daisy chain peripherals such as disk drives, scanners, etc.

2.3.1 The System Bus

The CPU has to be able to send various data values, instructions and information to all the devices and components inside your computer as well as the different peripherals and devices attached. If you look at the bottom of a motherboard you will see a whole network of lines or electronic pathways that join the different components together. These electronic pathways are nothing more than tiny wires that carry information, data and different signals throughout the computer between the different components. This network of wires or electronic pathways is called the 'Bus'.

It is further divided into:

- Address Bus
- Data Bus
- Control Bus

They together connect the microprocessor to each of the memories and I/O elements, which facilitate the information transformation between them.

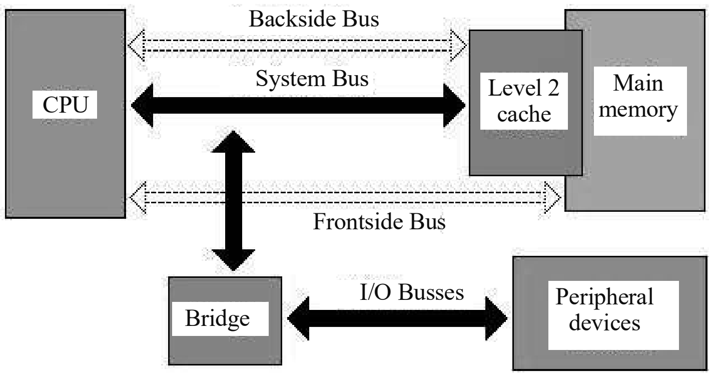


Fig. 2.26 *System Bus*

In Figure 2.26, you will find that the system bus connects the CPU and Level 2 cache to the main memory. It connects the bridge known as Front Side Bus (FSB), that is, an electrical pathway on a computer's motherboard. It connects the various hardware components to the main microprocessor or Central Processing unit (CPU). The Level 2 cache is used to supply the stored information to the processor without any delay (wait-state). For this purpose, the system bus is used.

Address Bus

Most microprocessors store information and instructions in a wide range of memory locations. Usually, the memory locations are in a memory chip rather than in the microprocessor. The microprocessor needs a way to tell the memory chip which memory location it wants to put data into or take data from. It does this through the **address bus**.

The address bus refers to a connection between the microprocessor and the memory chips. Actually, it is simply a group of electrical paths which are connected to RAM, ROM and the I/O chips. Through this bus, the microprocessor can specify the address of any memory location in any chip or device. On the address bus, the information travels in one direction only; from the microprocessor to the memory and I/O.

The address bus is characterized by the following important points:

- Unidirectional flow of information – From microprocessor to memory or I/O elements only.
- Usually 8 to 32-bits wide - The amount of unique addresses a microprocessor can generate depends on the width of the bus. For example, 8085 has a 16-bits address bus. So, it can generate 2^{16} = 65,536 different addresses. A different memory location or an I/O element can be represented by these addresses.
- Identifies the memory location or I/O device (also called port) the processor intends to communicate with.
- 20 bits for the 8086 and 8088 - 8086 has a 20-bit address bus and therefore addresses all combinations of addresses from all 0s to all 1s. This corresponds to 2 20 addresses or 1M (1 Meg) addresses or memory locations.
- 32 bits for the 80386/80486 and the Pentium.
- 36 bits for the Pentium Pro.
- The size of the memory actually specifies the number of address lines that are addressed.

Table 2.3 provides information about the number of address lines and the corresponding memory space required.

Table 2.3 *Address Lines and Memory Space*

No. of Address Lines	Size of Memory Space
8	2^8 = 256
16	2^{16} = 65 536 = 64 K
20	2^{20} = 1 048 576 = 1 M
24	2^{24} = 16 777 216 = 16 M
32	2^{32} = 4 294 967 296 = 4 G

Data Bus

Once the microprocessor has specified which memory location or device it wants to put data into or take data from, then it needs a set of electrical paths for this information to travel on. This set of paths is electrical and is called the **data bus.**

This 'Set of electrical paths' allows data to flow from one chip to another. Figure 2.27 shows that information on the data bus travels both to and from the microprocessor, memory, and I/O devices. 8-bit microprocessors have a data bus that is 8 bits wide; 16-bit microprocessors have a data bus that is 16 bits wide. This implies that the bus consists of 8 or 16 parallel connecting paths.

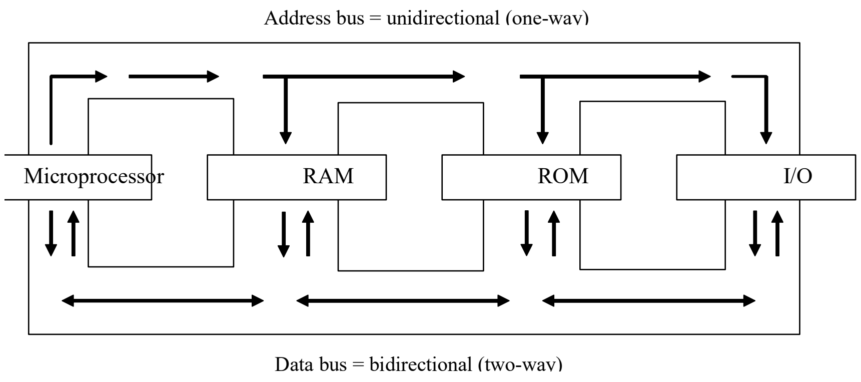

Address bus = unidirectional (one-way)
Microprocessor
RAM
ROM
I/O
Data bus = bidirectional (two-way)

Fig. 2.27 *Two-Way Information Flow in Data Bus*

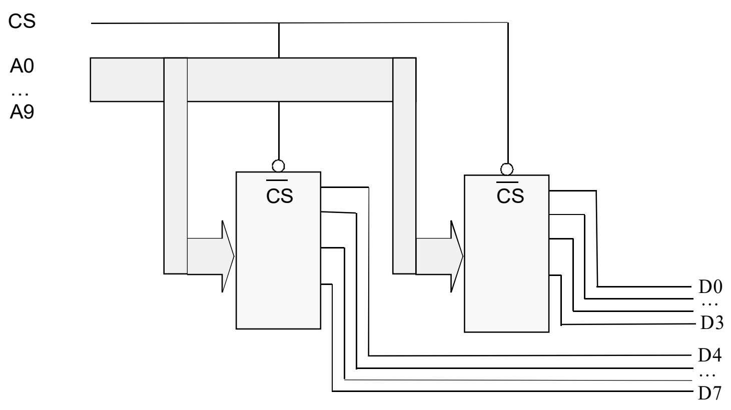


Fig. 2.28 *Chip Select Signal CS*

Control Bus

The control bus consists of wires, some of which transfer signals from the CPU to external devices and others that carry signals from external devices to the CPU. The number of wires present in the control bus varies from one mP to another. Examples of control bus signals are READ/, WAIT, READY, and HOLD.

The Control Bus is characterized by the following :

- Some signals are unidirectional and others, bidirectional.
- Transmits signals that coordinate the functions of the operations of individual microcomputer elements.
- Typical control signals are READ, WRITE, and RESET.
- Control bus is Unidirectional, i.e., in one direction only.

To identify which address is a memory address or an I/O port address of the following:

o Memory Read
o Memory Write
o I/O Read
o I/O Write

- When Memory Read or I/O Read are active, data is an **input** to the processor.

- When Memory Write or I/O Write are active, data is an **output** from the processor.
- The control bus signals are defined from the processor's point of view. Hence, these are microprocessor-specific.

Addressing Range

Let's look at the normal range of addresses possible with 8-bit computers. Assume that you have a fair understanding of the binary number system and learned that each position represents a certain power of 2. It is quite similar to decimal number system that represents a certain power of 10. This comparison is clearly explained in Figure 2.29.

Decimal	10^3	10^2	10^1	10^0
	1,000's	100's	10s	1s

Binary	2^3	2^2	2^1	2^0
	8s	4s	2s	1s

Fig. 2.29 *Comparison of Decimal and Binary System*

The decimal number like 9, 999_{10} where the subscript 10 means that you are using a number in base 10, not only tells you about a quantity of items, but also tells you about the possible combinations.

As the number 9,999 is a four-digit number, therefore, by using 10 different decimal digits at a time, there would be 9,999 + 1 = 10,000, possible numbers that can be created. (You add the 1 because the number 0000 or simply 0 must also be included.) This can also be calculated as 10^4 = 10,000.

If you are interested in giving unique addresses to 10,000 homes on the same street (quite a long street), then it would be possible by using only four digits. The first house would have the address 0, and then continue numbering up to 9,999.

Example 2.5

Using only three digits, how many unique addresses could you give to homes on a single street (a decimal number)?

Solution

Since 10^3 = 1,000 this is the number of unique addresses that are possible.

Now, let's try the same problem in binary: 1111_2 is a binary number. (The subscript 2 denotes the usage of base 2 or binary numbers.) The size of this number is shown as follows.

Binary	2^3	2^2	2^1	2^0
	8s	4s	2s	1s
	1	1	1	1

There is an 8, a 4, a 2, and a 1. If we add this up, we get

8	+	4	+	2	+	1	=	15

NOTES

The number 1111_2 is the same as 15_{10} (decimal 15). This means that by using only 4 binary digits or bits, there are a total of 15 + 1, or 16 unique numbers possible. This can be calculated by using $2^4 = 16$.

If you wanted to give unique binary addresses to 16 houses on the same street , it would be possible to do so with only 4 bits. The first house would be 0000 or simply 0, the next would be 0001, the next 0010, and so on up to 1111.

Example 2.6

Using 12 binary digits, how many unique house addresses would be possible?

Solution

2^{12} = 4,096 unique addresses

This is essentially what is necessary in the matter of addressing memory locations. The highest number that exists in binary using only 4 bits is 1111_2 (15_{10}). That means, if you had only four address lines – that is, an address bus with only four lines, then you would be have only a maximum of 16_{10} different addresses. (0000 counts as one address.) Obviously, this is not enough. Consider Figure 2.30, which illustrates the number of unique addresses possible with different numbers of address lines.

2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	32,768		8,192		2,048		512	256	128	64	32	16	8	4	2	1
65,536		16,384		4,096		1,024										

Fig. 2.30 *Powers of 2 and the Number of Memory Addresses Available with Varying Numbers of Address Lines*

As evident in Figure 2.30, in case you decide to use only eight address lines, then it is possible to limit yourselves to 256 memory locations. (Add the values of the first eight positions starting from the far right + 1.) However, in reality, this is inadequate. Most 8-bit chips use 2 bytes for addressing purposes, which then alloy 65,536 different memory locations. (One byte is 8 bits; 2 bytes is 16 bits, which then allows 2^{16} combinations.) This is often adequate. If not, then there is another way of increasing this number by using a method known as bank switching.

Example 2.7

How many memory locations could be addressed by a 10-line address bus?

Solution

2^{10} = 1,024 memory locations can be addressed. 1024 is also referred as 1K.

NOTES

Check Your Progress

1. What does the program counter store?
2. Define the term instruction register.
3. When is the overflow flag set?
4. How will you define the term address bus?
5. Define the term data bus.

2.4 MEMORY

Memory is required to store data and instructions. Data and instructions which can be lost after the power supply is stopped are stored in the Random-Access Memory (RAM). Data and instructions which should never be lost, even after the power is turned off, are stored in Read-Only Memory (ROM). ROM is a type of memory whose contents cannot be changed once the ROM chip is manufactured.

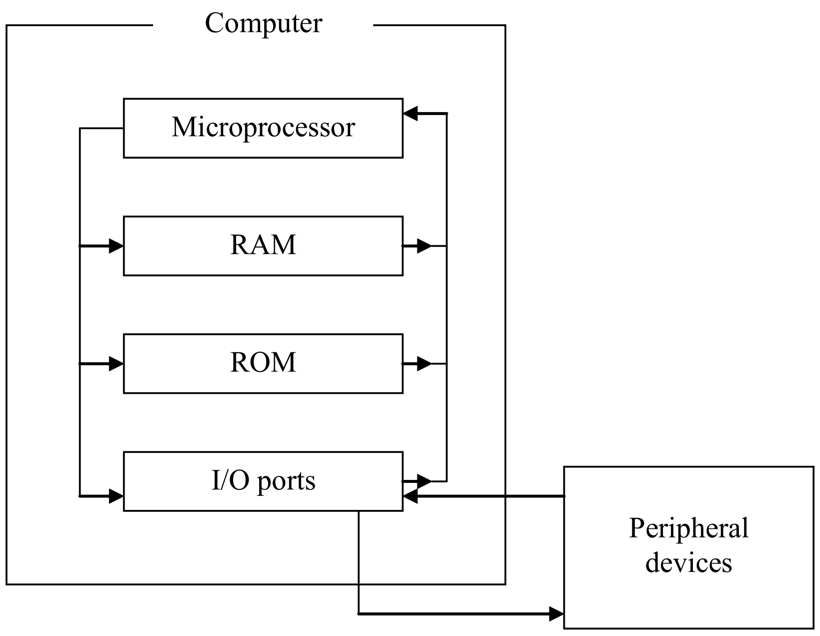


Fig. 2.31 *Block Diagram of a Complete Computer with Peripheral Devices (Arrows Indicate Data Flow)*

Programmable Read-Only Memory (PROM) and Erasable Programmable Read-Only Memory (EPROM) are used in a way similar to the ROM. However PROM can be programmed after being manufactured or even programmed more than once (EPROM). PROM and EPROM differ from RAM in the sense that they require special equipment to program them.

Memory Addressing

As there are several memory locations, it is necessary to have a way of referring to exact locations. This is done through addressing. Typically, memory locations are numbered from 0000 (in hexadecimal numbering) to the highest location used by that particular trainer or computer. This sequential number which is assigned to each location is its **address.** Figure 2.32 diagrammatically represents memory.

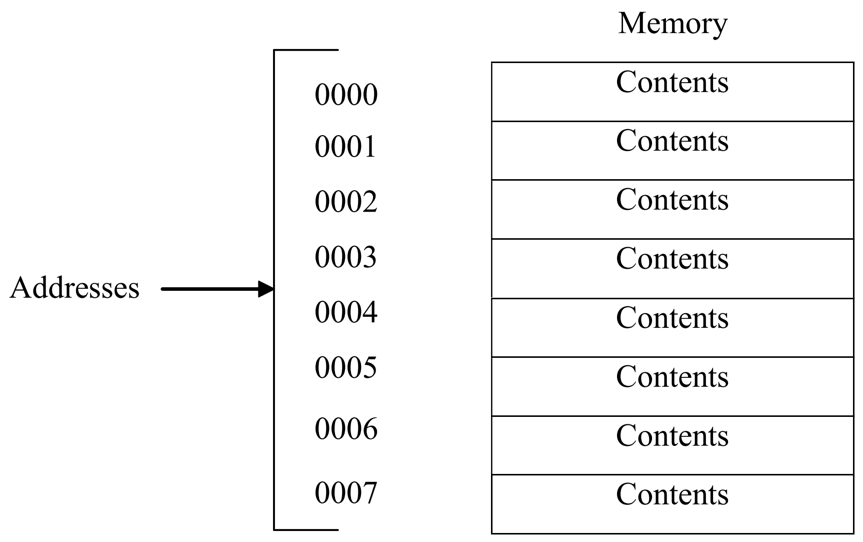


Fig. 2.32 *Memory*

A memory address is similar to the address of your home. Your house has a number or address assigned that no other house on your street has . Inside your house are its *contents;* chairs, beds, and so on. Notice that your home's address and your home's contents are not the same. Basically, each memory address points towards a memory location and each of these memory location contains the actual data. The same is illustrated in Figure 2.33:

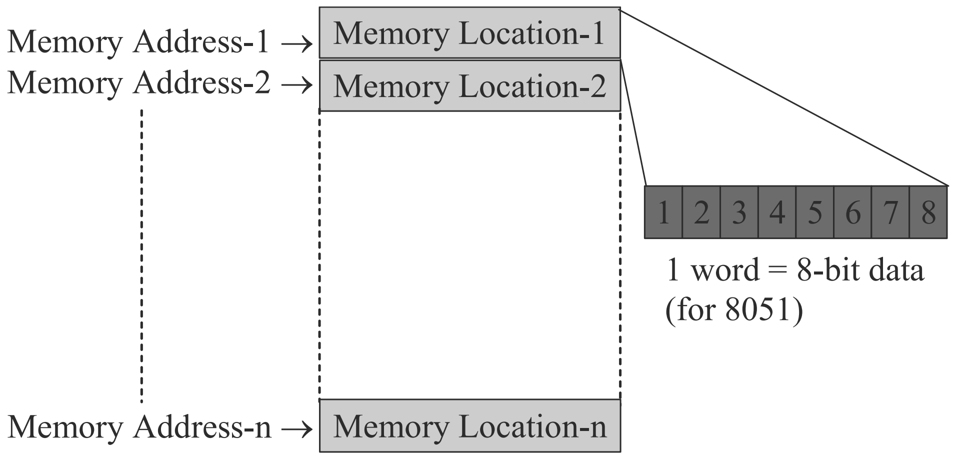


Fig. 2.33 *Memory Addressing*

Each memory location has an address and contents. The address is necessary, to specify which memory location to *read* information from or *write* information to. The contents are the information itself.

Therefore, the characteristics of memory unit are**:**

(i) The memory unit is the integral part of any microcomputer system and its primary purpose is to hold programe and data.

(ii) The major design goal of the memory unit is to permit it to operate at a speed close to that of the processor.

(iii) The cost factor impedes the design of the entire memory unit with a single technology that guarantees high speed.

(iv) To seek a trade-off between the cost and operating speed, a memory system is usually designed with different technologies such as solid, magnetic and optical state.

In a broader sense, a microcomputer memory can be logically divided into three groups:

(i) Processor Memory

(ii) Primary or Main Memory

(iii) Secondary Memory

Processor Memory consists of a set of CPU registers. These registers are useful to hold temporary results when a computation is in progress. In addition, there is consistency in the speed between the registers and the microprocessor because they are fabricated by using the same technology.

The most important disadvantage is the cost involved which forces the architect to include very few registers (usually 8 to 16 only) in the microprocessor.

Primary Memory is the storage area in which all the programs are executed. The size of the primary memory is much larger as compared to processor memory but its operating speed is slower than processor registers by a factor of 25. The processor is able to access only those items that are stored in the primary memory. All the programs and corresponding data must be within the primary memory prior to execution. The MOS technology is normally used in primary memory design.

Secondary Memory refers to the storage medium for huge files, such as program source codes, compilers, operating systems, RDBMS, etc. These are files that are not required frequently. They comprise of slow devices such as magnetic tapes and optical disks.

Sometimes, they are referred to as auxiliary or backup store.

Classification of Primary Memory

Primary memory normally includes ROM and RAM. As the name implies, a **ROM** permits only a read access. There are many kinds of ROMs. For example,

- Some ROMS are custom made. Their contents are programmed by the manufacturer and are called mask programmable ROMs. As they are mass produced, they are inexpensive.
- Sometimes, a user has to program the ROM. Such types of ROMs which allow this operation are called PROMs (Programmable ROMs). The main disadvantage is that they cannot be reprogrammed.
- In practice, it is necessary to alter the programs before they are introduced into the market. ROMs that allow reprogramming are called EPROMs. In an EPROM, programs are entered by using electrical impulses and the stored information is erased by using UV rays.

- With advances in IC technology, it is possible to achieve an electrical means of erasure. These new ROMs are called Electrically Alterable ROMs (EAROMs) or Electrically Erasable PROMs (EEPROMs).

These memories are usually called Read Mostly Memories (RMMs), since they have much slower writing times than read times.

Information stored in semiconductor random access memories will be lost if the power is turned off. This property is known as volatility and hence, **RAMs** are usually called volatile memories. Stored information in a magnetic tape or magnetic disk is not lost when the power is turned off. Therefore, these storage devices are called nonvolatile memories. ROM is a nonvolatile memory.

In a semiconductor, memory is constructed using bipolar transistors, the information is stored in the form of voltage levels in flip-flops. These voltage levels do not usually get drifted away. Such memories are called **static RAMs** because stored information remains constant for a stipulated period of time.

On the other hand, the semiconductor memories that are designed using MOS transistors, the information is held in the form of electrical charges in capacitors. Here, the stored charge has the tendency to be leaked. These memories are referred to as **dynamic RAMs**. To prevent information loss, dynamic RAMs must be refreshed regularly. Refreshing refers to the boosting of the signal level and writing it back. This activity is performed by using a hardware unit called 'refresh logic'.

- Static RAMs require power supplies in large quantities as they maintain information in active circuits, for which power is required even when the chip is inactive or in standby mode. Also, each static RAM cell is about four times larger in area than an equivalent dynamic cell.

The differences between static and dynamic RAMs have been stated in Table 2.4:

Table 2.4 *Difference between Static and Dynamic RAM*

Static RAM	**Dynamic RAM**
1. This semiconductor memory is constructed by using bipolar transistors	1. This semiconductor memory is constructed using MOS transistors.
2. Information is stored in the form of voltage levels in flip-flops	2. Information is stored in the form of electrical charges in capacitors
3. These voltage levels do not get drifted away	3. Has tendency of leakage
4. No refresh logic is needed	4. Refresh logic is necessary due to leakage of electrical charges
5. Power is required even when the chip is in standby mode	5. Refresh login is inbuilt, so comparatively draws less power.
6. Four time larger in size as compared to an equivalent dynamic cell	6. Four times as many bits as a static RAM chip.

NOTES

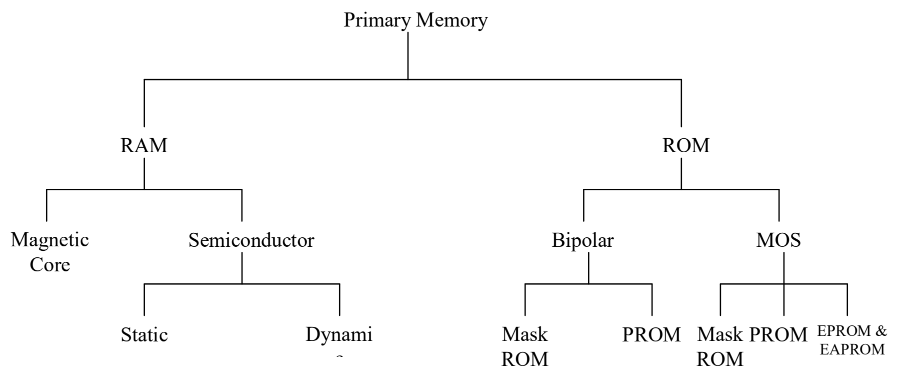


Fig. 2.34 *Classifications of Primary Memories*

Basic Memory Element

The basic memory element is similar to a D latch. This is also refereed as the D – Flip Flop. This latch has an input where the *data* comes in. It has an enable signal on which output data comes out. The D-latch is shown in Figure 2.35(a):

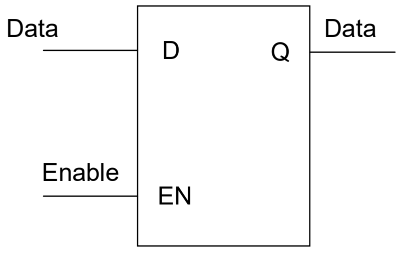


Fig. 2.35(a) *A D Latch*

However, this is not safe as data is always present on the input and the output is always set to the contents of the latch.

– To avoid this, tri-state buffers are added at the input and output of the latch as illustrated in the Figure 2.35(b):

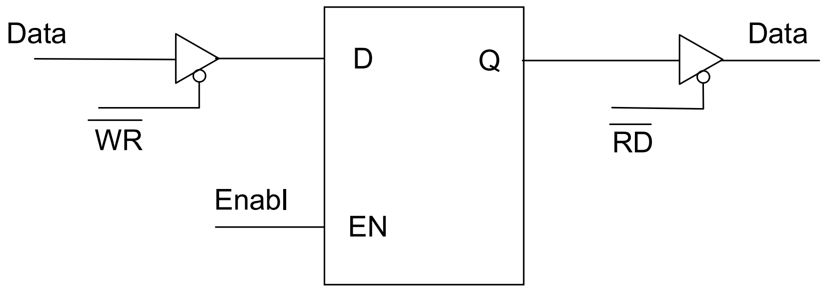


***Fig. 2.35(b) A** D Latch with Tri-state Buffers*

The WR signal controls the input buffer.

– The bar over WR means that this is an active low signal.

– Therefore, if WR is 0, then the input data reaches the latch input.

– If WR is 1, then the input of the latch looks like a wire connected to nothing.

Similarly, the RD signal controls the output.

Therefore, if four of these latches are connected to each other, then you would have a 4-bit memory register, which has been illustrated in Figure 2.36:

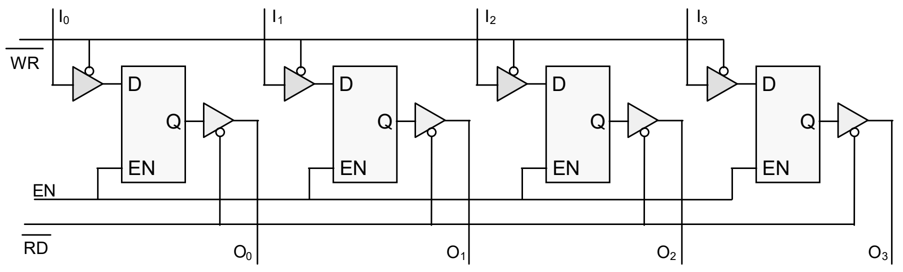


Fig. 2.36 *A 4-Bit Memory Register by Using a D Latch*

Usually an array of registers is formed for memory usage. The same is illustrated in Figure 2.37:

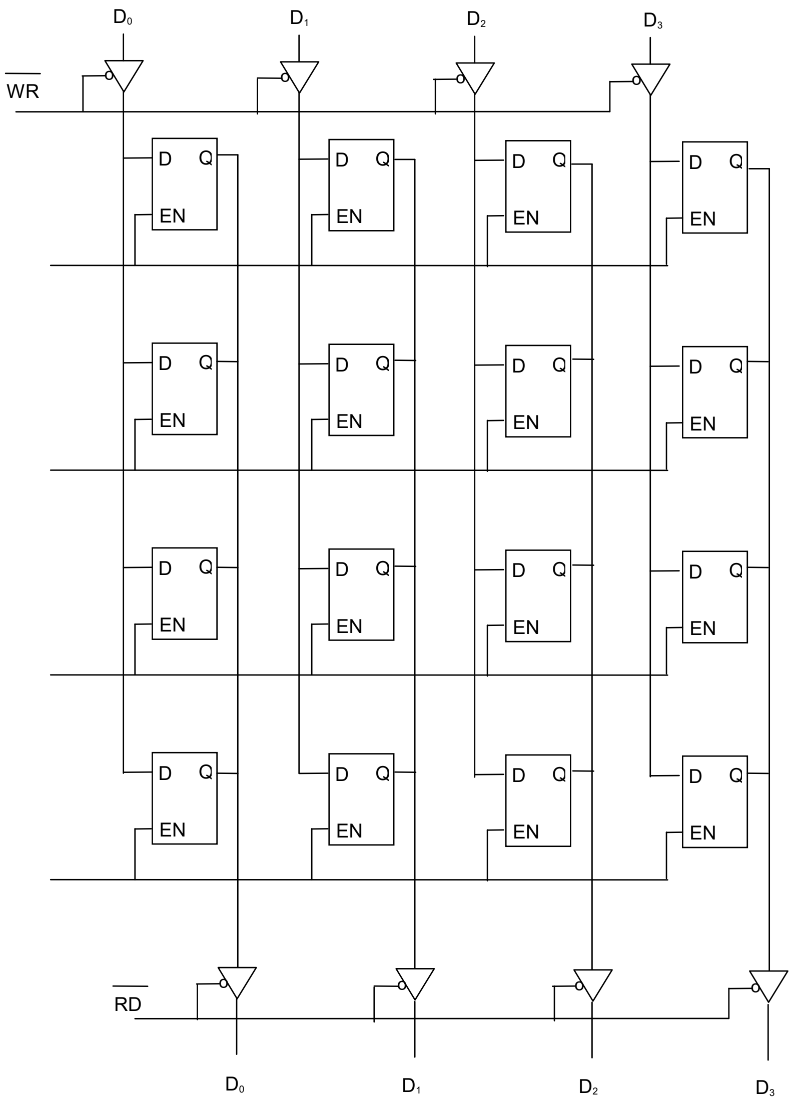


Fig. 2.37 *Array of Registers for Memory Usage*

NOTES

If each memory location (register) is represented as a block, you get the following D latch system :

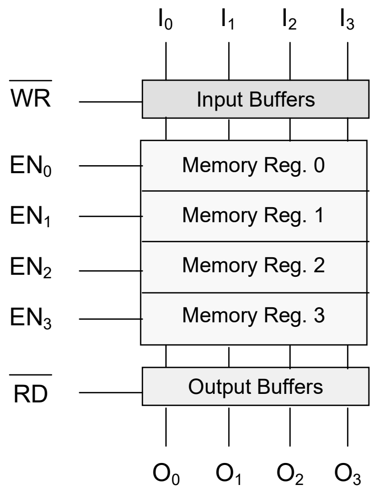


Fig. 2.38 *D-latch Represented as Blocks.*

Now in this system, by using the RD and WR controls, you can determine the direction of flow either into or out of the memory. Then using the appropriate Enable input you can enable an individual memory register.

The present model that has been designed is a memory with 4 locations and each location has 4 elements (bits). This memory would be called 4 X 4 [Number of location X number of bits per location].

Now the question arises - How do we produce these enable line?

Since it is not possible to have more than one of these enables active simultaneously, these lines can be encoded to reduce the number of lines coming into the chip. These encoded lines are the address lines for memory. Therefore, the preceding figure (Refer Figure 2.38) would now be changed and look like Figure 2.39:

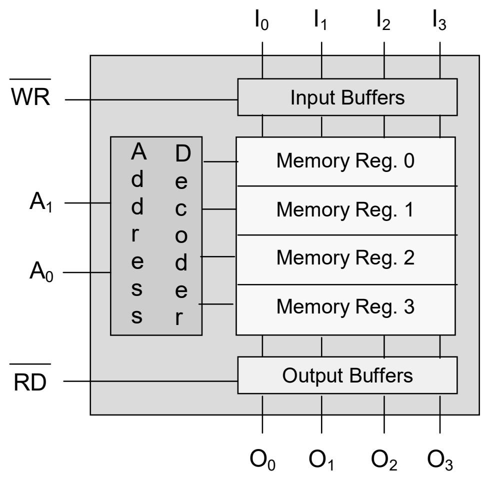


Fig. 2.39 *D Latch with Enable Lines*

Main Memory Array Design

In many applications, a memory of large size capacity is often realized by interconnecting several small size memory blocks. There are two techniques used for designing the main memory in such cases. They are:

a) Linear decoding

b) Fully decoding

First, let us consider the block diagram of a typical RAM IC.

NOTES

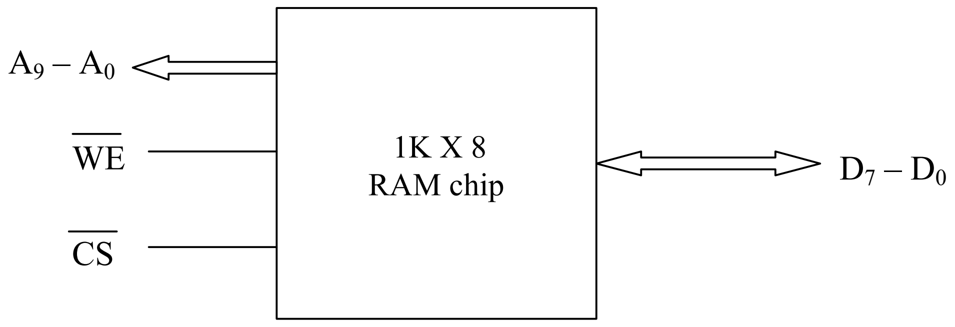


Fig. 2.40 *Typical RAM IC*

The capacity of this chip is 1Kb(Kilobytes), i.e., they are organized in the form of 1024 words with 8 bits/word. Each word has a unique address and is specified on 10-bit address lines A9 – A0. The inputs and outputs are routed through the 8-bit bidirectional data bus (D7 – D0). The operation of this chip is governed by two control inputs: $\overline{WE}$ (Write Enable) and $\overline{CS}$ (Chip Select).

The following truth table describes the operation of this chip:

Table 2.5 *Truth table of 1 K RAM Chip*

$\overline{CS}$	$\overline{WE}$	MODE	Status (D7 – D0)	Power
H	X	Not selected	High Impedance	Standby
L f s	L	Write	Input Bus	Active
L	H	Read	Output Bus	Active

(i) When $\overline{CS}$ is high, the chip is not selected at all, hence D7 to D0 are driven to high impedance state

(ii) When = 0 and = 0, data on lines D7 – D0 are written into the word addressed by A0 through A9.

(iii) When = 0 and = 1, the contents of memory word whose address is on A9 – A0 will appear on lines D7 – D0

Linear Decoding

Consider the problem where you have to connect 6 Kb memory to an 8-bit microprocessor whose address bus width is 16 – bits. The memory chips are available as 1K X 8.

In linear decoding,

(i) Address lines A9 through A0 of the microprocessor is used as the common input to address lines of all memory chips

(ii) The data lines of microprocessor are connected to data lines of all memory chips.

(iii) The remaining address lines are used to select one of the chips () at a time. For example, 000001 selects chip 1, 100000 selects chip 6, etc.

(iv) R/W from microprocessor is connected to for all RAM Chips

The primary advantage of this technique is that it does not need any decoding circuit.

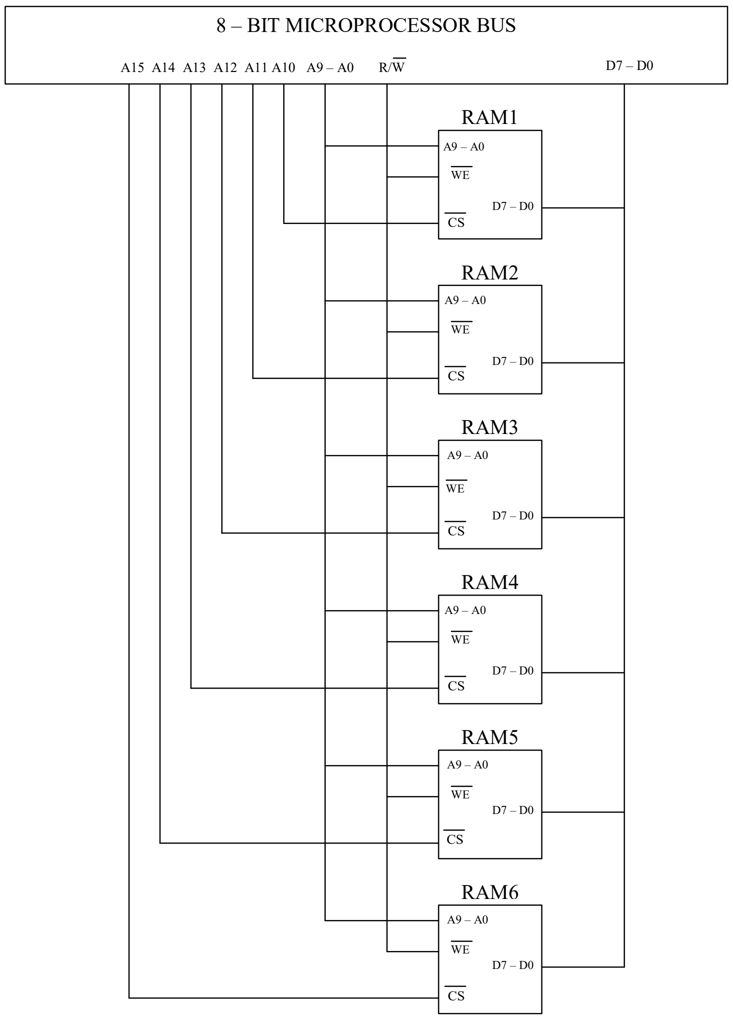

8 – BIT MICROPROCESSOR BUS
A15 A14 A13 A12 A11 A10 A9 – A0 R/$\overline{W}$ D7 – D0
RAM1
RAM2
RAM3
RAM4
RAM5
RAM6
A9 – A0
$\overline{WE}$
$\overline{CS}$
D7 – D0

Fig. 2.41 *Linear Decoding*

Some of the disadvantages of this approach are::

(i) Although there is an address bus of 16-bits wide, you were able to connect only 6Kb of RAM. This clearly wasted address space

(ii) Address map is not contiguous. It is sparsely distributed.

(iii) Conflicts occur if two of the select lines become active at the same time.

(iv) If all unused address lines are not used as chip selectors, then these unused lines become don't cares. This results in foldback, meaning a memory location will have its image in memory map. For example, if A15 is don't care, then address 0000_{16} is same as address 8000_{16}. It wastes memory space.

Fully Decoding

The problems of bus conflict and sparse address distribution are eliminated by the use of fully decoding address technique. Consider an example where an interface 4Kb of RAM to an 8 – bit microprocessor. The RAM chips are available in the form of 1K X 8. For this, perform the following steps.:

First, write the memory map to identify the address lines to be given to decoder logic as provided in Table 2.6:

Table 2.6 *Memory Map to Detect Address Lines*

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Address
0 · 0	0 · 0	0 · 0	0 · 0	0 · 0	0 · 0	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0000 to 03FF
0 · 0	0 · 0	0 · 0	0 · 0	0 · 0	1 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0400 to 07FF
0 · 0	0 · 0	0 · 0	0 · 0	1 · 1	0 · 0	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 0	0800 to 0BFF
0 · 0	0 · 0	0 · 0	0 · 0	1 · 1	1 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0 · 1	0CFF to 0FFF

Observe A10 and A11, 2-to-4 decoder would be an obvious choice for CS signals. Therefore, the truth table can be written as follows:

A11	A10	Device selected
0	0	RAM chip1
0	1	RAM chip2
1	0	RAM chip3
1	1	RAM chip4

Here, when you observe the memory map, there is no windowing between memory addresses and the fold bask is also removed. Above that, address space does not go waste because the unused lines can be used in the future by employing a higher decoder.

NOTES

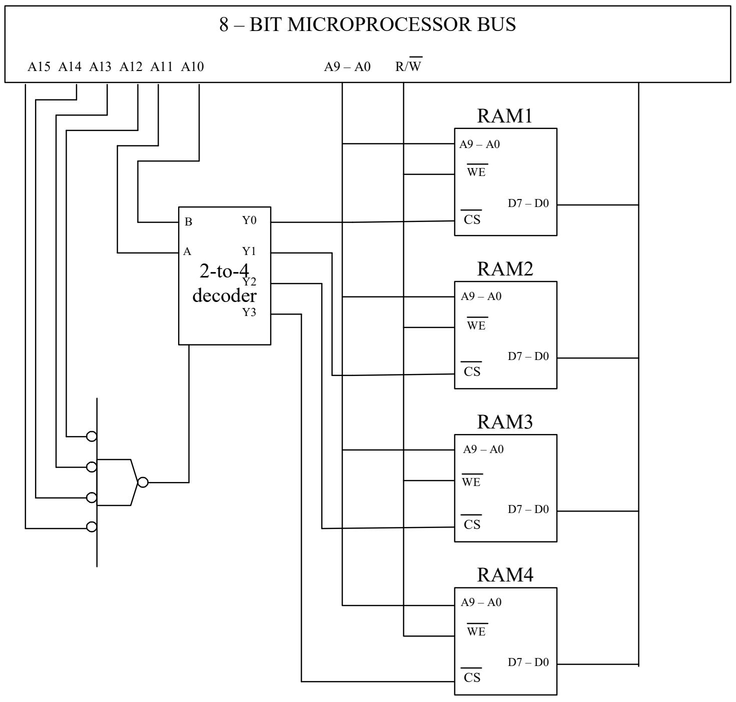


Fig. 2.42 An 8–bit Microprocessor Bus

The typical memory map of a microprocessor is represented in Figure 2.43:

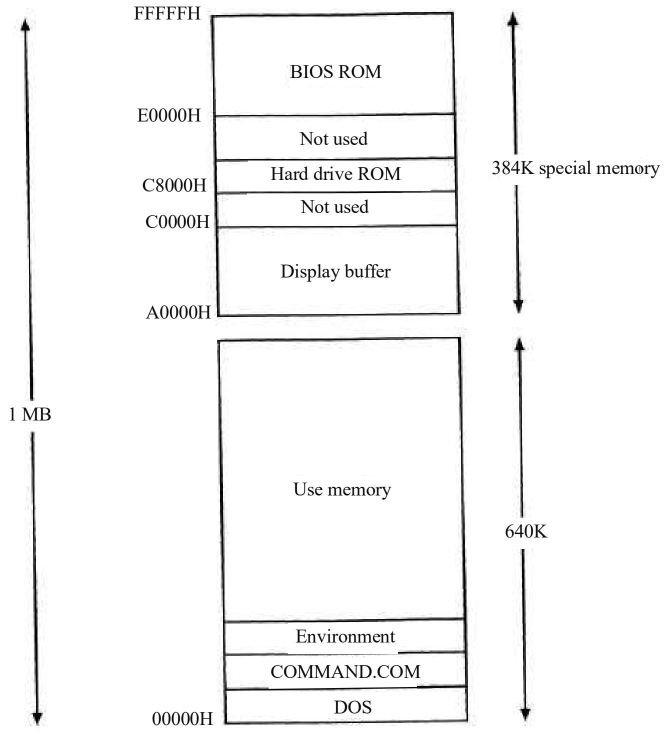


Fig. 2.43 Typical Memory Map

To execute programs, a microprocessor retrieves instructions from memory and executes them, again fetching data from the memory, if required. The two registers that have not been mentioned previously are, namely the **Memory Address Register** (**MAR)** and the **Memory Data Register** (**MDR)**. These registers have been shown in Figure 2.44:

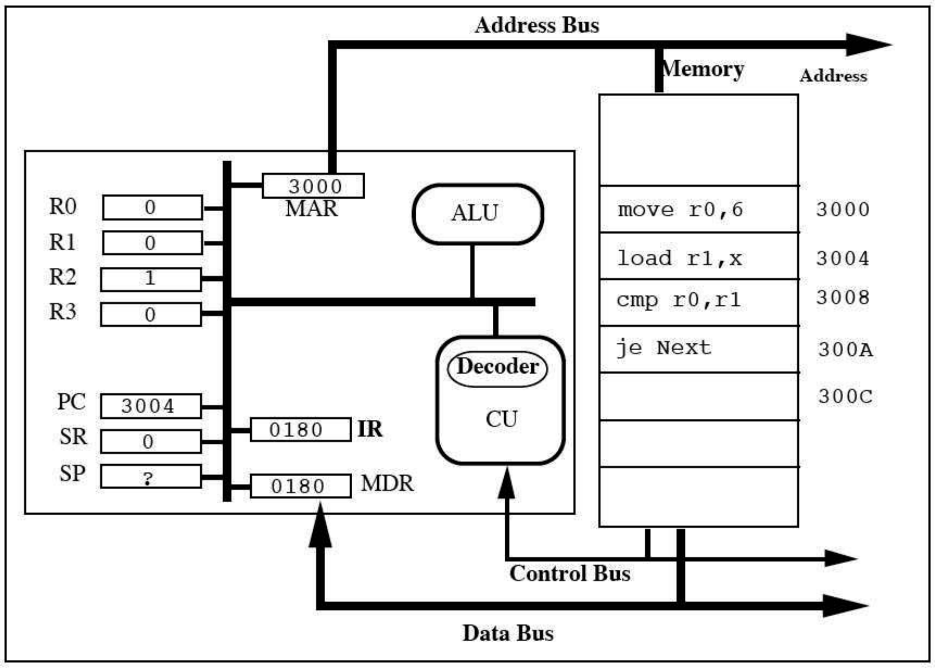


Fig. 2.44 *Location of MAR and MDR*

There are several CPU registers that do not appear in the programming model of a CPU. These registers will be referred as hidden CPU registers to distinguish them from other programming model registers such as R0 to R4, PC, SR and SP registers.

The MAR and MDR registers are used to communicate with the memory (and other devices attached to the system bus). In addition, the preceding figure (Refer Figure 2.44), shows the buses that allow the devices making up a simple Architecture Machine (SAM) to communicate with each other. The MAR register is used to store the address of the location in memory that is accessed for reading or writing. When you retrieve information from the memory, it is referred as the process of reading from memory. When an item is stored in the memory, then it is referred as the process of writing to memory.

In either case, before you can access memory, you must specify the location you want to access, i.e., the address of the location in memory. This address must be stored in the MAR register.

The MAR register is connected to the memory by way of the address bus whose function is to transfer the address in the MAR register to the memory, which in turn informs the memory unit about the location that needs to be accessed. As known, the address bus is a unidirectional bus, i.e. information can travel only in a single direction, from the CPU to the memory and other devices. The MAR register is a 16-bit register and similar to all other SAM registers, it can contain the maximum address of 216 - 1 (65,535) bytes, i.e., it can address up to 64Kb of memory.

The MDR register on the other hand is used either to store information that is to be written to memory or to store information that has been read from memory. The MDR register is connected to memory by the data bus whose function is to transfer information to or from memory and other devices. The data bus is a bi-directional bus, i.e., information travels both, to and from the CPU.

The control bus plays a crucial role in I/O. It carries control signals specifying what operation is to be executed and to synchronize the transfer of information. For example, one line of the control bus is the Read/Write (R/W) line which used to specify whether a read or write operation is to be carried out.

The Valid Memory Address (VMA) line is a line which signals that the address bus carries a valid memory address. This indicates to the memory unit, when to refer to the address bus, for the purpose of finding the address of the location to be accessed. A third line is the Memory Operation Complete (MOC) line. The MOC line indicates that the read/write operation has been completed. At this time, the other devices connected to the computer, (like the I/O and storage devices), generally communicate with the CPU in a way that is similar to the one described for communicating with the memory.

Reading from Memory

The following steps are carried out by the SAM microprocessor to read an item from memory. The item may be an instruction or a data operand.

1. The address of the item in memory is stored in the MAR register.
2. This address is transferred to the address bus.
3. The VMA line and R/W line of the control bus are used to indicate to the memory that there is a valid address on the address bus and that a **read** operation is to be carried out.
4. Memory responds by placing the contents of the desired address on the data bus.
5. Memory enables the MOC line to indicate when the memory operation is complete, i.e., the data bus contains the required data.
6. The information on the data bus is transferred to the MDR register.
7. The information is transferred from the MDR register to the specified CPU register.

Considering the Address, Data and Control Bus, the **READ** operation can be illustrated in three simple Steps provided in Figure 2.45, Figure 2.46 and Figure 2.47:

NOTES

Step 1

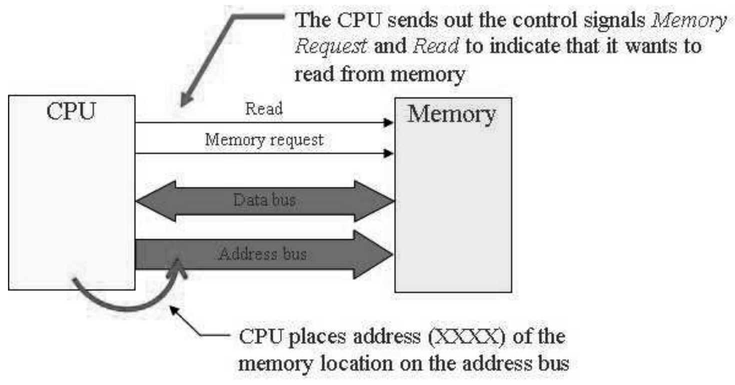

The CPU sends out the control signals *Memory Request* and *Read* to indicate that it wants to read from memory
CPU
Read
Memory request
Memory
Data bus
Address bus
CPU places address (XXXX) of the memory location on the address bus

Fig. 2.45 *Step One of the READ Operation*

In the above Figure 2.45, the upper red arrow shows that CPU sends out the control signals **Memory Request** and **Read** that indicates that data is read from the memory. The lower red curvy arrow shows that CPU places the address (XXXX) extracted from the memory location on the address bus.

Step 2

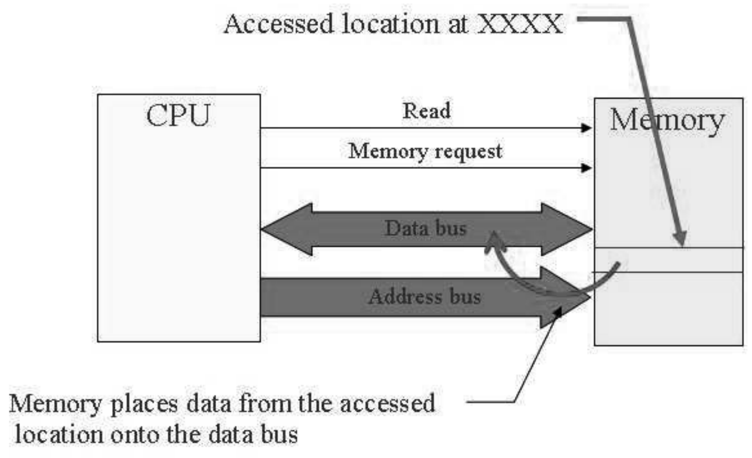

Accessed location at XXXX
CPU
Read
Memory request
Memory
Data bus
Address bus
Memory places data from the accessed location onto the data bus

Fig. 2.46 *Step Two of the READ Operation*

In the above Figure 2.46, the Step 2 is being processed in which upper above red arrow shows that the address (XXXX) is accessed from the memory and lower curvy red arrow shows that the memory places the data from the accessed location onto the data bus.

Step 3

In the Figure 2.47, the upper red arrow depicts that the CPU removes the Memory Request and Read signals. But, the lower curvy red arrow shows that CPU latches the data into the register that has to be processed.

NOTES

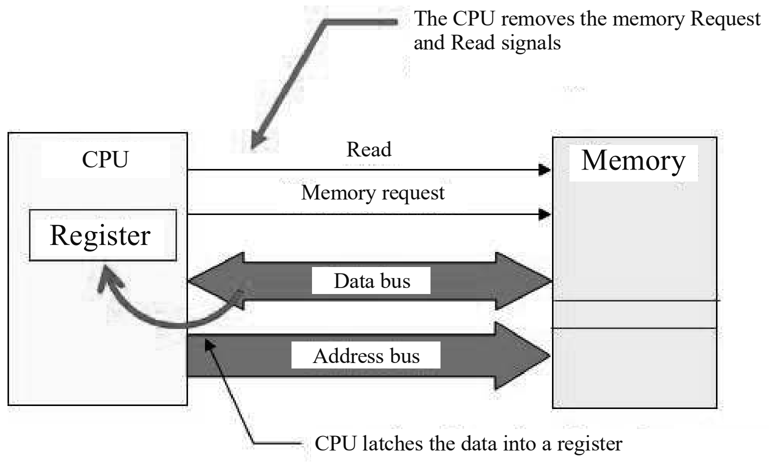

The CPU removes the memory Request and Read signals
CPU
Register
Read
Memory request
Data bus
Address bus
Memory
CPU latches the data into a register

Fig. 2.47 *Step Three of the READ Operation*

Now, it is clear to you how the above three steps are stepped-out to perform the **READ** operation by the Address bus, Data bus and Control Bus. The address bus is the set of wire traces that is used to identify which address in memory the CPU is accessing. A data bus is a group of wires connecting different parts of a circuit with wire carrying a different signal. The data bus is connected to the inputs of several gates and to the outputs of several gates. The address bus is a collection of wires connecting the CPU with main memory that is used to identify particular locations (addresses) in main memory. The width of the address bus, i.e., the number of wires determines how many unique memory locations can be addressed. All the three types of buses are involved together to perform the **READ** operation. The number of wire traces in the address bus limits the maximum amount of RAM which the CPU can address.

Writing to Memory

This process is similar to the procedure of reading from memory:

1. The address of the item in memory is stored in the MAR register.
2. This address is transferred to the address bus.
3. The item to be written to memory is transferred to the MDR register.
4. This information is transferred to the data bus.
5. The VMA line and R/W line of the control bus are used to indicate to memory that there is a valid address on the address bus and that a **write** operation is to be carried out.
6. Memory responds by placing the contents of the data bus in the desired memory location.
7. Memory uses the MOC line to indicate that the memory operation is complete, i.e. the data has been written to memory.

Considering the Address, Data and Control Bus, the **Write** operation can be illustrated by three simple steps provided in Figure 2.48, Figure 2.49 and Figure 2.50:

NOTES

Step 1

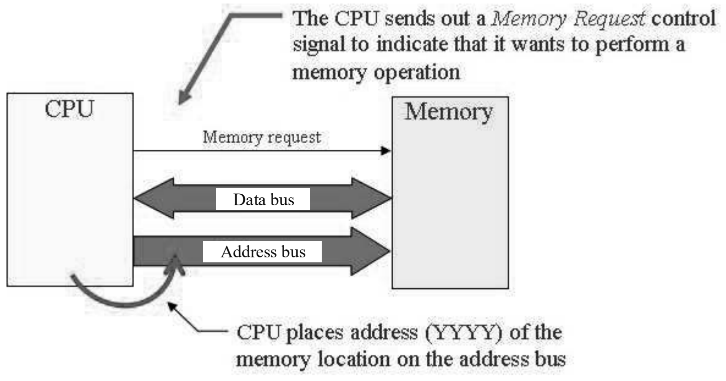

The CPU sends out a *Memory Request* control signal to indicate that it wants to perform a memory operation
CPU
Memory
Memory request
Data bus
Address bus
CPU places address (YYYY) of the memory location on the address bus

Fig. 2.48 *Step One of the Write Operation*

In Figure 2.48, the Step 1 is being processed in which upper above red arrow shows that the CPU sends out a Memory Request control signal to perform the tasks related to memory operation. But, the lower curvy red arrow shows that CPU places address (YYYY) of the memory location on the address bus.

Step 2

In Figure 2.49, Step 1 is being processed in which the upper above red arrow shows that the CPU sends out a **Write** control signal to indicate that valid data is available on the data bus that is later processed from a register onto the data bus.

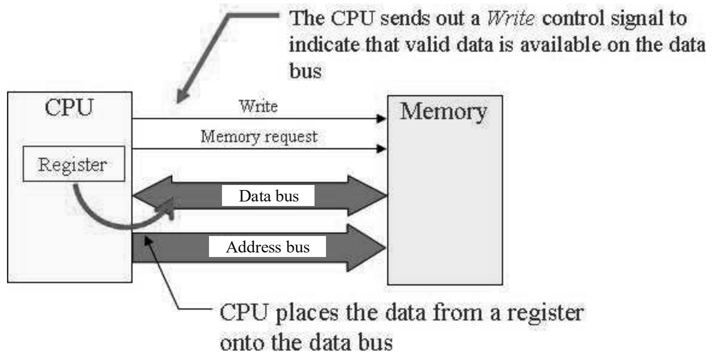

The CPU sends out a *Write* control signal to indicate that valid data is available on the data bus
CPU
Register
Write
Memory request
Memory
Data bus
Address bus
CPU places the data from a register onto the data bus

Fig. 2.49 *Step Two of the Write Operation*

Step 3

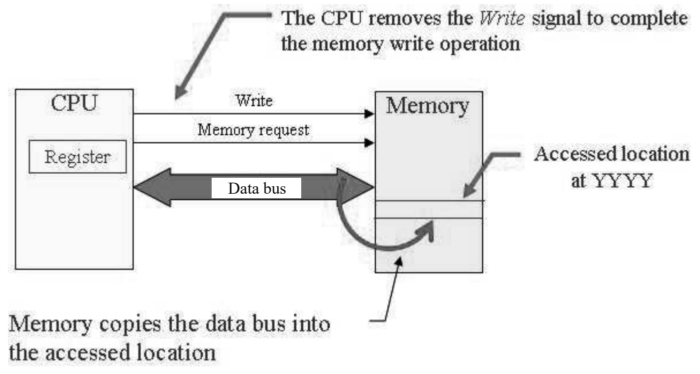

The CPU removes the *Write* signal to complete the memory write operation
CPU
Register
Write
Memory request
Memory
Data bus
Accessed location at YYYY
Memory copies the data bus into the accessed location

Fig. 2.50 *Step Three of the Write Operation*

In Figure, the Step 3 is being processed in which upper above red arrow shows that the CPU removes the **Write** signal to complete the memory write operation and the lower curvy red arrow shows that memory copies the data bus into the accessed location to perform successfully the **Write** operation.

It is evident from the simplified descriptions provided, that accessing memory or any device is quite complicated from an implementation point of view. Therefore, when an instruction to load a register with the contents of a memory variable are executed, such as load r1,i , a lot of activities must be performed.

Firstly, the instruction must be fetched from RAM, then the value of i must be fetched from RAM and finally, the value of i is transferred to register r1. It is important to realize that every operation concerning memory involves either reading from or writing to memory.

Memory is a passive device. It can only store information. No processing can be performed on the information stored in the memory. The information, stored in memory, must be transferred to a CPU register for processing and the result written back to memory. Take for instance, when an instruction, such as 8086 INC instruction is carried out to increment a memory variable (as in INC memvar), its execution involves both a memory read operation and a memory write operation.

First, the value of memvar must be transferred to the CPU where it can be incremented by the ALU. This transfer is carried out by a memory read operation. Then, once this value has been incremented by the ALU, the new value of memvar must be written to the address of memvar in the memory, by the use of a memory write operation.

2.5 INPUT/OUTPUT (I/O) INTERFACE

An individual communicates with the microcomputer system by using the I/O devices interface. The user can enter the program and data using the keyboard on the terminal (input) and execute the program to obtain results (output). Therefore, I/O devices, commonly known as peripherals provide an efficient mode of communication between the computer and the outside world. They consist of a keyboard, Cathode Ray Tube (CRT) display, printers, disks, etc.

The characteristics of I/O devices are (as compared to microcomputers) are:

1. The speed of operation of peripherals is usually slower .
2. Word length of the microcomputer may be different from the data format of the peripheral device.

To make these characteristics compatible, interface hardware circuitry is used, which provide input/output operations between microcomputer and peripherals by using an I/O bus which carries three types of signals: device address, data and command status. The microprocessor uses I/O bus when it executes an I/O instruction.

To make 8-bit microprocessors inexpensive, a separate interface is provided with I/O device. However, a separate 'Intelligent' I/O processor or data channel is provided to 16 – and 23- bit microprocessors, to route all I/O transfers,

When the microcomputer executes an I/O instruction, the following steps are involved:

1. The control unit decodes the **op-code** field and identifies that it is an I/ O instruction.
2. Then the microprocessor places the device address and command from the respective fields of the I/O instruction on to the I/O bus.
3. The interface of the various devices connected to this I/O bus, decode this address, and the appropriate interface is selected.
4. The identified interface decodes the command line and determines the function to be performed. Typical functions include receiving data from an input device into the microprocessor or sending data to an output device from the microprocessor.

In a typical microcomputer system, the user gets involved with two types of I/O devices:

- *Physical I/O*
- *Virtual I/O*

Physical I/O

When the microcomputer has no operating system, then the user works directly with physical I/O devices and performs detailed I/O design. There are three ways of transferring data between the microprocessor and a physical I/O device.

1. Programmed I/O
2. Interrupt Driven I/O
3. Direct Memory Access (DMA)
   - Data transfer between microcomputer's memory and an external device occurs without microprocessors involvement in Direct Memory Access.

For a microcomputer with an operating system, the user works with virtual I/O devices. The user does not have to be familiar with the characteristics of the physical I/O device. Instead, the user performs data transfers between the microcomputer and the physical I/O device indirectly by calling the I/O routines provided by operating system using virtual I/O instructions.

Programmed I/O

The microprocessor executes a program to communicate with an external device via a register called I/O port for programmed I/O. The microcomputer communicates with an external device through one or more registers called I/O ports. They are occasionally fabricated by the manufacturer in the same chip as the memory chip to achieve minimum chip count. There are usually 2 types of I/O ports:

1. Each bit in the port can be individually configured as either input or output port

NOTES

2. For the other type, all bits in a port can be set up as either all parallel outputs or parallel inputs. Each port can be configured as an input or output port by another register called **command** or **data direction** register. The port contains the actual data. The command register says whether they are inputs or outputs.

In the first method, the command register is loaded with 0s and 1s to indicate the action of the corresponding port bits. For example, consider the command register loaded with 65H; the corresponding port acts as follows:

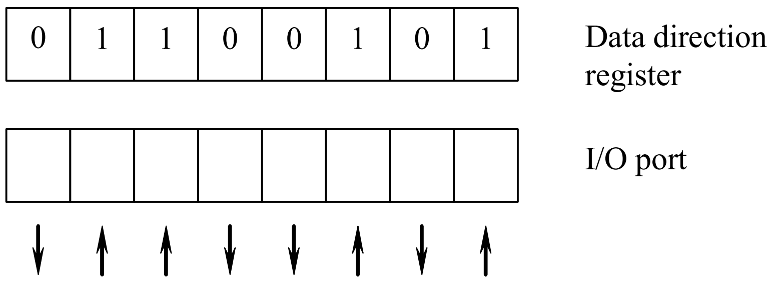


Fig. 2.51 *Data Direction and I/O Port*

In the preceding example, since 65H is sent as output into the data direction register, bits 1, 3, 4, and 8 of the port are set up as outputs and bits 0, 2, 5, and 6 are set as outputs as evident in Figure 2.51.

For parallel I/O, on the other hand, there is only one data-direction register for all ports. A particular bit in command register configures all the bits in a port as either inputs or outputs.

Figure 2.52 provides a clear idea of this type of configuring I/O ports:

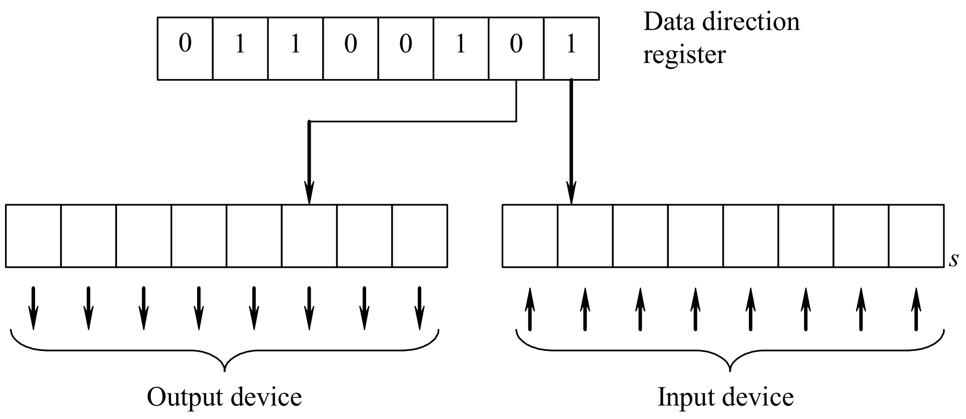


Fig. 2.52 *Data Direction Register and Different Input and Output Device*

Standard I/O vs Memory – Mapped I/O

I/O ports are addressed using either standard I/O or memory mapped I/O.

The standard I/O is also called isolated I/O. It uses $IO/\overline{M}$ control pin on the microprocessor. The processor provides the outputs as **high** on this pin to indicate that an I/O operation is taking place. A **low** on this pin indicates a memory operation.

Address will be of 8 – bits and hence they can address up to 256 different devices. Here, you can use only the IN and OUT instructions

In memory mapped I/O, the processor does not differentiate between I/O and memory.

The processor uses a portion of the memory address to represent I/O ports. The I/O ports are mapped into the processors main memory. Hence, they are called memory mapped I/O. Here, you can use all the instructions that are used to work with memory. The address bus width is the same as that of the microprocessor and hence decoding the logic becomes a bit complicated.

Unconditional and Conditional I/O

In unconditional I/O, the processor sends the data to the device at any time. The external device must always be ready for data transfer. A typical example is when the processor outputs seven segment code through an I/O port to drive a seven segment display that is connected to this I/O port.

In conditional I/O, on the other hand, the processor outputs data to an external device through handshaking. Data transfer occurs by the exchanging of the control signals between the processor and the external device. Data transfer takes place only when the device is ready.

A disadvantage of conditional programmed I/O is that the microcomputer needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop. This type of I/O transfer is dependent on the speed of the external device. For a slow device, this waiting may slow down the capability of the microprocessor to process other data. The interrupt I/O technique is efficient in this type of situation.

Interrupt Driver I/O

Interrupt I/O is a device-initiated I/O transfer. An external device requests the microprocessor to transfer data by activating a signal on the microprocessor's interrupt line during interrupt I/O. The external device is connected to a pin called the Interrupt (INT) pin on the processor chip. When the device needs an I/O transfer with the microcomputer, it activates the interrupt pin of the processor chip. In response, the microprocessor executes a program called the interrupt-service routine to carry out the function desired by the external device. The microcomputer usually completes the current instruction and saves at least the contents of the current program counter on the stack.

The microcomputer then automatically loads an address into the program counter to branch to a subroutine like program called the interrupt service routine. This program is written by the user. The external device wants the microcomputer to execute this program to transfer data. The last instruction of the service routine is a RETURN, which is typically the same instruction used at the end of a subroutine. This instruction normally loads the address (saved in the stack before going to the service routine) in the program counter. Then, the microcomputer continues executing the main program.

Virtual I/O

The virtual I/O processes the high packet rate in which data must often be copied on receive high-bandwidth, high throughput and low latency. In the microcomputer, if system unit is assembled with virtual I/O, the virtual machines use three models for I/O. They are known as emulation, para-virtualized drivers and pass-through access. Emulation process incorporates the real I/O devices, interrupts and DMA, para-virtualized drivers use network device level or higher up the stack and pass-through Without any software intermediaries between the virtual machine and the device, for example legacy adapters and self-virtualizing adapters. The virtual I/O is not aware of host physical memory instead aware of its own guest 'physical' memory for processing the data. In the virtual I/O settings, methods and apparatus are disclosed for coupling external, bi-directional ports to an onboard port of a single chip microprocessor such that the external ports can be addressed in the same manner as onboard ports. Using the disclosed method and apparatus, external ports, which are referred to herein as virtual ports, a single onboard port may be configured that it can be used to address up to twelve virtual ports. The port is used as a multiplexed bus to communicate with the virtual ports. **Read** (RD) and **Write** (WR) signals are used to perform read and write operations on the virtual ports. Basically in the era of Internetworking, the system unit includes virtual memory work with virtual I/O. This term is used to describe a set of storage virtualization and network virtualization features, including virtual Ethernet, shared Ethernet adapter, and storage virtualization.

Memory system uses the Virtual Input/Output (VIO) that are embedded in the circuit that is to be controlled and monitored. The VIO includes a control module that acts as a virtual input module for the user circuit, and can optionally include a status module that acts as a virtual output module for the user circuit. A bi-directional data interface is provided between the user circuit and the VIO that is coupled through input/output pads to an external communication link, and finally to a host computer in which resides software that controls the communication link. Thus, by interfacing with the host computer, a user can control the user circuit via the control modules and monitor output signals from the user circuit via the status modules.

Virtual I/O sets with the functioning of virtual memory. Here, you must know the concept of virtual memory. Virtual memory system came into existence in the very early age of computer programs development. The programs are so complex and big and sometimes not able to fit in the specified memory. The better option is to be chosen to divide the programs into subprograms. These subprograms are known as **overlays**. The overlays are swapped dynamically in the memory processed by OS.

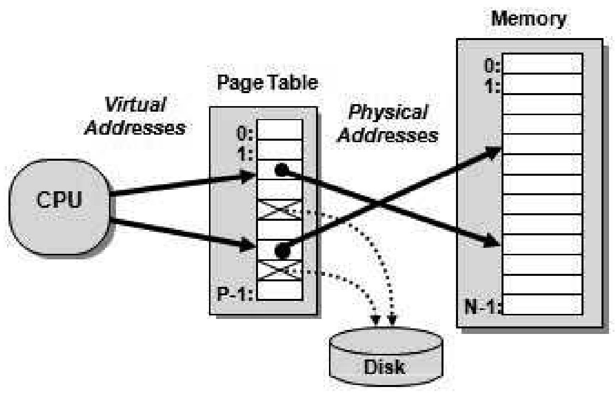


Fig. 2.53 *System With Virtual Memory*

In Figure 2.53, page table communicates virtual addresses with physical addresses. The statement V={0,1, …, N – 1} is defined as virtual memory space and P = {0,1, …, M – 1} is defined as physical address space. Virtual memory is used to manage the shortage of memory. If processes are very complex and big, VMS is needed for efficient protection scheme. It is also used to maintain paging system and sharing the processes. The virtual memory (virtual I/O) uses cache in physical DRAM in the system unit. The VM address space exceeds physical memory size. The VM system simplifies the complete memory management which resides in the main memory. The involved process has own address space. This system provides protection because one process can not intervene with other process because both processes operate in different address spaces. The concept of DMA is closely related to virtual I/O.

Dynamic Memory Access (DMA)

DMA stands for Direct Memory Access, a capability in modern computers that allows peripheral devices to send data to the motherboard's memory without intervention from the CPU. The DMA controllers are special hardware embedded into the chip in modern integrated processors that manage the data transfers and arbitrate access to the system bus. The controllers are programmed with source and destination pointers where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory. Transfers are initiated when the DMA controller is notified of the need to move data to the memory by some event (keyboard press or mouse click, for examples). The controller asserts a DMA request signal to the CPU to use the system bus. The CPU completes its current operation and yields control of the bus to the DMA controller via a DMA acknowledge signal. The controller then reads and writes data and controls signals as if it is the CPU, which at that instant is tri-stated (idled). Upon completion of the transfer, DMA controller de-asserts the DMA request signal and the CPU in turn removes its DMA acknowledge signal and resumes control of the bus. DMA is implemented in computer bus architectures to speed up computer operations and allow multitasking. Normally, the CPU will be fully occupied in any read/write operation; enabling DMA allows reading/writing data in the internal memory, external memory and peripherals without CPU involvement, thus making the processor available for other tasks. This ensures streamlined operations, as movement of data to/from memory is one of the most common computer operations and freeing the CPU. Cache and virtual memory effects can greatly affect program performance with reference to virtual I/O. Adapting program to characteristics of memory system can lead to major speed improvements. DMA assumes that both the destination and source can take transfers as quickly as the controller can make them. The CPU sets up the controller, and after a signal from the I/O port, the entire data is copied to the destination. The DMA controller has sole access to the system bus during the transfer which is very rapid compared to synchronous DMA. For this , DMA uses dynamic memory allocator.

NOTES

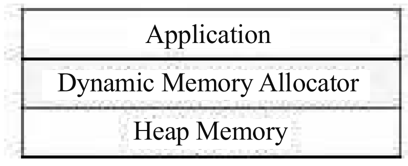


Fig. 2.54 *Dynamic Memory Allocator*

In Figure 2.54, you can see that the dynamic memory allocator is associated with application and heap memory. It is of two types—explicit and implicit memory allocator. The explicit is used for application allocates and frees space, e.g., malloc and free in C, whereas implicit is used for application allocates, but does not free space, e.g. ,garbage collection in Java, ML or Lisp. In both the cases, the memory allocator provides an abstraction of memory as a set of blocks. In the following figure, you see the memory is addressed for each word and each word holds a pointer. The two types of blocks, such as allocated block and free block are involved in this operation.

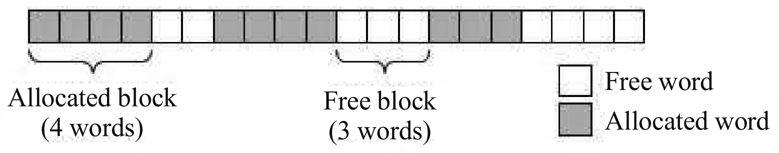


Fig. 2.55 *Allocation Example*

The throughput of DMA is sequenced as follows:

$R_0, R_1, ..., R_k, ... , R_{n-1}$

in which sequence $\boldsymbol{R_0, R_1, ..., R_k, ... , R_{n-1}}$ is maintained by malloc() function in C, where malloc(p) results in a block with a *payload* of p bytes.

Direct memory access is a system that can control the memory system without using the CPU. On a specified stimulus, the module will move data from one memory location or region to another memory location or region. While it is limited in its flexibility, there are many situations where automated memory access is much faster than using the CPU to manage the transfers. Systems like the ADC, DAC and PWM capturing all require frequent and regular movements of memory out of their respective systems. The DMA can be configured to handle moving the collected data out of the peripheral module and into more useful memory locations (like arrays). Only memory can be accessed this way, but most peripheral systems, data registers, and control registers are accessed as if they were memory. The DMA is intended to be used in low power mode because it uses the same memory bus as the CPU and only one or the other can use the memory at the same time. The DMA system is organized into three largely independent parts. Though the three compete for the same memory bus, they can be configured for independent triggers and memory regions.

DMA Operation

There are three independent channels for DMA transfers. Each channel receives its trigger for the transfer through a large multiplexer that chooses from among a large number of signals. When these signals activate, the transfer occurs. The DMAxTSELx bits of the DMA Control Register 0 (DMACTL0). The DMA controller receives the trigger signal but will ignore it under certain conditions. This is necessary to reserve the memory bus for reprogramming and non-maskable interrupts etc. The controller also handles conflicts for simultaneous triggers. The priorities can be adjusted using the DMA Control Register 1 (DMACTL1). When multiple triggers happen simultaneously, they occur in order of module priority. The DMA trigger is then passed to the module whose trigger activated. The DMA channel will copy the data from the starting memory location or block to the destination memory location or block. There are many variations on this, and they are controlled by the DMA Channel x Control Register (DMAxCTL). The mode of DMA are explained as follows::

Single Transfer: Each trigger causes a single transfer. The module will disable itself when DMAxSZ number of transfers have occurred (setting it to zero prevents transfer). The DMAxSA and DMAxDA registers set the addresses to be transferred to and from. The DMAxCTL register also allows these addresses to be incremented or decremented by 1 or 2 bytes with each transfer. This transfer halts the CPU.

Block Transfer: An entire block is transferred on each trigger. The module disables itself when this block transfer is complete. This transfer halts the CPU, and will transfer each memory location one at a time. This mode disables the module when the transfer is complete.

Burst-Block Transfer: This is very similar to Block Transfer mode except that the CPU and the DMA transfer can interleave their operation. This reduces the CPU to 20% while the DMA is going on, but the CPU will not be stopped altogether. The interrupt occurs when the block has completely transferred. This mode disables the module when the transfer is complete.

Repeated Single Transfer: It is the same as Single Transfer mode above except that the module is not disabled when the transfer is complete.

Repeated Block Transfer: It is the same as Block Transfer mode above except that the module is not disabled when the transfer is complete.

Repeated Burst-Block Transfer: It is the same as Burst Block Transfer mode except that the module is not disabled when the transfer is complete.

Check Your Progress

6. Write the different categories of microcomputer memory.
7. Why is memory a passive device?
8. What are the three ways of transferring data between the microprocessor and a physical I/O device?
9. Define the term programmed I/O ports.
10. State about the virtual I/O.

2.6 ANSWERS TO 'CHECK YOUR PROGRESS'

NOTES

1. The program counter or PC stores the address of the next instruction to be executed. Its contents are automatically updated by the ALU.
2. An instruction on our simple computer consists of one 12-bit word. The leading four bits form the operation code (opcode) which specifies the action to be taken, and the remaining 8 bits, when used; indicate the memory address of one of the instruction's operands. For those instructions that have two operands, the other operand is always contained within the accumulator. The instructions stored in memory constitute a program.
3. The Overflow (O) flag is set if the result of an arithmetic and logical operation on signed numbers is too large for the microprocessor's maximum word size.
4. Most microprocessors store information and instructions in a wide range of memory locations. Usually, the memory locations are in a memory chip rather than in the microprocessor. The microprocessor needs a way to tell the memory chip which memory location it wants to put data into or take data from. It does this through the address bus.
5. Once the microprocessor has specified which memory location or device it wants to put data into or take data from, then it needs a set of electrical paths for this information to travel on. This set of paths is electrical and is called the data bus.
6. There are three categories of memories:
   - Processor Memory
   - Primary or Main Memory
   - Secondary Memory
7. Memory is a passive device because it can only store information. No processing can be carried out on information in memory.
8. The three ways of transferring data between the microprocessor and a physical I/O device are:
   - Programmed I/O
   - Interrupt Driven I/O
   - Direct Memory Access (DMA)
9. The microprocessor executes a program to communicate with an external device via a register called I/O port for programmed I/O. The microcomputer communicates with an external device through one or more registers called I/O ports.
10. The virtual I/O processes the high packet rate in which data must often be copied on receive high-bandwidth, high throughput and low latency. In the microcomputer, if system unit is assembled with virtual I/O, the virtual machines use three models for I/O. They are known as emulation, para virtualized drivers and pass-through access.

2.7 SUMMARY

- The microprocessor is a Central Processing Unit (CPU) that is the 'Brain' of the computer.
- The PC stores the address (location) of the next instruction that needs to be executed. Its contents are automatically updated by the ALU.
- An instruction on our simple computer consists of one 12-bit word. The leading four bits form the operation code (opcode) which specifies the action to be taken, and the remaining 8 bits, when used; indicate the memory address of one of the instruction's operands.
- The stack is a special place in memory, which is often used to store critical pieces of data during subroutines and interrupts. In other words, a stack is a part of the memory, which temporarily stores data.
- ALU (Arithmetic and Logical Unit) performs arithmetic and logic operations on the data. The size of ALU defines the size of the microprocessor.
- A bus is a subsystem that transfers data between computer components inside a computer or between computers.
- The microprocessor needs a way to tell the memory chip which memory location it wants to put data into or take data from. It does this through the address bus.
- Once the microprocessor has specified which memory location or device it wants to put data into or take data from, then it needs a set of electrical paths for this information to travel on. This set of paths is electrical and is called the data bus.
- The control bus consists of wires, some of which transfer signals from the
- CPU to external devices and others that carry signals from external devices to the CPU.
- Memory is required to store data and instructions. Data and instructions which can be lost after the power supply is stopped are stored in the Random-Access Memory (RAM).
- Processor memory consists of a set of CPU registers. These registers are useful to hold temporary results when a computation is in progress.
- Primary memory is the storage area in which all the programs are executed. The size of the primary memory is much larger as compared to processor memory but its operating speed is slower than processor registers by a factor of 25.
- Secondary memory refers to the storage medium for huge files, such as program source codes, compilers, operating systems, RDBMS, etc.
- To execute programs, a microprocessor retrieves instructions from memory and executes them, again fetching data from the memory, if required.
- Memory is a passive device. It can only store information. No processing can be performed on the information stored in the memory. The information,

stored in memory, must be transferred to a CPU register for processing and the result written back to memory.

- An individual communicates with the microcomputer system by using the I/O devices interface. The user can enter the program and data using the keyboard on the terminal (input) and execute the program to obtain results (output).
- The microprocessor executes a program to communicate with an external device via a register called I/O port for programmed I/O. The microcomputer communicates with an external device through one or more registers called I/O ports.
- In unconditional I/O, the processor sends the data to the device at any time. In conditional I/O, on the other hand, the processor outputs data to an external device through handshaking. Data transfer occurs by the exchanging of the control signals between the processor and the external device. Data transfer takes place only when the device is ready.
- A disadvantage of conditional programmed I/O is that the microcomputer needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop. This type of I/O transfer is dependent on the speed of the external device.
- Interrupt I/O is a device-initiated I/O transfer. An external device requests the microprocessor to transfer data by activating a signal on the microprocessor's interrupt line during interrupt I/O. The external device is connected to a pin called the interrupt (INT) pin on the processor chip.
- The virtual I/O processes the high packet rate in which data must often be copied on receive high-bandwidth, high throughput and low latency. In the microcomputer, if system unit is assembled with virtual I/O, the virtual machines use three models for I/O. They are known as emulation, para virtualized drivers and pass-through access.
- DMA stands for Direct Memory Access, a capability in modern computers that allows peripheral devices to send data to the motherboard's memory without intervention from the CPU.
- The DMA controllers are special hardware embedded into the chip in modern integrated processors that manage the data transfers and arbitrate access to the system bus.

2.8 KEY TERMS

- **Microprocessor:** An integrated circuit (a semiconductor chip) that is the brain of the computer and performs almost all the operations of the computer.
- **Program counter:** The PC stores the address (location) of the next instruction that needs to be executed.
- **Instruction register:** An instruction on our simple computer consists of one 12-bit word.

- **Address bus:** The CPU has to be able to send various data values, instructions and information to all the devices and components inside your computer as well as the different peripherals and devices attached.
- **Computer memory:** A critical part of the computer that is actually either an internal or external device that stores information and programs for applications and future use.
- **Primary memory:** Primary memory is the storage area in which all the programs are executed.
- **Physical I/O:** When the microcomputer has no operating system, then the user works directly with physical I/O devices and performs detailed I/O design.

2.9 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What do you understand by the general purpose registers? Why are they important?
2. Why are dedicated registers so named?
3. Define the term accumulator.
4. What is the meaning of utility of barrel shifter?
5. Define the term control bus.
6. What is the data bus bi-directional?
7. Differentiate between static RAM and dynamic RAM.
8. What is the importance of interrupt-driven I/O?
9. Differentiate between linear decoding and fully decoding.

Long-Answer Questions

1. Discuss the classification of microprocessors on the basis of register section. Give suitable examples for each one of the categories.
2. Briefly explain about the ‘Stack’, How does it work? Discuss the PUSH and POP operations giving suitable examples.
3. Describe the different types of buses available with the microprocessor and what is the utility of each.
4. How is the memory accessed by the microprocessor? Discuss the steps.
5. Briefly explain the primary memory and its classification from the usage point of view.
6. How are I/O devices accessed by the microprocessor? Discuss the different techniques employed in I/O by the microprocessor.

2.10 FURTHER READING

NOTES

Goankar, Ramesh. 2002. *Microprocessor Architecture, Programming and Application with the 8085*. Mumbai: Penram International Publishing.

Ram, B. 2005. *Fundamentals of Microprocessor and Microcomputers*. New Delhi: Dhanpat Rai Publications.

Lotia, Manahar and Pradeep Nair. 2003. *Modern All About Motherboard*. New Delhi: BPB Publications.

Mueller, Scott, and Craig Zacker. 2002. *Upgrading and Repairing PCs*. New Jersey: Pearson Education (Que Publishing).

Godse, D.A. and A.P. Godse. 2007. *Microprocessor and Assembly Language*

Programming. Pune: Technical Publications.

Kleitz, William. 2009. *Digital and Microprocessor Fundamentals: Theory and*

Applications. New Jersey: Prentice Hall.

Ray, A.K. and K.M. Bhurchandi. 2000. *Advanced Microprocessors and Peripherals*. New Delhi: Tata McGraw-Hill.

UNIT 3 PROGRAMMING MODE

NOTES

Structure

3.0 INTRODUCTION

A computer is a device that can function only under a set of certain software instructions. Without software, microprocessors cannot be used for any practical purpose. Simultaneously, microprocessors cannot work in isolation without communicating with the outside world. The memory as well as the I/O devices are an integral part of a microprocessor-based system. These 1-byte instructions perform three different tasks. In the first instruction, both the operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bits binary format in memory.

The cache is a small amount of high-speed memory used to work directly with the microprocessor. The memory cycle time of cache is usually in the order of the time required by the CPU to fetch an instruction. The cache usually gets its data from the main memory whenever the instructions or data is required by the CPU. The main memory can contain wider data, whereas the CPU requirement may be of the data with less width. The amount of information which can be placed at one time in the cache memory is called the line size for that particular cache. Cache controllers are being commonly used in microcomputer systems to effectively reduce the load on the microprocessor.

In this unit, you will study about the register organization of 8086, memory addressing, instruction formats, memory interfacing, cache memory and cache controllers.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic of register organization of 8086

- Explain the memory addressing
- Discuss about the instruction formats
- Analyse the cache memory and cache controllers
- Define the memory interfacing

NOTES

3.2 REGISTER ORGANIZATION OF 8086

A computer is a device that can function only under a set of certain software instructions. Without software, microprocessors cannot be used for any practical purpose. Simultaneously, microprocessors cannot work in isolation without communicating with the outside world. The memory as well as the I/O devices are an integral part of a microprocessor-based system. Thus, for the programmer, it becomes necessary to conceptualize the following parts of the system:

- The architecture of the microprocessor with complete knowledge of the control unit as well as the register set.
- Memory addressing capability of the microprocessor and modes of operation.
- I/O compatibility of the microprocessor and modes of operation.

A programmer's model of the microprocessor-based system is shown in Figure 3.1.

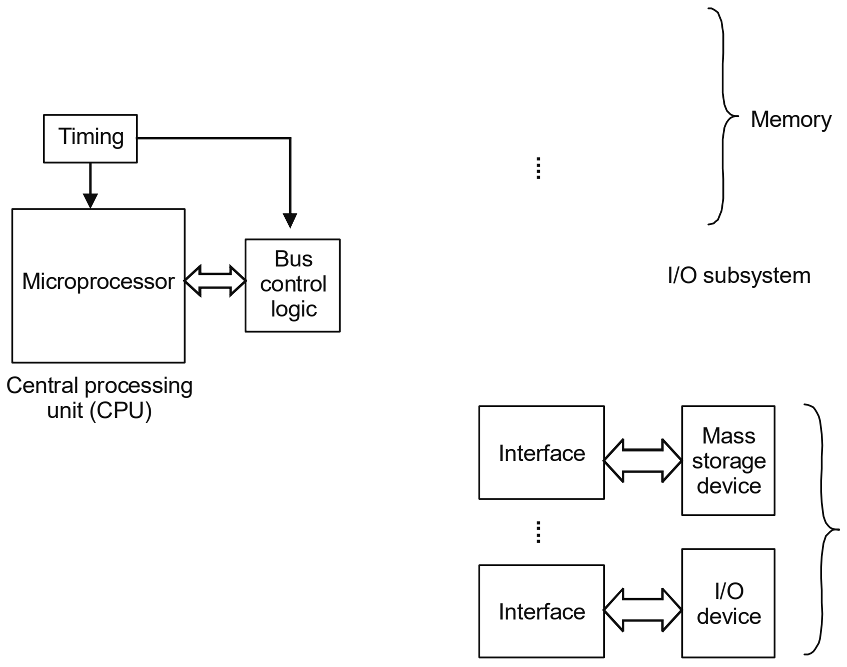


Fig. 3.1 *Model of a Microprocessor-Based System*

Figure. 3.2 displays the same specifically for the microprocessor 8086.

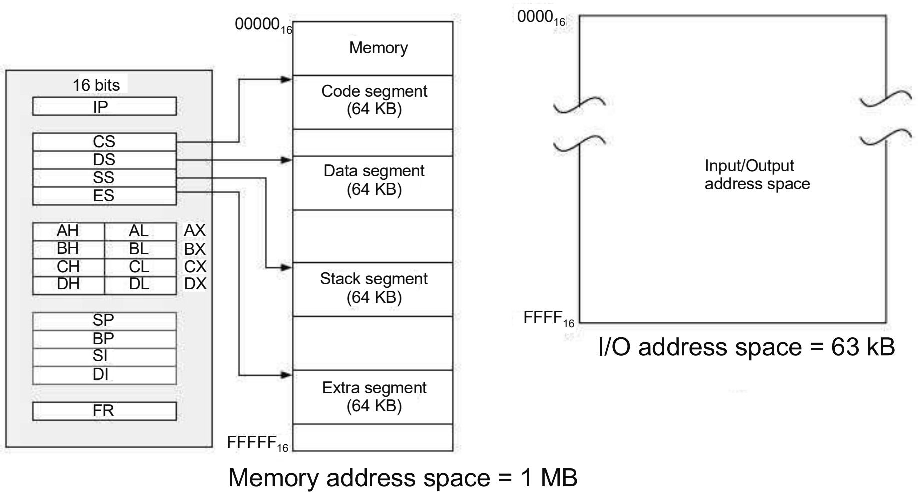


Fig. 3.2 Microprocessor 8086-Based Computer Model

NOTES

3.2.1 The Architecture of 8086

The basic architecture of microprocessor 8086 is divided into two parts (Refer Figure 3.3):

(1) The Bus Interface Unit

(2) The Execution Unit

Generally, the Bus Interface Unit (BIU) is responsible for the communication by the microprocessor. This unit basically controls the buses and handles all the data code, address etc. from the memory or other devices. The Execution Unit (EU), on the other hand, is responsible for the actual execution of the program code. It performs all the operations, generates the status signals and passes the results to the bus interface unit for further processing through the I/O devices or the memory.

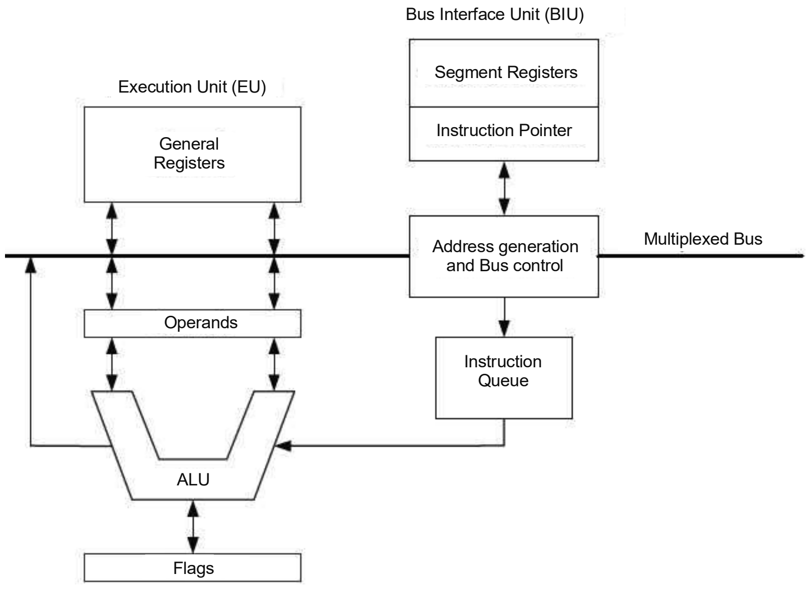
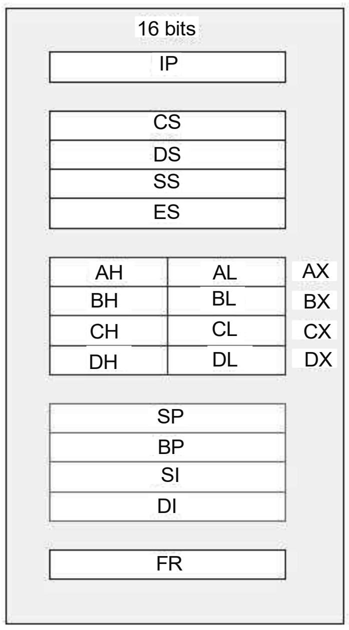


Fig. 3.3 Basic Architecture Showing BIU and EU

The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer. Eight of the registers are known as general-purpose registers, i.e., they can be used by the programmer for data manipulation. Each register is 16-bits long i.e. it can contain a 16-bit binary number. The first four registers are sometimes referred to as data registers. They are the AX, BX, CX and DX registers. The next four are referred to as index registers or pointer registers. They are the SP, BP, SI and DI registers. The data registers can be treated as 16-bit registers or they can each be treated as two 8-bit registers. All of these registers are shown in Figure 3.4.



Instruction	IP : Instruction Pointer
Segment	CS : Code Segment
	DS : Data Segment
	SS : Stack Segment
	ES : Extra Segment
General	AX (AH:AL): Accumulator
	BX (BH:BL): Base
	CX (CH:CL): Count
	DX (DH:DL: Data
Pointer	SP: Stack
	BP: Base
Inbox	SI: Source
	DI: Destination
Status	FR: Flag

Fig. 3.4 *Registers*

Each 8-bit register can be used independently. The AX register may be accessed as AH and AL (H and L refer to high-order and low-order bytes).

Similarly, BX can be used as BH, BL; CX can be used as CH, CL and DX can be used as DH, DL.

If you use a data register as an 8-bit register, you cannot use its 16-bit parent at the same time. The four index registers can be used for arithmetic operations but their use is usually concerned with the memory addressing modes of the 8086 microprocessor, which will be discussed later in this unit. The two remaining registers are the Instruction Pointer (IP) and the status word, or flags register. Neither of these is referenced directly by the program. The registers can be divided into the following four categories:

1. General-Purpose Registers
2. Pointer or Index Registers
3. Segment Registers
4. Other Registers

NOTES

1. General-Purpose Registers

The **accumulator** is 16-bits wide and is called AX. The upper 8 bits are called AH (accumulator high) and the lower 8 bits are called AL (accumulator low). The 8086 accumulator is shown in Figure 3.5.

The 8086/8088 has three 16-bit or six 8-bit general-purpose registers (besides the accumulator). These are shown in Figure 3.6. 15–34-bits registers are called BX, CX and DX registers. Each can be divided into an upper and lower byte called BH, BL, CH, CL, DH and DL, respectively.

Observe that in Figure 3.6, A stands for accumulator, B for base, C for count and D for data. This can help you remember the main functions of each register.

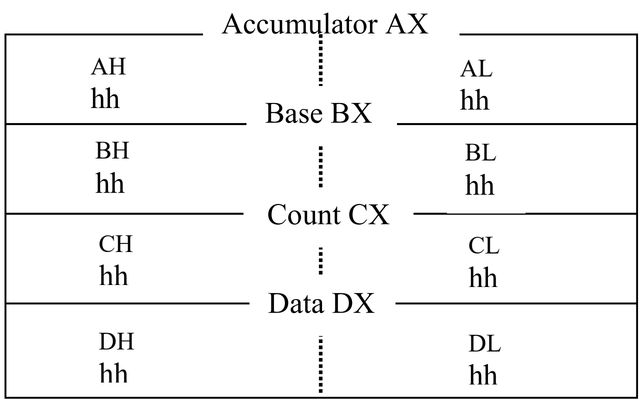


Fig. 3.5 *8086 Accumulator*

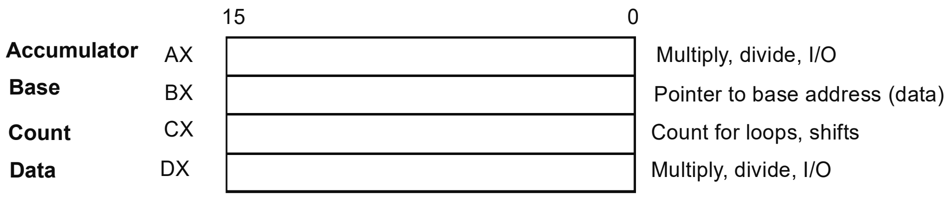


Fig. 3.6 *The A, B, C, D Registers*

Although these four registers are **general-purpose registers** and are used for any instructions in general, however each of them has a specific purpose also corresponding to some specific instructions. The register AX works as the accumulator, while the register BX is used as a pointer to the base address in case of based addressing. Similarly, the register CX is used as a default counter in case of the loop and shift, rotate instructions. The register DX has a greater significance as it is used to store the reminder or dividend in a default way during the division process. These four 16-bit registers can also be accessed as 8-bit registers as shown in Figure 3.7.

NOTES

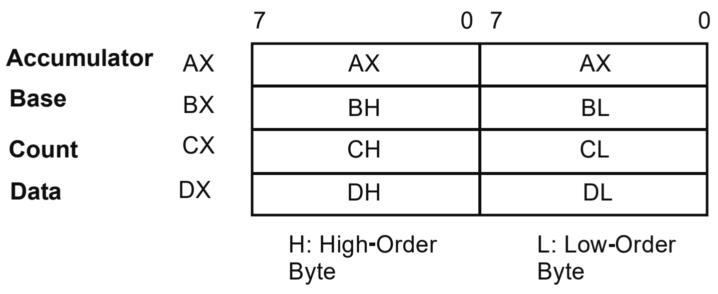


Fig. 3.7 *16-bit Registers*

2. Pointer and Index Registers

The 8086 microprocessor has several index registers and pointers including the base pointer, source index and destination index. All are 16 bits wide. Unlike 8-bit chips, these are not used alone, but are used in combination with registers called **segment registers**. Figure 3.8 shows a model consisting of pointers and index registers.

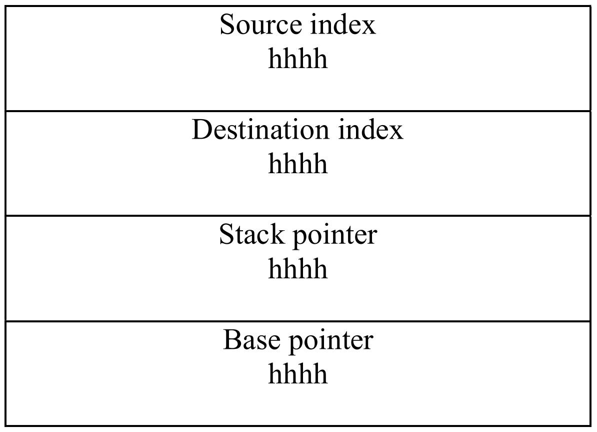


Fig. 3.8 *Pointers and Index Registers*

The 8086 stack is a standard memory stack. The 8086 however, is a large stack, which contains up to 64K (65,536 bytes). The location of the top-of-the-stack is calculated by using both the Stack Pointer (SP) and the stack segment. The register SP always points to the top of the stack and basically the offset is provided by SP. The base address is provided by the stack Segment Register (SS).

The base pointer, on the other hand, is a pointer to the base address and is used in the based addressing mode. Both the Source Index (SI) and the Destination Index (DI) are registers that useful for providing the offset during string operations. The SI is used for the source strings and the DI is used for the destination. The same is illustrated in the Figure 3.9.

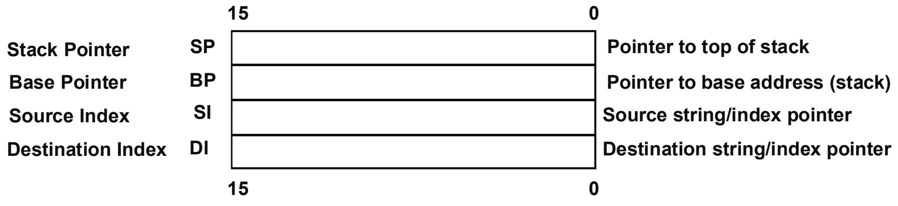


Fig. 3.9 *SP, Base Pointer, SI and DI*

3. Segment Registers

The 8086 microprocessor has several other registers, which do not exist on 8-bit chips. These are called as the segment registers. Actually, in case of 8086, the concept of segmentation is one of the biggest innovations that Intel did. This concept was first introduced in 8086 and is still retained by all modern-day microprocessors. The 8086 contains the four segment registers shown in Figure 3.10.

Code Segment
CS
Data Segment
DS
Stack Segment
SS
Extra Segment
ES

Fig. 3.10 *Segment Registers*

All the pointers and index registers in the 8086 microprocessor are 16 bits wide. Therefore, they will be able to address 2^{16} as 65,536 (64K) bytes. The address bus, however, is 20 bits wide. It is possible to have memory locations extending up to 2^{20} or 1,048,576 (1 mega-) bytes. However, pointers would be unable to point this wide a range of addresses, so segment registers are used. Their contents are combined with the contents of the various pointers and index registers to form an address which is 20 bits wide. This will be illustrated while calculating the memory address. Each one of these segment registers has a specific purpose and utility. The following is a brief description of this:

1. Code segment – CS:IP

   Code segment is the most important segment and it contains the actual assembly language instructions to be executed by the microprocessor. During execution of the program, the program code is actually retrieved from the code segment instruction.

2. Data segment – DS:BX, SI, DI

   The data segment stores all the information or data to be processed. During the execution of any instruction, the data to be processed is based on the data obtained from the data segment.

3. Stack segment – SS:SP, BP

   The stack segment stores all the data related to stack. The stack is used to temporarily store information like the return addresses etc or the intermediary results.

4. Extra segment – ES:BX, SI, DI

   The extra segment is actually an extended data segment, which is particularly used for the string operations. In addition, in case the data segment gets full, then automatically the extra segment works as the data segment.

NOTES

Boundaries of a Segment

Each segment has a particular boundary and all the activities are performed inside this boundary only. The offset of a particular segment varies from 0000H to FFFFH. Hence, the physical boundaries are calculated by adding the 0000 and FFFFH to the base address. The same has been illustrated in Figure 3.11.

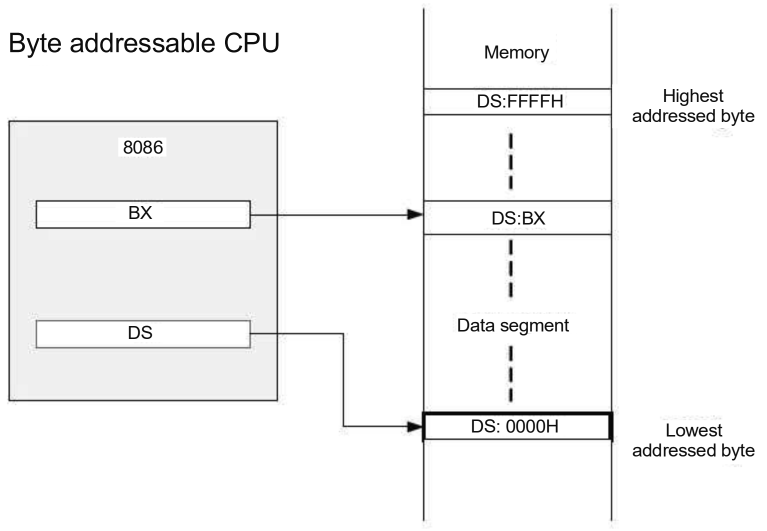


Fig. 3.11 *Physical Boundaries of a Segment*

Stack Segment of Memory

The stack segment is an important segment as all the data related to the stack is held by this segment only. Certainly, it has great importance in modular programming and the programs in which a lot of jumping and conditions are involved. To calculate the physical address for an item in the stack, the base address is that of the stack segment, while the offset is taken from the stack pointer register.

Stack segment is illustrated in Figure 3.12.

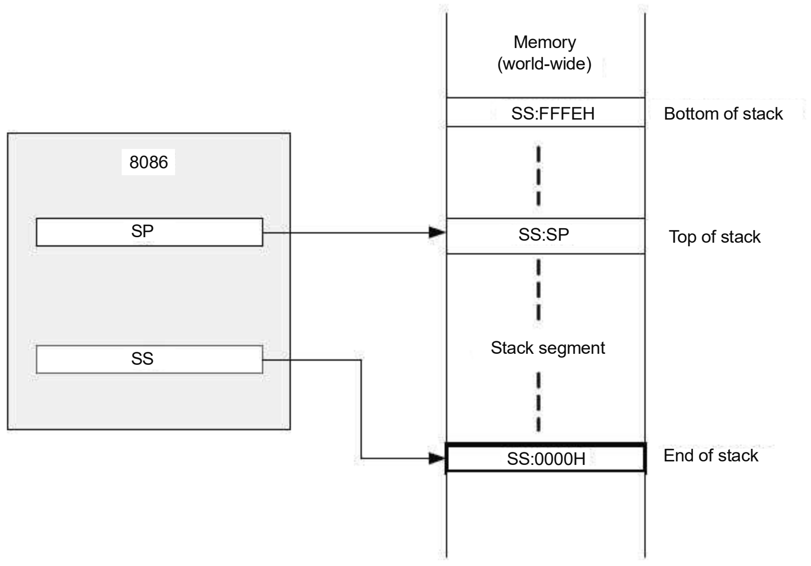


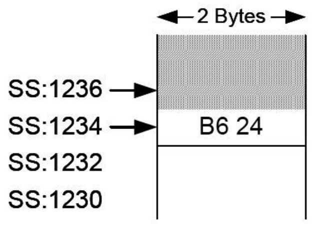
Fig. 3.12 *Stack Segment*

The PUSH and POP operation using this segment can be illustrated as follows:

Logical address SS:SP
SP = 1236
AX = 24B6

Pushing onto stack :
PUSH AX
SP = 1234
AX = 24B6

Popping the stack :
POP CX
SP = 1236
CX = 24B6



The segmentation concept actually helps in simplifying the programming process, as the programmer always knows what kind of data is available at what place in the memory. This basically is referred to as ease of programming as illustrated in Figure 3.13.

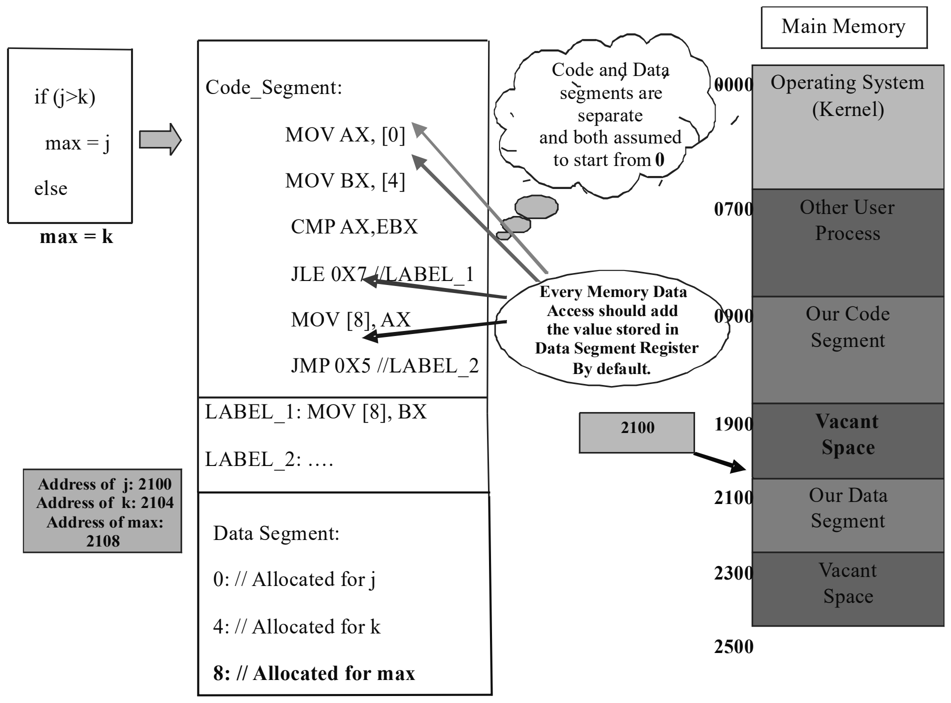


Fig. 3.13 Segmentation and Ease of Programming

4. Other Registers

The status register (flags) and the instruction pointer are categorized as the other registers (Refer Figure 3.14). However, both these registers are essential and play a critical role in the execution of programs.

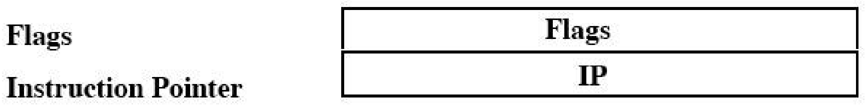


Fig. 3.14 Status Registers (Flags)

NOTES

(a) Instruction Pointer Register

Instead of using a program counter, 8086/8088 microprocessor uses an instruction pointer which performs the same activity as the program counter. This is a crucially important register which is used to control which instruction the CPU executes. The instruction pointer is 16 bits wide. The IP, or Program Counter *(PC)*, is used to store the memory location of the next instruction to be executed. The microprocessor checks the program counter to ascertain which instruction to carry out next. It then updates the program counter to point to the next instruction. Thus, the program counter always points to the next instruction that needs to be executed.

(b) Status (Flags) Register

The status register is the register containing the 8086/8088 flags. This register is 16 bits wide, although not all the 16 bits are used. This register, shown in Figure 3.15, has a lower byte (8 bits) which is exactly the same as the 8-bit 8085 microprocessor's status register. In fact, it has the same flags in the same positions. The upper byte has four flags which the 8085 does not have and are new to 8086.

New	8085-like
- - - - O D I T	S Z – A – P – C
- - - - b b b b	b b – b – b – b
h ⋮ h	h ⋮ h

Fig. 3.15 *The 16-Bit Status Register*

The first flag is the **Trap flag** (T), which controls a single-step mode of operation. The **Interrupt-Enable flag** (I) controls the interrupt request pin on the microprocessor chip. The **Direction flag** (D) controls whether the source index and destination index increment or decrement during string operations. Finally, the **Overflow flag** (O) alerts the programmer to the existence of an arithmetic overflow when set. This is a condition in which the legal range for the signed binary numbers of a particular word size have been exceeded.

Out of 16 bits, nine individual bits of the status register are used as control flags (3 of them) and status flags (6 of them).The remaining 7 are not used. A flag can only take the values 0 and 1. A flag is considered to be set if it has the value 1. The status flags are used to record specific characteristics of arithmetic and logical instructions. For example, the zero flag (Z-flag) is set to 1 if the result of an arithmetic operation by the ALU is zero. The control flags, on the other hand, are used to control the specific modes of the CPU (Refer Figure 3.16).

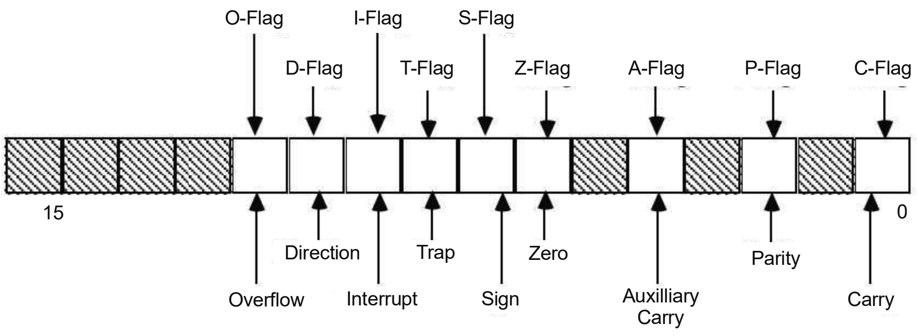


Fig. 3.16 Various Flags of the Status Register

The flags are categorized as the conditional flags and the control flags. The conditional flags are set or reset as per the conditions of the ALU (i.e. after the execution of the program), whereas the control Flags (D, I, T) are set even before the execution to actually control the program flow. These two categories of the flags are shown in Figure 3.17.

C = Carry
P = Even parity
A = Auxiliary carry
Z = Zero
S = Sign
O = Overflow
	Condition Flags (after execution)

D = Direction
I = Interrupt
T = Trap
	Control Flags (before execution)

Fig. 3.17 Condition Flags and Control Flags

3.3 MEMORY ADDRESSING

Each memory address in the 8086 addressing is represented by two different kinds of addresses:

(1) Logical Address

(2) Physical Address

Each segment in 8086 is of 64 KB, hence the logical address and is represented by Segment : Offset, while, the physical address is represented by Segment 0 : Offset. The physical address is basically calculated by adding the segment address and the offset depicted in Figure 3.18.

NOTES

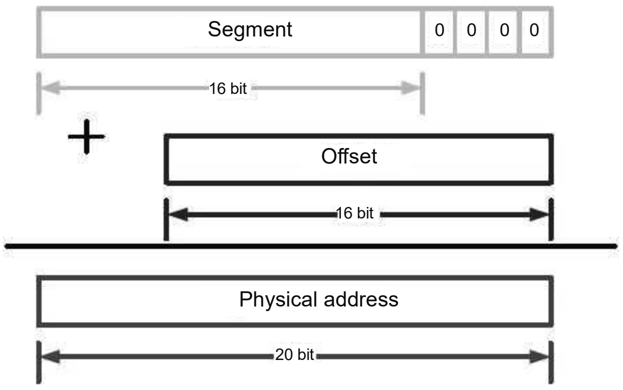


Fig. 3.18 Calculating Physical Address by Adding Segment Address

The physical address of a memory is a 20-bit address for the 8086 microprocessor. However, it is also set in two components of 16 bit each. One is referred as the segment address and another as the offset address. The actual address is calculated by adding these two addresses in a simple manner as illustrated in Figure 3.19.

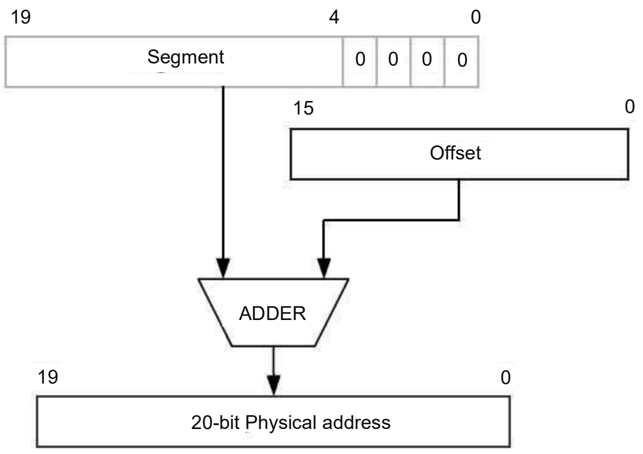
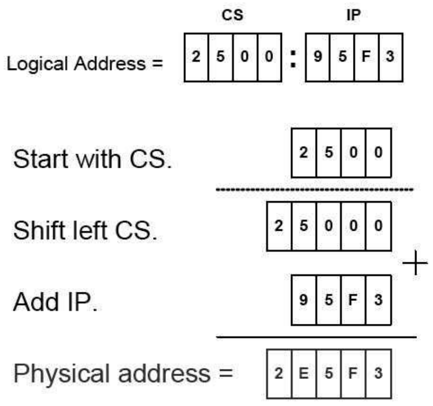


Fig. 3.19 Finding an Actual Address by Using the Segment and Offset Address

Let us consider the example of calculating the physical address, when the address is given as 2500: 95F3. The steps for calculating the physical address are provided sequentially as follows:



Therefore, if DS = 7FA2H and the offset is 438DH, then:

- The logical address is: 7FA2:438D
- The physical address is: 7FA20 + 438D = 83DAD
- The upper range of the data segment is: 7FA20 + FFFF = 8FA1F
- The lower range of the data segment is: 7FA20 + 0000 = 7FA20

NOTES

Complete Model of the 8086 Microprocessor

Figure 3.20 shows a complete model of the 8086 microprocessor. Instead of mentioning all the placeholders for the binary digits, the hexadecimal digits have been indicated as these would be seen on a computer or trainer. To provide clarity in case of the status register, both binary and hexadecimal placeholders are displayed. The small d's and b's represent the data that would be in each register or memory location. Each 'h' stands for one hexadecimal digit or nibble, which is 4 bits. Each 'b' stands for 1 bit.

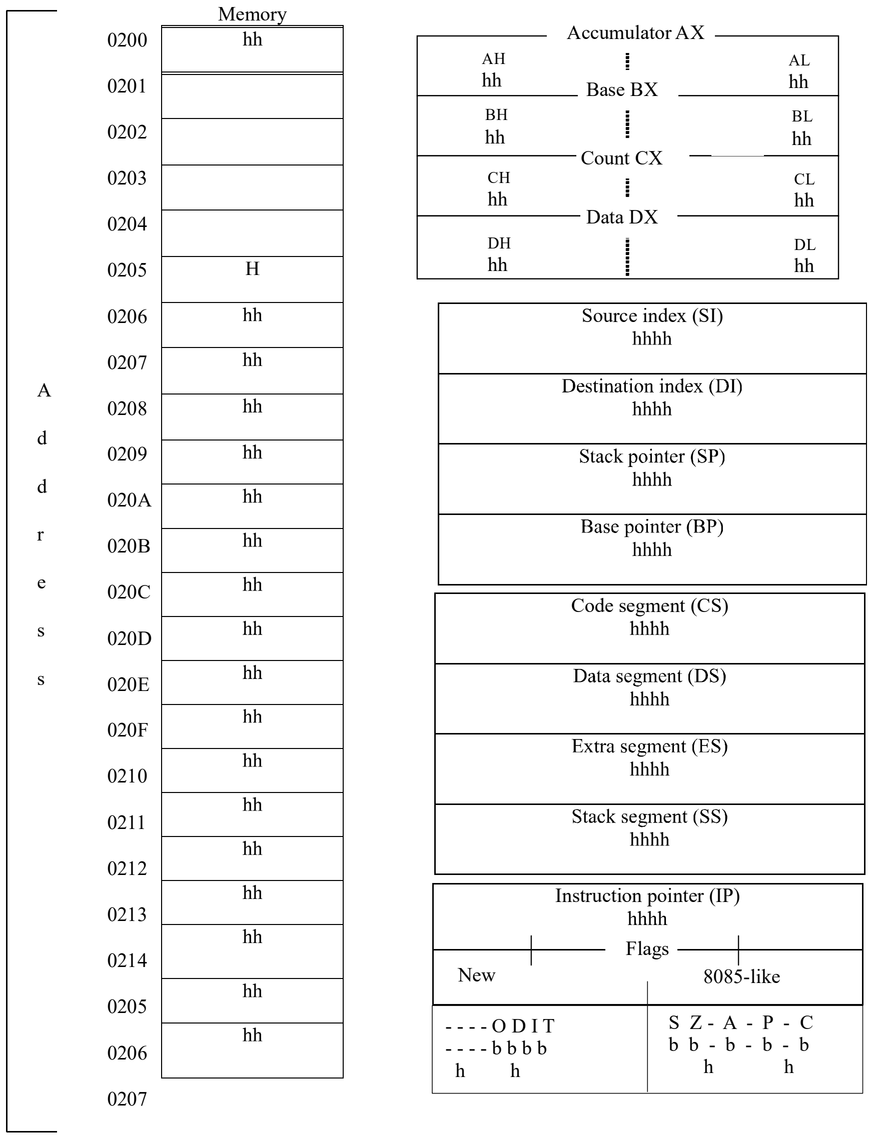


Fig. 3.20 8086 Microprocessor Complete Model

The Endian Conventions

All Intel microprocessors and scores of minicomputers use the little endian convention. According to this convention, a higher byte of the address is sent to the higher address, whereas the lower byte of the address is sent to the lower address. This convention is followed for all the instructions whenever dealing with the memory. Example:

MOV AX, 35F3H;	AL = F3 , AH = 35
MOV [1500], AX;	DS: 1500 ← F3
	DS: 1501 ← 35

On the other hand, Motorola microprocessors and mainframes use big endian conventions, which are just the opposite of the Intel microprocessors. In case of these conventions, the high byte goes to the lower address, and the lower byte to the higher address.

Pin Configuration of 8086

The pin configuration of 8086 is available in the 40-pin Dual Inline Package (DIP). The pin configuration is shown in Figure 3.21 and the important pin signals have been discussed:

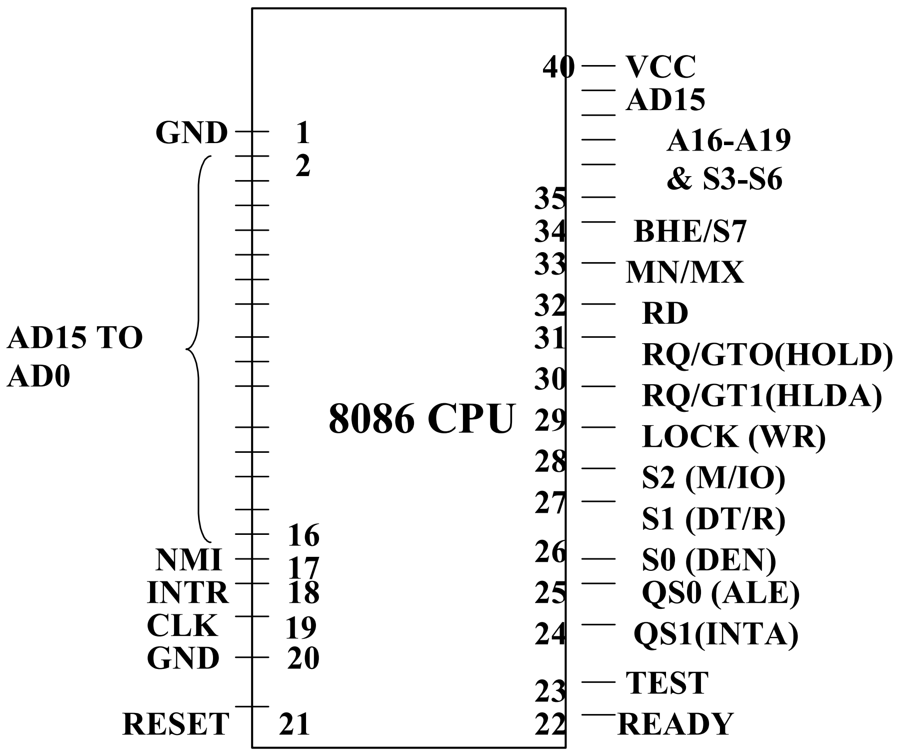


Fig. 3.21 *Pin Configuration of 8086*

AD15 to AD0

Also known as address data bus, these lines constitute the time multiplexed memory/IO address and data bus. They can be used both as the address and data bus depending upon the ALE signal.

ALE

Also known as address latch enable, A high on this line causes the lower-order 16-bit address bus to be latched that stores the addresses and then the lower-order 16-bit of the address bus can be used as data bus.

READY

This signal is very useful for the memory and I/O read/ write operation. READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer.

INTR

INTR is the INTERRUPT REQUEST signal. This is a level-triggered input which is sampled during the last clock cycle of each instruction to determine if the microprocessor should enter into an interrupt-acknowledge operation. This signal can be internally masked by software resetting the interrupt enable bit.

INTA

Also known as Interrupt Acknowledge signal, this signal goes from the microprocessor to the device, which has sent the interrupt signal.

NMI

NMI means the non-maskable interrupt. This is another edge-triggered input, which causes an interrupt request to the microprocessor. NMI cannot be masked internally by the software.

RESET

The RESET signal causes the microprocessor to immediately terminate its present activity. The signal must be active high for at least four clock cycles. It restarts the execution by the microprocessor.

M/IO

This particular signal is used to differentiate between the memory and I/O operation. Alow on this pin indicates the I/O operation and a high indicates a memory operation.

MN/MX

This stands for the Minimum/ Maximum mode. It indicates what mode the processor is to operate in. In case of the minimum mode, the 8086 microprocessor works in a single processor environment, thus all control signals for memory and I/O are generated by the microprocessor itself. On the other hand, maximum mode is designed to be used when a coprocessor exists in the system. In this case the 8086 works in a multiprocessor environment. Control signals for memory and I/O are not generated by the microprocessor, instead an external BUS controller is used for this purpose.

HOLD

The 8086 has a pin called hole. This pin is used by external devices to gain control of the microprocessor buses. This signal is normally activated by the DMA controller to take control of the buses in the DMA mode of data transfer.

HLDA

HLDA is a kind of acknowledgement to the HOLD signal. Whenever the HOLD signal is activated by an external device, the 8086 stops executing instructions as well as using the buses. This would allow external devices to control buses. The HLDA signal is used to tell the DMA that the microprocessor has released the buses for external use.

S0, S1 and S2

These are three status signals, which are of particular use when the microprocessor is working in the maximum mode**.**

Minimum and Maximum Mode of 8086

The microprocessor 8086 can be configured to work in two modes: minimum mode and maximum mode. The minimum mode is used for single-processor systems, where 8086 directly generates all the necessary control signals. The maximum mode is designed for multiprocessor systems, where an additional 'Bus-Controller' IC is required to generate the control signals. The processors control the bus controller using statuscodes. The minimum and maximum mode implementation of 8086 / 8088 is shown in the following Figure (Refer Figure 3.22 (a) and (b)):

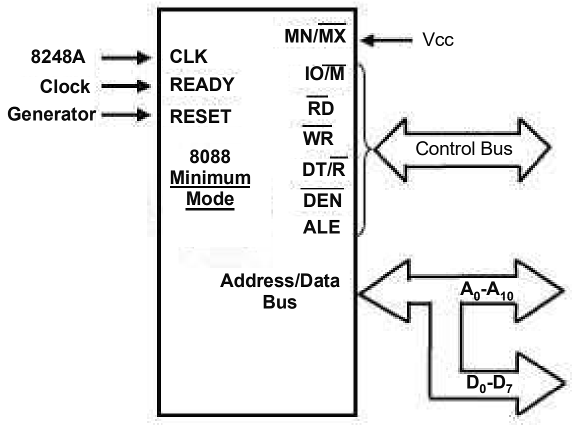


Fig. 3.22 (a) *Minimum Mode Implementation of 8088*

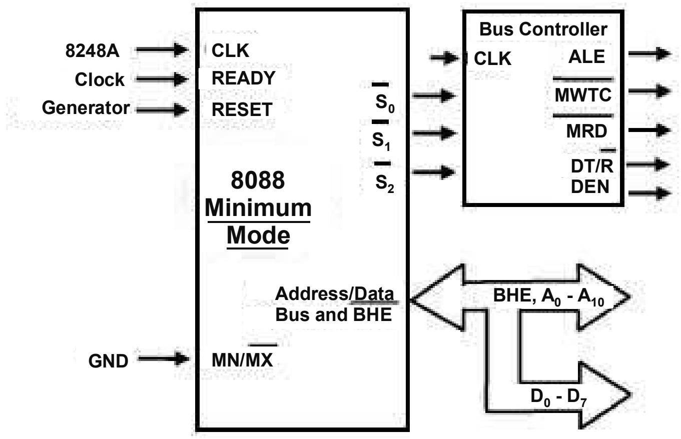


Fig. 3.22 (b) *Maximum Mode Implementation of 8088*

Addressing Modes of 8086

Whenever the microprocessor 8086 executes an instruction, it performs a very specific function on data. These data items, often referred to as operands, may be part of the instruction, may be residing in one of the internal registers of the microprocessor or may be stored at a specific address in memory. To access these different types of operands, the microprocessor is provided with various addressing modes. An addressing mode is a method of specifying an operand. The addressing modes are categorized into three types: register addressing, immediate addressing and memory-operand addressing.

Register-Addressing Mode

Both operands are registers in the register-addressing mode. With the register-addressing mode, operand to be accessed is specified as residing in an internal register of 8086. Any general-purpose register can be used as a source or destination operand. For example,

MOV AX, BX

This stands for copying the contents of register BX into register AX.

Operand is always found in the specified register as follows:

```
MOV AX, DX
MOVAH, BL
MOV SP, BP
MOV DS, AX
MOV DI, SI
MOV ES, DS:  Not Allowed
MOVCS, AX: Not Allowed
```

The important points to be noted are:

- The operands can be located in any of the general-purpose registers, base and index registers, segment registers, FLAGS register depending upon the particular instruction.
- Some instructions, such as IMUL and IDIV, implicitly use operands contained in a pair of registers, e.g., in AX and DX.
- In most cases, destination and source operands must be of the same size.
- Segment register to segment register moves are not allowed.
- Nothing can be copied to code-segment register CS.

Immediate-Addressing Mode

In case of the immediate-addressing mode, the operand is in the instruction itself. Hence, the effective address is also within the instruction.

MOV AX, 2631

The content 2631 is copied to the register AX in this case. The operand can be 8-bits (Imm8) or 16-bits (Imm16) in length, can be included as part of the instruction. Since the data is kept directly into the instruction, immediate operands normally represent constant data. This addressing mode can only be used to specify a source operand. Operand is an immediate (constant) value (Refer Figure 3.23).

NOTES

MOV AL, 22

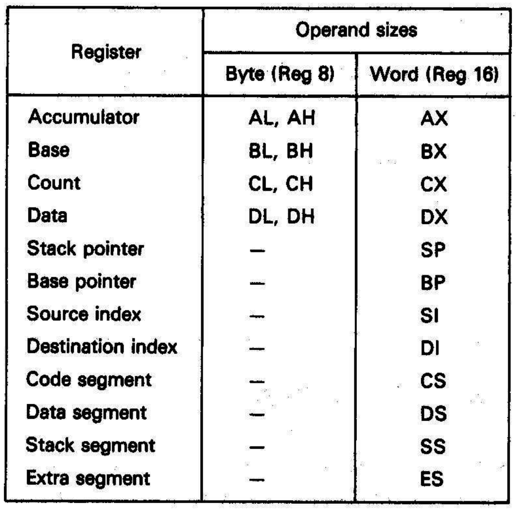
MOVAX, 1645

MOV BX, 1011

MOV BX, Age (Where Age is a variable)

MOV AL, 'A'

MOVAX, 'MP' (A and MP are the string)



Register	Operand sizes	
	Byte (Reg 8)	Word (Reg 16)
Accumulator	AL, AH	AX
Base	BL, BH	BX
Count	CL, CH	CX
Data	DL, DH	DX
Stack pointer	–	SP
Base pointer	–	BP
Source index	–	SI
Destination index	–	DI
Code segment	–	CS
Data segment	–	DS
Stack segment	–	SS
Extra segment	–	ES

Fig. 3.23 *Operand Sizes for Different Registers*

Memory-Operand Addressing Modes

To refer to an operand in memory, the 8088 must calculate the Physical Address (PA) of the operand and then initiate a read or write operation to this storage location. As we discussed earlier, the physical address is formed from a Segment Base Address (SBA) and an Effective Address (EA). The base address identifies the starting location of the segment in memory and EA represents the offset of the operand from the beginning of this segment of memory.

The value of the EA can be specified in a variety of ways. One way is to encode the effective address of the operand directly in the instruction. This represents the simplest type of memory addressing, known as the direct-addressing mode. Figure 3.24 shows that an effective address can be made up from as many as three elements: the base, index, and displacement. Using these elements, the effective address calculation is made by the general formula:

EA = Base + Index + Displacement

PA = Segment base : Base + Index + Displacement

PA = Segment base : Base + Index + Displacement

$$\text{PA} = \left\{ \begin{matrix} \text{CS} \\ \text{SS} \\ \text{DS} \\ \text{ES} \end{matrix} \right\} : \left\{ \begin{matrix} \text{BX} \\ \text{BP} \end{matrix} \right\} + \left\{ \begin{matrix} \text{SI} \\ \text{DI} \end{matrix} \right\} + \left\{ \begin{matrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{matrix} \right\}$$

Fig. 3.24 *Direct-Addressing Mode*

The registers that can be used to hold the values of the segment base, base, and index have also been shown in the figure. A number of addressing modes are defined by various combinations of these elements. They are called register indirect addressing, based addressing, indexed addressing and based-indexed addressing.

Direct-Addressing Mode

Direct-addressing mode is similar to immediate addressing in that information is encoded directly into the instruction. However, in this case, the instruction opcode is followed by an effective address, instead of the data. This is shown in the following example:

MOV DL, [2400H]; Move contents of DS:2400H into DL

The effective address is used directly as the 16-bit offset of the storage location of the operand from the location specified by the current value in the selected segment register. The default segment register is DS. Therefore, the 20-bit physical address of the operand in memory is normally obtained as DS:EA. However, by using a Segment-Override Prefix (SEG) in the instruction, any of the four segment registers can be used. Figure 3.25 shows the computation of direct memory address.

$$\text{PA} = \text{Segment base: Direct address}$$

$$\text{PA} = \left\{\begin{matrix}\text{CS}\\ \text{DS}\\ \text{SS}\\ \text{ES}\end{matrix}\right\} : \left\{\text{Direct address}\right\}$$

Fig. 3.25 Computation of Direct Memory Address

Specified constant value (called the displacement) gives offset. The examples of this particular mode are:

MOV AX, [22]

MOVAX, ES:[22]

MOV [16H], AX

MOVBYTE PTR [22], 98

MOV BX, [1234H]

MOV CX, Users

MOV[MyPointer], AH

The important points to be noted are:

- Displacements are encoded directly into the instruction.
- This facilitates global variables with fixed offset values to be addressed directly.

NOTES

The example instruction MOV AX, [22] is illustrated in Figure 3.26.

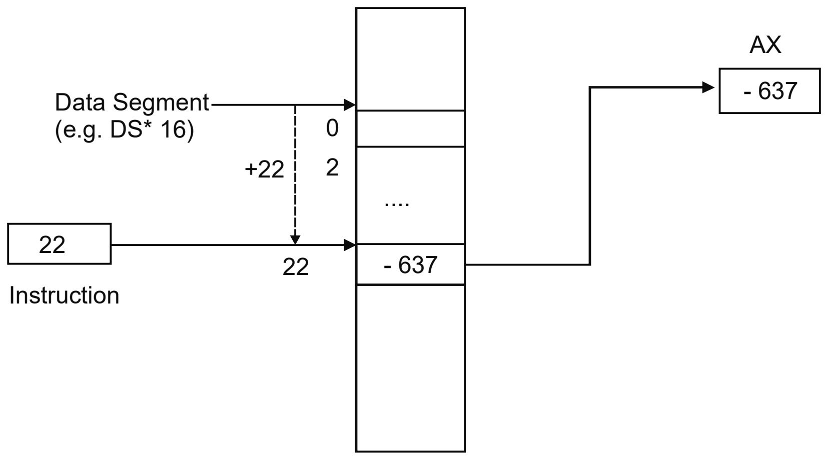


Fig. 3.26 *Instruction MOV AX, [22]*

Register Indirect-Addressing Mode

Register indirect-addressing mode is similar to direct-addressing mode in that an effective address is combined with the contents of DS to obtain a physical address. However, it differs in the way the offset is specified. This can be explained by the following examples:

MOV AL, [BX]; Copy the contents of the memory location DS:BX into AL

MOV [DI], AH; Copy the contents of AH into DS:DI

Figure 3.27 shows that in this case, EA comes from either a base register or an index register within the 8086. The base register can be either base register BX or base-pointer register BP, and the index register can be source-index register SI and destination-index register DI.

$$\text{PA} = \text{Segment base} : \text{Indirect address}$$

$$\text{PA} = \left\{ \begin{matrix} \text{CS} \\ \text{DS} \\ \text{SS} \\ \text{ES} \end{matrix} \right\} : \left\{ \begin{matrix} \text{BX} \\ \text{BP} \\ \text{SI} \\ \text{DI} \end{matrix} \right\}$$

Fig. 3.27 *Register Indirect-Addressing Mode*

Indirect addressing allows us to store the address of a location in a register and use this register to access the value stored at that location. This means that we can store the address of the string in a register and access the first character of the string via the register. If we increase the register content by 1, we can access the next character of the string. By continuing to increase the register, we can access each character of the string, in turn, processing it as we see fit. Figure 3.28 illustrates how indirect addressing operates, using register BX to contain the address of a string "Hello" in memory. Here, register BX has the value 1024 which is the address of the first character in the string. Another way of phrasing this is to say that BX points to the first character in the string.

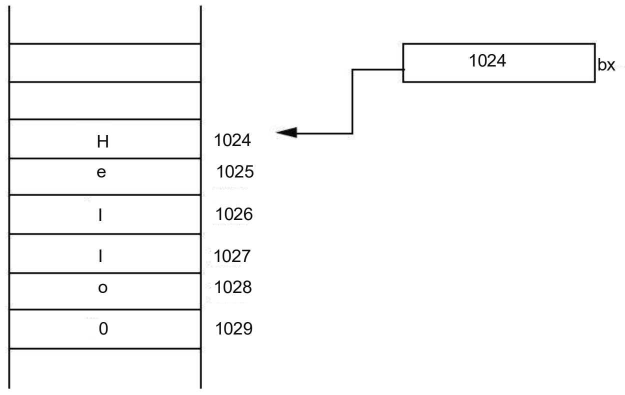


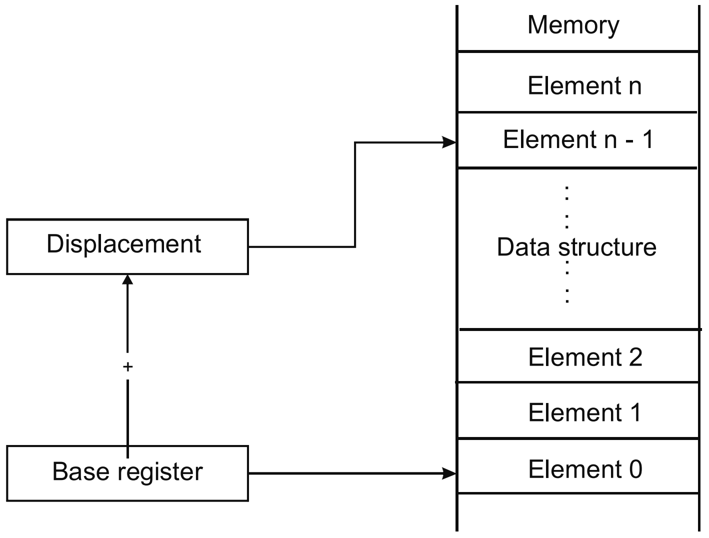
Fig. 3.28 *Indirect Addressing*

Base-Addressing Mode

In the base-addressing mode, the effective address of the operand is obtained by adding a direct or indirect displacement to the contents of either base register BX or Base Pointer BP. The physical address calculation is shown in Figure 3.29 (a) and (b). The value in the base register defines the beginning of a data structure, such as, a record in memory and the displacement selects an element of a data within this structure. To access a different element in the record, we can simply change the value of the displacement. To access the same element in another similar record, we can change the value in the base register so that it points to the beginning of the new record.

$$PA = \begin{Bmatrix} CS \\ SS \\ DS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{Bmatrix}$$

(a)



(b)

Fig. 3.29 *Physical Address Calculation in Base-Address Mode*

NOTES

In 8086 assembly language, we denote this by enclosing BX in square brackets, [BX], which reads as the value pointed to by BX, i.e., the contents of the location whose address is stored in the BX register. The other examples of this mode are:

MOV AX, [BX]

MOV [BP], AL

MOV AX, [DI]

MOV [SI], AH

MOV BX, [SI]

MOV [AX], CX

MOV CX, [BX] + 10H; move DS:BX+10H and DS:BX + 10H + 1H into CX

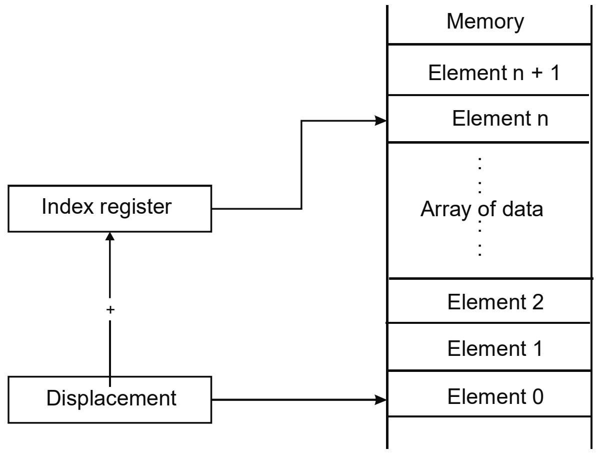
MOV AL, [BP] + 5H; PA = SS:BP+5H

Indexed-Addressing Mode

Indexed-addressing mode is similar to the base-addressing mode. The example of the indexed-relative mode is as follows:

MOV DX, [SI] + 5H ; PA = DS:SI+5H

As illustrated in Figure 3.30 (a), indexed-addressing mode uses the value of the displacement as a pointer to the starting point of an array of data in memory, and the contents of the specified register as an index that selects the specific element in the array, which is to be accessed. For instance, for the byte-size element array in Figure 3.30 (a), the index register holds the value *n*. In this way, it selects data element *n* in the array. Figure 3.30 (b) shows how the physical address is obtained from the value in a segment register, an index in the SI or DI register, and a displacement.



(a)

PA = Segment base : Index + Displacement

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{Bmatrix}$$

(b)

Fig. 3.30 *Indexed-Addressing Mode*

NOTES

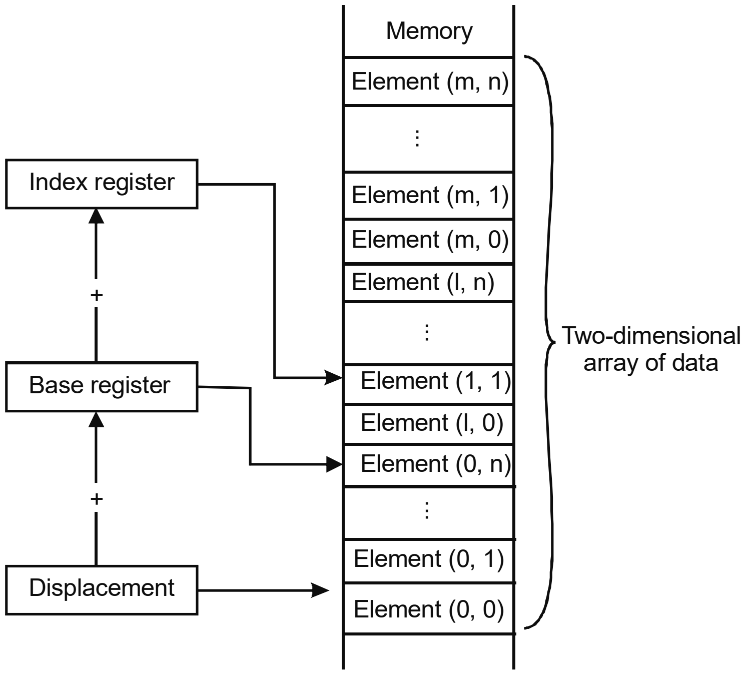
Based-Indexed Addressing Mode

Combining the based-addressing mode and the indexed-addressing mode results in a more powerful mode known as based-indexed addressing mode. This addressing mode can be used for complex data structures such as two-dimensional arrays. The example of based-indexed relative addressing is as follows:

MOV CL, [BX][DI]+8H; PA = DS:BX+DI+8H

MOV AH, [BP][DI]+12H; PA = SS:BP+DI+12H

Figure 3.31 shows how based-indexed addressing mode can be used to access elements in an *m* ´ *n* array of data. It can be seen that the displacement, which is a fixed value, locates the array in memory. The base register specifies the *m* coordinate of the array and index register identifies the *n* coordinate. Any element in the array can be accessed simply by changing the value in the base and index register.



(a)

PA = Segment base : Base + Index + Displacement

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{Bmatrix}$$

(b)

Fig. 3.31 Based-Indexed Addressing Mode

Basically, the sum of specified base register and index register gives offset. The examples of this addressing mode are as follows:

MOV CX, [BX + DI]

MOV[AX + BX], CX

MOV CH, [BP] [SI]

MOV[BX] [SI], SP

MOV CL, [DX] [DI]

NOTES

MOV[AX] [BX], CX

MOV[BP + DI], CX

A summary of the 8086 addressing modes is shown in Table 3.1 along with specific examples:

Table 3.1 *8086 Addressing Modes*

Addressing Mode	Operand	Default Segment
Register	Reg	none
Immediate	Data	none
Direct	[offset]	DS
Register indirect	[BX] [SI] [DI]	DS DS DS
Based relative	[BX]+disp [BP]+disp	DS SS
Indexed relative	[DI]+disp [SI]+disp	DS DS
Based indexed relative	[BX][SI]+disp [BX][DI]+disp [BP][SI]+disp	DS DS SS

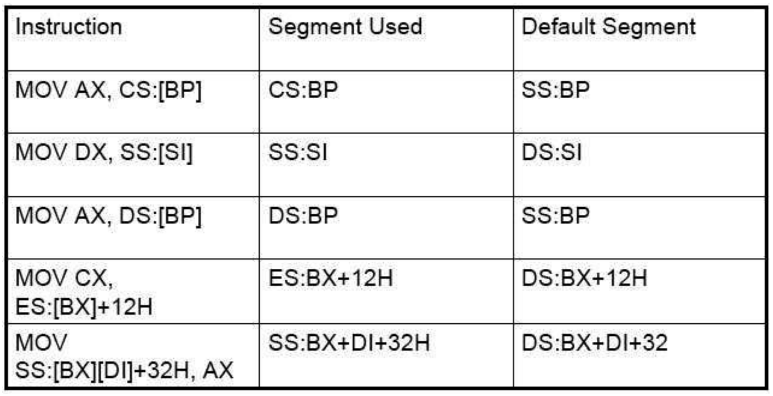
1. Register	MOV AX, BX
2. Immediate	ADD AL, 10H
3. Direct	MOV CX, [1200H]
4. Register indirect	MOV AL, [$\underset{\underset{\text{SI, DI}}{\uparrow}}{\text{BX}}$]
5. Based relative	MOV CX, [$\underset{\underset{\text{BP}}{\uparrow}}{\text{BX}}$ + 10H]
6. Indexed relative	MOV AX, [$\underset{\underset{\text{DI}}{\uparrow}}{\text{SI}}$ + 15H]
7. Based indexed relative	MOV AH, [$\underset{\underset{\text{BP}}{\uparrow}}{\text{BX}}$ + $\underset{\underset{\text{SI}}{\uparrow}}{\text{DI}}$ + 5H]

The offset registers for various segments are shown as follows:

Segment register	CS	DS	ES	SS
Offset register(s)	IP	SI, DI, BX	SI, DI, BX	SP, BP

Although in different addressing modes, a particular combination of segment and offset register is used, in some cases, instead of a default segment, another segment is used for calculating the address. This is referred to as *segment override*. In case of segment override too, different combinations are possible and these are listed in Table 3.2.

Table 3.2 *Various Combinations in Segment Override*



Instruction	Segment Used	Default Segment
MOV AX, CS:[BP]	CS:BP	SS:BP
MOV DX, SS:[SI]	SS:SI	DS:SI
MOV AX, DS:[BP]	DS:BP	SS:BP
MOV CX, ES:[BX]+12H	ES:BX+12H	DS:BX+12H
MOV SS:[BX][DI]+32H, AX	SS:BX+DI+32H	DS:BX+DI+32

3.4 INSTRUCTION FORMATS

The 8085 instruction set is classified into the following three groups according to word size or byte size:

(i) 1-Byte Instruction

(ii) 2-Byte Instruction

(iii) 3-Byte Instruction

1-Byte Instruction

A 1-byte instruction includes the opcode and operand in the same byte, as shown in Table 3.3.

Table 3.3 *1-Byte Instruction*

Task	***Opcode***	***Operand***	***Binary Code***	***Hex Code***
Copy the contents of the accumulator in register C.	MOV	C.A	0100 11 11	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert each bit in the accumulator.	CMA		0010 11 11	2 FH

These 1-byte instructions perform three different tasks. In the first instruction, both the operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bits binary format in memory; each requires one memory location.

NOTES

2-Byte Instruction

In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand as shown in Table 3.4.

Table 3.4 *2-Byte Instruction*

Task	*Opcode*	*Operand*	*Binary Code*	*Hex Code*
Load an 8-bit data byte in the accumulator.	MVI	A, 32H	0011 1110 0011 0010	3E First Byte 32 Second Byte
Load an 8-bit data byte in register B.	MVI	B, F2H	1100 0011 1000 0101 0010 0000	06 First Byte F2 Second Byte

These instructions require two memory locations each to store the binary codes. The data bytes 32H and F 2H are selected arbitrarily as examples.

3-Byte Instruction

In a 3-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third-byte is the high-order address, as shown in Table 3.5.

Table 3.5 *3-Byte Instruction*

Task	*Opcode*	*Operand*	*Binary Code*	*Hex Code*
Load contents of memory 2050H into A.	LDA	2050H	0011 1010 0101 0000 0010 0000	3A First Byte 50 Second Byte 20 Third Byte
Transfer the program Sequence to memory location 2085H	JMP	2085H	1100 0011 1000 0101 0010 0000	C3 First Byte 85 Second Byte 20 Third Byte

These instructions require three memory locations each to store the binary codes.

The various ways of specifying data are called **addressing modes.** Although microprocessor instructions require one or more words to specify the operands, the notations and conventions used in specifying the operands have very little to do with the operation of the μP. The mnemonic letters used to specify a command are chosen by the manufacturer. When an instruction is stored in memory, it is stored in binary code, the only code the μP is capable of reading and understanding. The conventions used in specifying the instructions are valuable in terms of keeping uniformity in different programs and in writing assemblers. Note that the μP neither reads nor understands mnemonics or hexadecimal numbers.

Programs Example

To learn assembly language programming, a beginner should write simple programs. The memory addresses given in the program are for a particular μP kit. These addresses can be changed to suit the μP bit available in the laboratory. Before writing an assembly language program, one should learn some important Intel 8085 instructions such as MOV, MUI, ADD, SUB, LXI, LDA, INX, INR and HLT.

NOTES

Example 3.1: Object: Place 05 in Register B.

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
FC 00	06, 05	MUI	B, 05	Get 05 in register B
FC 02	76	HLT		Stop

The instruction MUI B, 05 moves 05 to register B. HLT halts the program. A program is fed to up kit in machine codes. The machine code for the instruction MVI B, 05 is 06, 05. The first byte of the instruction is 06 which is the machine code for the instruction MVI B. The second byte of the instruction, 05, is the dates which is to be moved to register B. The code for HLT is 76. The machine codes for a program are entered in the memory. In this case, the memory addresses from FC 00 to FC 02 have been used. The machine code 06 is entered in the memory location FC 00 H; 05 in FC 01 H and 76 in FC 02 H.

Example 3.2: Object: Place 05 in the accumulator. Increment it by one and store the result in the memory location FC 50 H.

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
FC 00	3E, 05	MVI	A, 05	Get 05 in the accumulator
FC 02	3C	INR	A	Increment the content of the accumulator by one
FC 03	32, 50, FC	STA	FC 50 H	Store the result in FC 50 H
FC 06	76	HLT		Halt

The instruction MVI A, 05 moves 05 to the accumulator. INR A increases the content of the accumulator from 05 to 06. STA FC 50 stores the content of the accumulator in the memory location FC 50 H. After the execution of the above program the memory location FC 50 H will contain 06.

Addition of Two 8-bit Numbers; Sum 8-bits

Example 3.3: Add 49 H and 56 H.

The first number 49 H is in the memory location 2501 H.

The second number 56 H is in the memory location 2502 H.

The result is to be stored in the memory location 2503 H.

Numbers are represented in hexadecimal number system.

NOTES

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
2000	21, 01, 25	LXI	H, 2501 H	Get address of first number in H-L pair.
2003	7 E	MOV	A, M	First number in accumulator.
2004	23	INX	H	Increment content of H-L pair.
2005	86	ADD	M	Add first and second numbers.
2006	32, 03, 25	STA	2503 H	Store sum in 2503 H.
2009	76	HLT		Stop.

Data

2501—49 H

2502—56 H

The sum is stored in the memory location 2503 H.

Result

2503—9F H

2501 H is the address of the memory location for the first number. 2501 is placed in H–L pair by the instruction LXI H, 2501 H. The next instruction is MOV A, M moves the content of the memory location addressed by H–L pair to the accumulator. In this case, H–L pair contains 2501 H and, therefore, the content of the memory location 2501 H is moved to the accumulator. Thus, the first number 49 H has been moved to the accumulator. The instruction INX-H increases the content of H–L pair by one. Previously, the content of H–L pair was 2501 H. After the execution of INX-H, it becomes 2502 H. ADD M adds the contents of the accumulator and the content of the memory location addressed by the H–L pair. The content of 2502 H is the second number 56 H. So 56 H is added to 49 H. The sum resides in the accumulator. The instruction STA 2503 H stores the sum in the memory location 2503 H. The instruction HLT ends the program.

Addition of Two 8-bit Numbers; Sum 16-bits

Example 3.4: Add 98 H and 9A H.

Sum = 98 H + 9A H = 01, 32 H

The first number 98 H is in the memory location 2501 H.

The second number 9A H is in the memory location 2502 H.

The results are to be stored in 2503 H and 2502 H.

In this case, the sum is to be stored in two consecutive memory locations. The Least Significant Bits (LSBs) of the sum is 32 H and it will be stored in the memory location 2503 H. The Most Significant Bits (MSB) of the sum is 01, which will be stored in 2504 H.

Program

Memory Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	21, 01, 25		LXI	H, 2501 H	Address of first number in the H–L pair.
2003	0E, 00		MVI	C, 00	MSBs of sum in register C. Initial value = 00.
2005	7E		MOV	A, M	First number in accumulator.
2006	23		INX	H	Address of second number 2502 in N–L pair.
2007	86		ADD	M	First number + Second number.
2008	D2, 0C, 20		JNC	AHEAD	Is carry? No, go to the label AHEAD.
200B	0C		INR	C	Yes, increment C.
200C	32, 03, 25	AHEAD	STA	2503 H	LSBs of sum in 2503 H.
200F	79		MOV	A, C	MSBs of sum in accumulator.
2010	32, 04, 25		STA	2504 H	MSBs of sum in 2504 H.
2013	76		HLT		Halt.

NOTES

8-Bit Multiplication; Product 16-Bit

Decimal multiplication. For a decimal multiplication, the procedure is as shown in the following example:

Example 3.5:

36D, Multiplicand

× 29D, Multiplier

324D,

72D4,

Product: $1044D = 1044_{10}$

Binary Multiplication

Example 3.6: Multiply 7 by 5.

First, represent 7 and 5 in binary form:

$7_{10} = 0111_2$ and $5_{10} = 0101_2$

0111_2, Multiplicand

$\times\ 0101_2$, Multiplier

0111_2,

00001_2,

011111_2,

0000111_2,

0100011_2, $= 35_{10}$

Example 3.7: Multiply 84 H by 56 H.

Here, 84 H is the multiplicand and 56 H is the multiplier. 84 H is extended to 16 bits and stored in the two consecutive memory locations 2501 and 2502 H. 56 H is stored in 2503 H. The product is a 16-bit number and it is stored in 2504 H and in 2505 H. The data and results are in hexadecimal. The program flow chart is shown in Figure 3.32.

NOTES

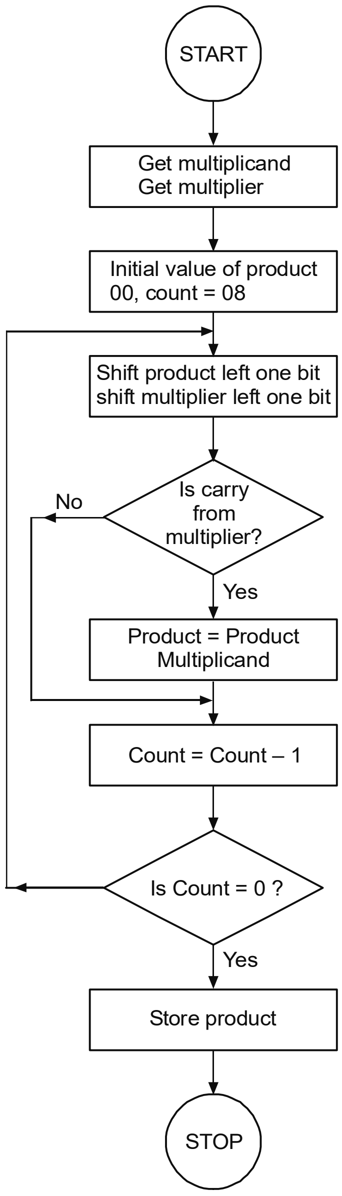


Fig. 3.32 *Program Flow Chart for an 8-bit Multiplication*

Program

Memory Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	2A, 01, 25		LHLD	2501 H	Get multiplicand in H–L pair.
2003	EB		XCHG		Multiplicand in D–E pair.
2004	3A, 03, 25		LDA	2503 H	Multiplier in accumulator.
2007	21, 00, 00		LXI	4,000	Initial value of product = 00 in H–L pair.
200A	0E, 08		MVI	C, 08	Count = 8 in register C.
200C	29	Loop	DAD	H	Shift partial product left by 1 bit.
200D	17		RAL		Rotate multiplier left one bit. Is multiplier's bit = 1?

```
200E    D2, 12, 20           JNG    AHEAD    No, go to AHEAD.
2011       19                DAD      D      Product = Product + Multiplicand
2312       OD       AHEAD    DCR      C      Decrement Count.
2013    C2, OC, 20           JNZ     Loop
2016     22, 04, 25          SHLD   2504 H   Store result.
2019       76                HLT             Stop.
```

The instruction LHLD 2501H transfers the 16-bit multiplicand from the memory locations 2501 and 2502 H to H–L pair. By executing the instruction XCHG, the contents of H–L pair are exchanged with the contents of D–E pair. Thus, the multiplicand is placed in D–E pair. The instruction LDA 2503 H transfers the multiplier from the memory location 2503 H to the accumulator. LXIH, 0000 makes the initial value of the product equal to zero and it is placed in H–L pair. The count is equal to the bits of the multiplier. In this case, it is 08 and is placed in register C. DADH is an instruction for 16-bit addition. It adds the contents of H–L pair to itself. Thus, the partial product which is in H–L pair is shifted to the left by one bit. RAL rotates the content of the accumulator to the left by one bit. The accumulator contains the multiplier and hence it is rotated to the left by one bit. The instruction DAD D adds the content of D–E pair and H–L pair and places the result in H–L pair. D–E pair and H–L pair contains the multiplicand and partial product respectively. Thus, the execution of the instruction DAD D adds the multiplicand to the partial product and places the sum which is the new partial product in H–L pair. The instruction DAD D is executed only when the bit of the multiplicand under consideration is one; otherwise, it is not executed. To get the result, the program moves in the loop eight times, as there are eight bits in the multiplier.

Example 3.8

Data

2501—84 H, LSBs of multiplicand

2502—00, MSBs of multiplicand

2503—56 H, Multiplier

Result

2504—58 H, LSBs of product

2505—26 H, MSBs of product

Example 3.9:

Data

2501—84 H, LSBs of multiplicand

2502—00, MSBs of multiplicand

2503—52 H, Multiplier

Result

2504—34 H, LSBs of product

2505—2C H, MSBs of product

Check Your Progress

1. What are the four categories of registers?
2. Define the term segment registers.
3. What is the function of interrupt-enable flag?
4. What does the little endian convention state?
5. Define the term READY signal.
6. What are addressing modes?
7. Does the microprocessor read the mnemonics?

3.5 CACHE MEMORY AND CACHE CONTROLLERS

An effective memory system is the key to any successful computer system. The memory system is said to be effective if the access time of the cache is close to the effective access time of the processor. Cache is usually the first level of memory access by the microprocessor. The achievement of this goal is mainly governed by the following factors:

(a) Architecture of the Microprocessor

(b) Properties of the Programs being Executed

(c) Size and Organization of the Cache

The principle of working of the cache memory largely depends upon the locality of program behaviour. Mainly, there are three types of localities to be considered:

1. **Spatial Locality:** If an access to a particular location in memory is given, there is a high probability that other accesses will be made to either that or the neighbouring locations during the lifetime of the program.
2. **Temporal Locality:** Temporal locality works as a complementary to spatial locality. If a sequence of references to *n* locations is given, there is a high probability that references following this sequence will be made into the sequence. Elements of the sequence will again be referenced during the lifetime of the program.
3. **Sequentiality:** If a reference has been made to a particular location *s*, it is likely that within the next several references, a reference to the location of $s + 1$ will be made. Sequentiality is normally considered to be a restricted type of spatial locality and can be regarded as a subset of it.

Cache Memory

The *cache* is a small amount of high-speed memory used to work directly with the microprocessor. The memory cycle time of cache is usually in the order of the time required by the CPU to fetch an instruction. The cache usually gets its data from the main memory whenever the instructions or data is required by the CPU. The main memory can contain wider data, whereas the CPU requirement may be of the data with less width. The amount of information which can be placed at one time in the cache memory is called the *line size* for that particular cache. This normally resembles the width of the data bus between the cache memory and the main memory. If a cache has a wide-line size, then it means that several instruction or data words are loaded into the cache at one time. This itself becomes a kind of prefetching for instructions or data. The memory hierarchy of a modern computer is shown in the following Figure (Refer Figure 3.33):

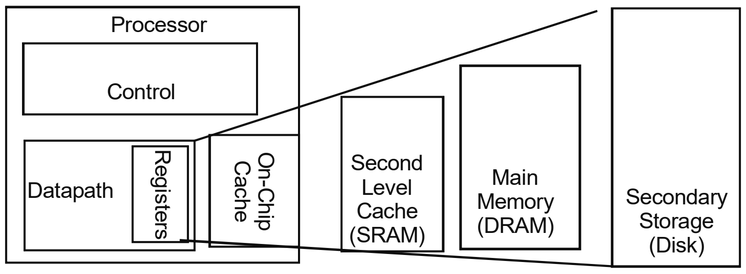


Fig. 3.33 *Memeory Heirarchy in Computer*

However, depending upon the usage, almost everything can be considered as a cache, such as:

- Registers work as cache on the variables.
- First level of cache (normally Referred to as L1) works as a cache on second-level cache (normally Referred to as L2).
- Second-level cache is a cache on the main memory (RAM).
- Main memory works as a cache for the secondary storage disk (virtual memory).
- TLB works as a cache for the page table.
- Branch prediction works as a cache for the prediction information.

However, the speed of cache is different at different levels. The complete memory structure has been compared in the speed and the size in the following Figure (Refer Figure 3.34).

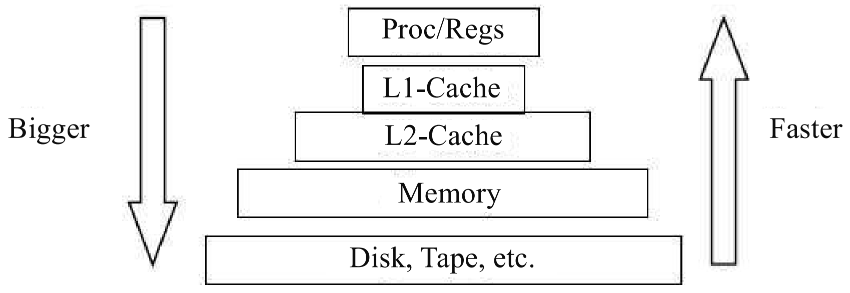


Fig. 3.34 *Speed and Size of Memory Structure*

Since the cache is small, the effectiveness of the cache relies on the following properties of most programs:

- Most of the programs are highly sequential in nature. Thus, the next instruction to be executed usually comes from the next memory location. Data is usually structured and data in these structures normally are stored in continuous memory locations.

NOTES

- Most of the programs have short loops as a common program structure, especially for the innermost sets of nested loops. Thus, the same small set of instructions is used over and over (during execution). Generally, several operations are performed on the same data values or variables. You may also assume that the code remains the same for particular kind of operations, which is to be carried out for different data sets.

Whenever the cache memory is used, there must be some way in which the memory controller determines whether the value currently being addressed in memory is available from the cache or not. There are several ways in which this can be accomplished. One possibility is to store both the address and the value from main memory in the cache, with the address stored in a type of memory called associative memory or, more descriptively, content-addressable memory. Within the cache, there are three basic types of organization:

1. Fully Associative Cache
2. Direct-Mapped Cache
3. Set-Associative Cache

Fully Associative Cache

An associative memory, or content-addressable memory, works on a simple principle. It has the property that when a value is presented to the memory, the address of the value is returned if the value is available in the memory, otherwise an indication that the value is not in the associative memory is returned. Since all of the comparisons are done simultaneously, the search is performed very quickly. However, the cost for this type of memory becomes very expensive, because each memory location must have both a comparator for comparison purpose as well as a storage element. The cache memory can be implemented with a block of associative memory, together with a block of 'ordinary' memory. In this case, the associative memory holds the address of the data stored in the cache, and the ordinary memory contains the data at that address. Such a cache memory might be configured as shown in Figure 3.35.

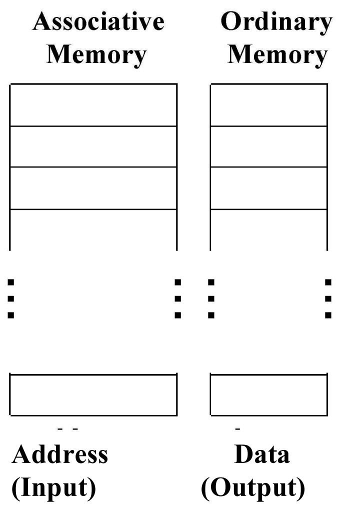


Fig. 3.35 Cache Implemented with Associative Memory

In fully associative mapping, when a request is made to the cache, the requested address is compared with all the available entries in the directory. Now, if the particular address is found (this is referred to as a directory hit), then the corresponding location is fetched from the cache memory and sent to the processor. The alternate case, when the particular address is not available, is referred to as a miss. In this case, if the address is not found in the associative memory, then the value is obtained from main memory. The process has been illustrated in Figure 3.36 considering the data and the tags:

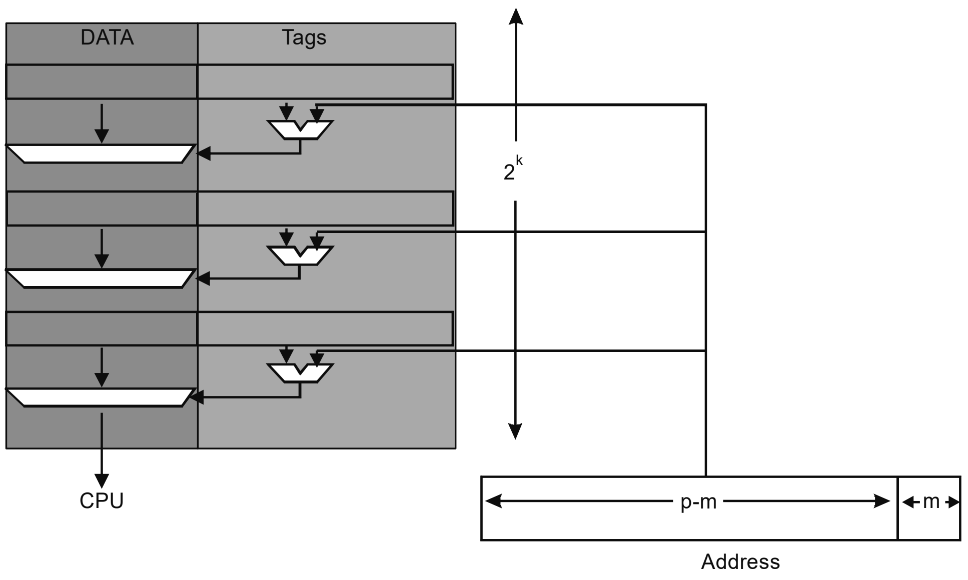


Fig. 3.36 *Associative Memory using Data and Tags*

Direct-Mapping Cache

Associative memory proves to be expensive when it comes to implementation, because a comparator is required for every word in the memory to perform all the comparisons in parallel. This poses a problem while designing cost-effective system. Thus, the designers are always looking for cheaper methologies. Direct mapping is a cheaper way to implement a cache memory, without using expensive associative memory. In this case, only a part of the memory address (usually the low-order digits of the address) is used to address a word in the cache. This part of the address is usually referred to as the index. The remaining high-order bits in the address, known as tags, are stored in the cache memory along with the data.

For example, if the processor has 18-bit address bus for memory, and a cache of 1K words of 2 bytes (16 bits) length, and the processor can address single bytes or 2 byte words, we might have the memory address field and cache organized as in the following Figure (Refer Figure 3.37).

NOTES

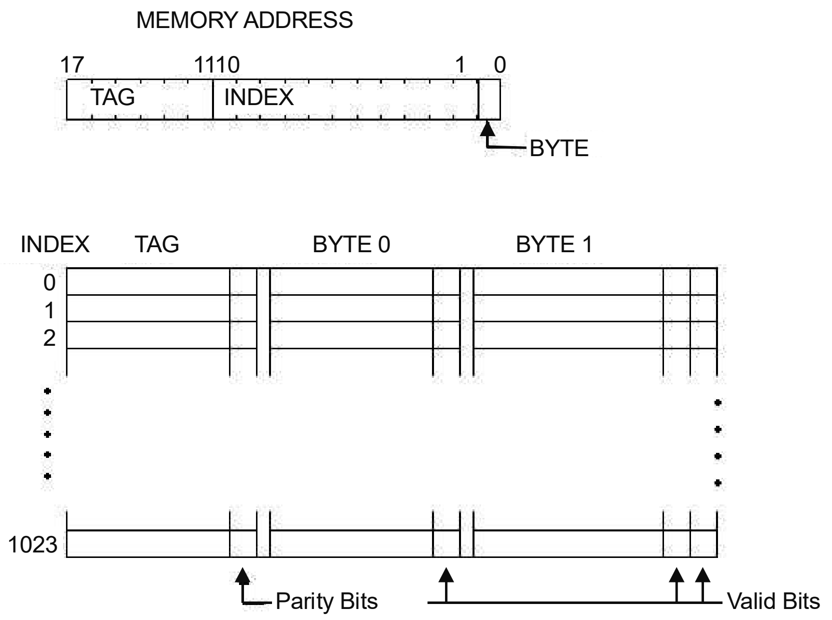


Fig. 3.37 A Direct-Mapped Cache Configuration

This is similar to the way in which the cache is organized in the PDP-11/60 processor. In this processor, however, there are 4 other bits that are used to ensure that the data in the cache is valid. Three of these bits are the *parity bits*; essentially one for each byte and one for the tag. The parity bits are used to check that a single-bit error has not occurred to the data while in the cache. A fourth bit, called the *valid bit,* is used to indicate whether or not a given location in cache is valid. In case of the PDP-11/60 processor and also in many other processors, the cache is not updated if memory is altered by a device other than the CPU. For example, when a disk stores new data in memory and microprocessor is not responsible for that, it is possible that it has been done through the DMA. When such a memory operation occurs to a location which has its value stored in cache, the valid bit is reset to show that the data is 'stale' and does not correspond to the data in main memory. Also, the valid bit is reset when power is first applied to the processor or when the processor recovers from a power failure. In these cases, the data available in the cache at that time will be invalid.

If we consider the example of PDP-11/60 microprocessor, the data path from memory to cache was the same size (16 bits) as that from cache to the CPU. However, in faster machines like the PDP-11/70, the data path from the CPU to cache is 16 bits, while from memory to cache it is 32 bits which means that the cache can effectively prefetch the next instruction, approximately in half the time. The information (instructions or data) stored with each tag in the cache is called the line size of the cache. It is usually equivalent to the data path from main memory to the cache. In case of large line size, it allows the prefetching of a number of instructions or data words. All items in a line of the cache are replaced in the cache simultaneously, however, resulting in a larger block of data being replaced for each cache miss.

In case of MIPS R2000/R3000 processor, the architecture itself has a built-in cache controller and this controller could control a cache up to 64K bytes. For a similar 2K word (or 8K byte) cache, the MIPS processor typically follows a cache configuration as shown in Figure 3.38.

NOTES

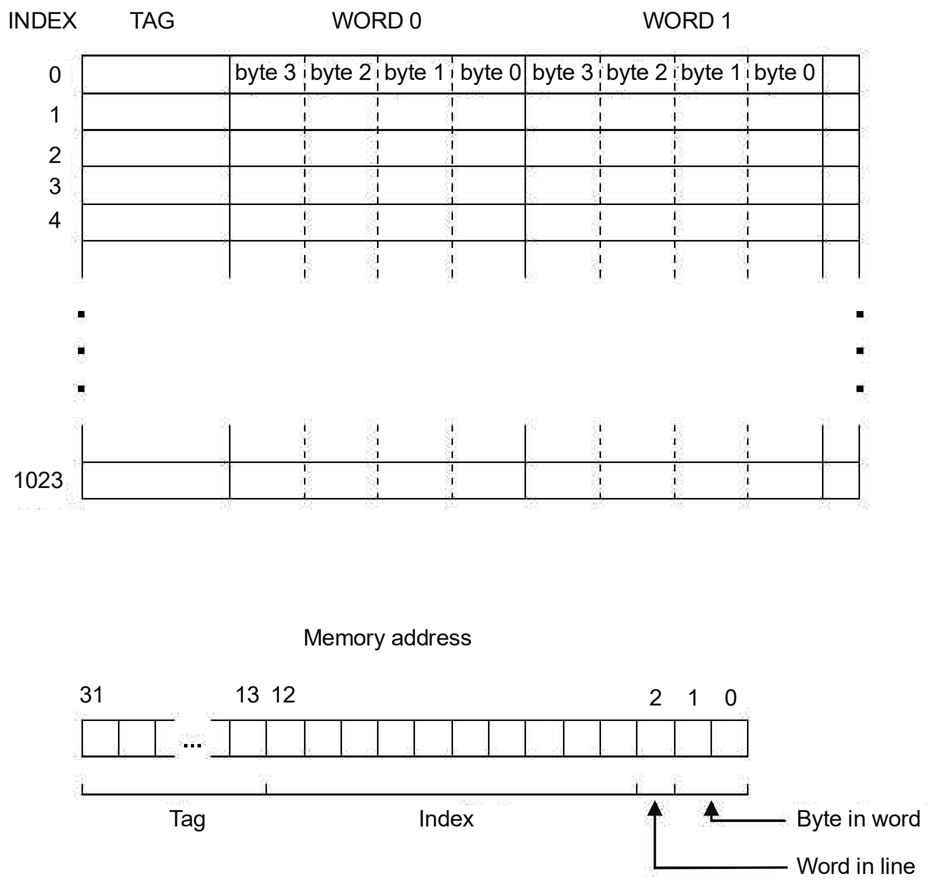


Fig. 3.38 *MIPS Cache Organization*

In case of direct-mapped cache, lower-order line address bits are used to access the directory. Since multiple-line addresses map into the same location in the cache directory, the upper-line address bits (tag bits) must be compared with the directory address to ensure a hit. If the comparison results are false, a cache miss or simply a miss happens. However, the significant thing is that the address given to the cache by the microprocessor is actually subdivided into several pieces. Each of these pieces has a different role in accessing data. The concept of direct-mapping cache is illustrated Figure 3.39.

NOTES

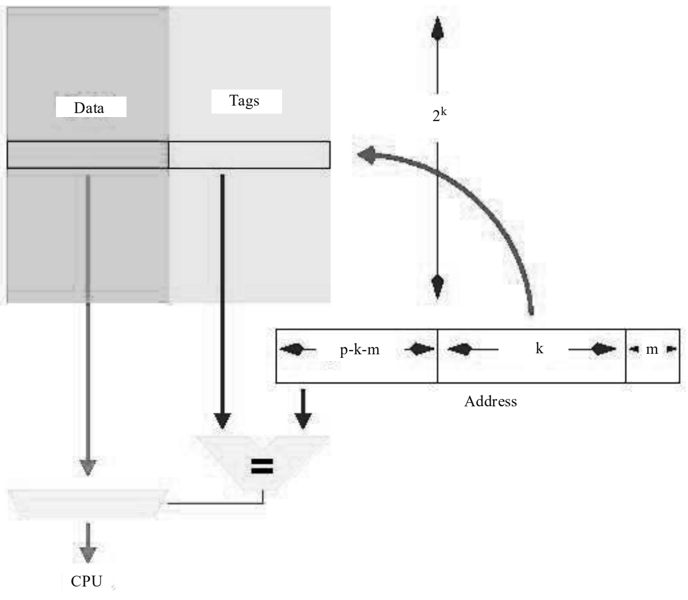


Fig. 3.39 *Direct-Mapped Cache*

An important characteristic of the direct-mapped cache is that a particular memory address can only be mapped into one cache location. *M*any memory addresses are however mapped to the same cache location. Practically, all addresses with the same index field are mapped to the same cache location. Whenever a ‘cache miss’ happens, the cache line is replaced by a new line of information from the main memory at an address with the same index but with a different tag. Thus, if the program ‘jumps around’ in memory, this cache organization is less likely to be effective because the index range is limited. However, if both instructions and data are stored in the same cache memory, both will be mapped into the same area of cache, and both often replace each other. An example of this kind of situation can be considered as a matrix operation. The code for the matrix operation as well as the matrix data will have the same index values. Thus, it can be concluded that the directly-mapped cache is a nice implementation considering the cost. The implementation can be well illustrated with Figure 3.40.

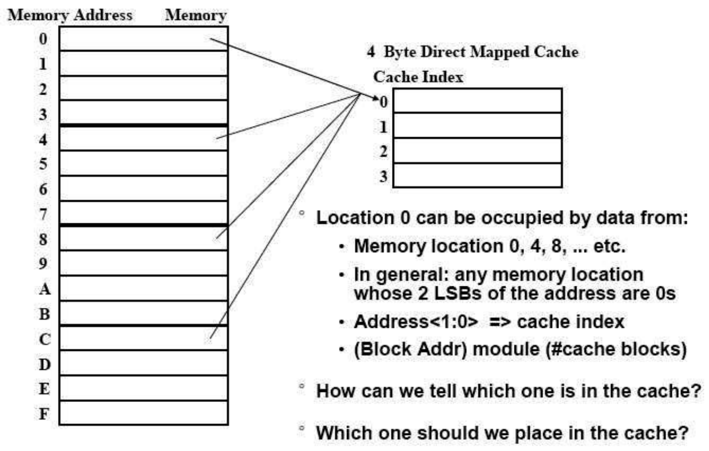

Memory Address
Memory
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
4 Byte Direct Mapped Cache
Cache Index
0
1
2
3
Location 0 can be occupied by data from:
• Memory location 0, 4, 8, ... etc.
• In general: any memory location whose 2 LSBs of the address are 0s
• Address<1:0> => cache index
• (Block Addr) module (#cache blocks)
How can we tell which one is in the cache?
Which one should we place in the cache?

Fig. 3.40 *Implementation of Direct-Mapped Cache*

Set-Associative Cache

A more interesting configuration for the cache implementation is the *set-associative* cache. This configuration uses another kind of mapping known as the *set-associative mapping*. The simple technique used here is that a given memory location is allowed to be mapped to more than one cache locations. Hence, each index usually corresponds to two or more data words, each with a corresponding tag. A set-associative cache with n tag and data fields is called an 'n-way set-associative cache'. Usually $n = 2^k$, where $k = 1, 2, 3$, chosen for a set-associative cache, and $k = 0$ for direct mapping. Such n-way set-associative caches allow interesting tradeoff possibilities; cache performance can be improved by increasing the number of 'ways', or by increasing the line size for a given amount of memory. A 2-way set-associative cache is shown in Figure 3.41, which shows a cache containing a total of 2K lines, or 1K sets, each one of these sets being 2-way associative.The sets correspond to the rows in Figure 3.41.

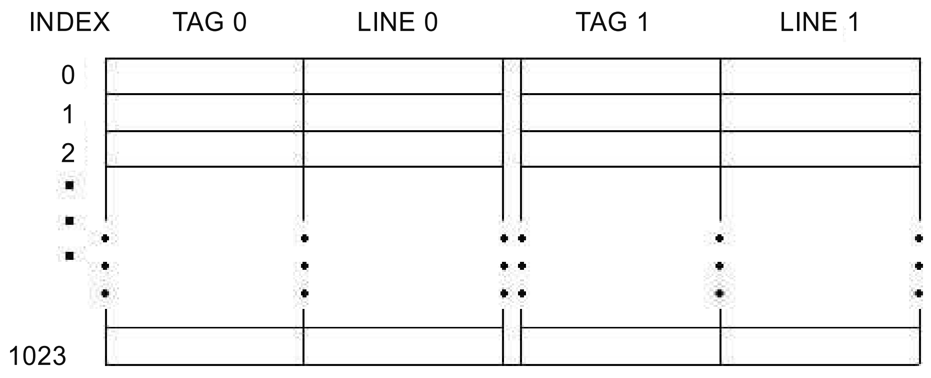


Fig. 3.41 *Set-Associative Cache Organization*

The working of set-associative cache is more or less on the similar lines to the direct-mapped cache. However, it uses another concept of sub cache. Bits from the line address are used to address a cache directory. However, there are multiple choices available for the programmer: (a) two, (b) four, or (c) more complete line addresses may be present in the directory. These line addresses basically correspond to a location in the sub cache. The collection of these sub caches forms a total cache array that can be accessed in the same manner as the direct-mapping cache. All of these sub arrays can be accessed simultaneously, together with the cache directory. If any of the entries in the cache directory match the reference address, and there is a hit, the particular sub-cache array is selected and the same is sent back to the processor. The concept of set-associative cache is illustrated in Figure 3.42.

NOTES

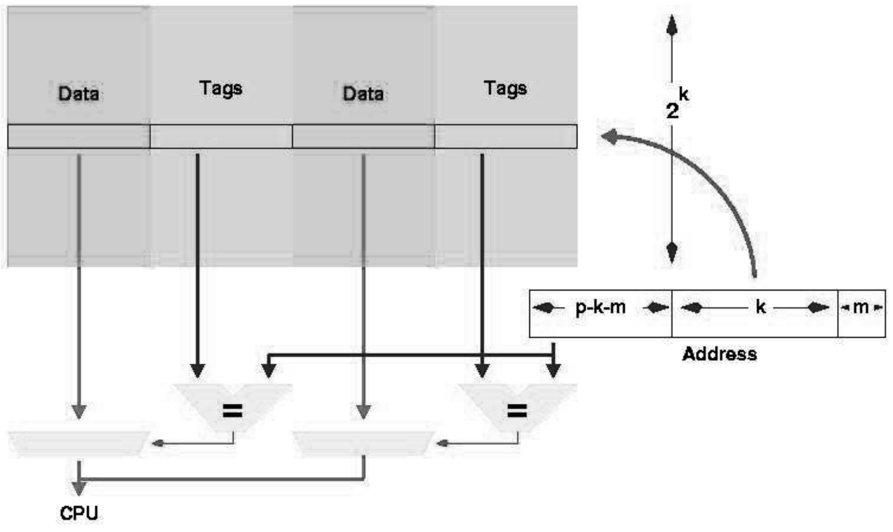


Fig. 3.42 *Set-Associative Cache*

In case of a 2-way set-associative cache, if one data word is empty for a read operation corresponding to a particular index, then it is filled. If both data words are filled, then one must be overwritten by the new data. Similarly, in an *n*-way set-associative cache, if all *n* data and tag fields in a set are filled, then one value in the set must be overwritten, or replaced, in the cache by the new tag and data values. Thus, an entire line must be replaced each time. For replacement, the following algorithms are most common:

1. Random: In case of random algorithm, the location for the value to be replaced is chosen at random from all memories of the cache locations at that index position. In a 2-way set-associative cache, this can be accomplished with a single modulo where two random variables are obtained (for example, from an internal clock).
2. First In, First Out (FIFO): In this case, the first value stored in the cache, at each index position, is replaced at the first time. Considering a 2-way set-associative cache, this replacement strategy can be implemented by setting a pointer to the previously loaded word each time a new word is stored in the cache. This pointer is required to be a single bit. However, for set sizes greater than 2, this algorithm can put another bit to each cache location. The strategy operates by setting a bit in the other word when a value is stored and resetting the corresponding bit for the new word. For an *n*-way set-associative cache, this strategy can be implemented by storing a modulo *n* counter with each data word. However, as the value of *n* becomes large, the circuitry being implemented becomes very complex.

Cache memories normally allow one of two things to happen when data is written into a memory location for which there is a value stored in cache. These are as follows:

- Write-through cache: In this particular case, both the cache and main memory are updated at the same time. This may actually slow down the execution of instructions which write data to memory because of the relatively longer write time with regard to main memory. Buffering the memory writes can help in speeding up the procedure. This is particularly useful if they are relatively infrequent.

- Write-back cache: In this case, the cache is updated directly by the microprocessor. The cache memory controller marks the value so that it can be written back into memory when the word is actually removed from the cache. This method is used because a memory location may often be altered several times while it is still in cache without having to write the value into main memory. This method is generally implemented using an 'ALTERED' bit in the cache. The ALTERED bit is set whenever a value is written in the cache by the microprocessor. Only if the ALTERED bit is set, then only it is necessary to write the value back into main memory. The value should be written back immediately before any value is replaced in the cache.

The microprocessor MIPS R2000/3000 uses the write-through approach with a buffer for the memory writes. Practically, the memory writes are less frequent than memory reads. Typically for each memory write, an instruction must be fetched from main memory and usually two operands are fetched as well. Therefore, we may say that read operation are three times more than the write operations. In fact, there is often many more memory read operations than memory write operations. Figure 3.43 shows the behaviour (actually the miss ratio, which is equal to 1 minus the hit ratio) for cache memories with various combinations of total cache memory capacity and width of the cache memory.

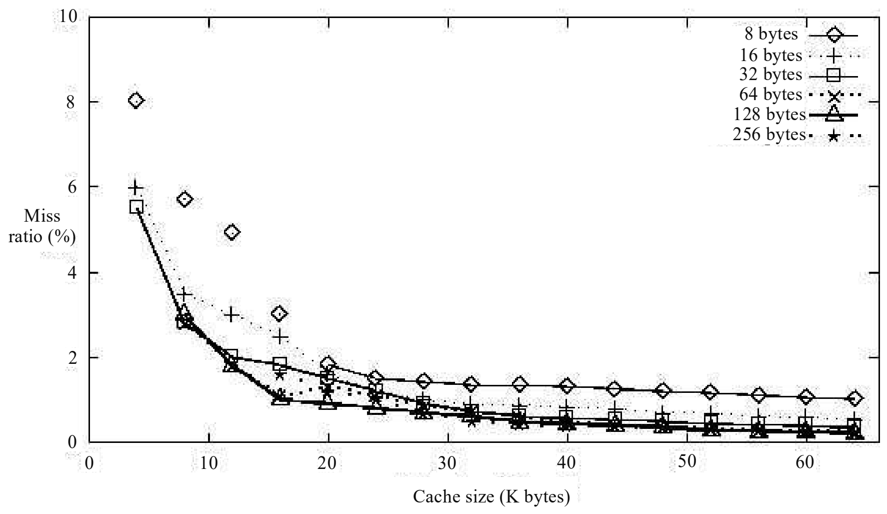


Fig. 3.43 *Cache Memory Performance for Various Line Sizes*

These results are of the simulations of the behaviour of several 'typical' program mixes. It can be seen that the miss ratio drops consistently with cache size. Also, increasing the line size is not always effective in increasing the throughput of the processor, although it does decrease the hit ratio. This is because of the additional time required to transfer large lines of data from the main memory to the cache. It becomes very interesting to plot the same data using log-log coordinates. In this case, the graph is roughly linear as shown in Figure 3.44.

NOTES

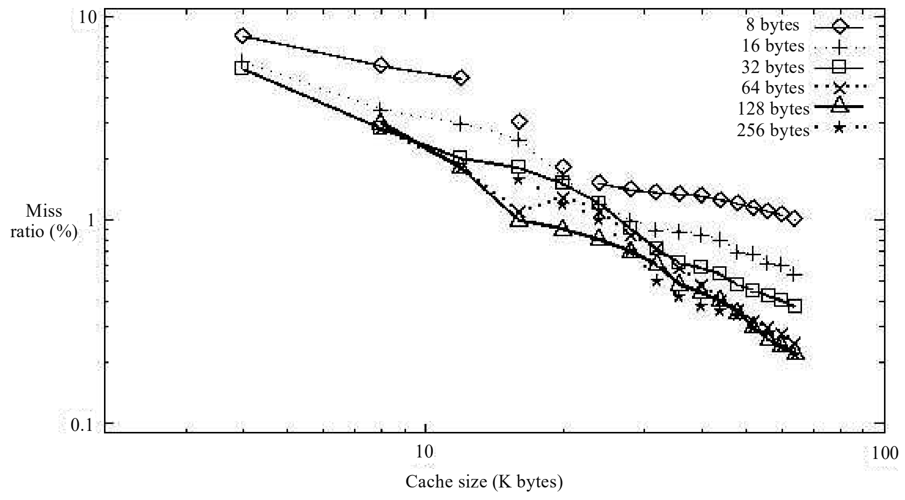


Fig. 3.44 *Plot of Cache Performance for Various Line Sizes*

What Happens When There is a Cache Miss?

Microprocessor references that are available in the cache are called *cache hits*, whereas those not available in the cache are called *cache misses*. On a cache miss, the cache control mechanism must fetch the missing data from memory and place it in the cache. Usually the cache fetches a spatial locality from memory , which is called a line. The physical word is the basic unit of access in the memory. The processor-cache interface can be characterized by a number of parameters. These parameters directly affect the microprocessor performance and include:

1. Access time for a reference found in the cache (a hit): This is actually the property of the cache size and organization.
2. Access time for a reference not found in the cache (a miss): This is a property governed by the memory organization.
3. Time to initially compute a real address given a virtual address (not-in-TLB-time): This is basically a property of the address translation facility, which actually is not a part of the cache but resembles the cache in most aspects.

A cache miss on an instruction fetch requires that the microprocessor should wait until the instruction becomes available from main memory. A cache miss on a data read may be less serious; instructions can, however, continue execution until the data to be fetched is actually required. In practice, however, the data is used almost immediately after it is fetched. A cache miss on a data word write may be even less serious. If the write is buffered, the processor can continue until the write buffer is full. If we know the miss rate for reads in a cache memory, we can calculate the number of read-stall cycles with the following simple formula:

Read-Stall Cycles = Reads × Read Miss Rate × Read Miss Penalty

The formula for writes is also similar, except that the effect of the write buffer must be added in the following way:

Write-Stall Cycle = (Writes × Write Miss Rate × Write Miss Penalty) + Write Buffer Stalls

In many cache memory systems, the penalties are the same for a cache read or write. Here, we can use a single miss rate, and miss penalty:

Memory-Stall Cycles = Memory Accesses × Cache Miss Rate × Cache Miss Penalty

To illustrate this with the help of an example, let us assume a cache with 'miss rate' of 5 per cent, (a 'hit rate' of 95 per cent) with cache memory of 20ns cycle time, and main memory of 150ns cycle time. We can calculate the average cycle time as:

$(1 - 0.05) \times 20\text{ns} + 0.05 \times 150\text{ns} = 26.5\text{ns}$

Table 3.6 records the effective memory cycle time as a function of cache hit rate for the above case (for the case hit ranging 80–100 %):

Table 3.6 *Effective Memory Cycle Time*

Cache Hit (%)	Effective Cycle Time (ns)
80	46
85	39.5
90	33
95	26.5
100	20

In conclusion, we can say that:

(i) Hit: Data appears in some block in the upper level (example: Block X)

(a) Hit Rate: The fraction of memory access found in the upper level

(b) Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss

(ii) Miss: Data needs to be retrieved from a block in the lower level (Block Y)

(a) Miss Rate = 1 - (Hit Rate)

(b) Miss Penalty: Time to replace a block in the upper level + Time to deliver the block to the processor

- Hit Time << Miss Penalty

The addition of an on-chip cache, secondary cache (also referred to as a level2 cache or L2 cache) is a very popular method of improving the performance of computer systems when a high amount of memory is used by the microprocessor. However, the secondary cache assumes the presence of a level1 or primary cache, closely coupled or internal to the CPU. Memory access is fastest to L1 cache, followed closely by the L2 cache controller. Memory access is significantly slower with L3 memory, main memory. Table 3.7 shows typical sizes and access times for different types of memory.

Table 3.7 *Sizes and Access Time for Different Types of Memory*

NOTES

Memory type	Size	Access time
Processor registers	128 Bytes	1 cycle
On-chip L1 cache	32KB	1-2 cycles
On-chip L210 Cache Controller	128KB	8 cycles
Main memory (L3) dynamic RAM	MB GB[a]	16 cycles
Back-up memory (hard disk) (L4)	MB GB	> 500 cycles

Cache controllers are being commonly used in microcomputer systems to effectively reduce the load on the microprocessor. L210 cache controller is the most commonly used cache controller with a number of processor sets. The block diagram of L210 cache controller is shown in Figure 3.45.

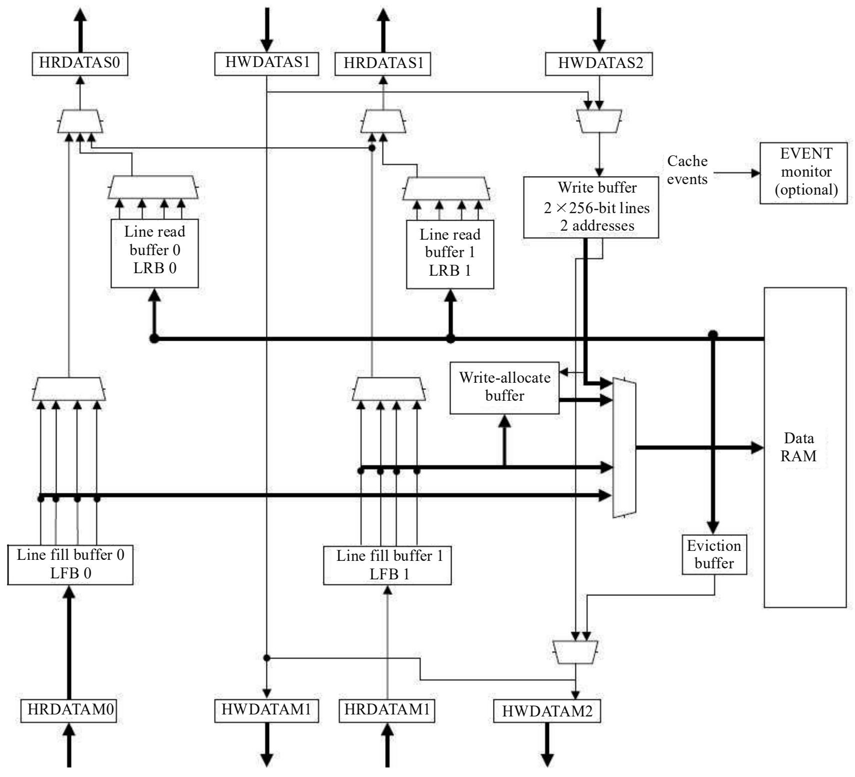


Fig. 3.45 *Block Diagram of L210 Cache Controller*

The features of this particular cache controller are:

- Cache of the size 16KB–2MB can be controlled.
- Fixed-line length of eight word or 32 bytes is used.

- Operating frequency range is 300 MHz in the worst case with 0.18 micron technology.
- This is a physically addressed and physically tagged device.
- Lockdown format C is supported, with separate way-locking mechanisms for data and instructions.
- Eight-way associativity is available; this can be direct mapped, depending on the use of lockdown registers.
- Data RAM is byte-writable.
- Different cache modes that are supported include:
  - (i) Write-through, read allocate
  - (ii) Write-back, read allocate
  - (iii) Write-back, read and write allocate, for systems using ARMv6 extensions only
- Provides a write-allocate override option to always have allocation on write misses in the cache controller.
- Performs critical word first refilling, with the option of refilling starting with the word 0. The same option is used to transform non-bufferable wrap write bursts and non-cacheable wrap read bursts into linear accesses.
- Pseudo-random victim selection policy can be made deterministic with the use of lockdown registers.
- You can statically configure the following options:
  - (i) Single master port, master port 1
  - (ii) Two master ports, master port 0 and master port 1
- The cache controller has the following buffers:
  - (i) **Two Line-Fill Buffers (LFB):** These buffers capture line-fill data from main memory, waiting for a complete line before writing to level2 cache. This buffer makes the cache controller non-blocking for requests from other slave ports.
  - (ii) **Two Line-Read Buffers (LRB):** These buffers hold a line from the cache controller, for subsequent requests that hit on the line.
  - (iii) **One Eviction Buffer (EB):** This buffer holds an evicted line from the cache controller, which can be written back to main memory.
  - (iv) **One Write Buffer (WB):** This particular buffer is used to hold the buffered writes before their being drained to the main memory as well as to the cache controller. The write buffer is made of two slots, each with a 256-bit data line and one address per slot. It enables multiple writes to the same line to be merged.
  - (v) **One Write-Allocate Buffer (WA):** If the write buffer line is not full, this buffer merges data from the write buffer and missing data from master port 1 before requesting an allocation to the cache.

Ports of the Cache Controller

A cache controller can be configured to use one, two or three ports. The cache controller ports are:

1. 64-bit AHB-Lite slave ports:
   (i) One slave (S1)
   (ii) Two slaves (S0 and S1)
   (iii) Three slaves (S0, S1 and S2)

   The cache controller arbitrates internally among them.
2. 64-bit AHB-Lite master ports:
   (i) One master (M1)
   (ii) Two masters (M0 and M1)
   (iii) Three masters (M0, M1 and M)

Check Your Progress

8. Define the term cache hits and cache miss.
9. What do you understand by line size of the cache?
10. Define the term cache controllers.
11. What do you understand by maximum memory?

3.6 MEMORY INTERFACING

Maximum memory that can interface with 8086 microprocessor is 1 megabyte. 1 megabyte memory divides into four segments of continuous memory size of 64 kilobyte and segment can define in anywhere in 1 MB memory. Bottom address of each segment define in such away that last four bit should be zero.

Normally, the segment base register contains four zeroes, so that each segment can start from say E0000 to EFFFF.

The segments namely, Code Segment (CS), Data Segment (DS), Stack Segment (SS) and Extra Segment (ES) for a particular program can be contiguous, separate or in case of small programs, even overlapping. For example, the code segment is supposed to have 64kb and in case of small programs data segment may be within the code segment. The memory segmentation for CS, DS, SS and ES have been shown in Figures 3.46, 3.47, 3.48, and 3.49 respectively.

NOTES

Now let us consider an example of the segment base and offset. We can consider the telephone numbers, for example, 23322651 is a telephone number out of which, 2 is a universal code, 332 is the area code, and 2651 is the offset in that area. In other words, the area telephone numbers can occupy 23320000 to 23329999.

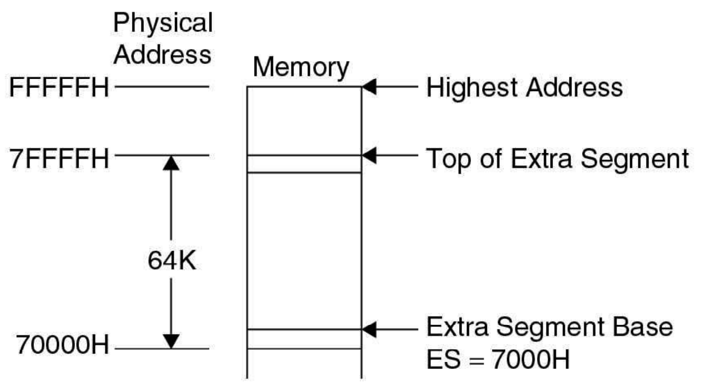


Fig. 3.46 *Memory Segmentation Range for ES*

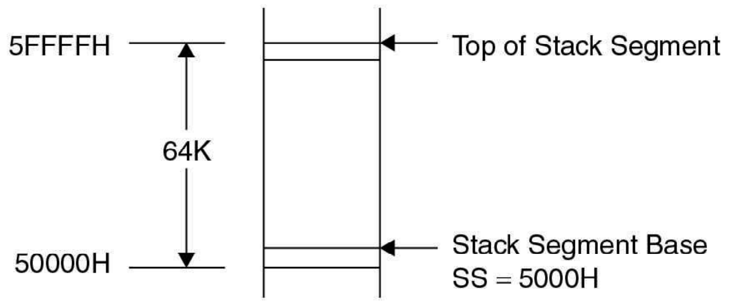


Fig. 3.47 *Memory Range for SS*

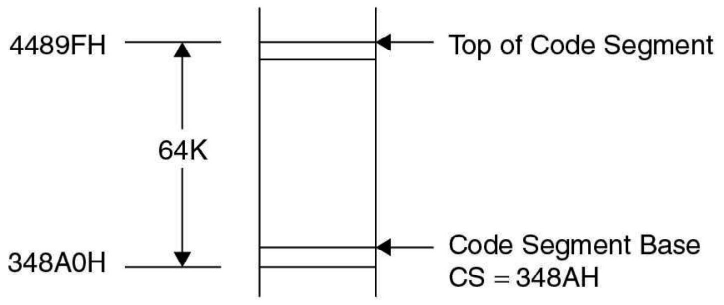


Fig. 3.48 *Memory Range for CS*

NOTES

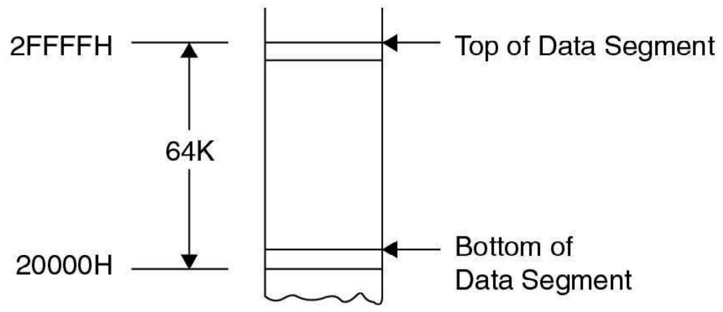


Fig. 3.49 *Memory Range for DS*

NOTE: One way of four 64-kbyte segment might be positioned within the 1-mbyte address space of 8086.

Each segment is used for the storage of specific type of information. These are as follows:

1. Code segment is used to store instruction.
2. Stack segment is used to store temporary information.
3. Data segment and extra segment are used to store data before and after execution.

In 8086 microprocessor, each segment a 16-bit register that hold 16 for bit of bottom address of particular segment.

$$\left.\begin{array}{l}\text{Code register} = \text{CS}\\ \text{Stack register} = \text{SS}\\ \text{Data register} = \text{DS}\\ \text{Extra register} = \text{ES}\end{array}\right\}\text{16-bit register}$$

Regarding each segment, another 16-bit register that hold the effective address of memory locations in range of 0000H to FFFFH.

$$\left.\begin{array}{ll}\text{CS Code segment} = & \text{Instruction pointer}\\ & \text{(IP)}\\ \text{SS Stack segment} = & \text{Stack pointer (SP)}\\ \text{DS Data segment} = & \text{Source index (SI)}\\ \text{ES Extra segment} = & \text{Destination index}\\ & \text{(DI)}\end{array}\right\}\begin{array}{l}\text{16-bit}\\ \text{register}\end{array}$$

Physical Address/Actual Memory Address/Memory Address/Effective Memory Address

$$\underset{\text{Base address}}{\underset{\uparrow}{\text{CS}}} \quad : \quad \underset{\text{Offset address}}{\underset{\uparrow}{\text{IP}}}$$

IP in 8086 stores effective address of next instruction fetched from the memory, and it is equivalent to PC of 8085. Refer Figure 3.50.

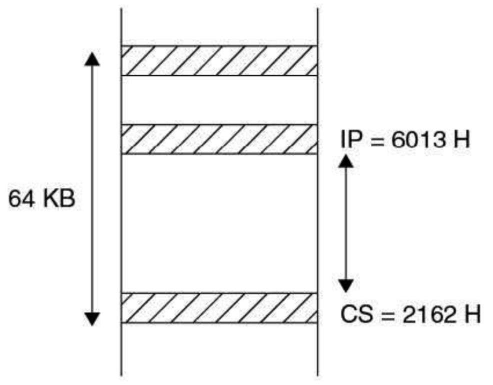


Fig. 3.50 *Physical Address Range*

$$\begin{array}{rccl} & & \overset{\text{Hardwared zero}}{} & \\ CS = & 2\,1\,6\,2 & 0 & H \\ \underline{IP =} & \underline{6\,0\,1} & \underline{3} & \\ & 2\,7\,6\,3 & 3 & H = \text{Physical address} \end{array}$$

Memory Address Calculation or Generation of 20-Bit Physical Address

The 8086 microprocessor has 20-bit address pins.

These are capable of addressing 2^{20} = 1 mega byte memory. To generate this 20-bit physical address from two 16-bit registers, the following procedure is adopted.

The first 16-bit register is called the segment **base register.** These are code segment registers to hold programs, data segment register to keep data, stack segment register for stack operations and extra segment register to keep strings of data. The contents of the segment registers are shifted left four times with zeros (0's) filling on the right hand side. This is similar to multiplying four hex numbers by the base 16. This multiplication process takes place in the adder and thus a 20 bit number is generated. This is called the base address. To this a 16-bit offset is added to generate the 20-bit physical address.

Segment addresses must be stored in segment registers. The offset is derived from the combination of pointer registers, the Instruction Pointer (IP), and immediate values. Refer Figure 3.51.

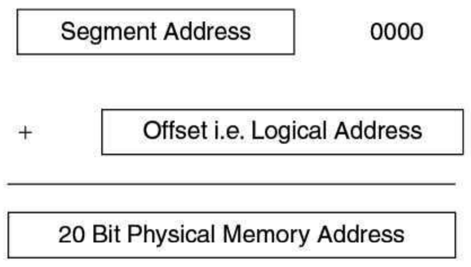


Fig. 3.51 *Physical Address Calculation*

Case 1: One of the technique used to obtain different address is the instruction address = CS + IP. Refer Figures 3.52 and 3.53.

NOTES

CS	3	4	8	A	0
IP +		4	2	1	4
Instruction Address	3	8	A	B	4

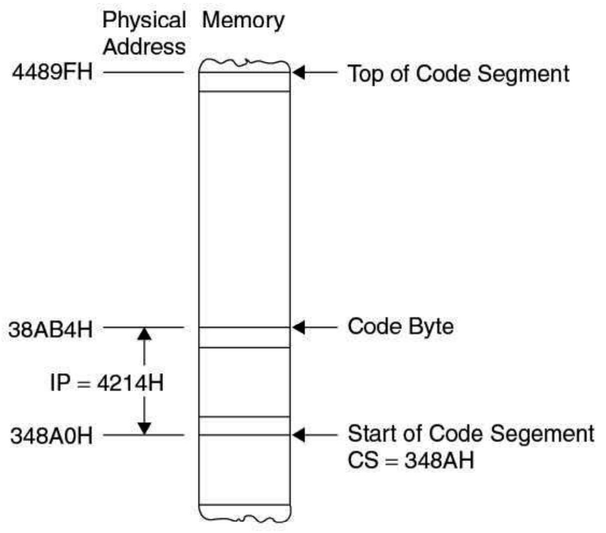


Fig. 3.52 Presentation of CS Memory and its Physical Address

CS	3	4	8	A	0	◀── Hardwired Zero
IP +		4	2	1	4	
Physical Address	3	8	A	B	4	

Fig. 3.53 Addition of IP to CS to Produce the Physical Address of the Code Byte

Case 2: Data Address = DS + DI. Refer Figure 3.54.

DS	1	2	3	4	0
DI +		0	0	2	2
Data Address	1	2	3	6	2

Fig. 3.54 Addition of DS to DI to Produce the Data Address of Data Segment Byte

Case 3: Stack Address = SS + SP. Refer Figure 3.55.

SS	5	0	0	0	0
SP +		F	F	E	0
Stack Address	5	F	F	E	2

Fig. 3.55 Addition of SS to SP to Produce the Stake Address of Stake Segment Byte

3.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Registers can be categorized into:
   - General-Purpose Registers
   - Pointer or Index Registers
   - Segment Registers
   - Other Registers
2. The 8086 microprocessor has several other registers, which do not exist on the 8-bit chips. These are called as the segment registers.
3. The Interrupt-enable flag (I) controls the interrupt request pin on the microprocessor chip.
4. According to the little endian convention, a higher byte of the address is sent to the higher address, whereas the lower byte of the address is sent to the lower address.
5. This signal is very useful for the memory and I/O read/ write operation. READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer.
6. The various ways of specifying data are called addressing modes.
7. No, the microprocessor does not read the mnemonics.
8. Microprocessor references that are available in the cache are called cache hits, whereas those not available in the cache are called cache misses.
9. The information (instructions or data) stored with each tag in the cache is called the line size of the cache.
10. Cache controllers are being commonly used in microcomputer systems to effectively reduce the load on the microprocessor.
11. Maximum memory that can interface with 8086 microprocessor is 1 megabyte. 1 megabyte memory divides into four segments of continuous memory size of 64 kilobyte and segment can define in anywhere in 1 MB memory. Bottom address of each segment define in such away that last four bit should be zero.

3.8 SUMMARY

- A computer is a device that can function only under a set of certain software instructions. Without software, microprocessors cannot be used for any practical purpose.
- The Bus Interface Unit (BIU) is responsible for the communication by the microprocessor. This unit basically controls the buses and handles all the data code, address etc. from the memory or other devices.

NOTES

- The Execution Unit (EU), on the other hand, is responsible for the actual execution of the program code. It performs all the operations, generates the status signals and passes the results to the bus interface unit for further processing through the I/O devices or the memory.
- The 8086 microprocessor has several index registers and pointers including the base pointer, source index and destination index. All are 16 bits wide. Unlike 8-bit chips, these are not used alone, but are used in combination with registers called segment registers.
- The 8086 microprocessor has several other registers, which do not exist on 8-bit chips. These are called as the segment registers.
- The stack segment is an important segment as all the data related to the stack is held by this segment only. Certainly, it has great importance in modular programming and the programs in which a lot of jumping and conditions are involved.
- The status register is the register containing the 8086/8088 flags. This register is 16 bits wide, although not all the 16 bits are used.
- The physical address of a memory is a 20-bit address for the 8086 microprocessor. According to this convention, a higher byte of the address is sent to the higher address, whereas the lower byte of the address is sent to the lower address.
- The pin configuration of 8086 is available in the 40-pin Dual Inline Package (DIP).
- READY signal is very useful for the memory and I/O read/ write operation. READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer.
- The microprocessor 8086 can be configured to work in two modes: minimum mode and maximum mode. The minimum mode is used for single-processor systems, where 8086 directly generates all the necessary control signals. The maximum mode is designed for multiprocessor systems, where an additional 'Bus Controller' IC is required to generate the control signals.
- Direct-addressing mode is similar to immediate addressing in that information is encoded directly into the instruction.
- Register indirect-addressing mode is similar to direct-addressing mode in that an effective address is combined with the contents of DS to obtain a physical address.
- 1-byte instructions perform three different tasks. In the first instruction, both the operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand.
- An effective memory system is the key to any successful computer system. The memory system is said to be effective if the access time of the cache is close to the effective access time of the processor.

- The cache is a small amount of high-speed memory used to work directly with the microprocessor. The memory cycle time of cache is usually in the order of the time required by the CPU to fetch an instruction.
- An associative memory, or content-addressable memory, works on a simple principle. It has the property that when a value is presented to the memory, the address of the value is returned if the value is available in the memory, otherwise an indication that the value is not in the associative memory is returned.
- Associative memory proves to be expensive when it comes to implementation, because a comparator is required for every word in the memory to perform all the comparisons in parallel.
- Cache controllers are being commonly used in microcomputer systems to effectively reduce the load on the microprocessor.

3.9 KEY TERMS

- **8086 Stack:** It is a standard memory stack with a capacity of 64K.
- **Interrupt-enable flag:** The Interrupt-enable flag (I) controls the interrupt request pin on the microprocessor chip.
- **INTR:** It implies the INTERRUPT REQUEST signal.
- **Addressing modes:** The various ways of specifying data.
- **Cache:** It refers to a small amount of high-speed memory that is used to work directly with the microprocessor.

3.10 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write the basic architecture of microprocessor 8086.
2. What is memory addressing?
3. Differentiate between the general-purpose and specific-purpose registers.
4. Define the term addressing modes of 8086.
5. Define the term 1-byte instructions.
6. What is the main purpose of cache memory?
7. Differentiate between the fully associative and set-associative cache.
8. Differentiate between the spatial locality and temporal locality.
9. What is maximum memory?

Long-Answer Questions

1. Briefly explain the architecture of 8086 and its types giving appropriate examples.

2. Describe memory addressing giving appropriate examples.
3. Discuss about the complete of the 8086 microprocessor with the help of diagram.
4. Briefly explain the instruction format with the help of examples.
5. Describe the cache memory with the help of diagram.
6. Discuss about the fully associative and direct associative with the help of diagram.
7. Briefly explain the block diagram of L210 cache memory.
8. Explain briefly about the maximum memory with the help of relevant examples.

3.11 FURTHER READING

Goankar, Ramesh. 2002. *Microprocessor Architecture, Programming and Application with the 8085*. Mumbai: Penram International Publishing.

Ram, B. 2005. *Fundamentals of Microprocessor and Microcomputers*. New Delhi: Dhanpat Rai Publications.

Lotia, Manahar and Pradeep Nair. 2003. *Modern All About Motherboard*. New Delhi: BPB Publications.

Mueller, Scott, and Craig Zacker. 2002. *Upgrading and Repairing PCs*. New Jersey: Pearson Education (Que Publishing).

Godse, D.A. and A.P. Godse. 2007. *Microprocessor and Assembly Language*

Programming. Pune: Technical Publications.

Kleitz, William. 2009. *Digital and Microprocessor Fundamentals: Theory and*

Applications. New Jersey: Prentice Hall.

Ray, A.K. and K.M. Bhurchandi. 2000. *Advanced Microprocessors and Peripherals*. New Delhi: Tata McGraw-Hill.

UNIT 4 BASIC I/O INTERFACE

Structure

4.0 INTRODUCTION

A standalone microprocessor is of no use to the world, unless it can communicate with the outside world. I/O interface provides a mechanism to establish a link between the microprocessor and the Input/Output devices like printer, keyboard, mouse, process control devices, etc. Connecting input and output devices with the microprocessor is an important and tedious task. These connections require specific handshaking, timing and controlling requirements. As a result, a number of controlling devices are required to be attached to the microprocessor.

8255 is a commonly used programmable interface and is particularly used to provide handshaking, whereas 8251 is a programmable communication interface. In order to generate timing signals, the 8254 programmable timer is used. Interrupts also find an important place in creating a successful communication interface. Different microprocessors have different hardware and software interrupts. However, 8259 is a very widely used interrupt controller with a number of microprocessors.

In this unit, you will study about the I/O interface, 8255 programmable interface, 8254 programmable timer, 8251 programmable/communication interface, interrupts, and 8259 programmable interrupts controller.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basics of I/O interface
- Explain the 8255 programmable interface
- Discuss about the 8254 programmable timer
- Define the 8251 programmable/communication interface

- Analyse the interrupts
- Elaborate the 8259 programmable interrupts controller

NOTES

4.2 I/O INTERFACE

A standalone microprocessor is of no use to the world, unless it can communicate with the outside world. I/O interface provides a mechanism to establish a link between the microprocessor and the Input/Output devices like printer, keyboard, mouse, process control devices, etc. The I/O devices can be accessed by the microprocessor using two methods:

1. Memory Mapped I/O
2. I/O Mapped I/O

(1) Memory Mapped I/O

In memory mapped I/O, the processor does not differentiate between I/O and memory.

The processor uses a portion of the memory address to represent I/O ports. The I/O ports are mapped into the processor's main memory and are hence called memory mapped I/O. Here, we can use all the instructions that are used to work with memory. The address bus width is the same as that of microprocessor and hence decoding logic becomes a bit complicated. However, a large number of I/O devices can be addressed using the Memory Mapped I/O.

(2) I/O Mapped I/O or Standard I/O

The standard I/O is also called isolated I/O. It uses $IO/\overline{M}$ control pin on the microprocessor. Processor outputs are high on this pin to indicate an I/O operation taking place. A low on this pin indicates a memory operation. I/O devices are not mapped into the memory. Instead, they are directly and specifically treated as I/O devices. Here, we can use only IN and OUT instructions.

4.3 8255 PROGRAMMABLE INTERFACE

The microprocessor is a fast device and the Input/Output devices are slow in comparison to the microprocessor. Thus, in order to communicate, the microprocessor needs to know which is the right time to send the next data. The process of knowing the status of devices and transferring the data with matching speeds is called handshaking. The parallel port devices, such as 8255 have been designed to automatically manage the handshake operation. The 8255A can be programmed to automatically receive the STB signal from a peripheral, send an interrupt signal to the processor, and send ACK signal to the peripheral at proper times. The block diagram of the 8255 is shown in Figure 4.2. The device has 24 input/output lines and these can be used with two 8-bit ports (A and B) and two 4-bit ports C_L and C_U. These ports are divided into two control groups A and B. It is clear from Figure 4.1 that 8 -data lines are used to write data bytes to a port or control registers and to read bytes from the port or a status register under the control of RD and WR lines. The address lines A0 and A1 (Refer Figure 4.1) are used to access one of the ports A, B and C.

A0	A1	Port
0	0	A
0	1	B
1	0	C
1	1	Control

Fig. 4.1 *Division of Ports into Control Groups*

The Chip Select Signal is used to select the chip similar to other chips in the circuitary. The RESET pin of the device is connected to system RESET, so that all the ports are RESET, when the system is RESET.

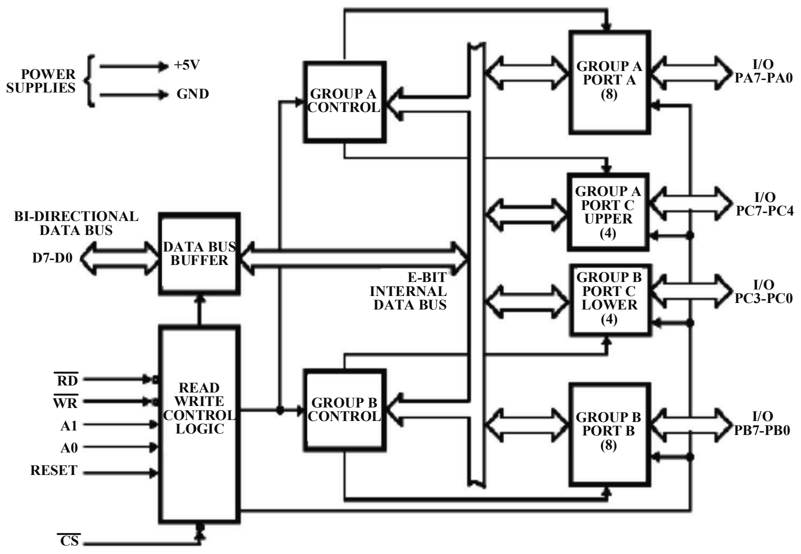


Fig. 4.2 *Block Diagram of 8255*

8255 can be used in three different modes of the I/O as per requirement. These have been shown in Figure 4.3.

Mode 0: Mode 0 is used for simple input or output without handshaking. Both ports A and B can be initialized in this mode, also the two halves of port C can be used separately or combined as port C in this mode. When used individually, the two halves of port C can be used as input and output ports independently.

Mode 1: Mode 1 is used for single handshake (Strobed). Both ports A and B can be initialized in this mode. However, the pins of port C are used to generate the control signals (BSR mode). PC0-PC2 are used for port B, whereas PC3-PC5 are used for port A. The remaining pins PC6-PC7 can be used as I/O lines.

Mode 2: Mode 2 is used for double handshake (bi-directional). Only port A can be initialized in this mode. However, the pins of port C are used to generate the control signals. PC3-PC7 are used as handshake lines (BSR mode) for port A. The remaining pins PC0-PC2 can be used as I/O lines. The generation of the control words for both the I/O mode as well as the BSR mode is shown in Figures 4.4 and 4.5.

NOTES

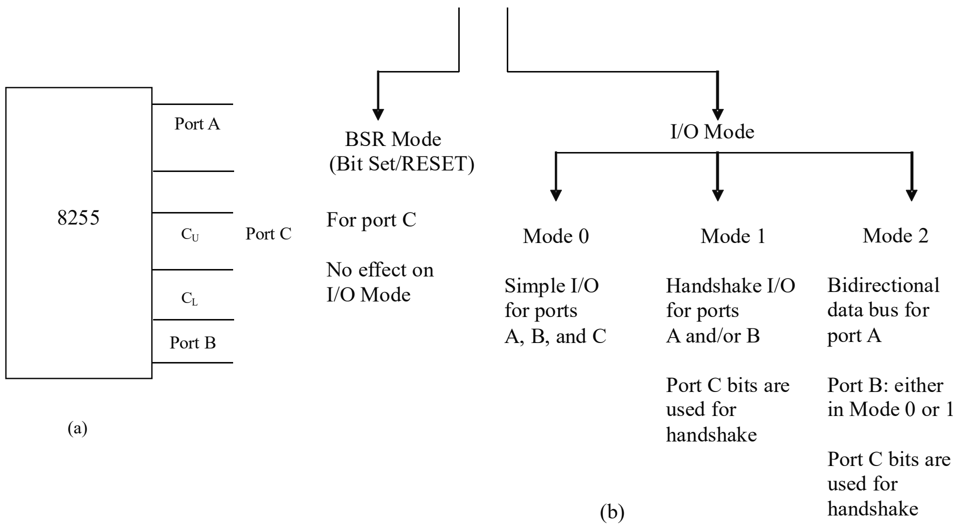

8255 Modes
8255
Port A
Port C
C_U
C_L
Port B
(a)
BSR Mode
(Bit Set/RESET)
For port C
No effect on I/O Mode
I/O Mode
Mode 0
Simple I/O for ports A, B, and C
Mode 1
Handshake I/O for ports A and/or B
Port C bits are used for handshake
Mode 2
Bidirectional data bus for port A
Port B: either in Mode 0 or 1
Port C bits are used for handshake
(b)

Fig. 4.3*(a) and (b) I/O Ports and their Modes*

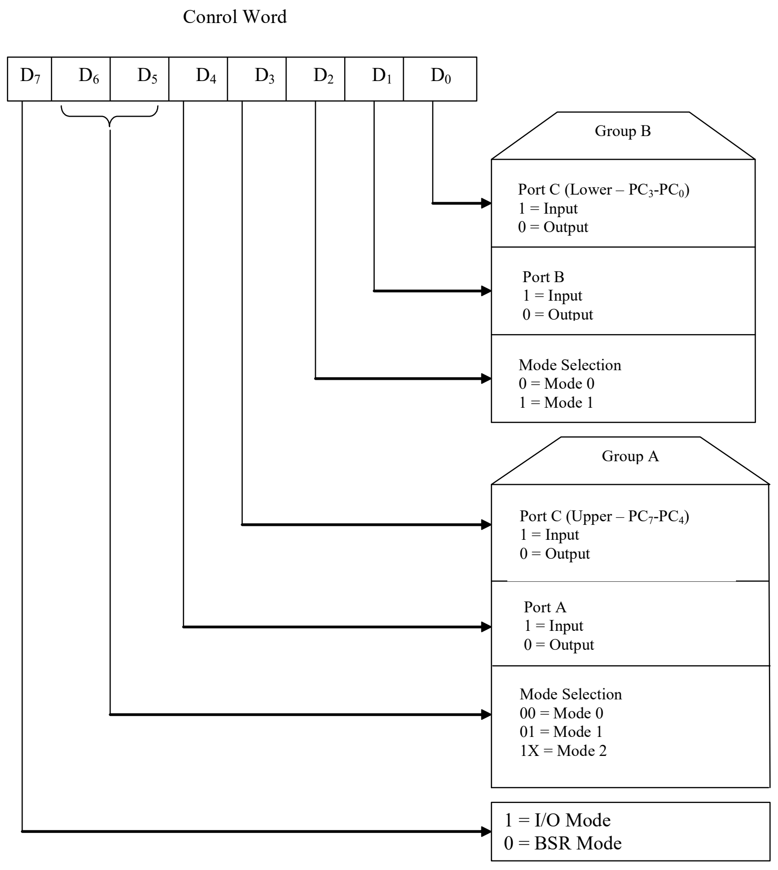


Fig. 4.4 *8255 A Control Work Format for I/O Mode*

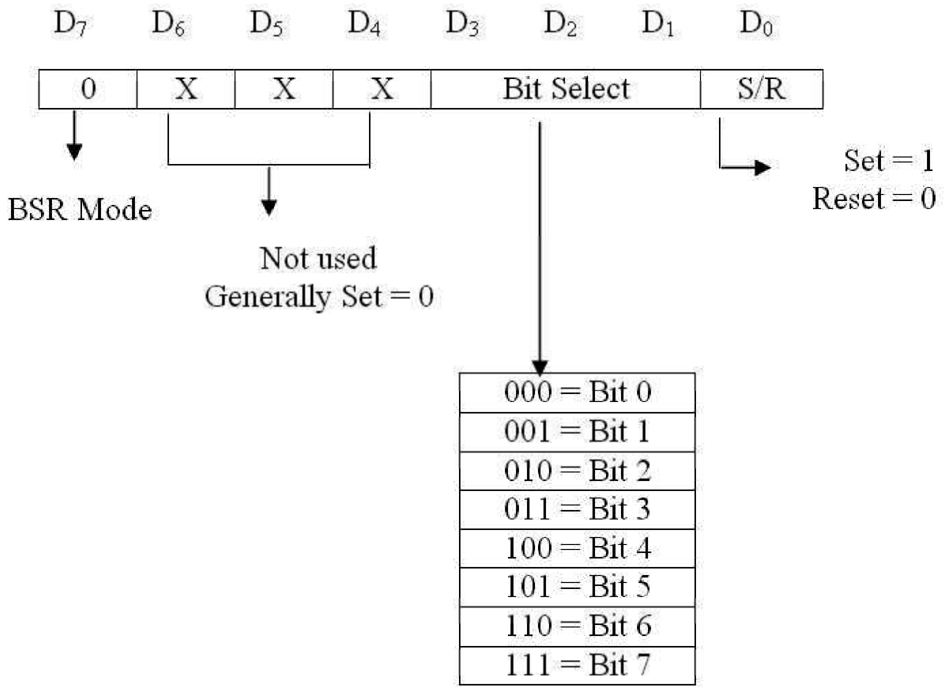


Fig. 4.5 *8255 A Control Work Format for BSR Mode*

Example 4.1: Find the control word if PA = out, PB = in, PC0 – PC3 = in and PC4- PC7 = out.

The Control word = 1000 00112 = 83H

Input/Output Using Microprocessor 8086

The **In** and **Out** instructions can be used for data transfer by the microprocessor through 8255. Both the direct and indirect addressing can be used as shown in Table 4.1.

Table 4.1 *Direct and Indirect Addressing*

Mnemonic	Meaning	Format	Operation

IN	Input direct
	Input indirect (variable)
		IN ACC, Port
		IN ACC, DX
			ACC ← [Port] ; ACC = AL or AX
			ACC ← [DX]

OUT	Output direct Output indirect (variable)	OUT Port, ACC OUT DX, ACC	[Port] ← ACC [DX] ← ACC

In case of direct addresing the port address is specified directly and cannot be larger than FFH.

Ex. IN AL, 99H ; bring a byte into AL from port 99H

OUT 34H, AX ; send out a word from AX to port addresses 34H -35H

In case of Register indirect addresing the port address is kept by the DX register. Therefore, it can be as high as FFFFH.

Ex. MOV DX, 64B1H DX = 64B1H

```
OUT DX, AX          send out a word from AX to port address
                    pointed to DX.
                    The byte from AL goes to port 64B1H and
                    byte from AH goes to port 64B2H
```

Example 4.3: Program the 8255 to get data from port B and send it to port A. In addition, data from PCL is sent out to the PCU. Use port addresses of 300H-3003H for the 8255 chip.

```
B8255C   EQU 300H        Base address of 8255 chip
CNTL     EQU 83H         PA = out, PB = in, PCL = in, PCU = out
MOV      DX, B8255C+3    Load control reg. address (300H+3 =303H)
MOV      AL, CNTL        Load control byte
OUT      DX,AL           Send it to control register
MOV      DX, B8255C+1    Load PB address
IN       AL, DX          Get the data from PB
MOV      DX, B8255C      Load PA address
OUT      DX,AL           Send it to PA
MOV      DX, B8255C+2    Load PC address
IN       AL, DX          Get the bits from PCL
AND      AL, 0FH         Mask the upper bits
ROL      AL,1            Shift the bits
ROL      AL,1            To upper position
ROL      AL,1
ROL      AL,1
OUT      DX,AL            Send it to PCU
```

Interfacing Analogue to Digital Converters Using 8255 PPI

A/D and D/A converters are very interfaced with the microprocessors using 8255 programmable interface. Let us consider the example of interfacing analogue to digital converters with the microprocessor 8086 using 8255. In most of the cases, the 8255 is used for interfacing the analogue to digital converters with microprocessor. We have already studied 8255 interfacing with 8086 as an I/O port, in the previous section. In this section, we will only emphasize on the interfacing techniques of analogue to digital converters with 8255. The analogue to digital converters are treaded as an input device by the microprocessor, that sends an initializing signal to the ADC to start the analogy of digital data conversation process. The start of conversation signal is a pulse of a specific duration. The process of analogue to digital conversion is a slow process, and the microprocessor has to wait for the digital data till the conversion is over. After the conversion is over, the ADC sends End of Conversion (EOC) signal to inform the microprocessor that the conversion is over and the result is ready at the output buffer of the ADC.

These tasks of issuing an SOC pulse to ADC, reading EOC signal from the ADC and reading the digital output of the ADC are carried out by the CPU using 8255 I/O ports. The time taken by the ADC from the active edge of SOC pulse till the active edge of EOC signal is called the conversion delay of the ADC. It may range anywhere from a few microseconds in case of fast ADCs to even a few hundred milliseconds in case of slow ADCs. The available ADC in the market uses different conversion techniques for conversion of analogue signal to digital.

Successive approximation techniques and dual slope integration techniques are the most popular techniques used in the integrated ADC chip. The general algorithm for ADC interfacing contains the following steps:

1. Ensuring the stability of analogue input applied to the ADC.
2. Issuing start of conversion pulse to ADC.
3. Marking the end of the conversion processes by the.
4. Read digital data output of the ADC as equivalent digital output.

Analogue input voltage must be constant at the input of the ADC right from the start of conversion till the end of the conversion to get correct results. This may be ensured by a sample and hold circuit which samples the analogue signal and holds it constant for a specific time duration. The microprocessor may issue a hold signal to the sample and hold circuit. If the applied input changes before the complete conversion process is over, the digital equivalent of the analogue input calculated by the ADC may not be correct.

ADC 0808/0809

The analogue to digital converter chips 0808 and 0809 are 8-bit CMOS, successive approximation converters. This technique is one of the fastest techniques for analogue to digital conversion. The conversion delay is 100µs at a clock frequency of 640 KHz, which is quite low as compared to other converters. These converters do not need any external zero or full-scale adjustments as they are already taken care of by internal circuits. These converters internally have a 3:8 analogue multiplexer so that at a time eight different analogue conversions by using address lines - ADD A, ADD B, ADD C, can be used. Using these address inputs, multichannel data acquisition system can be designed using a single ADC. The CPU may drive these lines using output port lines in case of multichannel applications. In case of single input applications, these may be hardwired to select the proper input. There are unipolar analogue to digital converters, i.e., they are able to convert only positive analogue input voltage to their digital equivalent. These chips do not contain any internal sample and hold circuit.

NOTES

Analogue I/P selected	**Address lines**		
	C	B	A
I / P_0	0	0	0
I / P_1	0	0	1
I / P_2	0	1	0
I / P_3	0	1	1
I / P_4	1	0	0
I / P_5	1	0	1
I / P_6	1	1	0
I / P_7	1	1	1

If you need a sample and hold circuit for the conversion of fast signal into equivalent digital quantities, it has to be externally connected at each of the analogue inputs. The different signals to be used have been shown in Table 4.2.

Table 4.2 *Different Signals*

Vcc	Supply pins +5V
GND	GND
Vref +	Reference voltage positive +5 Volts maximum
Vref _	Reference voltage negative 0Volts minimum
I/P0 –I/P7	Analogue inputs
ADD A, B, C	Address lines for selecting analogue inputs
O7 –O0	Digital 8-bit output with O7 MSB and O0 LSB
SOC	Start of conversion signal pin
EOC	End of conversion signal pin
OE	Output latch enable pin, if high enables output
CLK	Clock input for ADC

Figure 4.6 shows ADC 0808 / 0809. It shows both the Input as well as Output sections of the ADC. The timing diagram of the 0808 has also been shown—it shows the clock end of conversion, start of conversion and enable

signals. The happenings with each clock tick can be well understood with Figure 4.6.

NOTES

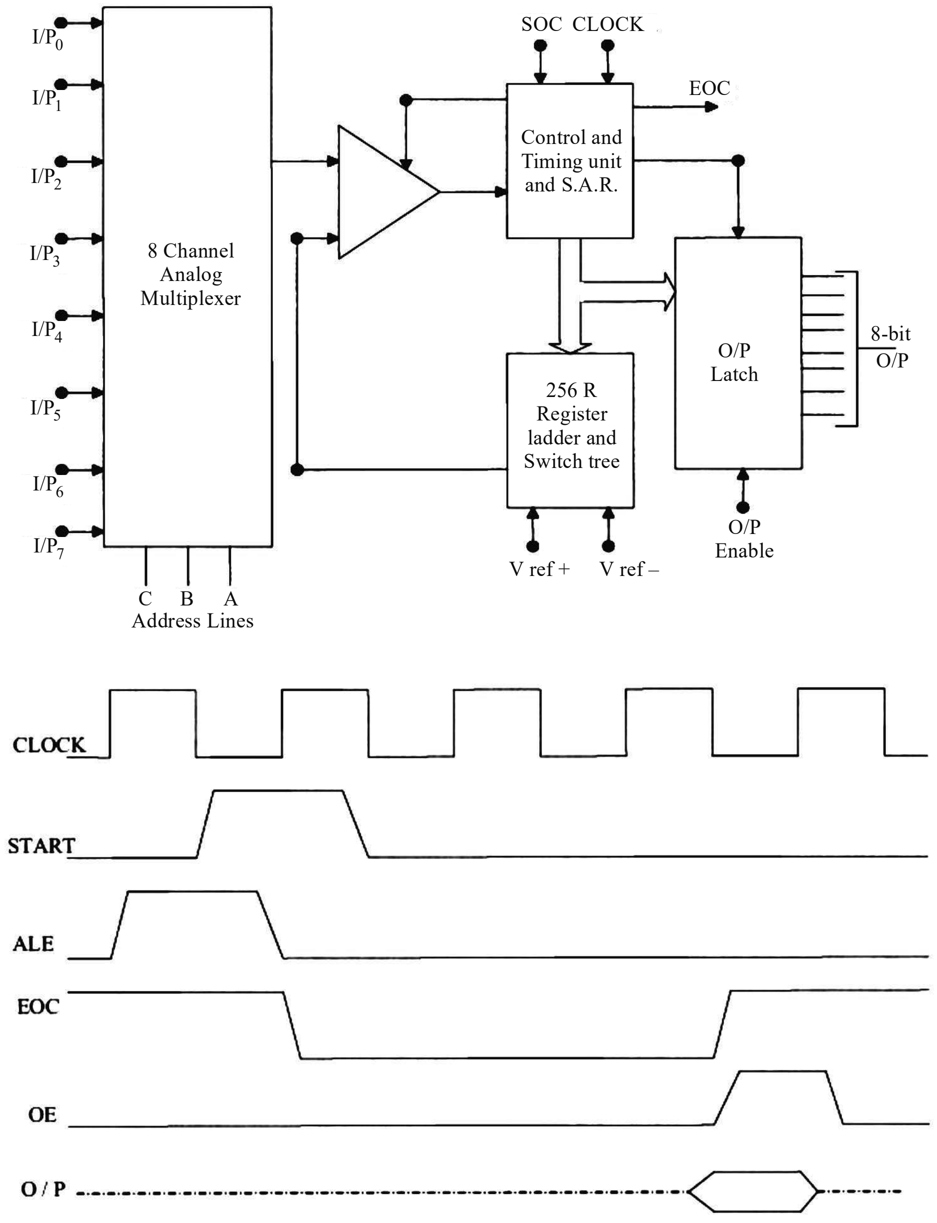


Fig. 4.6 Block Diagram of ADC 0808/0809

Let us consider the following interfacing example:

Example 4.3: Interfacing ADC 0808 with 8086 using 8255 ports. Use port A of 8255 for transferring digital data output of ADC to the CPU and port C for control signals. Assume that an analogue input is present at I/P2 of the ADC and a clock input of suitable frequency is available for ADC.

Solution: The analogue input I/P2 is used and therefore address pins A, B and C should be 0, 1 and 0 respectively to select I/P2. The OE and ALE pins are already kept at +5V to select the ADC and enable the outputs. Port C upper acts as the input port to receive the EOC signal while port C lower acts as the output port to send SOC to the ADC. Port A acts as an 8-bit input data port to receive the digital data output from the ADC. The 8255 control word is written as follows:

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

1 0 0 1 1 0 0 0

The required Assembly Language Program is shown below:

```
MOV AL, 98H               Initialize 8255 as discussed above
OUT CWR, AL
MOV AL, 02hH              Select I/P2 as analogue input
OUT Port B, AL
MOV AL, 00hH              Give start of conversion
OUT Port C, AL             pulse to the ADC
MOV AL, 01H
OUT Port C, AL
MOV AL, 00H
OUT Port C, AL
WAIT: IN AL, Port C       Check for EOC by
RCR                       Reading port C upper and
JNC WAIT                  Rotating through carry.
IN AL, Port A             If EOC, read digital equivalent in AL
HLT                       Stop
```

The final interfacing circuit with the 8255 PPI is shown in Figure 4.7.

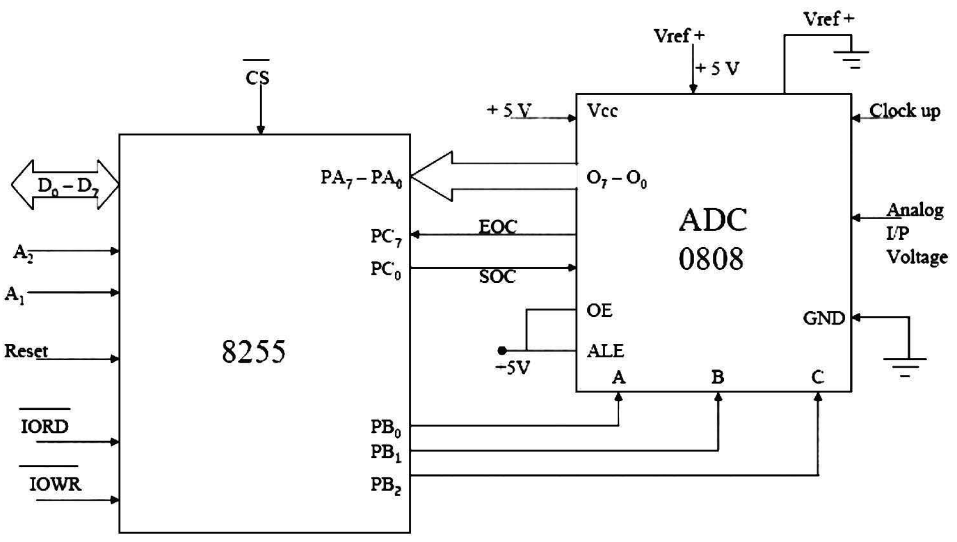
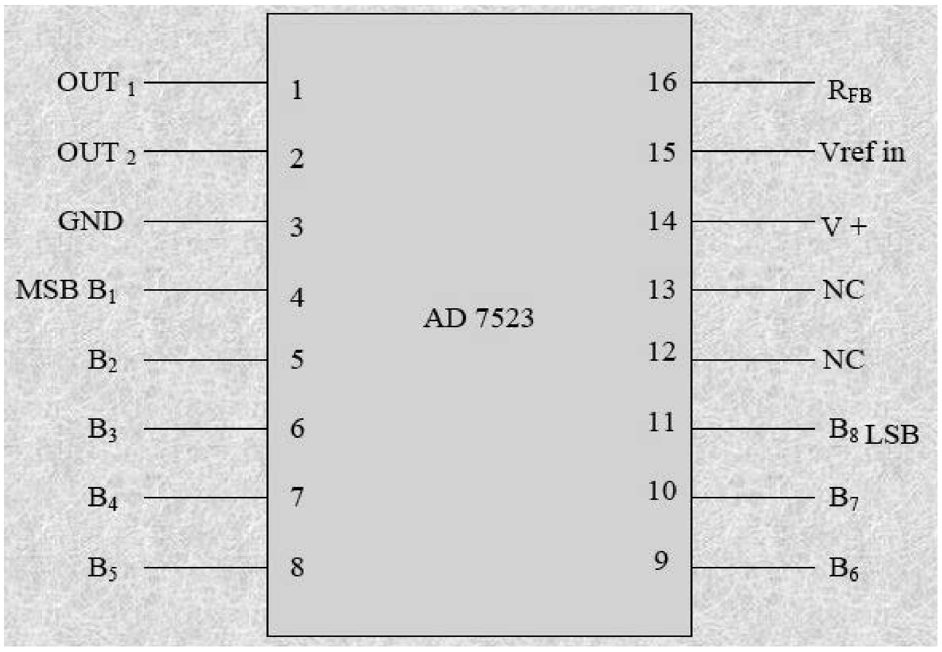


Fig. 4.7 *Final Interfacing Circuit with 8255 PPI*

Interfacing Digital to Analogue Converters Using 8255 PPI

The digital to analogue converters convert binary number into their equivalent voltages. The DAC find applications in areas like digitally controlled gains, motors speed controls, programmable gain amplifiers etc. AD 7523 8-bit multiplying DAC is a 16 pin DIP, multiplying digital to analogue converter, containing R-2R ladder for D-A conversion along with single pole double thrown NMOS switches to connect the digital inputs to the ladder. This is a very commonly used D/A converter with the microprocessors. The pin diagram of AD7523 is shown in Figure 4.8 along with actual working configuration. The supply range is from +5V to +15V, while Vref may be any where between -10V to +10V. The maximum analogue output voltage will be any where between -10V to +10V, when all the digital inputs are at logic high state. Usually a zener is connected between OUT1 and OUT2 to save the DAC from negative transients. An operational amplifier is used as a current to voltage converter at the output of AD to convert the current out put of AD to a proportional output voltage. It also offers additional drive capability to the DAC output. An external feedback resistor acts to control the gain. One may not connect any external feedback resistor, if no gain control is required.



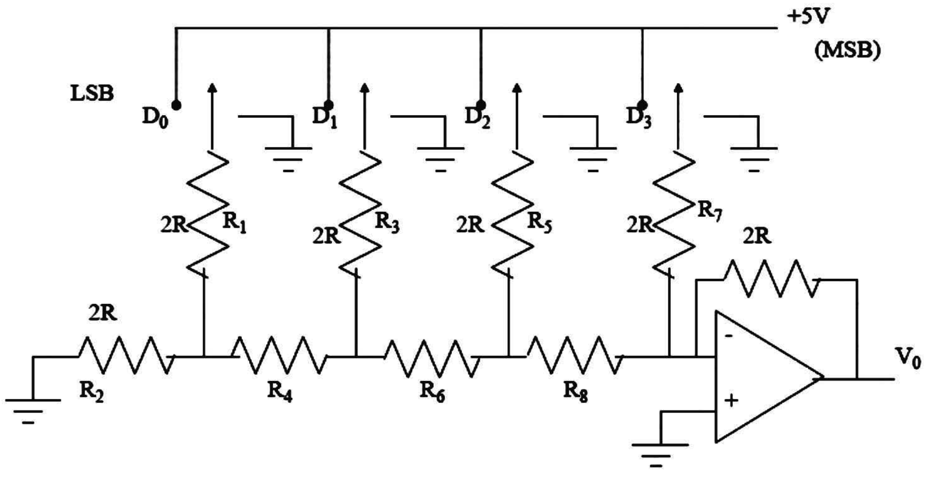


Fig. 4.8 *Pin Diagram of AD7523*

Example 4.4: Interfacing DAC AD7523 with an 8086 CPU running at 8MHZ and write an assembly language program to generate a sawtooth waveform of period 1ms with Vmax = 5V.

Solution: Figure 4.9 shows the interfacing circuit of AD 74523 with 8086 using 8255. Assembly Language Program is also given below, which generates a sawtooth waveform using circuit.

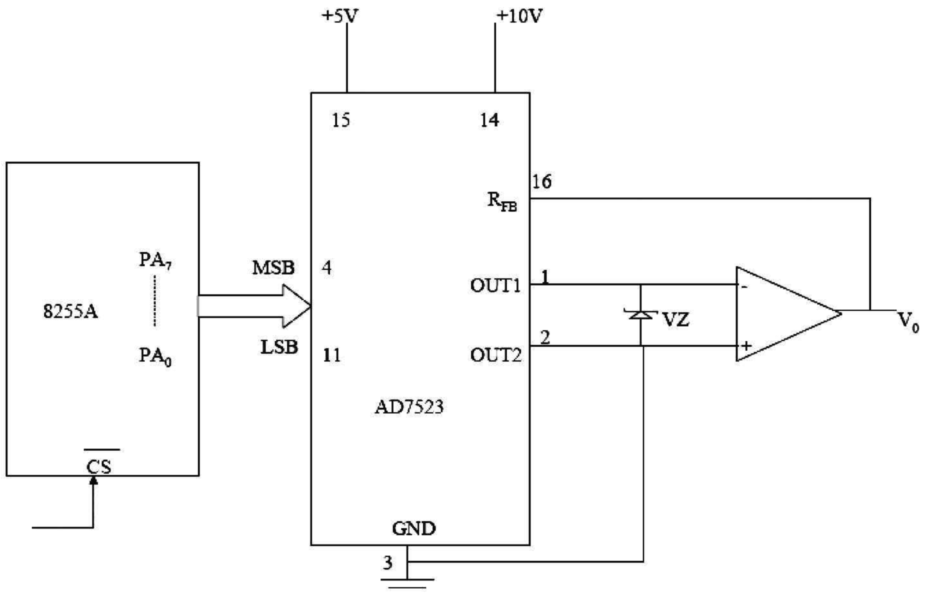


Fig. 4.9 Interfacing Circuit of AD 74523

The Assembly Language Program

```
ASSUME        CS:CODE
CODE          SEGMENT
START:        MOV AL, 80H      Make all ports output
              OUT CW, AL
AGAIN:        MOV AL, 00H      Start voltage for ramp
BACK :        OUT PA, AL
              INC AL
              CMP AL, 0FFH
              JB BACK
              JMP AGAIN
              CODE ENDS
END                            START
```

In the above program, port A is initialized as the output port for sending the digital data as input to DAC. The ramp starts from the 0V (analogue), hence AL starts with 00H. To increment the ramp, the content of AL is increased during each execution of loop till it reaches F2H. After that the saw tooth wave again starts from 00H, i.e. 0V(analogue) and the procedure is repeated. The ramp period given by this program is precisely 1.000625 ms. Here the count F2H has been calculated by dividing the required delay of 1ms by the time required for the execution of the loop once. The ramp slope can be controlled by calling a controllable delay after the OUT instruction.

4.4 8254 PROGRAMMABLE TIMER

NOTES

The 8254 programmable Timer is functionally similar to the software designed timers ad counters. It is used to generate accurate time delays and can be used for other timing applications, such as a real time clock, an event counter, a digital one shot, a square wave generator and a complex waveform generator. The Intel 8253/ 8254 contains three 16-bit counters and each of them can be programmed to operate in several different modes. The block diagram of 8254 is shown in Figure 4.11. It includes three counters (0, 1, 2), a data bus buffer, Read/Write Control Logic and control registers. Each counter has two input signals (CLK and GATE) and one OUtput Signal (OUT). The control word registers and the counters are selected according to the signals on A0 and A1 (as Shown in Figure 4.10).

A0	A1	Selection
0	0	Counter 0
0	1	Counter 1
1	0	Counter 3
1	1	Control Register

Fig. 4.10 *Selection of Counters and Registers According to Signals*

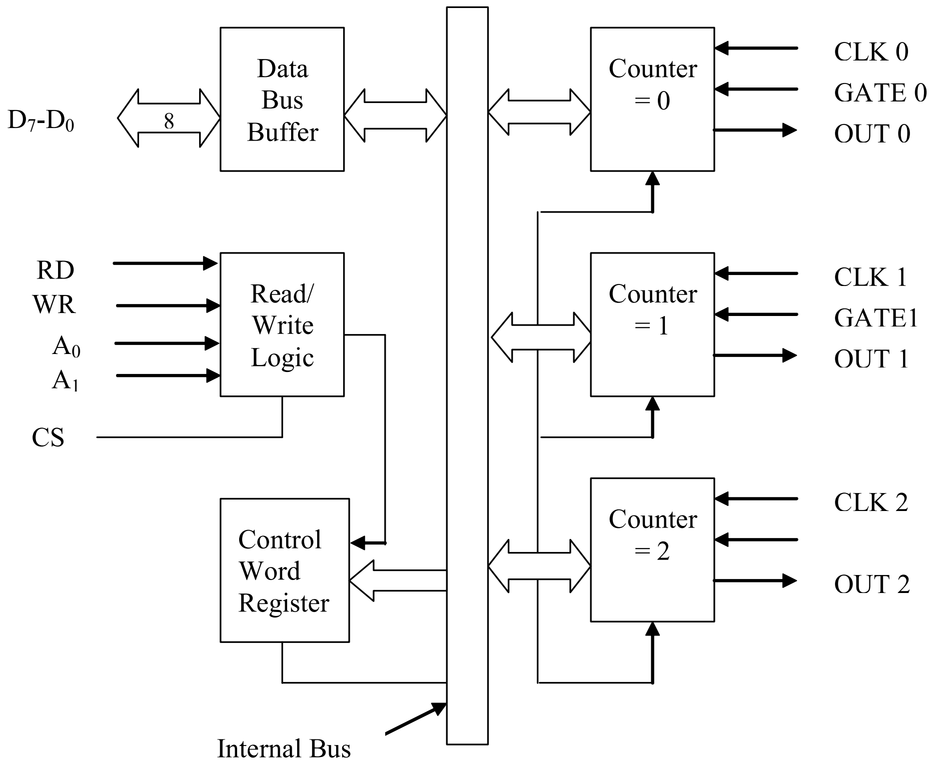


Fig. 4.11 *8254 Operating in Six Operating Modes*

8254 can operate in six different operating modes (Refer Figure 4.11) and gate of a counter is used to either disable or enable counting. Counters can count in the binary or decimal format. The control word of 8254 is illustrated in Figure 4.12 and the possible values of each of the bits are also being illustrated with different combinations. Mode 0 of the counter is most prominently used in general purpose applications.

8254 Control Word Format

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC 1	SC 0	RW 1	RW 0	M2	M1	M0	BCD

SC – Select counter:

SC1 SC 0

0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (See Read Operations)

M – MODE

M2 M1 M0

0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

RW – Read/Write

RW1 RW0

0	0	Computer Latch Command
0	1	Read/Write least significant byte only
1	0	Read/Write most significant byte only
1	1	Read/Write least significant byte first, then most significant byte.

BCD:

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

Fig. 4.12 *Control Word of 8254*

The six different modes of 8254 are:

Mode 0: Interrupt on Terminal Count

Mode 1: Hardware Triggerable One Shot

Mode 2: Rate Generator

Mode 3: Square Wave Generator

Mode 4: Software-Triggered Strobe

Mode 5 : Hardware Triggered Strobe

Mode 0: Interrupt on Terminal Count

The salient features of the mode are:

- It is used to generate an interrupt to the microprocessor after a certain interval of time.
- The output is initially low after the mode is set. The output remains LOW after the count value is loaded in the counter.
- The process of decrementing the counter continues till the terminal count is reached, i.e. the count becomes zero and the output goes HIGH. The output remains high until it reloads a new mode of operation or new count.
- The GATE signal is high for normal counting. When GATE goes low counting is terminated and the current count is latched till the GATE goes high again.
- The signal waveforms are shown in Figure 4.13.

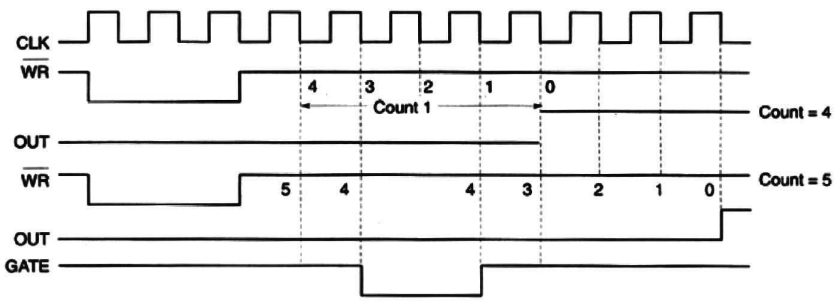


Fig. 4.13 Mode 0

Mode 1: Hardware Triggerable One-Shot

The salient features of the mode are:

- The 8253/54 can be used as a monostable multivibrator.
- The gate input is used as trigger input in this mode. Normally, the output remains high until the count is loaded and a trigger is applied.
- The duration of the quasistable of the monostable multivibrator is decided by the count loaded in the count register.
- The signal waveforms are shown in Figure 4.14.

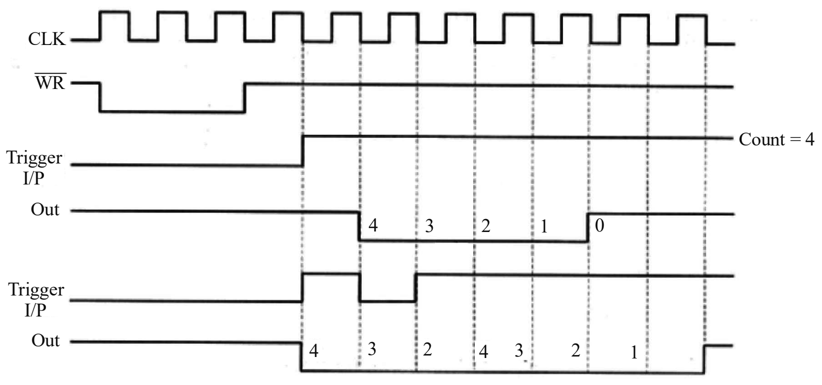


Fig. 4.14 Mode 1

Mode 2: Rate Generator

The salient features of the mode are:

- Divide by N counter.
- The output is normally high after initialization.
- If N is loaded as the count value, after N pulses, the output becomes low for one clock cycle.
- Whenever the count becomes zero another low pulse is generated at the output.
- The signal waveforms are shown in Figure 4.15.

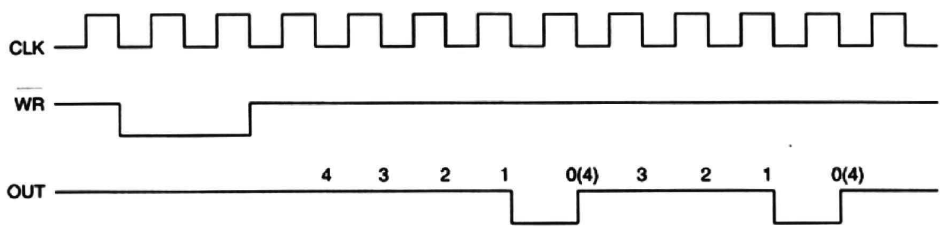


Fig. 4.15 Mode 2

Mode 3: Square Wave Generator

The salient features of the mode are:

- It is similar to mode 2.
- When, the count N loaded is EVEN, half of the count will be high and half of the count will be low.
- When, the count N loaded is ODD, the first clock pulse decrements it by 1. Then half of the remaining count will be high and half of the remaining count will be low.
- The signal waveforms are shown in Figure 4.16.

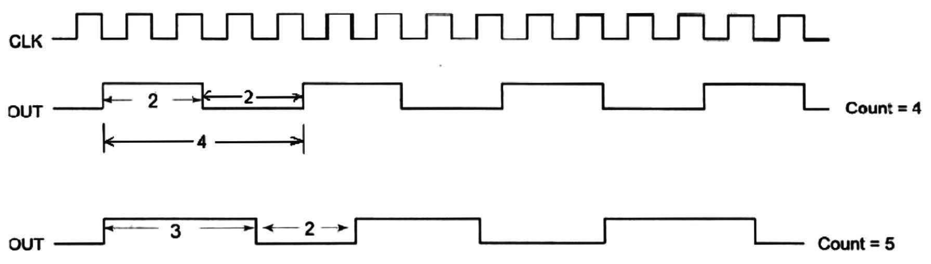


Fig. 4.16 Mode 3

Mode 4: Software Triggered Strobe

The salient features of the mode are:

- After the mode is set, the output goes high.

- The counter automatically begins to decrement (count down) one clock pulse after it is loaded with the initial value through software.
- When the GATE signal goes low, the count is latched.
- On the terminal count, the output goes low for one clock cycle and then again goes high. This low pulse can be used as a strobe.
- The signal waveforms are shown in Figure 4.17.

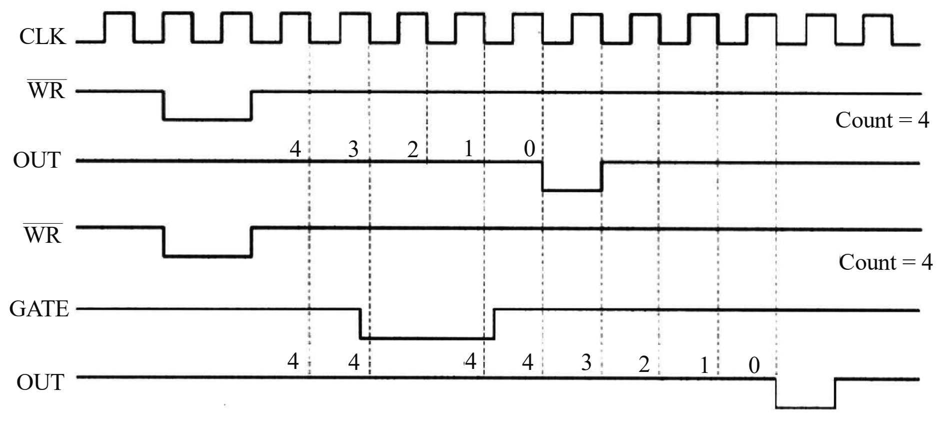


Fig. 4.17 *Mode 4*

Mode 5: Hardware Triggered Strobe

The salient features of the mode are:

- This mode generates a strobe in response to an externally generated signal.
- It is similar to mode 4 except that the counting is initiated by a signal at the gate input, i.e., it is hardware triggered instead of software triggered.
- After it is initialized, the output goes high.
- The counter starts counting after the rising edge of the trigger input (GATE).
- When the terminal count is reached, the output goes low for one clock cycle.
- The signal waveforms are shown in Figure 4.18.

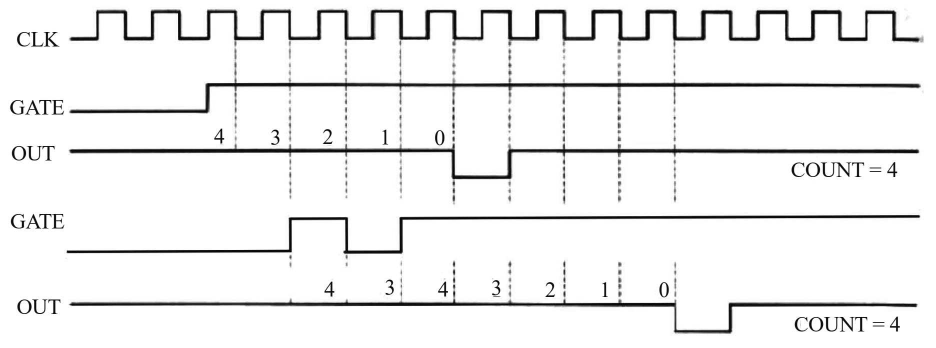


Fig. 4.18 *Mode 5*

NOTES

Check Your Progress

1. Define the term memory mapped I/O.
2. What is handshaking?
3. Which port of 8255 is used for generating control signals for mode 1 and mode 2?
4. How many counters are available in 8254?
5. What are the possible applications of 8254?

4.5 8251 PROGRAMMABLE/COMMUNICATION INTERFACE

There are two modes of data transmission—serial transmission and parallel transmission. Each mode has its own advantages and applications. In serial data transmission, the data is sent bit by bit (i.e., one bit at a time). However, in case of parallel data transmission, a number of bits (particularly one byte at a time) are sent as illustrated in Figure 4.19.

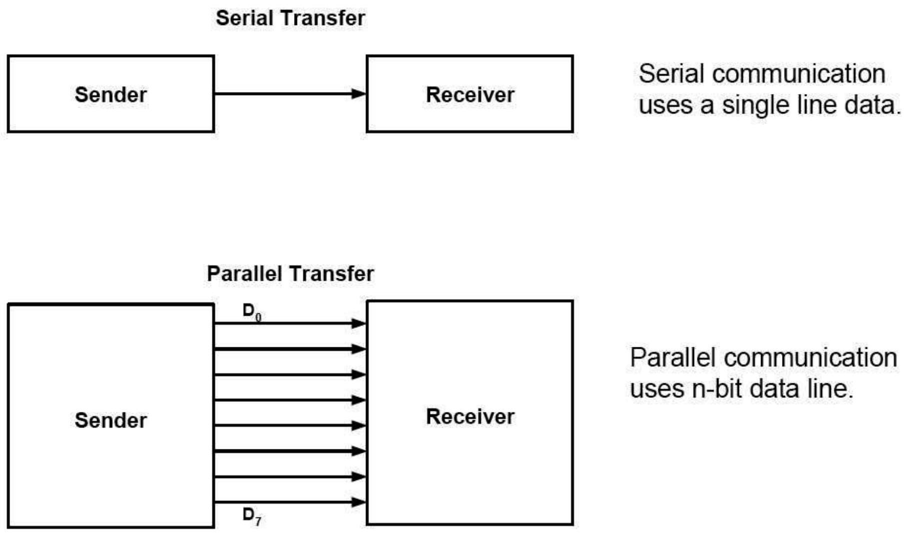

Serial Transfer
Sender
Receiver
Serial communication uses a single line data.
Parallel Transfer
D0
D7
Sender
Receiver
Parallel communication uses n-bit data line.

Fig. 4.19 *Modes of Data Transmission*

Communication protocol is a convention for data transmission that includes such functions as timing, control, formatting and data presentation. There are two categories depending on the clocking of the data on the serial link:

- **Synchronous Protocols**—Each successive datum in a stream of data is governed by a master clock and appears at specific intervals of time.
- **Asynchronous Protocols**—Successive data appears in the data stream at arbitrary times, with no specific clock control governing the relative delays between data.

The 8251A is a USART (Universal Synchronous Asynchronous Receiver Transmitter), which is used to convert serial data into parallel and vice-versa. It is a programmable chip used for both synchronous and asynchronous communication.

The device is very important as a microprocessor deals with the parallel data most of the time. The block diagram of 8251A is shown in Figure 4.20. It contains five sections as read/write control logic, transmitter, receiver, data bus buffer and modem control.

The control logic interfaces the chip with the microprocessor and determines the functions of the chip according to the control word in its register and also monitors the data flow. The Transmitter section converts parallel data received from the microprocessor into serial data and transmits this to the peripheral over the TxD line.The receiver section receives the serial bits from the peripheral, coverts them into the parallel word and transmits them to the CPU. Thus, the transmitter and receiver sections are mostly responsible for the conversions and are the most important parts of the chip. An expanded block diagram of the transmitter and receiver sections is shown in Figure 4.20. It contains registers to receive the data and buffer registers to temporarily store the data. Both transmitter and receiver sections have their separate control logic to control the conversions.

The modem control is used to establish data communication through modem over the telephone lines. The 8251A is a complex device (Refer Figure 4.21), capable of performing various functions. Although both synchronous and asynchronous communication is possible, the asynchronous mode is normally the focus of study. The reason is that the asynchronous mode is normally used for data transfer between the microprocessor and the serial peripheral devices such as video terminals and floppy disks.

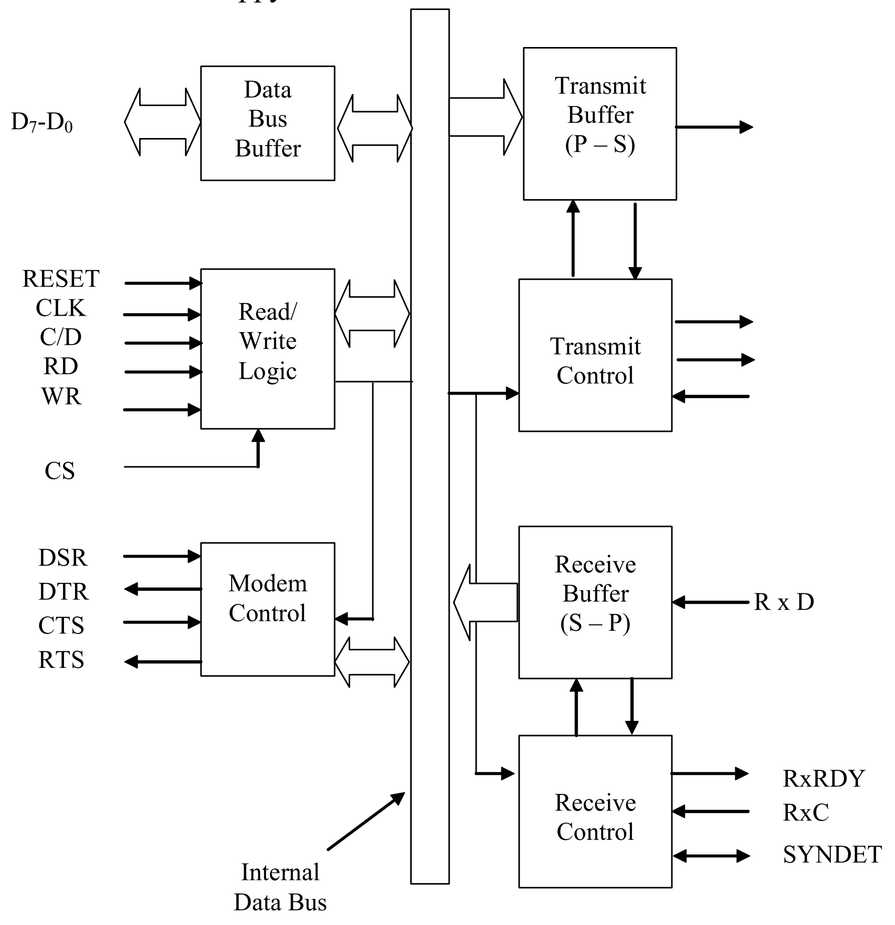


Fig. 4.20 Block Diagram of 8251

NOTES

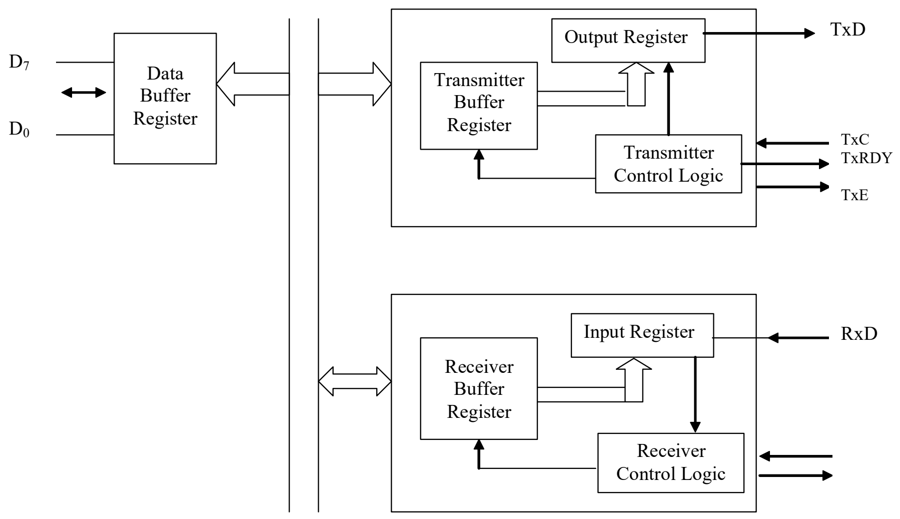


Fig. 4.21 *The 8251A Expanded Block Diagram of Transmitter and Receiver Section*

Before 8251 can be used to receive or transmit characters, its mode control and command registers must be initialized. The 8251 has only one address for a few control registers. The only readable register is a status register. The other registers must be written in sequence. The sequence to be followed for initialization of 8251 is shown in the flow chart in Figure 4.22.

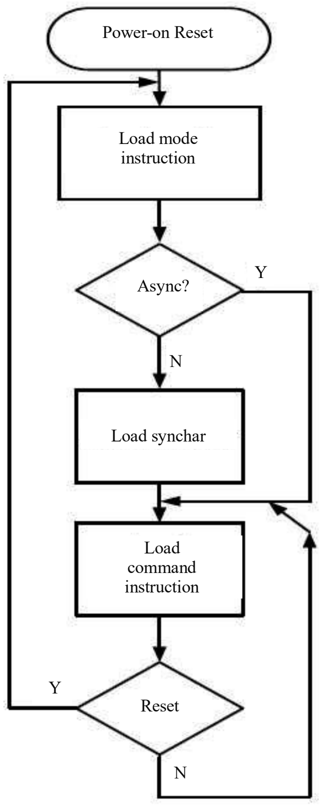


Fig. 4.22 *Sequence for Initialization*

4.6 INTERRUPTS

NOTES

An interrupt is an external event which informs the CPU that a device needs its service. While executing an interrupt, a microprocessor automatically saves the flag register, the instruction pointer and the code segment register on the stack, and goes to a fixed memory location to serve the interrupt. There are typically three types of interrupts: external interrupts, traps or internal interrupts and software interrupts.

1. **External Interrupts:** External interrupts are initiated through the microcomputer's interrupt pins by external devices such as A/D converters. A simple example of an external interrupt was given in the previous section. External interrupts can further be divided into two types: *maskable and nonmaskable*. A maskable interrupt can be enabled or disabled by executing specific instructions of microprocessor. If the microcomputer's interrupt is disabled, the microcomputer ignores the maskable interrupt. Some processors, such as the Intel 8086, have an interrupt flag bit in the processor status register. When the interrupt is disabled, the interrupt flag bit is 1, so no maskable interrupts are recognized by the processor. The interrupt flag bit resets to zero when the interrupt is enabled. The nonmaskable interrupt has higher **priority** than the maskable interrupt. If both maskable and nonmaskable interrupts are activated at the same time, the processor will service the nonmaskable interrupt first. Considering the case of 8086, the following are the external interrupts in 8086:
   (a) INTR (Interrupt Request): This is an input signal into the CPU which can be masked (ignored) and unmasked through CLI (disabled) and STI (enabled). No specific location in the vector table is assigned for INTR. It uses any "INT nn: space that has not been assigned. The 8259 chip connects to INTR to expand # of hardware interrupts up to 64.
   (b) NMI (NonMaskable Interrupt): This is an input signal into the CPU that cannot be masked and unmasked using software instructions. CPU will go to memory location 0008 to get the address (CS:IP) of the Interrupt Service Routine associated with NMI.

   Corresponding to each Interrupt Signal, INTA (interrupt acknowledge) signal is sent by the microprocessor. This is an output of the microprocessor to signal the external circuitry that the interrupt request has been acknowledged and to prepare to put its interrupt type.
2. **Internal Interrupts:** Internal Interrupts, or traps, are activated internally by exceptional conditions such as overflow, division by zero, or execution of an illegal op-code. Traps are handled the same way as external interrupts. The user writes a service routine to take corrective measures and provide an indication to inform the user that an exceptional condition has occurred.
3. **Software Interrupts:** Many processors include software interrupts or system calls. When one of these instructions is executed, the processor is interrupted and serviced similarly to external or internal interrupts. Software

interrupt instructions are normally used to call the operating system. Software interrupt instructions allow the user to switch from the user to the supervisor mode.

NOTES

8086 Interrupts

An 8086 interrupt can come from any of the following three sources:

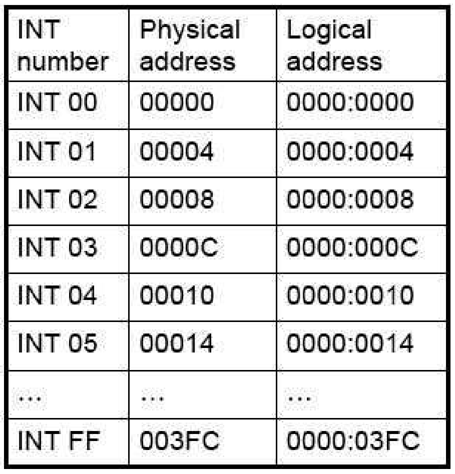
(1) An External Signal applied to the NMI input pin or INTR input pin of 8086. These interrupts are basically termed as hardware interrupts.

(2) Execution of software interrupts through INT instructions. There are 256 interrupts (types): INT 00, INT 01,…, INT FF in the 8088/8086.

(3) Due to some error condition generated during the execution of instructions, e.g. divide by Zero interrupt, if you try to divide an operand by zero, 8086 will automatically interrupt the currently executed programme.

Now, if an interrupt has been requested by any one of these reasons, the 8086 using responds the following steps:

1. Decrements the stack pointer by 2 and pushes the flag register on the stack.
2. Disables the INTR input by clearing the IF flag.
3. Resets the Trap Flag in the flag register.
4. Decrements the stack pointer by 2 and pushes the current Code Segment register contents on the stack.
5. Decrements the stack pointer again by 2 and stores the current instruction pointer contents on the stack.
6. Does an indirect far jump to the start of the procedure (Interrupt Service Procedure).
7. The location for the indirect far jump is provided by the interrupt vector table giving physical and logical addresses as shown in Figure 4.23.
8. The values of the CS and IP are basically copied from these physical locations and the Interrupt Service Procedure is stored at these locations as illustrated in Figure 4.23.



INT number	Physical address	Logical address
INT 00	00000	0000:0000
INT 01	00004	0000:0004
INT 02	00008	0000:0008
INT 03	0000C	0000:000C
INT 04	00010	0000:0010
INT 05	00014	0000:0014
…	…	…
INT FF	003FC	0000:03FC

Fig. 4.23 *Interrupt Service Procedure*

Out of these 256 interrupts, some software interrupts have been fixed for specific conditions like divide by zero, overflow, breakpoint, etc., whereas the remaining interrupts (INT 05 – INT FF) are free and can be used by the programmer to implement hardware of software interrupts. Intel's list of designated interrupts of 8086 has been shown in Figure 4.24.

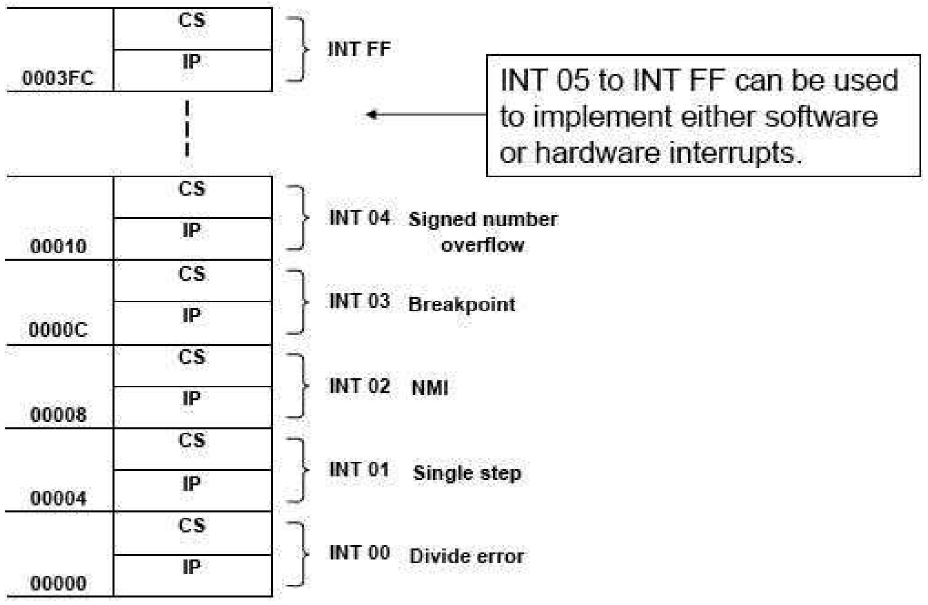

CS
IP
0003FC
INT FF
INT 05 to INT FF can be used to implement either software or hardware interrupts.
00010
INT 04 Signed number overflow
0000C
INT 03 Breakpoint
00008
INT 02 NMI
00004
INT 01 Single step
00000
INT 00 Divide error

Fig. 4.24 Intel's List of Designated Interrupts of 8086

Whenever an interrupt occurs, a specific program is run by the microprocessor. This specific program is called Interrupt Service Routine. An Interrupt vector table is used to store the addresses of Interrupt Service Routines. An interrupt vector table contains 256 table entries. Each table entry takes 4 bytes—two bytes are for IP values and two bytes for CS values. interrupt vector table locates the reserved memory space from 00000H to 003FFH (Refer Figure 4.25). The actual service routine is stored at these memory locations. This service routine actually tells the microprocessor, what to do, when an interrupt occurs.

For example, assume that the Interrupt Service Routine for the Type-40 interrupt is located at address 28000H. How do you write this address to the vector table? Surely, you have to think this physical address in terms of the CS:IP values and then you have to save these CS and IP values in the interrupt vector table locations corresponding to Type 40.

NOTES

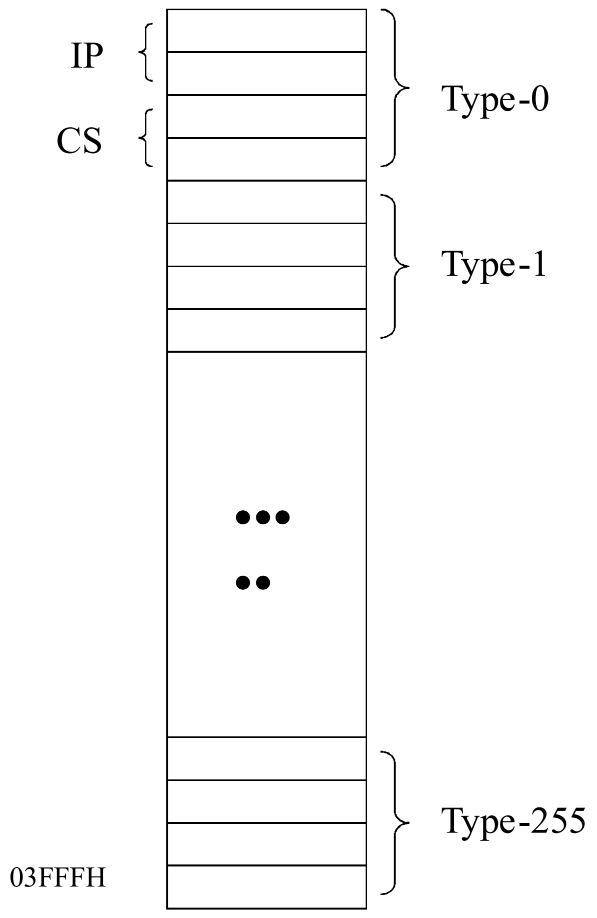


***Fig. 4.25** Interrupt Service Routines*

In order to process the particular Interrupt Service Routine the following set of steps are followed by the microprocessor 8086.

1. Get Vector Number (get the interrupt type)
    - Caused by NMI, it is type 2
    - Caused by INTR, the type number will be fed to the processor through data bus
    - Caused by executing INT instructions, the type number is given by the operand
    - •••
2. Save Processor Information
    - Push flag register into stack
    - Clear trace flag and interrupt-enable flag
    - Push CS and IP into stack
3. Fetch New Instruction Pointer
    - Load new CS and IP values from the instruction vector table
4. Execute Interrupt Service Routine
5. Return from Interrupt Service Routine
    - Pop flag register from stack
    - Pop CS and IP from stack

Some ISRs also contain instructions that save and restore general purpose registers. An example code for the Interrupt Service Routine has been shown as follows:

```
1234:   00AE PUSH AX
        PUSH DX
        MOV AX, 5
        MUL BL
        MOV [DI], AX
        MOV [DI+2], DX
        POP DX
        POP AX
        IRET
```

The Interrupt Vector Table Corresponding to this ISR has been shown in Figure 4.26.

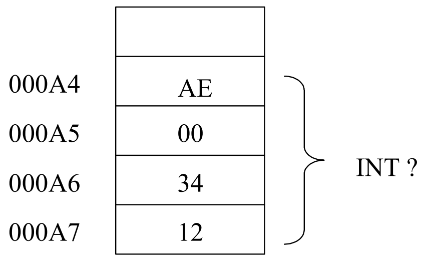


Fig. 4.26 *Interrupt Vector Table Corresponding to ISR*

During the processing of the interrupt, the Programming Environment needs to be stored. The most important is the return address of the main program and the status of the Flags. The sequence of steps to store this environment is shown in Figure 4.27.

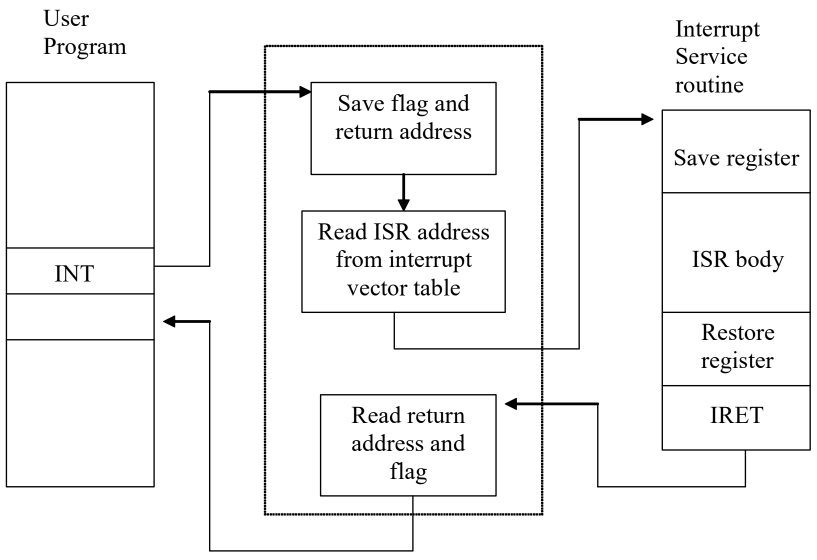

User Program
INT
Save flag and return address
Read ISR address from interrupt vector table
Read return address and flag
Interrupt Service routine
Save register
ISR body
Restore register
IRET

Fig. 4.27 *Storing of Programming Environment*

4.7 8259 PROGRAMMABLE INTERRUPTS CONTROLLER

NOTES

The 8259A is a programmable interrupt controller designed to work with Intel Microprocessors like 8085, 8086 and 8088. This has the following features:

1. It can manage eight interrupts according to the instructions written in the control registers.
2. It vectors an interrupt anywhere in the memory map. However, all eight interrupts are spaced at the interval of either four or eight locations.
3. It resolves eight levels of interrupt priority in a variety of modes, such as fully nested, automatic rotation and specific rotation.
4. It mask each interrupt request individually.
5. It reads the status of in-service, masked and pending interrupts.
6. It can be expanded to 64 priority levels by cascading additional 8259As.
7. It can work with the 8085 8-bit processor or a 16-bit 8086/8088.

The block diagram of 8259A is shown in Figure 4.28. It consists of eight blocks. The Read/Write logic is used to write a command or read a status whereas the control logic has two connections INT and $\overline{\text{INTA}}$ connected to the INT and INTA pin of the microprocessor. The interrupt Request Register (IRR) has right input lines (IR_0-IR_7) for interrupts. When these lines go high, the requests are stored in the register. The In-Service Register (ISR) stores all the levels that are currently being serviced, and the Interrupt Mask Register (IMR) stores the masking bits of the interrupt lines to be masked. The Priority Resolver (PR) examines these three registers and determines, whether the INT should be sent to the microprocessor.

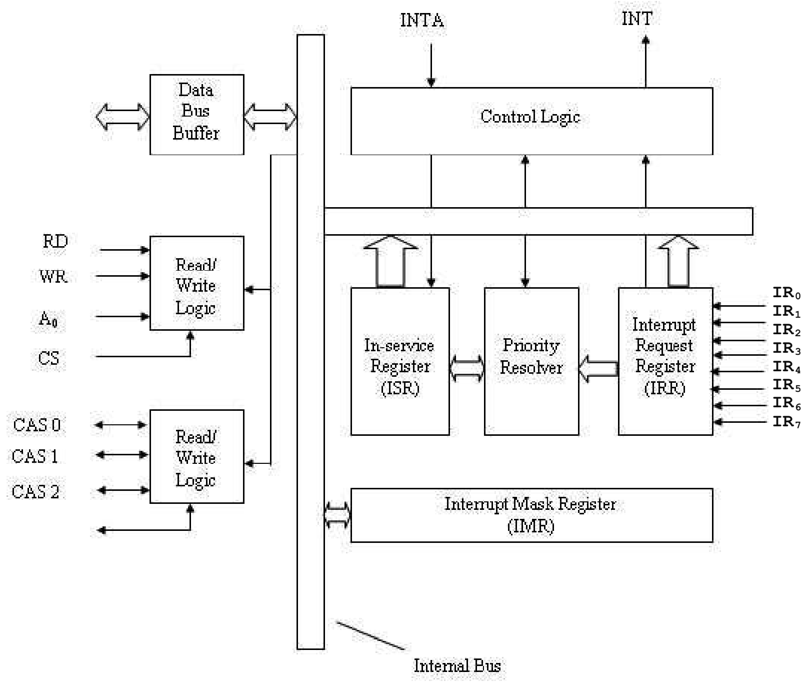


Fig. 4.28 *8259A Block Diagram*

Basically, the 8259A is a device specifically designed for use in real time, interrupt-driven microcomputer systems. It manages eight levels or requests and has built-in features for expandability to other 8259A's (up to 64 levels). It is programmed by the system's software as an I/O peripheral. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the 8259A can be configured to match his system requirements. The priority modes can be changed or reconfigured dynamically at any time during the main program. This means that the complete interrupt structure can be defined as required, based on the total system environment.

Interrupt Request Register (IRR) and In-Service Register (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all the interrupt levels which are requesting service; and the ISR is used to store all the interrupt levels which are being serviced.

Priority Resolver

This logic block determines the priorites of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during INTA pulse.

Interrupt Mask Register (IMR)

The IMR stores the bits which mask the interrupt lines to be masked. The IMR operates on the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower quality.

INT (Interrupt)

This output goes directly to the CPU interrupt input. The VOH level on this line is designed to be fully compatible with the 8080A, 8085A and 8086 input levels.

INTA (Interrupt Acknowledge)

INTA pulses will cause the 8259A to release vectoring information onto the data bus. The format of this data depends on the system mode (mPM) of the 8259A.

Data Bus Buffer

This 3-state, bidirectional 8-bit buffer is used to interface the 8259A to the system data bus. Control words and status information are transferred through the data bus buffer.

Read/Write Control Logic

The function of this block is to accept output commands from the CPU. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the 8259A to be transferred onto the Data Bus.

CS (Chip Select)

A LOW on this input enables the 8259A. No reading or writing of the chip will occur unless the device is selected.

WR (Write)

A LOW on this input enables the CPU to write control words (ICWs and OCWs) to the 8259A.

RD (Read)

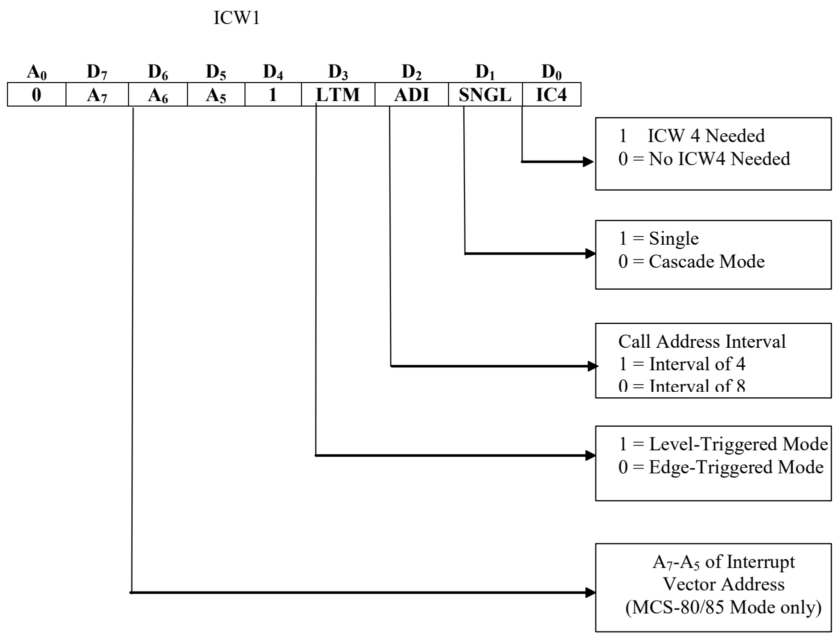
A LOW on this input enables the 8259A to send the status of the Interrupt Request Register (IRR), In Service Register (ISR), the Interrupt Mask Register (IMR), or the Interrupt level onto the Data Bus.

A0

This input signal is used in conjunction with WR and RD signals to write commands into the various command registers, as well as to read the various status registers of the chip. This line can be tied directly to one of the address lines.

Programming 8259A

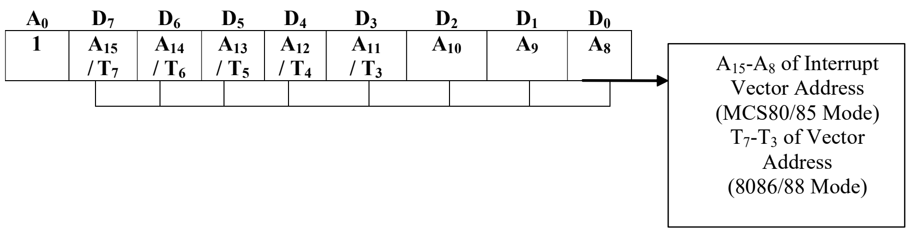
The 8259A requires two types of command words, the Initialization Command Words (ICW) and Operational Command Words (OCW).The 8259A can be initialized by four ICWs. The first two are essential and the remaining two are mode based. These words must be issued in sequence. Once initialized, the 8259A can be set up to operate in various modes by using three different OCWs, however they need not be in the sequence. The ICW1 format is shown in Figure 4.29. The total address is generated by keeping in mind the Interrupt Request Address given in the Table (interval of 4 or 8) and the ICW2. The choice of ICW3 and ICW4 is made on the basis of certain conditions as illustrated in the flow chart shown in Figure 4.29.



IR	Interval = 4 (8086 mode)							
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
7	A_7	A_6	A_5	1	1	1	0	0
6	A_7	A_6	A_5	1	1	0	0	0
5	A_7	A_6	A_5	1	0	1	0	0
4	A_7	A_6	A_5	1	0	0	0	0
3	A_7	A_6	A_5	0	1	1	0	0
2	A_7	A_6	A_5	0	1	0	0	0
1	A_7	A_6	A_5	0	0	1	0	0
0	A_7	A_6	A_5	0	0	0	0	0

IR	Interval = 8 (8085 mode)							
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
7	A_7	A_6	1	1	1	0	0	0
6	A_7	A_6	1	1	0	0	0	0
5	A_7	A_6	1	0	1	0	0	0
4	A_7	A_6	1	0	0	0	0	0
3	A_7	A_6	0	1	1	0	0	0
2	A_7	A_6	0	1	0	0	0	0
1	A_7	A_6	0	0	1	0	0	0
0	A_7	A_6	0	0	0	0	0	0

ICW2



A_0 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
1
A_{15} / T_7
A_{14} / T_6
A_{13} / T_5
A_{12} / T_4
A_{11} / T_3
A_{10}
A_9
A_8
A_{15}-A_8 of Interrupt Vector Address (MCS80/85 Mode) T_7-T_3 of Vector Address (8086/88 Mode)

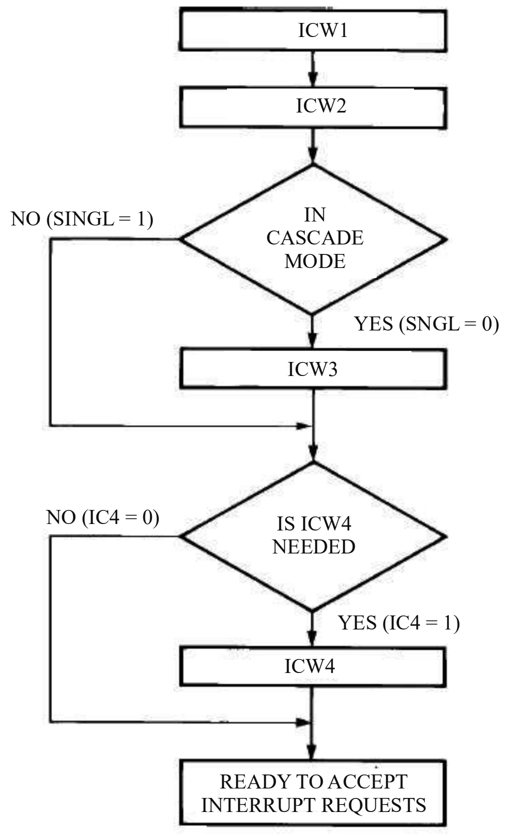


Fig. 4.29 *Initialization Control Word for 8259A*

Check Your Progress

6. Write the use of USART.
7. Define the term interrupts.
8. Write the command of 5259.

NOTES

4.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The processor uses a portion of the memory address to represent I/O ports. The I/O ports are mapped into the processor's main memory and are hence called memory mapped I/O.
2. The microprocessor is a fast device and the Input/Output devices are slow in comparison to the microprocessor. Thus, in order to communicate, the microprocessor needs to know which is the right time to send the next data. The process of knowing the status of devices and transferring the data with matching speeds is called handshaking.
3. Port C can be used to generate control signals for mode 1 and mode 2.
4. Three counters are available in 8254.
5. 8254 is used to generate accurate time delays and can be used for other timing applications such as a real time clock, an event counter, a digital one shot, a square wave generator and a complex waveform generator.
6. USART (Universal Synchronous Asynchronous Receiver Transmitter), which is used to convert serial data into parallel and vice-versa. It is a programmable chip used for both synchronous and asynchronous communication. The device is very important as a microprocessor deals with the parallel data most of the time.
7. An interrupt is an external event which informs the CPU that a device needs its service. While executing an interrupt, a microprocessor automatically saves the flag register, the instruction pointer and the code segment register on the stack, and goes to a fixed memory location to serve the interrupt.
8. The 8259 requires two types of command:
   - Initialization Command Words (ICU)
   - Operational Command Words (OCW)

4.9 SUMMARY

- A standalone microprocessor is of no use to the world, unless it can communicate with the outside world.
- I/O interface provides a mechanism to establish a link between the microprocessor and the Input/Output devices like printer, keyboard, mouse, process control devices, etc.

- The processor uses a portion of the memory address to represent I/O ports. The I/O ports are mapped into the processor's main memory and are hence called memory mapped I/O.
- The standard I/O is also called isolated I/O. It uses IO/M control pin on the microprocessor. Processor outputs are high on this pin to indicate an I/O operation taking place. A low on this pin indicates a memory operation.
- The microprocessor is a fast device and the Input/Output devices are slow in comparison to the microprocessor. Thus, in order to communicate, the microprocessor needs to know which is the right time to send the next data.
- Mode 0 is used for simple input or output without handshaking. Both ports A and B can be initialized in this mode, also the two halves of port C can be used separately or combined as port C in this mode.
- Mode 1 is used for single handshake (Strobed). Both ports A and B can be initialized in this mode. However, the pins of port C are used to generate the control signals (BSR mode). PC0-PC2 are used for port B, whereas PC3-PC5 are used for port A. The remaining pins PC6-PC7 can be used as I/O lines.
- Mode 2 is used for double handshake (bi-directional). Only port A can be initialized in this mode. However, the pins of port C are used to generate the control signals. PC3-PC7 are used as handshake lines (BSR mode) for port A.
- A/D and D/A converters are very interfaced with the microprocessors using 8255 programmable interface.
- The analogue to digital converter chips 0808 and 0809 are 8-bit CMOS, successive approximation converters. This technique is one of the fastest techniques for analogue to digital conversion.
- The digital to analogue converters convert binary number into their equivalent voltages. The DAC find applications in areas like digitally controlled gains, motors speed controls, programmable gain amplifiers etc.
- The 8254 programmable Timer is functionally similar to the software designed timers ad counters. It is used to generate accurate time delays and can be used for other timing applications, such as a real time clock, an event counter, a digital one shot, a square wave generator and a complex waveform generator.
- The Intel 8253/8254 contains three 16-bit counters and each of them can be programmed to operate in several different modes.
- There are two modes of data transmission serial transmission and parallel transmission. Each mode has its own advantages and applications. In serial data transmission, the data is sent bit by bit (i.e., one bit at a time).
- USART (Universal Synchronous Asynchronous Receiver Transmitter), which is used to convert serial data into parallel and vice-versa. It is a programmable chip used for both synchronous and asynchronous communication. The device is very important as a microprocessor deals with the parallel data most of the time.

- An interrupt is an external event which informs the CPU that a device needs its service. While executing an interrupt, a microprocessor automatically saves the flag register, the instruction pointer and the code segment register on the stack, and goes to a fixed memory location to serve the interrupt.
- External interrupts are initiated through the microcomputer's interrupt pins by external devices, such as A/D converters. A simple example of an external interrupt was given in the previous section.
- Internal Interrupts, or traps, are activated internally by exceptional conditions, such as overflow, division by zero, or execution of an illegal op-code. Traps are handled the same way as external interrupts.
- Many processors include software interrupts or system calls. When one of these instructions is executed, the processor is interrupted and serviced similarly to external or internal interrupts.
- The 8259A requires two types of command words, the Initialization Command Words (ICW) and Operational Command Words (OCW).

4.10 KEY TERMS

- **Mode 0:** Mode 0 is used for simple input or output without handshaking.
- **Serial transmission:** A mode of data transmission where the data is sent bit by bit, one bit at a time.
- **Parallel transmission:** A mode of data transmission where the data is sent number of a bits at a time.
- **Communication protocol:** It is a convention for data transmission that includes functions, such as timing, control, formatting and data presentation.
- **Interrupt:** An external event which informs the CPU that a device needs its services.

4.11 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Differentiate between the memory mapped I/O and mapped I/O.
2. What is the need of 8255 microprocessor?
3. Name the different modes of 8254.
4. Differentiate between the maskable and non-maskable interrupts.
5. What do you mean by cycle stealing?
6. How will you generate a real time clock using microprocessor 8086?

Long-Answer Questions

1. Discuss the architectural diagram of 8255 giving the description of various ports and operating modes. How do we generate control words in the I/O mode or in the BSR mode?
2. Describe the interfacing of digital to analogue converter with 8086 using the 8255 PPI.
3. Briefly explain about the block diagram of 8254 with initialization control words.
4. Discuss the block diagram of 8251 giving a brief description of each block.
5. Briefly explain about the interrupt and different categories of interrupts giving appropriate examples.
6. Discuss the block diagram of 8259A giving a description of various block and essential signals. How do we decide the need for various initialization control words?
7. Describe the operation of programmable timer 8254 in different operating modes with the help of example diagrams.

4.12 FURTHER READING

Goankar, Ramesh. 2002. *Microprocessor Architecture, Programming and Application with the 8085*. Mumbai: Penram International Publishing.

Ram, B. 2005. *Fundamentals of Microprocessor and Microcomputers*. New Delhi: Dhanpat Rai Publications.

Lotia, Manahar and Pradeep Nair. 2003. *Modern All About Motherboard*. New Delhi: BPB Publications.

Mueller, Scott, and Craig Zacker. 2002. *Upgrading and Repairing PCs*. New Jersey: Pearson Education (Que Publishing).

Godse, D.A. and A.P. Godse. 2007. *Microprocessor and Assembly Language Programming*. Pune: Technical Publications.

Kleitz, William. 2009. *Digital and Microprocessor Fundamentals: Theory and Applications*. New Jersey: Prentice Hall.

Ray, A.K. and K.M. Bhurchandi. 2000. *Advanced Microprocessors and Peripherals*. New Delhi: Tata McGraw-Hill.

UNIT 5 8086 ASSEMBLY LANGUAGE PROGRAMMING

NOTES

Structure

5.0 INTRODUCTION

The 8086 was Intel's first 16-bit microprocessor. It was launched in 1978. Its design is based on the 8080 but is not directly compatible with the 8080. The introduction of the 16-bit processor was the result of the increasing demand for more and more powerful and high speed computational resources. An interesting feature of 8086 is that it pre fetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution. Directives are statements written in the source program but are meant for use by the assembly language program. Assembly directives may be divided into several groups, such as data directives, segment control directives, modular programming directives, primary usage directives, etc.

To convert an assembly language program to a machine code, you must convert each assembly language instruction to its equivalent machine code instruction. Assembly language programming the beginner should write simple programs. The memory addresses given in the program are for a particular microprocessor kit. To develop a program that implements an application, the programmer goes through a multi-step process, called the program development cycle.

In this unit, you will study about the introduction set of 8086, assembler directives and operators, machine coding and programs, a few machine level programs, programming with a assembler, assembly language example programs.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic of introduction set of 8086

- Explain the assembler directives and operators
- Discuss about the machine coding and programs and a few machine level programs
- Define the programming with a assembler
- Analyse the assembly language example programs

5.2 INSTRUCTION SET OF 8086

The 8086 was Intel's first 16-bit microprocessor. It was launched in 1978. Its design is based on the 8080 but is not directly compatible with the 8080. The introduction of the 16-bit processor was the result of the increasing demand for more and more powerful and high speed computational resources. An interesting feature of 8086 is that it prefetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.

The memory in the 8086-based microcomputer is organized as bytes. Each byte can be uniquely addressed with 20-bit addresses of 00000_{16}, 00001_{16}, ... $FFFFF_{16}$. An 8086 word in memory consists of any two consecutive bytes; the smaller address containing the high byte specifies the word, while the higher address contains the low byte. The 8086 always accesses a 16-bit word to or from memory. A word instruction accessing a word starting at an even address can perform its function with one memory access while a word instruction starting at an odd address performs two memory accesses to two consecutive memory even addresses, discarding unwanted bytes of each. For byte access starting at odd address N, the byte at the previous even address N– 1 is also accessed but discarded. Similarly, for byte access starting at even address N, the byte with odd address N +1 is also accessed but discarded. The 8086 register names followed by the letters X, H or L in an instruction for data transfer between register and memory specify whether the transfer is 16-bit or 8-bit. The 8086 is packaged in a 40-pin Dual Inline Package (DIP). A single + 5 V power supply is required. The clock input signal is generated by the 8284 clock generator chip. Instruction execution times vary between 2 clock cycles and 3D clock cycles.

The architecture of the 8086 processor provides a number of improvement over its predessecor 8085. It supports a 16-bit ALU, a set of 16-bit registers and provides segment memory addressing capability, a rich instruction set, powerful interrupt structure and a six byte instruction queue.

Main Features of the 8086

The following are the main features of the 8086:

- It is a 16- bit microprocessor.
- It is a 16- bit ALU.
- Maximum clock frequency is 5MHz.
- Memory is divided in two banks: (*i*) even bank (*ii*) odd bank.
- Maximum memory that can be connected is 1MB (2^{20} = 1,048,576 bytes).
- Address range of memory is from 00000 H – FFFFF H (= 32 address lines).

NOTES

- It requires a single +5V power supply.
- It has a 20 bit address bus.
- It can generate 16-bit input/output address, hence it can access $2^{16} = 65536$ I/O ports.
- It is designed to operate in two modes, namely the minimum mode and maximum mode.
- It supports multiprogramming.

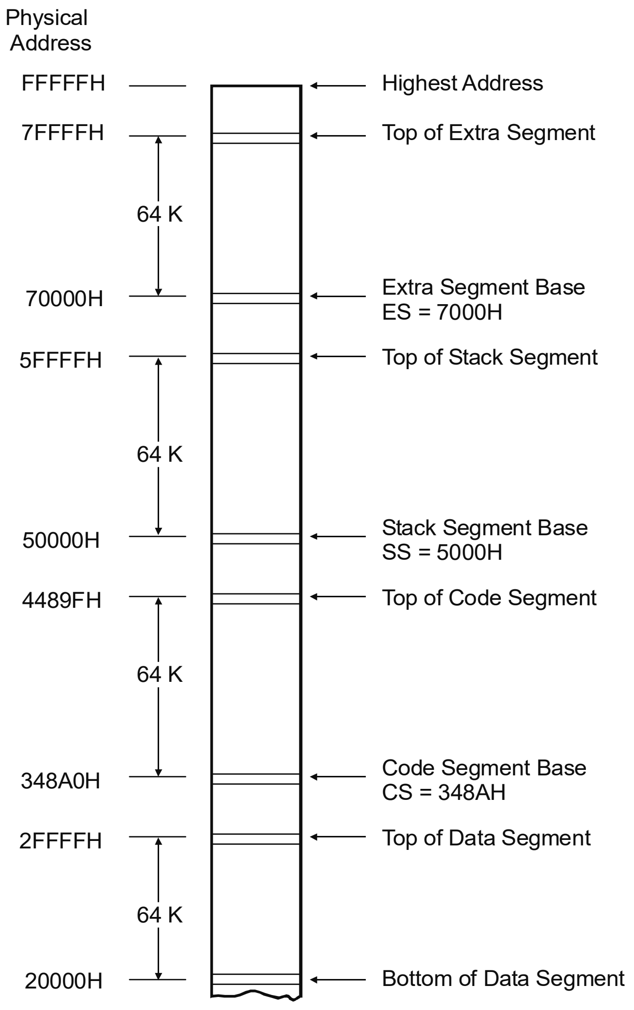


Fig. 5.1 *8086 Internal Block Diagram*

- It can fetch up to six instruction bytes (four instruction bytes for 8088) from memory and can queue them in order to speed up instruction execution.

The internal block diagram of Intel 8086 is as shown in Figure 5.1. It can be divided into two parts: (*a*) Bus Interface Unit (BIU) and (*b*) Execution Unit (EU). The two units function independently. The BIU fetches instructions, reads operands and writes results. The EU executes instructions that have already been fetched by the BIU so that the instruction fetch overlaps with execution.

Bus Interfacing Unit (BIU)

The BIU (Bus Interfacing Unit) provides all external bus operations. It contains segment registers, instruction pointer, instruction queue and address generation/

bus control circuitry to provide functions such as the fetching and queueing of instructions and bus control.

NOTES

It performs the following functions:

- It sends instruction from memory.
- It reads data from port/memory.
- It writes data into port/memory.
- It supports instruction queueing.
- It provides address relocation facility.

The BIU's instruction queue is a First-In-First-Out (FIFO) group of registers in which up to 6 bytes of instruction code are prefetched from memory ahead of time. This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as *queue* or *pipelining*. If the queue is full and the EU does not request BIU to access memory, then BIU does not perform any bus cycle. On the other hand, if the BIU is not full and if it can store at least two bytes and the EU does not request it to access memory, the BIU may fetch instructions. However, if BIU is interrupted by the EU for memory access while the BIU is in the process of fetching an instruction, the BIU first completes fetching and then services the EU. The queue allows the BIU to keep the EU supplied with prefetched instructions without tying up the system bus. If an instruction, such as a JUMP or CALL SUBROUTINE, is encountered, the BIU will reset the queue and begin refilling after passing the new instruction to the EU.

BIU contains a dedicated adder, which is used to produce a 20-bit physical address that is output on the address bus. This address is formed by adding 20-bit segment address called *base address* and 16-bit offset address called *effective address*.

The bus control logic of the BIU generates all the bus control signals, such as the READ and WRITE signals, for memory and I/O.

The BIU also has four 16-bit segment registers. They are:

(*a*) Code Segment (CS) Register

(*b*) Data Segment (DS) Register

(*c*) Extra Segment (ES) Register

(*d*) Stack Segment (SS) Register

The 8086's 1-MB memory is divided into segments of up to 64 K bytes each. The 8086 can directly address fair segments (256 K bytes within the 1MB of memory) at a particular time. Programs obtain access to code and data in the segments by changing the segment register contents to point to the desired segments.

CS Register

This holds the program code, and its contents indicate where exactly the code segment start in memory.

All program instructions are located in main memory pointed to by the 16-bit *CS register* with a 16-bit offset in the segment contained in the 16-bit *Instruction*

Pointer (IP). The BIU computes the 20- bit physical address internally using the programmer-provided logical address by logically shifting the contents of CS four bits to the left and then adding the 16- bit contents of IP. The BIU always inserts four zeros for the lowest four bits of the 20-bit starting address of a segment. Immediate data is considered part of the code segment.

SS Register

This points to the current stack. The 20-bit physical stack address is calculated from the SS and Stack Pointer (SP) for stack instructions, such as PUSH and POP. The programmer can use Base of Pointer (BP) instead of SP for accessing the stack using the base addressing mode. The stack is a Last-In-First-Out (LIFO) data structure implemented in RAM. In 8086, there will be a separate stack segment.

DS Register

This points to the current data segment. Operands for most instructions are fetched from data segment. The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address. DS holds the data, constants and work areas needed by the program.

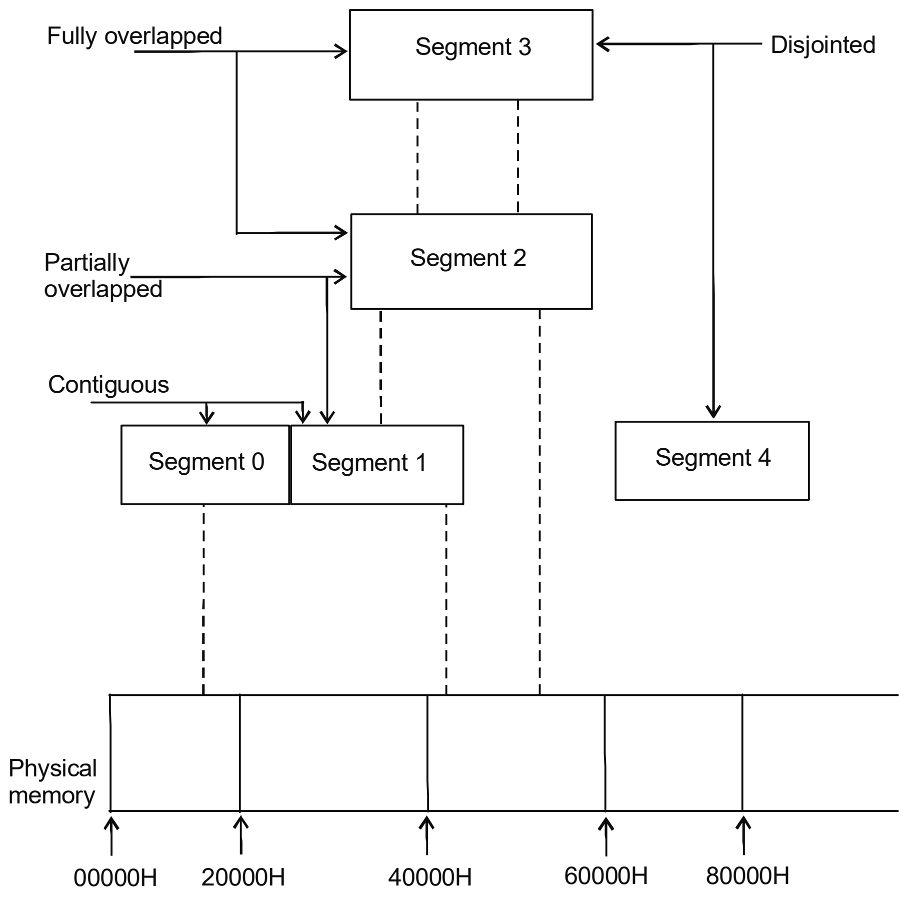


Fig. 5.2 *Segments that are Stored in Physical Memory*

NOTES

ES Register

It points to the extra segment in which data is stored. String instructions always use the ES and DI to determine the 20-bit physical address for the destination.

The segments can be contiguous, partially overlapped, fully overlapped or disjointed. Five segments from 0 to 4 may be stored in physical memory, as shown in Figure 5.2.

Every segment must start on 16-byte memory boundaries. A segment can be pointed to by more than one segment register.

It is very important to note that codes should not be written within 6-bytes of the end of physical memory.

Execution Unit (EU)

EU consists of an ALU, status and control flags and queue control logic. It also contains nine 16-bit registers. These are: AX, BX, CX, DX, SP, BP, SI and DI and flag register. The 16-bit general registers AX, BX, CX, and DX can be used as two 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL). The general-purpose registers AX, BX, CX and DX are named after special functions carried out by each one of them.

EU decodes and executes instructions. A decoder in the EU control system translates instructions. It has a 16-bit ALU for performing arithmetic and logic operations. The EU may pass the results to the BIU for storing them in memory.

After extracting instruction from the BIU in the top of the queue, EU decodes it, generating an operands address if needed, and requesting it to perform read or write bus cycles to memory or I/O to perform the operation specified by instructions on operands. While executing the instructions, EU tests status and control flags, updating them on the basis of the execution results. In case the queue is empty, EU waits for the next instruction data to be fetched, which is then moved to the queue's top.

AX Register: This is called a 16-bit accumulator while the AL is the 8-bit accumulator. The I/O instructions use the AX or AL for inputting/outputting 16 or 8-bit data to or from an I/O port. AX or AL are also used in multiplication and division instructions. The AL is the same as the 8085 A register.

BX Register: This is also known as the base register. It is a general-purpose register, the only one, whose contents can be used to address 8086 memory. Memory references that utilize BX register content to address use DS as default segment register.

CX Register: This is also known as the counter register as instructions such as SHIFT, ROTATE and LOOP use its contents as counter.

DX Register: This is also known as data register. It holds the high 16-bit result (data) in 16x16 multiplication or the high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit remainder after the division. For in and out instructions, 16-bit port address is stored in DX register.

SP and BP: These are two pointer registers that access data in a stack segment. SP is used as an offset from the current SS while executing instructions involving stock segment in external memory. SP contents are updated automatically as POP or PUSH instructions are executed. BP contains an offset address in the current SS that is used by instructions that utilize based addressing mode.

SI and DI: These are two index registers used in indexed addressing. Instructions that process data strings are SI and DI index registers together with DS and Es for distinguishing between source and destination addresses.

Flag Register

A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU.

The *flag register* holds the status flags, typically after an ALU operation. The EU sets or resets these flags to reflect certain properties of the results of arithmetic and logic operations. Figure 5.3 summarizes the 8086 registers and Figure 5.4 shows the 8086 flag register. The 8086 has six 1-bit status flags.

There are six type of flags.

Auxiliary Carry Flag (AF): This is set if there is a carry from the low nibble into the high nibble or a borrow from the high nibble into the low nibble of the low-order 8 bits of a 16-bit number. This flag is used by BCD arithmetic instructions; otherwise, AF is zero.

Carry Flag (CF): This is set if there is a carry from addition or a borrow from subtraction.

Overflow Flag (OF): This is set if there is an arithmetic overflow. An interrupt on overflow instruction is available to generate an interrupt in this situation; otherwise, it is zero.

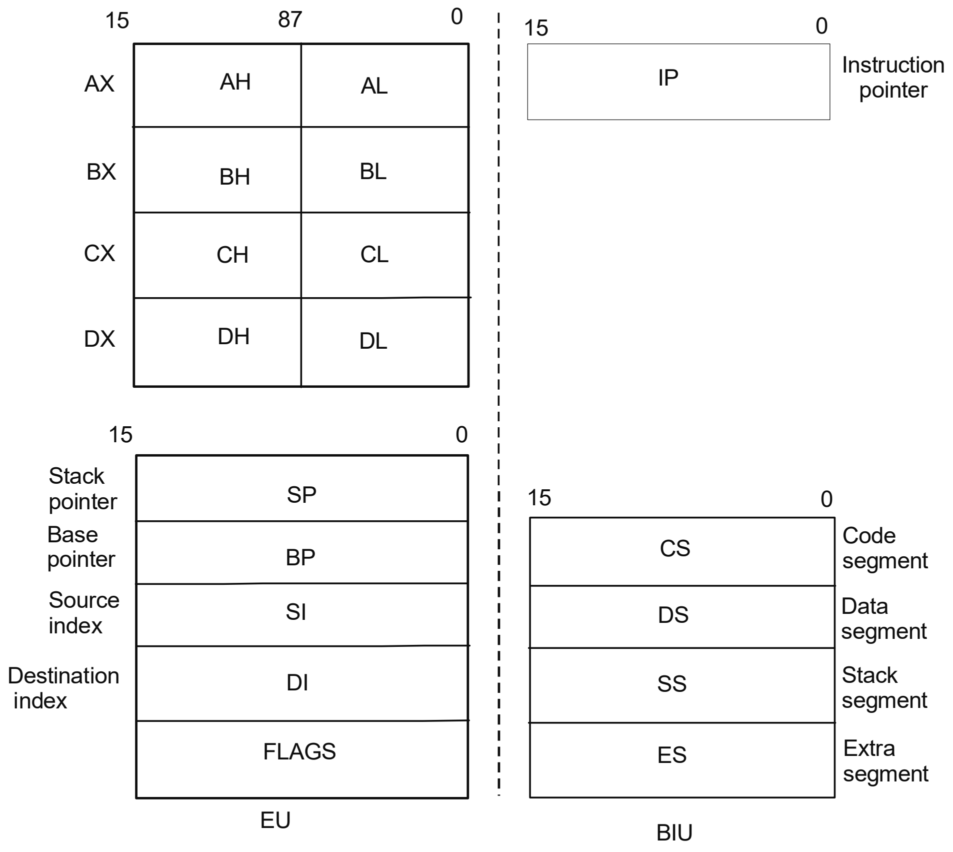


Fig. 5.3 *Format of 8086 Registers*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

U–undefined

Fig. 5.4 *8086 Flag Register Format*

NOTES

Sign Flag (SF): This is set if the Most Significant Bit (MSB) of the result is one; otherwise it is zero.

Parity Flag (PF): It is set if the result has even parity; PF is zero for odd parity of the result.

Zero Flag (ZF): It is set if the result is zero; ZF is zero for a non-zero result.

The 8086 has three control bits in the flag register that can be set or cleared by the programmer.

Both BIU and EU operate synchronously to give 8086 an overlapping instruction fetch and instruction execution mechanism. This parallel processing of BIU and EU eliminates the time ceded to fetch many of the instructions. This results in efficient use of the system bus and significantly improve the system performance.

5.3 ASSEMBLER DIRECTIVES AND OPERATORS

Directives are statements written in the source program but are meant for use by the assembly language program. Assembly directives may be divided into several groups, such as data directives, segment control directives, modular programming directives, primary usage directives, etc.

Data Directives

The primary function of the directives in the data group is to define values for constants, variables and labels. Other functions that can be performed by them are assigning a size to a variable and reserving storage locations in memory.

The directive equate and equal to are used to assign a constant value to a symbol. The value of the operand may also be assigned using the arithmetic, relational and logical expressions. The other directives are Define Byte(DB), Define Word (DW) and Define Double Word (DDW). The functions of these directives is to define the size of the variables as being byte, word or double word of memory.

Segment Control Directives

Memory of the 8088/8086-based microcomputer is portioned into the code, data and stack segment. Using segment control directives, the statements of the source program can be portioned and assigned to a specific memory segment. These directives can be used to specify the beginning and end of a segment in a source program and to assign to them attributes, such as a start address, the kind of segment and the method for conbining a segment with other segments of the same name. The beginning of the segment is identified by the segment directive and the end is marked by the end of segment directive.

Modular Programming Directives

For the purpose of development, large programs are broken down into small segments called modules. Each module implements a specific function and has its own code segment and data segment. However, it is common that during execution of a module, some other part of the module may need to be accessed for processing. The Microsoft Macro Assembler (MASM) provides modular programming directives.

Directives for Memory Usage

If the machine code generated by the assembler must reside on a specific part of the memory address space, an origin (ORG) directive can be used to specify the starting point of that memory area.

End of Program Directive

The (END) directive tells the assembler when to stop assembling. It must be included at the end of the source program.

Directive for Program Listing Control

The purpose of listing control directives is to give the programmer some options related to the way in which source program listings are produced by the assembler.

The page directive lets us set the page width and length of the source listing produced as a part of the assembly process.

The most commonly used directives are discussed as follows:

1. **ORG (Origin)**
   - Indicate the beginning of the offset address.
2. **DB (Define Byte)**
   - Allocate memory in byte-sized chunks
   - ' ' indicates ASCII strings
   - Example Program is Given below

```
DATA1          DB        25              ; Decimal
DATA2          DB 10001001B              ; Binary
DATA3          DB         12H            ; HEX
               ORG       0010H
DATA4          DB        '2591'          ; ASCII Numbers
               ORG 0018H
DATA5          DB ?                      ; Set aside a byte
ORG            0020H
DATA6          DB         'Hello world' ; ASCII Characters
```

3. **DUP (DUPlicate)**
   - Duplicate a given number of characters.
   - The Sample Program is Given Below

```
ORG    0030H
DATA7DB       0FFH, 0FFH, 0FFH            ; Fill 3 bytes with FFH
              ORG    38H
```

```
DATA8DB 3   DUP (0FFH)                ; Fill 3 bytes with FFH
            ORG    40H
DATA9DB 32  DUP (?)                   ; Set aside 32 bytes
DATA10       DB 5 DUP (2 DUP (99H))   ; Fill 10 bytes with 99H
```

4. **DW (Define Word)**
   - Allocate memory 2 bytes (one word) at a time.
   - The sample program is as follows:

```
DATA11‘    DW     954                    ; Decimal
DATA12     DW     100110101110B          ; Binary
DATA13     DW 2   53FH                    ; HEX
           ORG    78H
DATA14     DW     9,2,7,0CH,00100001B    ; Miscellaneous data
DATA15     DW     8 DUP(?)                ; Set aside 8 words
```

5. **EQU (EQUate)**
   - Defines a constant without occupying a memory location.
   - The sample program is as follows

```
COUNT        EQU    25H
MOV CX, COUNT
```

6. **DD (Define Doubleword)**
   - Allocates 4-byte memory locations.
7. **DQ (Define Quadword)**
   - Allocates 8-byte memory locations.
8. **DT (Define Ten Bytes)**
   - It is used for memory allocation of packed BCD numbers.
   - The example for DD, DQ and DT is as follows:

```
             DD      1023                       ; Decimal
DATA17       DD 1    0010010110011000110B       ; Binary
DATA18       DQ      5C348FH                    ; HEX
DATA19       DT      123922910650               ; BCD
DATA20       DT       ?                         ; Nothing
```

9. **PAGE [Lines], [Column]**
   - Tells the printer how the list should be printed.
   - Default setting PAGE (no numbers after it): 66 lines per page and max. 80 characters
   - PAGE 60,132 the range of number of lines = 10 to 255 and range of columns is = 60 to 132.
10. **TITLE**
    - Puts the name of the program and a brief description of the function of the program.
11. **.STACK**
    - Marks the beginning of the stack segment

12. **.DATA**
    - Marks the beginning of the data segment
13. **.CODE**
    - Marks the beginning of the code segment
14. **NAME**
    - The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.
    - The statement NAME PC_BOARD, might be used an assembly module, which contains the instructions for controlling the temperature in a bulding
15. **PTR- Pointer**
    - The PTR operator is used to assign a specific type to a variable or to a label.
    - It is necessary to do this in any instruction, where the type of operand is not clear.
    - When the assembler reads an instruction INC[BX], for example, it is not clear whether to increment the byte pointed by BX or a word pointed by BX. The statement INC WORD PTR [BX] removes this confusion and tell that you specifically want the word to be incremented.
    - The PTR operator can also be used to override the declared type of the variable.
16. **PROC (Procedure)**
    - The PROC directive is used to specify the start of the procedure. This directive follows a name you give to a procedure.
    - After the PROC directive, the term NEAR or the term FAR is used to specify the type of the procedure.
    - For example, the statement SMART_DIVIDE PROC FAR, identifies the start of a procedure named SMART_DIVIDE and tells the assembler that it is a FAR procedure.
17. **Public**
    - Large programs are usually written as several separate modules. Each module is individually assembled, tested and debugged.
    - When all modules are working together, their object code files are linked together to form the complete program.
    - In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared Public by using the Public Directive.

NOTES

18. **OFFSET**
    - OFFSET is used to tell the assembler to determine the OFFSET or displacement of the named data item or the procedure which contains this.
    - This operator is used to load the offset of a variable into a register, so that the variable can be accessed by using an indexed addressing mode.
19. **TYPE**
    - The TYPE operator is used to specify the type of the variable. The assembler actually determines the number of bytes in the Type of the variable to reserve the Space.
    - For example, it can be used as:

      ADD BX, TYPE WORD_ARRAY, where you want to increment BX to point to the next word in an array of word.
20. **SHORT**
    - The SHORT Operator is used to tell the assembler that only one byte displacement is needed to code a jump instruction. Thus the assembler will automatically reserve two bytes for the instruction followed.

5.4 MACHINE CODING AND PROGRAMS

To convert an assembly language program to a machine code, you must convert each assembly language instruction to its equivalent machine code instruction. In general, the machine code for an instruction specifies things, such as what operation is to be performed on byte or what operand or operands are to be used, whether the operation is performed on byte or word data, whether the operation involves operands that are located in registers or a register and a storage location in memory, and if one of the operands is in memory, how its address is to be generated. All this information is encoded into the bits of the machine code for the instructions. The machine code instructions of the 8086 vary in the number of the bytes used to encode them. Some instructions can be encoded with just one byte, others in two bytes and many require even more. The machine code for instructions can be obtained by following the formats used in encoding the instructions of the 8086 microprocessor. Most multibyte instructions use the general-instruction format (Refer Figure 5.5). You can see that byte 1 contains three kinds of information: the operation code (opcode), the register direction bit (D) and the data size bit (W). The summary of the function of each of these pieces of information is as follows:

- Opcode occupies six bits and it defines the operation to be carried out by the instruction.
- D occupies one bit. It defines whether the register operand in byte 2 is the source or destination operand.
  - o If D=1, it specifies that the register operand is the destination operand.
  - o If D=0, it indicates that the register is a source operand.

- W defines whether the operation to be performed is an 8 bit or 16 bit data.
  - o If W=0, it indicates 8 bit operation.
  - o If W=1, it indicates 16 bit operation.

Note: The 8086 instruction sizes vary from one to six bytes.

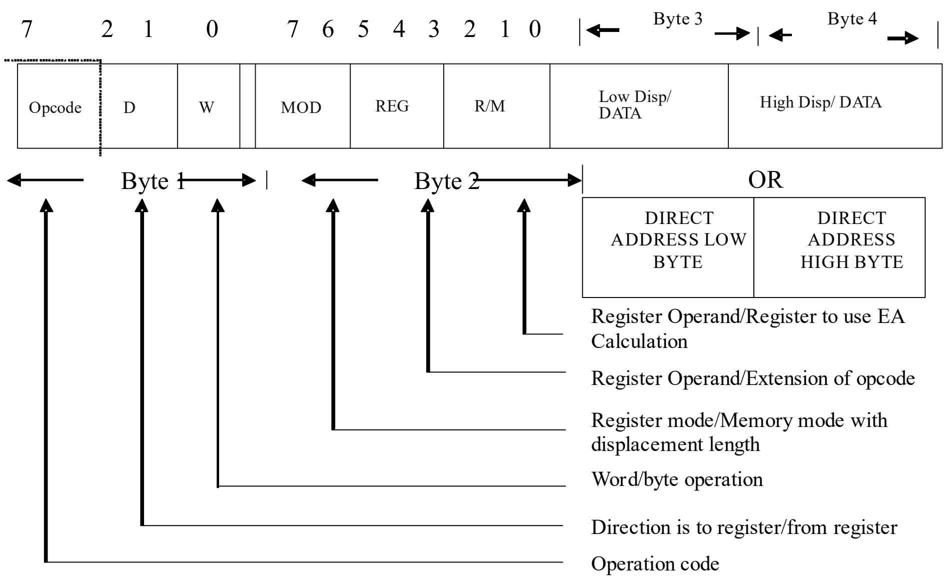


Fig. 5.5 *General-Instruction Format Used by Multibyte*

The second byte of the instruction usually identifies whether one of the operands is in memory or whether both are registers. This byte contains three fields. These are the mode (MOD) field, the register (REG) field and the register/ memory (R/M) field.

Tables 5.1–5.3 explan MOD, REG and R/M fields.

Table 5.1 *MOD Field*

MOD (2 bits)	*Interpretation*
00	Memory mode with no displacement follows except for 16 bit displacement when R/M=110
01	Memory mode with 8 bit displacement
10	Memory mode with 16 bit displacement
11	Register mode (no displacement)

Register field occupies 3 bits. It defines the register for the first operand which is specified as source or destination by the D bit.

Table 5.2 *REG Field*

REG	*W=0*	*W = 1*
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP

NOTES

The R/M field occupies three bits. The R/M field along with the MOD field defines the second operand as follows:

Table 5.3 *Encoding of R/M Field*

R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

In Table 5.4, encoding of the R/M field depends on how the mode field is set. If MOD=11 (register to register mode), this R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Bytes 3 through 6 of an instruction are optional fields that normally contain the displacement value of a memory operand and / or the actual value of an immediate constant operand.

Table 5.4 *Encoding of R/M Field*

R/M	MOD=00	MOD 01	MOD 10
000	(BX) + (SI)	(BX)+(SI)+D8	(BX)+(SI)+D16
001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16
010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16
011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D10
100	(SI)	(SI) + D8	(SI) + D16
101	(DI)	(DI) + D8	(DI) + D16
110	Direct address	(BP) + D8	(BP) + D16
111	(BX)	(BX) + D8	(BX) + D16

Example 5.1: MOV CH, BL

This instruction transfers 8-bit content of BL.

The 6-bit opcode for this instruction is 100010_2. D bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand.

D=0 indicates BL is a source operand.

W=0 byte operation

In byte 2, since the second operand is a register MOD field is 11_2.

The R/M field = 101 (CH)

Register (REG) field = 011 (BL)

Hence the machine code for MOV CH, BL is

10001000	11 011 101
Byte 1	Byte2

= 88DD16

NOTES

Example 5.2: SUB BX, (DI)

This instruction subtracts the 16-bit content of memory location addressed by DI and DS from BX. The 6-bit opcode for SUB is 001010_2.

D = 1 so that REG field of byte 2 is the destination operand. W=1 indicates 16-bit operation.

MOD = 00
REG = 011
R/M = 101

The machine code is $\underset{2}{\underline{0010}}\ \underset{B}{\underline{1011}}\ \underset{1}{\underline{0001}}\ \underset{D}{\underline{1101}}$

$2B1D_{16}$

MOD / R/M	Memory Mode (EA Calculation)			Register Mode	
	00	**01**	**10**	**W=0**	**W=1**
000	(BX)+(SI)	(BX)+(SI)+d8	(BX)+(SI)+d16	AL	AX
001	(BX) + (DI)	(BX)+(DI)+d8	(BX)+(DI)+d16	CL	CX
010	(BP)+(SI)	(BP)+(SI)+d8	(BP)+(SI)+d16	DL	DX
011	(BP)+(DI)	(BP)+(DI)+d8	(BP)+(DI)+d16	BL	BX
100	(SI)	(SI) + d8	(SI) + d16	AH	SP
101	(DI)	(DI) + d8	(DI) + d16	CH	BP
110	d16	(BP) + d8	(BP) + d16	DH	SI
111	(BX)	(BX) + d8	(BX) + d16	BH	DI

Example 5.3: Code for MOV 1234 (BP), DX

Here you have to specify DX using REG field. The D bit must be 0, indicating the DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. 1234 [BP] is specified using MOD value of 10 and R/M value of 110 and a displacement of 1234H. The 4-byte code for this instruction would be 89 96 34 12H.

Opcode	D	W	MOD	REG	R/M	LB displacement	HB displacement
100010	0	1	10	010	110	34H	12H

Example 5.4: Code for MOV DS : 2345 [BP], DX

Here you have to specify DX using REG field. The D bit must be o, indicating that Dx is the source register. The REG field must be 010 to indicate DX register. The you bit must be 1 to indicate it is a word operation. 2345 [BP] is specified with MOD=10 and R/M = 110 and displacement = 2345 H.

Whenever BP is used to generate the Effective Address (EA), the default segment would be SS. In this example, you want the segment register to be DS, so, you have to provide the Segment Override Prefix Byte (SOP byte) to start with. The SOP byte is 001 SR 110, where SR value is provided as per the following table.

NOTES

SR	Segment register
00	ES
01	CS
10	SS
11	DS

To specify DS register, the SOP byte would be 001 11 110 = 3E H. Thus, the 5-byte code for this instruction would be 3E 89 96 45 23 H.

SOP	Opcode	D	W	MOD	REG	R/M	LB disp.	HD disp.
3EH	1000 10	0	1	10	010	110	45	23

Suppose you want to code MOV SS : 2345 (BP), DX. This generates only a 4-byte code, without SOP byte, as SS is already the default segment register in this case.

Example 5.5: Give the instruction template and generate code for the instruction ADD OFABE [BX], [DI], DX (code for ADD instruction is 000000)
ADD OFABE [BX] [DI], DX

Here you have to specify DX using REG field. The bit D is 0, indicating that DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. FABE (BX + DI) is specified using MOD value of 10 and R/M value of 001 (from the summary table). The 4-byte code for this instruction would be as follows:

Opcode	D	W	MOD	REG	R/M	16 bit disp.		= 01 91 BE FAH
000000	0	1	10	010	001	BEH	FAH	

Example 5.6: Give the instruction template and generate the code for the instruction MOV AX, [BX]

(Code for MOV instruction is 100010)

AX destination register with D=1 and code for AX is 000 [BX] is specified using 00 Mode and R/M value 111

It is a word operation.

Opcode	D	W	Mod	REG	R/M	=8B 07H
100010	1	1	00	000	111	

Example 5.7: Code for MOV BL,AL

Opcode for MOV = 100010

D = 0 (AL source operand)

W bit = 0 (8-bits)

Therefore, byte 1 is 100010002=8816

• MOD = 11 (register mode)

• REG = 000 (code for AL)

• R/M = 011 (destination is BL)

Therefore, byte 2 is 110000112=C316
Similarly you can see that:
MOV BL, AL = 10001000 11000011 = 88 C3h
ADD AX, [SI] = 00000011 00000100 = 03 04 h
ADD [BX] [DI] + 1234h, AX = 00000001 10000001 = 01 81 34 12 h

NOTES

Check Your Progress

1. Write the functions of BIU.
2. Define the term EU.
3. Define the term flag register.
4. What are directives?
5. Write the purpose of listing control directives.
6. What does a machine code specify?

5.5 PROGRAMMING WITH AN ASSEMBLER

To learn assembly language programming the beginner should write simple programs. The memory addresses given in the program are for a particular microprocessor kit. These addresses can be changed to suit the microprocessor bit available in the laboratory. Before writing an assembly language program, one should learn some important Intel 8085 instructions, such as `MOV`, `MUI`, `ADD`, `SUB`, `LXI`, `LDA`, `INX`, `INR` and `HLT`.

Example 5.8: Object: Place 05 in register B.

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
FC 00	06, 05	MUI	B, 05	Get 05 in register B
FC 02	76	HLT		Stop

The instruction MUI B, 05 moves 05 to register B. HLT halts the program. A program is fed to up kit in machine codes. The machine code for the instruction MVI B, 05 is 06, 05. The first byte of the instruction is 06 which is the machine code for the instruction MVI B. The second byte of the instruction, 05 is the dates which is to be moved to register B. The code for HLT is 76. The machine codes for a program are entered in the memory. In this case the memory addresses from FC 00 to FC 02 have been used. The machine code 06 is entered in the memory location FC 00 H; 05 in FC 01 H and 76 in FC 02 H.

Example 5.9: Place 05 in the accumulator. Increment it by one and store the result in the memory location FC 50 H.

NOTES

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
FC 00	3E, 05	MVI	A, 05	Get 05 in the accumulator
FC 02	3C	INR	A	Increment the content of accumulator by one
FC 03	32, 50, FC	STA	FC 50 H	Store result in FC 50 H
FC 06	76	HLT		Halt

The instruction MVI A, 05 moves 05 to the accumulator. INR A increases the content of the accumulator from 05 to 06. STA FC 50 stores the content of the accumulator in the memory location FC 50 H. After the execution of the above program the memory location FC 50 H will contain 06.

Addition of Two 8-bit Numbers; Sum 8-bits

Example 5.10: Add 49 H and 56 H.

The first number 49 H is in the memory location 2501 H.

The 2nd number 56 H is in the memory location 2502 H.

The result is to be stored in the memory location 2503 H.

Numbers are represented in hexadecimal number system.

Program

Memory Address	Machine Codes	Mnemonics	Operands	Comments
2000	21, 01, 25	LXI	H, 2501 H	Get address of 1st number in H-L pair.
2003	7 E	MOV	A, M	1st number in accumulator.
2004	23	INX	H	Increment content of H-L pair.
2005	86	ADD	M	Add 1st and 2nd numbers.
2006	32, 03, 25	STA	2503 H	Store sum in 2503 H
2009	76	HLT		Stop

DATA

2501—49 H

2502—56 H

The sum is stored in the memory location 2503 H.

Result

2503—9F H

2501 H is the address of the memory location for the 1st number. 2501 is placed in H–L pair by the instruction LXI H, 2501 H. The next instruction is MOVA, M moves the content of the memory location addressed by H–L pair to the accumulator. In this case H–L pair contains 2501 H and, therefore, the content of the memory location 2501 H is moved to the accumulator. Thus, the 1st number 49 H has been moved to the accumulator. The instruction INX-H increases the content of H–L pair by one. Previously, the content of H–L pair was 2501 H. After the execution of INX-H it becomes 2502 H. Add M which adds the contents of the accumulator and the content of the memory location addressed by H–L pair. The content of 2502 H is the 2nd number 56 H. So 56 H is added to 49 H. Sum resides in the accumulator. The instruction STA 2503 H stores the sum in the memory location 2503 H. The instruction HLT ends the program.

Addition of Two 8-bit Numbers; Sum 16-bits

Example 5.11: Add 98 H and 9A H.

Sum = 98 H + 9A H = 01, 32 H

The 1st number 98 H is in the memory location 2501 H.

The 2nd number 9A H is in the memory location 2502 H.

The results are to be stored in 2503 H and 2502 H.

In this case the sum is to be stored in two consecutive memory locations. The LSBs of the sum is 32 H and it will be stored in the memory location 2503 H. The MSB of the sum is 01 which will be stored in 2504 H.

Program

Memory Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	21, 01, 25		LXI	H, 2501 H	Address of 1st number in the H–L pair.
2003	0E, 00		MVI	C, 00	MSBs of sum in register C. Initial value = 00.
2005	7E		MOV	A, M	1st number in accumulator.
2006	23		INX	H	Address of 2nd number 2502 in N–L pair.
2007	86		ADD	M	1st number + 2nd number.
2008	D2, 0C, 20		JNC	AHEAD	Is carry? No, go to the label AHEAD.
200B	0C		INR	C	Yes, increment C.
200C	32, 03, 25	AHEAD	STA	2503 H	LSBs of sum in 2503H.
200F	79		MOV	A, C	MSBs of sum in accumulator.
2010	32, 04, 25		STA	2504 H	MSBs of sum in 2504H.
2013	76		HLT		Halt.

8-Bit Multiplication; Product 16-Bit

Decimal multiplication. For decimal multiplication the following procedure is followed:

Example 5.12: 36D, Multiplicand
× 29D, Multiplier
324
72

Product: 1044D $= 1044_{10}$

Binary Multiplication

Example 5.13: Multiply 7 by 5.

First, 7 and 5 are represented in binary form:

$7_{10} = 0111_2$ and $5_{10} = 0101_2$

```
    0111₂, Multiplicand
  × 0101₂, Multiplier
    0111
   0000
  0111
 0000
 0100011   = 35₁₀
```

Example 5.14: Multiply 84 H by 56 H.

Here, 84 H is the multiplicand and 56 H is the multiplier; 84 H is extended to 16-bits and stored in the two consecutive memory locations 2501 and 2502 H. 56 H is stored in 2503 H. The product is a 16-bit number and it is stored in 2504 H and in 2505 H. Data and results are in hexadecimal. The program flow chart is shown in Figure 5.6.

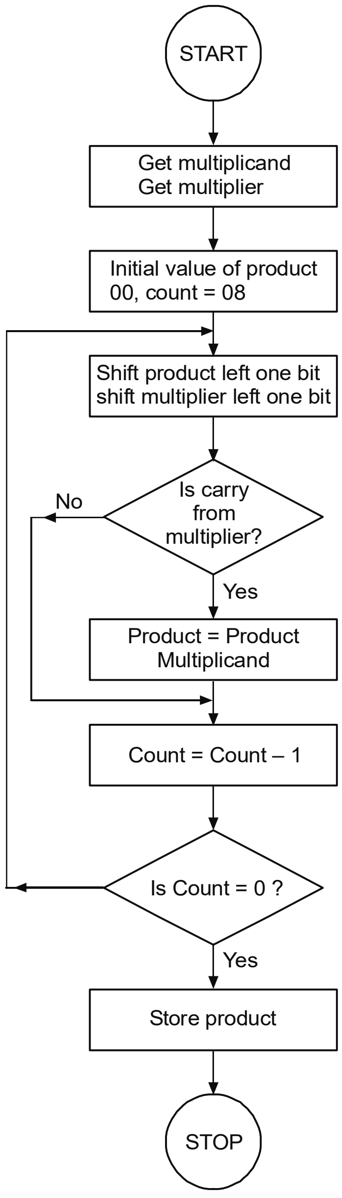


Fig. 5.6 Program Flow Chart for a 8-bit Multiplication

Program

Memory Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	2A, 01, 25		LHLD	2501 H	Get multiplicand in H–L pair.
2003	EB		XCHG		Multiplicand in D–E pair.
2004	3A, 03, 25		LDA	2503 H	Multiplier in accumulator.
2007	21, 00, 00		LXI	4,000	Initial value of product = 00 in H–L pair.
200A	0E, 08		MVI	C, 08	Count = 8 in register C.
200C	29	Loop	DAD	H	Shift partial product left by 1 bit.
200D	17		RAL		Rotate multiplier left one bit. Is multiplier's bit = 1?
200E	D2, 12, 20		JNG	AHEAD	No, go to AHEAD.
2011	19		DAD	D	Product = Product + Multiplicand
2312	OD	AHEAD	DCR	C	Decrement Count.
2013	C2, OC, 20		JNZ	Loop	
2016	22, 04, 25		SHLD	2504 H	Store result.
2019	76		HLT		Stop.

The instruction LHLD 2501H transfers 16-bit multiplicand from the memory locations 2501 and 2502 H to H–L pair. By the execution of the instruction XCHG the contents of H–L pair are exchanged with the contents of D–E pair. Thus, the multiplicand is placed in D–E pair. The instruction LDA 2503 H transfers multiplier from the memory location 2503 H to the accumulator. LXIH, 0000 makes the initial value of the product equal to zero and it is placed in H–L pair. The count is equal to the bits of the multiplier. In this case it is 08 and it is placed in register C. DADH is an instruction for 16-bit addition. It adds the contents of H–L pair to itself. Thus, the partial product which is in H–L pair is shifted left by one bit. RAL rotates the content of the accumulator to the left by one bit. The accumulator contains multiplier and hence it is rotated left by one bit. The instruction DAD D adds the content of D–E pair and H–L pair and places the result in H–L pair. D–E pair and H–L pair contains multiplicand and partial product respectively. Thus, the execution of the instruction DAD D adds the multiplicand to the partial product

NOTES

and places the sum which is the new partial product in H–L pair. The instruction DAD D is executed only when the bit of the multiplicand under consideration is 1, otherwise, it is not executed. To get the result the program moves in the LOOP 8 times as there are 8 bits in the multiplier.

Example 5.15

DATA

2501—84 H, LSBs of multiplicand

2502—00, MSBs of multiplicand

2503—56 H, Multiplier

Result

2504—58 H, LSBs of product

2505—26 H, MSB of product

Example 5.16

DATA

2501—84 H, LSBs of multiplicand

2502—00, MSBs of multiplicand

2503–52 H, Multiplier

Result

2504—34 H, LSBs of product

2505—2C H, MSBs of product

5.6 ASSEMBLY LANGUAGE EXAMPLE PROGRAMS

In this section, you will learn about the process by which the problems are solved using the software. An assembly language program is written to solve a specific problem. This problem is known as the application. To develop a program that implements an application, the programmer goes through a multi-step process. Figure 5.7 outlines the steps in the program-development cycle.

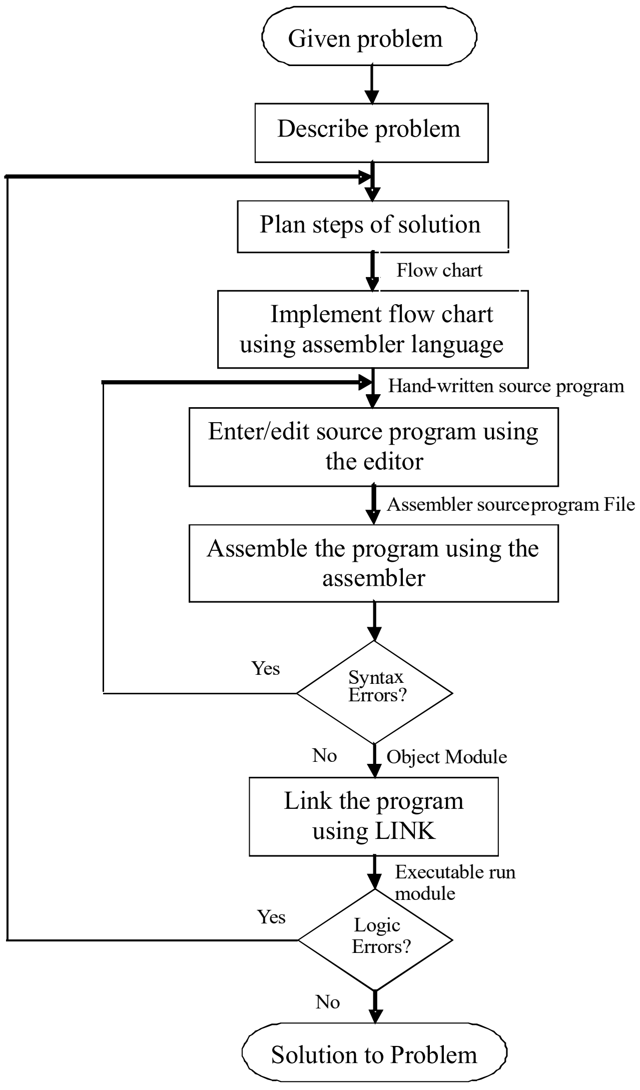

Given problem
Describe problem
Plan steps of solution
Flow chart
Implement flow chart using assembler language
Hand-written source program
Enter/edit source program using the editor
Assembler source program File
Assemble the program using the assembler
Yes
Syntax Errors?
No
Object Module
Link the program using LINK
Executable run module
Yes
Logic Errors?
No
Solution to Problem

Fig. 5.7 *Step in the Program Development Cycle*

The various steps involved in developing a program are discussed as follows.

1. Describing the Problem

The development cycle sequence begins by making a clear description of the problem to be solved and ends with a program that, when run, performs a correct solution. First, the programmer must understand and describe the problem that is to be solved. A clear, concise and accurate description of the problem is an essential part of the process of obtaining a correct and efficient software solution.

The program used here is an example of a simple software application. Its function is to move a fixed-length block of data, called the *source block*, from one location in memory to another location in memory, called the *destination block*. For the block- move program, a verbal or a written list of events may be used to describe this problem to the programmer.

On the other hand, in most practical applications, the problem to be solved is quite complex. The programmer must know what the input data are, what operations must be performed on this information, whether or not these operations need to be performed in a special sequence, whether or not there are time constraints on performing some of the operations, and if error conditions can occur during the process, and what results need to be output. For this reason, most applications are described with a written document called an application specification. The programmers study this specification before they begin to define a software solution for the problem.

2. Planning the Solution

Before writing an application program, a plan must be developed to solve the problem. Figure 5.7 shows that this is the second step in the program development process. The decision to move to this step assumes that a complete and clear description of the problem to be solved has been provided.

The programmer carefully analyses the application specification. Typically, the problem is broken into a series of basic operations, which, when performed in a certain sequence, produce a solution to the problem. This plan defines the method by which software solves the problem. The software plan is known as the *algorithm*.

Usually, the algorithm is described with another document called the *software specification*. Also, the proposed solution may be presented in a pictorial form known as a *flowchart* in the specification. A flowchart is an outline that both documents the operations the software must perform to implement the planned solution and shows the sequence in which they are performed. Figure 5.8 is the flowchart for a program that performs a block-move operation.

The flowchart identifie operations that can be implemented with assembly language instructions. For example, the first block calls for setting up a data segment, initializing the pointers for the starting address of the source and destination blocks, and specifying the count of the number of pieces of data that are to be moved. These types of operations can be achieved by moving either immediate data or data from a known memory location, into appropriate registers within the microprocesor.

A flowchart uses a set of symbols to identify both the operations required in the solution and the sequence in which they are performed. The operation to be performed is listed inside the symbol. Arrows are used to describe the flow of these operations as the block-move operation is performed.

The solution should be hand-tested to verify that it correctly solves the stated problem. Specifying test cases with known inputs and outputs can do this. Then, tracing through the operation sequence defined in the flowchart for these input conditions, the outputs are found and compared to the known test results. If the results are not the same, the cause of the error is found, the algorithm is modified and the tests are rerun. When the results match, the algorithm is assumed to be correct and the programmer is ready to move on to the next step in the development cycle. The process is called desk checking.

The flow chart representation of the planned solution is valuable aid to the programmer while coding the solution with assembly language instructions. When a problem is a simple one, the flowcharting step may be bypassed. However, for complex applications, a flowchart is an important program-development tool for obtaining an accurate and timely solution.

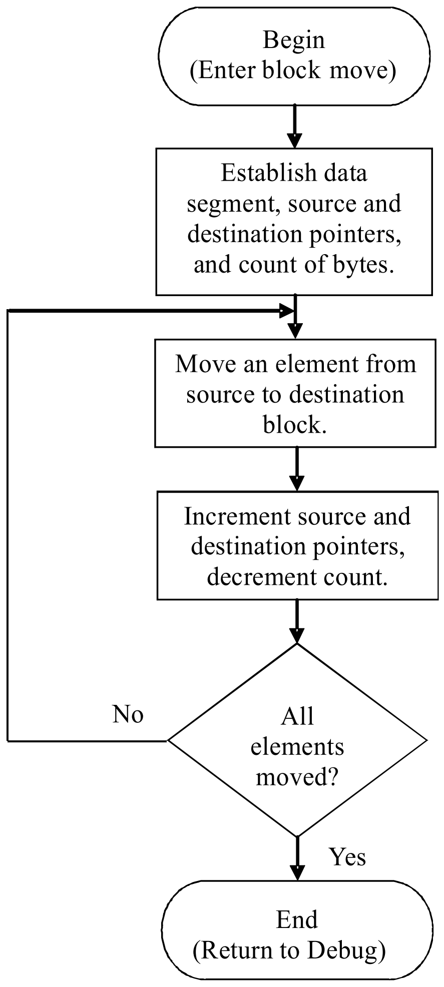

Begin
(Enter block move)
Establish data segment, source and destination pointers, and count of bytes.
Move an element from source to destination block.
Increment source and destination pointers, decrement count.
All elements moved?
No
Yes
End
(Return to Debug)

Fig. 5.8 *Flow Chart Representation of the Planned Solution*

3. Coding the Solution with Assembly Language

The application program is the step-by-step sequence of computer operations that must be performed to convert the input data to the required output results. In other words it is the software implementation of the algorithm. The third step of the program development cycle is the translation of the flowchart solution into its equivalent assembly language program. This requires the programmer to implement the operations described in each symbol of the flowchart with a sequence of assembly language instructions. These instruction sequences are then combined to form a hand written assembly language programs called the source program.

Two types of statements are used in the source program.

(i) There are the assembly language instructions. They are used to tell the microprocessor what operations are to be performed to implement the application. The program for this block-move operation is shown below. Comparing the program to the flowchart, it is easy to see that the initialization block is implemented with the assembly language statements

```
MOV AX, DATASEGADDR
MOV DS, AX
MOV SI, BLK1ADDR
MOV DI, BLK2ADDR
MOV CX, N
```

The first two move instructions load a segment base address called DATASEGADDR into the data segment register. This defines the data segment in memory where the two blocks of data reside. Next two more instructions are used to load SI and DI with the start offset address of the source (BLK1ADDR) and destination block (BLK2ADDR), respectively. Finally, the count N of the number of the bytes of data to be copied to the destination block is loaded into count register CX.

(ii) A source program can also contain another type of statement called the *directives*, which are the instructions to the assembler program that is used to convert the assembly language program into the machine code. The statements like BLOCK PROC FAR and BLOCK ENDP are examples of modular programming directives. They mark the beginning and end, respectively, of the software procedure called BLOCK.

To accomplish the third step of the development cycle, the programmer must know the instruction set of the microprocessor, basic assembly language programming techniques, the assembler's instruction statement syntax and the assembler's directives.

4. Creating the Source Program

After having handwritten the assembly language program, you are ready to enter into the computer. This step is identified as the enter/edit source program block in the program-development cycle (Refer Figure 5.7) and is done with a program called an *editor*. Using an editor, each of the statements of the program is typed into the computer. If errors are made as the statements are keyed in, the corrections can either be made at the time of entry or edited at a later time. The source program is saved in a file.

5. Assembling the Source Program into an Object Module

The fifth step of the flowchart is the point at which the assembly language source program is converted to its corresponding machine language program. To do this, you use a program called an *assembler*. A program originally available from Microsoft Corporation called MASM is an example of an 8088/8086 assembler that runs in DOS on a PC. The assembler program reads as its input the contents of the *assembler source file;* it converts this program statement by statement to machine code and produces a machine-code program as its output. This machine-code output is stored in a file called the *object module.*

If during the conversion operation, syntax errors are found, i.e, there are violations in the rules of writing the assembly language statements for the assembler- the assembler automatically flags them. As shown in Figure 5.7 before going on, the cause of each error in the source program must be identified and corrected. The corrections are made using the editor program. After the corrections are made, the source program must be reassembled. This edit-assemble sequence must be repeated until the program assembles with no error.

6. Producing a Run Module

The object module produced by the assembler cannot be run directly on the microcomputer. As shown in the figure above, a LINK program must process the module to produce an executable object module, which is known as a *run module*. The linker program converts the object module to a run module by making it-address compatible with the microcomputer on which it is to be run. For instance, if your computer is implemented with memory at addresses $0A000_{16}$, through $0FFFF_{16}$ the executable machine-code output by the linker will also have addresses in the range.

There is another purpose for the use of a linker: it links different object modules to generate a single executable object module. This allows program development to be done in modules, which are later combined to form the application program.

7. Verifying the Solution

Now the executable object module is ready to be run on the microcomputer. Once again, the PC's DOS operating system provides you with a program, called DEBUG, to perform this function. DEBUG provides environment in which you can run the program instruction or run a group of instructions at a time, look at intermediate results, display the contents of the registers within the microprocessor, and so on.

For instance, you could verify the operation of your earlier block-move program by running it for the data in the cases defined to test the algorithm. DEBUG is used to load the run module for block-move into PC's memory. After loading is completed and verified, other DEBUG commands are employed to run the program for the data in the test case. THE DEBUG program permits you to trace the operation as instructions are executed and observe each element of data as it is copied from the source to the destination block. These results are recorded and compared to those provided with the test case. If the program is found to perform the block-move operation correctly, the program-development process is complete.

On the other hand, the flow-chart shows that if errors are discovered in the logic of the solution, the cause must be determined, corrections must be made to the algorithm, and then the assembly language source program must be corrected using the editor. The edited source file must be reassembled re-linked and retested by running it with DEBUG. This loop must be repeated until it is verified that the program correctly performs the operation for which it was written.

Programs and Files Involved in the Program Development Cycle

The edit, assemble, link and debug parts of the general program development cycle in starting flow chart are performed directly on the PC. The flow chart shows the names of the programs and typical filenames with extensions used as inputs and outputs during this process. For example, the EDIT program is an editor used to create and correct assembly language source files. The program that results is shown to have the name PROG1.ASM. This stands for program 1 assembly source code.

MASM is a program that can be used to assemble source files into object modules. The assembler converts the contents of the source input file PROG1.ASM into two output files called PROG1.OBJ and PROG1.LST. The file PROG1.OBJ contains the object code module. The PROG1.LST file provides additional information useful for debugging the application program.

Object module PROG1.OBJ can be linked to other object modules with the LINK program. For instance, programs that are available as object modules in a math library could be linked with another program to implement math operations. A library is a collection of prewritten, assembled, and tested programs. Notice that this program produces a run module in file PROG1.EXE and a map file called PROG1.MAP as outputs. The executable object module, PROG1.EXE, is run with the debugger program, called DEBUG. Map file PROG1.MAP is supplied as support for the debugging operation by providing additional information, such as where the program will be located when loaded into the microcomputer's memory.

Figure 5.9 shows the programs and files involved in the program development cycle.

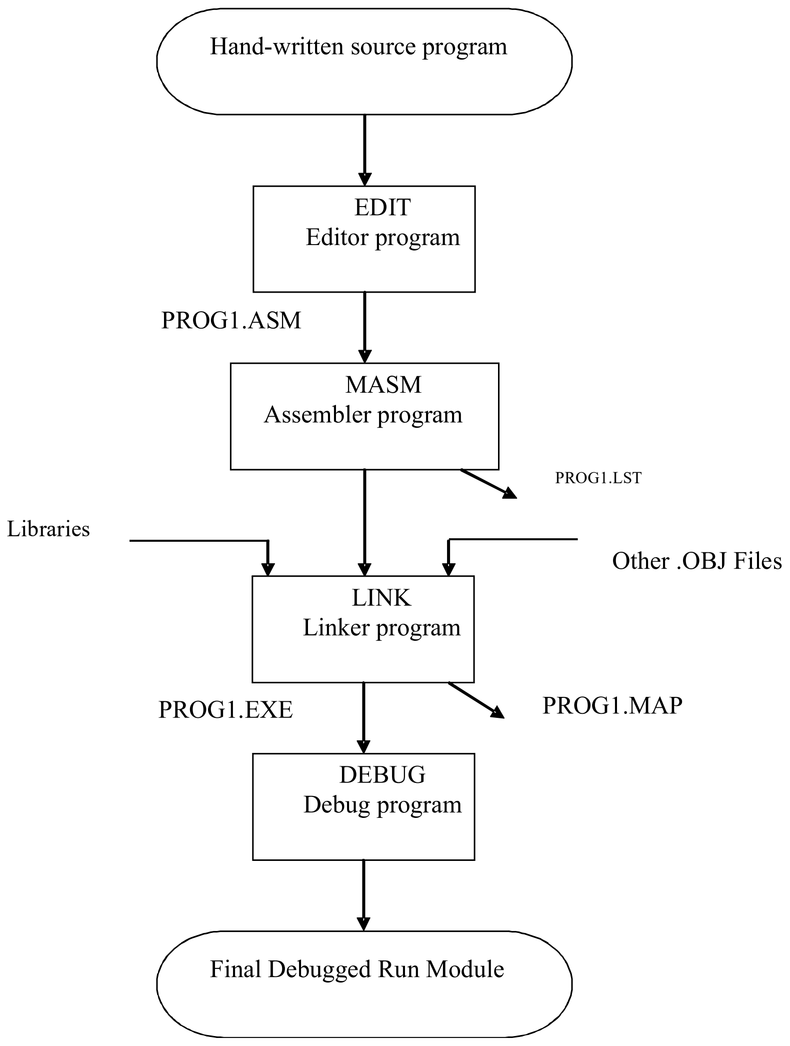


Fig. 5.9 *Programs and Files Involved in the Program Development Cycle*

The various steps in working with the assembly language can be illustrated with the the help of Figure 5.10.

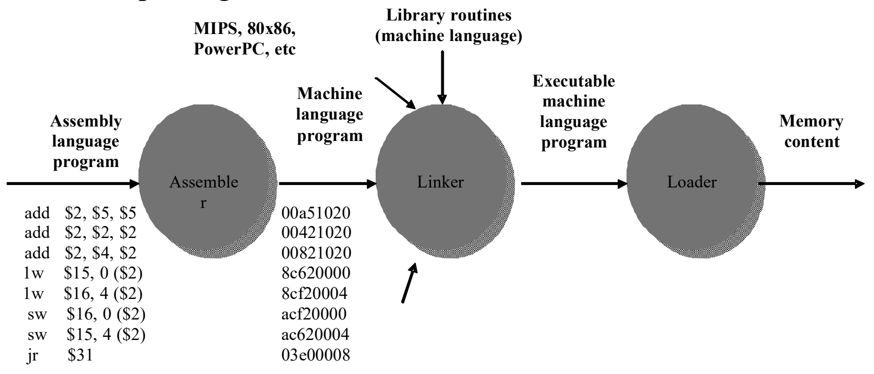


Fig. 5.10 *Steps in Working with Assembly Language*

The various extensions of the files used in this process are as follows:

- . **asm** : the source file
- . **obj** : the assembler converts the .asm file into machine code (object file).
- . **lst** : (optional) it lists all the opcodes and offset addresses as well as errors that the assembler detected.
- . **crf** : (optional) the cross-reference lists all the symbols and labels used in the program.
- . **map** : it gives the name of each segment, where it starts, where it stops, and its size in bytes.
- . **exe** : It can be executed by microprocessor. (ready-to-run version of a program)

The generalized format for the assembly-level program is as follows:

Label: Mnemonic Operands; Comment

In this regard, the important points to be noted are:

- Label cannot exceed 31 characters and must be unique.
- The mnemonic (instruction) and operands fields is used for the real tasks.
- Pseudo-instructions called 'Directive' give directions to the assembler on how it translate the instructions into a machine code.
- Comment field begins with a ";".

Program 5.1 illustrates the shell of an assembly language program.

Program 5.1

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
STSEG SEGMENT
            DB 64 DUP(?)
STSEGENDS
;– – – – – – – – – – – – – – – –
DTSEG       SEGMENT
;
;place data here
;
```

NOTES

```
DTSEG ENDS
;- - - - - - - - - - - - - - - - -
CDSEG          SEGMENT
MAIN           PROC FAR ;this is the program entry point
               ASSUME CS:CDSEG, DS:DTSEG, SS:STSEG
               MOV AX,DTSEG ;bring in the segment for data
               MOV DS,AX ;assign value to DS
;
;place code here
;
               MOV AH, 4CH ;return control to operating system
               INT 21H ;return to DOS
MAIN           ENDP
CDSEG          ENDS
               END MAIN ; this is the program exit point
```

Program 5.2 illustrates the actual assembly language program.

Program 5.2

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM TO SUM TWO
;NUMBERS IN MEMORY AND STORE THE RESULT
STSEG          SEGMENT
               DB 64 DUP(?)
STSEG          ENDS
;- - - - - - - - - - - - - - - - -
DTSEG          SEGMENT
DATA1          DB          52H
DATA2          DB          29H
SUM            DB          ?
DTSEG          ENDS
;- - - - - - - - - - - - - - - - -
CDSEG          SEGMENT
MAIN           PROC FAR                ;this is the program entry point
ASSUME         CS:CDSEG, DS:DTSEG, SS:
                STSEG MOV AX,DTSEG     ;load the data segment address
               MOV DS,AX               ;assign value to DS
               MOV AL,DATA1            ;get the first operand
               MOV BL,DATA2            ;get the second operand
               ADD AL,BL               ;add the operands
               MOV SUM,AL              ;store result in location SUM
               MOV AH,4CH              ;set up to
               INT 21H                  ;return to DOS
MAIN           ENDP
CDSEG          ENDS
               END MAIN                ;this is the program exit point
```

NOTES

Program 5.3

```
DOSSEG
IDEAL
MODEL SMALL
;- - - - - - - - - - - - - - - - - - - - - - - - - - - -
SEGMENT DDD
string1 db "chulalongkorn$"
inchar db "a"
org 0010h
poschar db ?
ENDS
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; Find 1st char. in string1 that matches the char.
; in inchar and store its position in poschar.
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
SEGMENT CCC
ASSUME CS:CCC,DS:DDD
MAIN:
MOV AX,DDD
MOV DS,AX
; begin your code
mov si,offset string1 ; si is the data pointer
mov al,[inchar] ; al will hold the ASCII
; code of letter "a"
mov cl,1 ; set count to 1
comp:
cmp [si], al ; compare characters
je find ; if equal goto find
inc cl ; else increment count
inc si ; and index
jmp comp ; then go back to
; compare
find:
mov [poschar],cl ; store count
;End of your code
MOV AH,4CH
INT 21H
ENDS
END MAIN
```

Program 5.4

```
DOSSEG
IDEAL
MODEL SMALL
```

NOTES

```
;_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
SEGMENT DDD
string1 db "chulalongkorn$"
inchar db "a"
endstring db "$"
org 0010h
poschar db ?
ENDS
;_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
; Modify code so that poschar is zero if there is
no match.
; Find 1st char. in string1 that matches the char.
in inchar
; and store its position in poschar.
;_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
SEGMENT CCC
ASSUME CS:CCC,DS:DDD
MAIN:
MOV AX,DDD
MOV DS,AX
; begin your code
mov si,offset string1 ; si is the data pointer
mov al,[inchar] ; al = the ASCII code of letter "a"
mov bl,[endstring] ; bl = ASCII code of letter "$"
mov cl,1 ; set count to 1
comp:
cmp [si], bl ; if end of string
je fail ; if equal goto fail
cmp [si],al
je find
inc cl ; else increment count
inc si ; and index
jmp comp ; then go back to compare
fail:
mov cl, 0
find:
mov [poschar],cl ; store count
;End of your code
MOV AH,4CH
INT 21H
ENDS
END MAIN
```

Check Your Progress

7. What do you mean by the assembly language programming?
8. Define the term program development cycle.
9. What is MASM?

5.7 ANSWERS TO 'CHECK YOUR PROGRESS'

NOTES

1. The BIU performs the following functions:
    - It sends instruction from memory.
    - It reads data from port/memory.
    - It writes data into port/memory.
    - It supports instruction queueing.
    - It provides address relocation facility.
2. Execution Unit (EU) decodes and executes instructions. A decoder in the EU control system translates instructions. It has a 16-bit ALU for performing arithmetic and logic operations. EU may pass the results to the BIU for storing them in memory. It extracts instruction from the top of the queue in the BIU, decodes them, generates operands address, if necessary, passes them to BIU and requests it to perform the read or write bus cycles to memory or I/O and performs the operation specified by the instruction on the operands.
3. A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. The flag register holds the status flags, typically after an ALU operation. The EU sets or resets these flags to reflect certain properties of the results of arithmetic and logic operations.
4. Directives are statements written in the source program but are meant for use by the assembly language program.
5. The purpose of listing control directives is to give the programmer some options related to the way in which source program listings are produced by the assembler.
6. In general, the machine code for an instruction specifies various things, such as what operation is to be performed on byte or what operand or operands are to be used, whether the operation is performed on byte or word data, whether the operation involves operands that are located in registers or a register and a storage location in memory, and if one of the operands is in memory, how its address is to be generated.
7. Assembly language programming the beginner should write simple programs. The memory addresses given in the program are for a particular microprocessor kit. These addresses can be changed to suit the microprocessor bit available in the laboratory. Before writing an assembly language program, one should learn some important Intel 8085 instructions, such as MOV, MUI, ADD, SUB, LXI, LDA, INX, INR and HLT.
8. To develop a program that implements an application, the programmer goes through a multi-step process, called the program development cycle.
9. MASM is a program that can be used to assemble source files into object modules.

5.8 SUMMARY

NOTES

- The 8086 was Intel's first 16-bit microprocessor. It was launched in 1978. Its design is based on the 8080 but is not directly compatible with the 8080.
- The memory in the 8086-based microcomputer is organized as bytes. Each byte can be uniquely addressed with 20-bit addresses of 0000016, 0000116, ... FFFFF16.
- The architecture of the 8086 processor provides a number of improvement over its predessecor 8085. It supports a 16-bit ALU, a set of 16-bit registers and provides segment memory addressing capability, a rich instruction set, powerful interrupt structure and a six byte instruction queue.
- The BIU provides all external bus operations. It contains segment registers, instruction pointer, instruction queue and address generation/ bus control circuitry to provide functions such as the fetching and queueing of instructions and bus control.
- Execution Unit (EU) decodes and executes instructions. A decoder in the EU control system translates instructions. It has a 16-bit ALU for performing arithmetic and logic operations. EU may pass the results to the BIU for storing them in memory. It extracts instruction from the top of the queue in the BIU, decodes them, generates operands address, if necessary, passes them to BIU and requests it to perform the read or write bus cycles to memory or I/O and performs the operation specified by the instruction on the operands.
- A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. The flag register holds the status flags, typically after an ALU operation. The EU sets or resets these flags to reflect certain properties of the results of arithmetic and logic operations.
- Directives are statements written in the source program but are meant for use by the assembly language program.
- Assembly directives may be divided into several groups, such as data directives, segment control directives, modular programming directives, primary usage directives, etc.
- The purpose of listing control directives is to give the programmer some options related to the way in which source program listings are produced by the assembler.
- Memory of the 8088/8086-based microcomputer is portioned into the code, data and stack segment. Using segment control directives, the statements of the source program can be portioned and assigned to a specific memory segment.
- To convert an assembly language program to a machine code, you must convert each assembly language instruction to its equivalent machine code instruction.

- In general, the machine code for an instruction specifies various things, such as what operation is to be performed on byte or what operand or operands are to be used, whether the operation is performed on byte or word data, whether the operation involves operands that are located in registers or a register and a storage location in memory, and if one of the operands is in memory, how its address is to be generated.
- Assembly language programming the beginner should write simple programs. The memory addresses given in the program are for a particular microprocessor kit. These addresses can be changed to suit the microprocessor bit available in the laboratory. Before writing an assembly language program, one should learn some important Intel 8085 instructions, such as MOV, MUI, ADD, SUB, LXI, LDA, INX, INR and HLT.
- To develop a program that implements an application, the programmer goes through a multi-step process, called the program development cycle.
- The development cycle sequence begins by making a clear description of the problem to be solved and ends with a program that, when run, performs a correct solution.
- The application program is the step-by-step sequence of computer operations that must be performed to convert the input data to the required output results. In other words it is the software implementation of the algorithm.

5.9 KEY TERMS

- **Bus Interfacing Unit (BIU):** The part of the Intel 8086 which provides all external bus operations.
- **Execution Unit (EU):** The part of the Intel 8086 which decodes and executes instructions.
- **Flag:** A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU.
- **Directives:** They are statements written in the source program but are meant for use by the assembly language program.

5.10 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write the main features of the 8086.
2. What is meant by flag register? Name the different types of flags.
3. Define the term segment control directives.
4. Give the instruction template and generate code for the instruction ADD OFABE [BX] [DI], DX (code for ADD instruction is 000 000).

5. What is the addition of two 8-bit numbers?
6. Define the term destination block.

Long-Answer Questions

1. Discuss the internal block diagram of the Intel 8086 and label the parts.
2. Briefly explain the functions of the bus interfacing unit.
3. Describe the assembler directives and operators giving appropriate examples.
4. Briefly explain about the machine coding and program with the help of examples.
5. Explain about the program flow chart of 8-bit multiplication process.
6. Discuss the steps involved in the program development cycle with the help of example flow chart.

5.11 FURTHER READING

Goankar, Ramesh. 2002. *Microprocessor Architecture, Programming and Application with the 8085*. Mumbai: Penram International Publishing.

Ram, B. 2005. *Fundamentals of Microprocessor and Microcomputers*. New Delhi: Dhanpat Rai Publications.

Lotia, Manahar and Pradeep Nair. 2003. *Modern All About Motherboard*. New Delhi: BPB Publications.

Mueller, Scott, and Craig Zacker. 2002. *Upgrading and Repairing PCs*. New Jersey: Pearson Education (Que Publishing).

Godse, D.A. and A.P. Godse. 2007. *Microprocessor and Assembly Language*

Programming. Pune: Technical Publications.

Kleitz, William. 2009. *Digital and Microprocessor Fundamentals: Theory and*

Applications. New Jersey: Prentice Hall.

Ray, A.K. and K.M. Bhurchandi. 2000. *Advanced Microprocessors and Peripherals*. New Delhi: Tata McGraw-Hill.

NOTES

NOTES