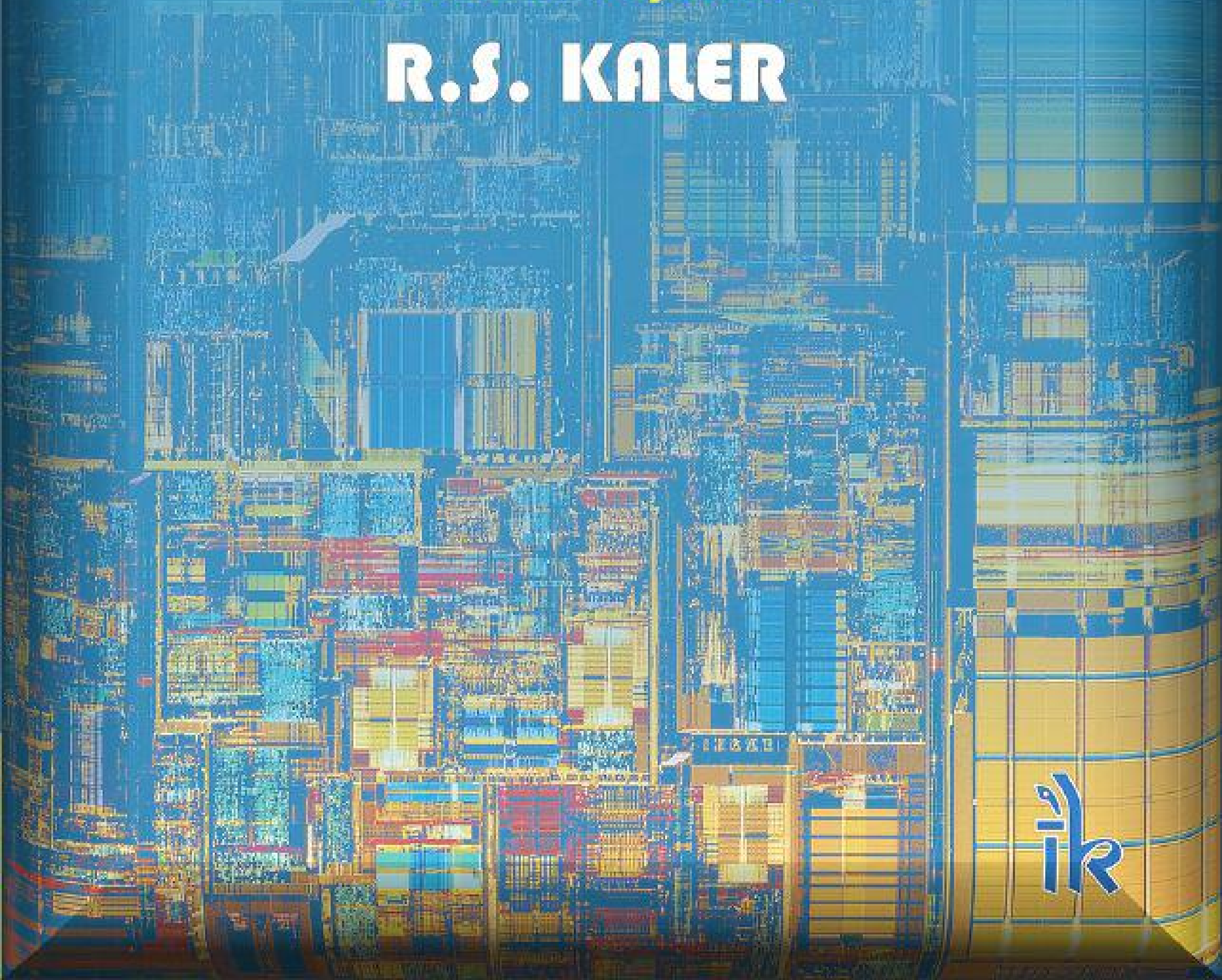
SECOND EDITION

Microprocessors AND Microcontrollers

Includes:
Microprocessors-8085, 8086 & 80XXX Series
Microcontrollers-8051, ARM, AVR & PIC Series
Embedded Systems

R.S. KALER



Microprocessors and Microcontrollers

Includes

Microprocessors–8085, 8086 & 80XXX Series
Microcontrollers–8051, ARM, AVR & PIC Series
Embedded Systems

R. S. KALER
Dean (Resource Planning and Generation) and Senior Professor
Electronics and Communication Engineering Department
Thapar University, Patiala, Punjab

I.K. International Publishing House Pvt. Ltd.
NEW DELHI • BANGALORE

Published by

I.K. International Publishing House Pvt. Ltd.
S-25, Green Park Extension
Uphaar Cinema Market
New Delhi–110 016 (India)
E-mail:info@ikinternational.com

ISBN 978-93-82332-68-8

Published by Krishan Makhijani for I.K. International Publishing House Pvt. Ltd., S-25, Green Park Extension, Uphaar Cinema Market, New Delhi–110 016 and Printed by Rekha Printers Pvt. Ltd., Okhla Industrial Area, Phase II, New Delhi–110 020.

This book is dedicated to my family and people whom I love and care.

Preface

It is a complete book for anyone interested in all aspects of the microprocessors and microcontrollers family. With the help of this book, anyone can learn to use and understand how microprocessors or microcontrollers operate. This book is basically based upon Microprocessor 8085, 8086 and Microcontroller 8051. All other related microprocessors and microcontrollers like 80186, 80286, 80386, Pentium-4, ARM and PIC are also discussed. All chapters are described with fundamental objectives. A review of important terms and concepts is also given at the end of each chapter that reinforces the idea and material presented. Each chapter also has questions and problems for reader's practice.

CONTENTS AND ORGANIZATION

This book covers the entire syllabus of microprocessors and microcontrollers by describing the key concepts and each step of important aspects. The flow of the book follows the evolution of microprocessor, definition of digital computer and computer languages in Chapter 1 and then a brief review of the digital concepts is given in Chapter 2. Number system and their conversion, logic gates and combinational logic, combinational circuits, complements, multiplexers-demultiplexers, Flip-Flops are covered in this chapter. It also includes counters, registers, Analog/Digital Conversion. Chapters 3 and 4 include Microprocessor 8085 and 8086 architecture, pin configuration, instructions set, stack and subroutines, addressing modes, interrupts. Chapter 3 also discusses the machine cycles and bus timings, generating control signals, peripheral I/O instructions, memory segmentation, flag register, minimum mode 8086 system and timings, assembler directives and operators are introduced in Chapter 4. Chapter 5 is devoted to interfacing devices. This chapter is about data transfer schemes, interfacing devices and I/O devices, programmable peripheral interface (PPI), programmable keyboard/display interface (Intel 8279). It also gives a look on centronix parallel communication, RS-232C, UART, programmable interval timer. It also discusses 8253, 8254, 8257 and 8259 in detail. Chapter 6 describes microprocessor applications. This chapter is devoted to seven-segment LED display, microprocessor-based traffic control, data acquisition system, analog to digital (A/D) converter, traffic signal controller, digital to analog converter (DAC). Chapter 7 includes Microprocessor 80XXX architecture, pin configuration, instructions set, addressing modes, interrupts. Multitasking and difference between different microprocessors is also discussed. Chapter 8 is about Microcontroller 8051. It starts with basic introduction to microcontroller and its differences from microprocessor. MCS-51 family overview is also given in this chapter. It also includes architecture, basic registers, counters and timers, timer counter interrupts, serial data input/output, addressing modes, push and pop opcodes, instructions set, arithmetic operations, programming and testing the design. The chapter ends with a brief description of real-time operating systems (RTOS). Chapter 9 describes ARM, AVR and PIC microcontrollers. It deals with their architecture, programming model, registers and flags, exception and interrupt modes, instructions set, PIC microcontroller family, PIC16F84 microcontroller, EEPROM data memory, PIC16Cxx microcontroller family. Chapter 10 describes the embedded systems and its characteristics. Some programming part is also included. In the appendix, the programming using KEIL software has been discussed and instructions set for 8085, 8086 and 8051 has been given. Special focus is made towards programming side where a large number of programs have been illustrated. To conclude, the book gives complete reference to 8085, 8086 and 8051.

R.S. Kaler

Acknowledgements

I would like to express a deep sense of gratitude and thanks to all who have directly or indirectly contributed for this book. I'll be failing in my duty if I do not acknowledge the contribution of my colleagues, friends and students. My thanks to Dr Surinder Singh, Associate Professor, Sant Longowal Institute of Engineering and Technology, Longowal for writing Chapter 10 "Embedded Systems".

The departmental faculty as well as my parents has been a constant source of inspiration and moral support for me. I would like to express a deep sense of gratitude and thanks to them. I would like to thank my father late Shri Hoshiar Singh who continuously encouraged me for my well-being. I sincerely dedicate this book to him. My thanks to my mother Smt. Joginder Kaur, my wife Dr. Rajneesh Randhawa, my son Harnoor (Harry) and my daughter Priyal (Saffy) for their patience and continuous encouragement.

R. S. Kaler

Contents

Chapter 1

Introduction to Microprocessor

- Introduction
- Integrated Circuit Technologies
- Digital Computer
- Processing Speed of a Processor
- Classification of Microprocessors
- Computer Generations
- Evolution of Microprocessors
- Intel Microprocessor Family
- Microprocessor by Various Manufacturers
- Computer Architecture
- Computer Languages
- Classification of Computers

1.1 INTRODUCTION

A computer may be defined as "an electronic device that processes binary data". Any computer based system will consist of microprocessor which is considered its brain. Microprocessor is an integrated circuit containing the arithmetic and logical unit, control circuitry and register arrays on a single chip required to interpret and execute instructions from program. It controls all of the computer's functions and data processing actions. When combined with other integrated circuits that provide storage for data, programs, input and output, it becomes the heart of a small computer, or microcomputer. Microprocessor can be considered silicon chip that processes data input by the user, and translates it into a binary code, which the device can understand.

Today, the microprocessor based products have revolutionized every area of human activity and have made a deep impact on quality of life. Right from small chip to supercomputer, microprocessor has become an integral part of every system. Microprocessor has been evolved from the developments in computers. Microprocessors are used in a wide range of commercial, industrial and domestic applications. General-purpose microprocessors in personal computers are used for computation, text editing, multimedia display and communication over the Internet. Examples of microprocessor based systems can be cellular phones, all computers, etc.

Microprocessor control of a system can provide control strategies that would be impractical to implement using electromechanical controls or purpose-built electronic controls. For example, an engine control system in an automobile can adjust ignition timings based on engine speed, load on the engine, ambient temperature, and any observed tendency for knocking–allowing an automobile to operate on a range of fuel grades.

A microprocessor can be viewed as a programmable integrated device which has computing and decision-making capabilities that is almost equivalent to the early CPU (Central Processing Unit) of computers. However, the microprocessor also has the advantage of low size, low cost and better speed.

1.1.1 Microprocessor Unit (MPU)

The MPU requires supporting ICs (Integrated Circuits) that include external RAM, ROM, I/O interface, Clock circuitry, etc. along with microprocessor. This forms a complete system which can be used to perform the general microprocessor based application. This complete setup is called Microprocessor Unit (MPU). The number of supporting chips depends upon the task to be performed, types of signals to be processed and the complexity involved. Microprocessor can handle 4-bit (nibble), 8-bit (byte), 16-bit (word) or 32-bit (double word) depending on the type of microprocessor.

The N bit data bus is deciding factor for N bit microprocessor. The microprocessors can process binary N bits of maximum length which is equal to its data bus width and accordingly has N bit registers internally; this decides how many binary bits a microprocessor can process during one T state (clock cycle). For example, a microprocessor having a data bus width of 8 and 8-bit internal registers internally can process maximum 8 bits in one T state and will be called 8-bit microprocessor.

1.1.2 Microcontroller

A microcontroller is a small computer on a single integrated circuit containing microprocessor, memory, timers and programmable input/output peripherals. Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control more devices and processes.

1.1.3 Central Processing Unit

The central processing unit (CPU) is the hardware within a computer system which carries out the instructions of a computer program by performing the basic arithmetical, logical and input/output operations of the system. CPU consists of microprocessor (ALU, control unit and register arrays) unit along with memory (RAM, ROM and hard disk) and other peripheral devices such as keyboard, display unit and other interfaces, which forms the hardware of computer system. The arithmetic and logical unit (ALU), performs arithmetic and logical operations, and the control unit (CU) performs control operations.

1.2 INTEGRATED CIRCUIT TECHNOLOGIES

Integrated circuit refers to a miniaturized electronic circuit consisting of semiconductor devices, as well as passive components bonded to a substrate or circuit board. This configuration is now commonly referred to as a hybrid integrated circuit.

1.2.1 Chip Density

The chip density refers to the number of transistors per cubic centimeter in an integrated circuit (IC). The advancement in IC manufacturing leads to increase in chip density from a few transistors to millions of transistors. The various generations of IC manufacturing can be classified as follows:

SSI: The first integrated circuits contained only a few transistors. An IC is said to use small-scale integration (SSI) if it contains fewer than 10 transistors with few logic gates. For example early linear ICs such as the Plessey SL201 or the Philips TAA320 had as few as two transistors.

MSI: An IC that contains from 10 to 100 transistors is said to use medium-scale integration. In the late 1960s, development of integrated circuits introduced devices which contained hundreds of transistors on each chip.

LSI: Further development, driven by the same economic factors, led to "large-scale integration" (LSI) in the mid 1970s. A large-scale integration IC contains from 100 to 1,000 transistors, Integrated circuits such as 1K-bit RAMs, calculator chips, and the first microprocessors, that began to be manufactured.

VLSI: The final step in the development process, starting in the 1980s and continuing through the present, was "very large-scale integration" (VLSI). It contains more than 1,000 transistors. The development started with hundreds of thousands of transistors in the early 1980s, and continues beyond several billion transistors as of 2009.

ULSI: The term ULSI that stands for "ultra-large-scale integration". The ULSI chips have very high density and complexity. It contains more than 1 million transistors.

1.2.2 IC Manufacturing Techniques

The advancement in the technology leads to various techniques which are used to produce microprocessors. Microprocessor can be made by using these technologies:

1.2.2.1 PMOS Logic

P-type metal-oxide-semiconductor logic uses p-channel enhancement type metal oxide semiconductor field effect transistors (MOSFETs) to implement logic gates and other digital circuits. PMOS transistors have four modes of operation: cut-off (or sub threshold), triode, sometimes called active, saturation and velocity saturation. While PMOS logic is easy to design and manufacture. It leads to static power dissipation even when the circuit sits idle. This technology was used in early microprocessors.

1.2.2.2 NMOS Logic

N-type metal-oxide-semiconductor logic uses n-type metal-oxide-semiconductor field effect transistors (MOSFETs) to implement logic gates and other digital circuits. It was also easier to manufacture NMOS than PMOS. The major advantage of this technology is that it is faster and has high chip density than P-MOS logic. The major problem with NMOS (and most other logic families) is that a DC current must flow through a logic gate even when the output is in a steady state (low in the case of NMOS). This means static power dissipation, i.e., power drain takes place even when the circuit is not switching. Also, NMOS circuits are slow in transition from low to high state.

1.2.2.3 CMOS Logic

Complementary metal oxide semiconductor (CMOS) is a technology used nowadays for constructing integrated circuits. CMOS technology is used in microprocessors, microcontrollers, static RAM, and other digital logic circuits. Two important characteristics of CMOS devices are high noise immunity and low static power consumption. Significant power is only drawn when the transistors in the CMOS device are switching between on and off states. Consequently, CMOS devices do not produce as much waste heat as other forms of logic, for example, transistor-transistor logic (TTL) or NMOS logic. CMOS also allows a high density of logic functions on a chip. It was primarily for this reason that CMOS became the most used technology to be implemented in VLSI chips.

1.3 DIGITAL COMPUTER

A computer may be defined as "an electronic device which accepts data and instructions in coded forms, stores them if necessary, processes the data as per the instructions and shows the outcome in some human language" or we can say that it is a device that consists of a processor that manipulates and executes the instructions. A set of instructions written for the computer to perform a job is called a program and a collection of programs is called software. Figure 1 shows the block diagram of a digital computer.

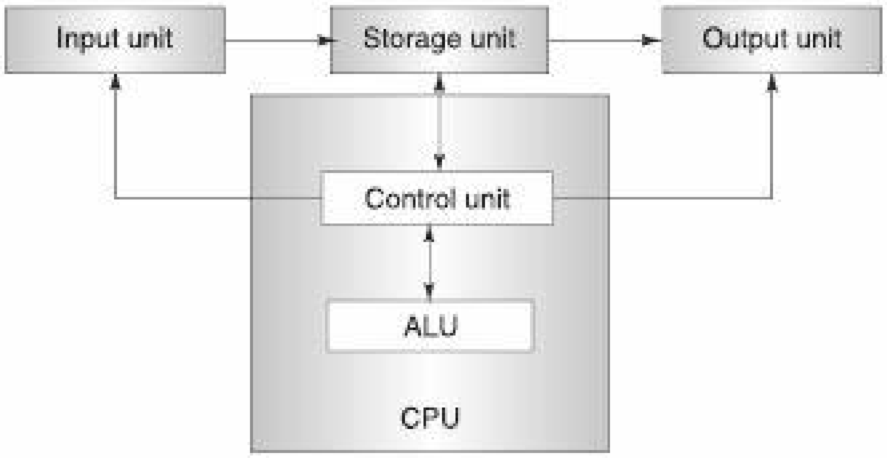


Fig. 1.1 Components of a Computer System.

1.3.1 Components of a Digital Computer

Input Unit: Computers need to receive data and instruction in order to solve any problem. Therefore, we need to input the data and instructions to the computer. For this input unit links the computer with its external environment. It accepts data and instructions from user and converts these data and instructions into computer acceptable form. The input unit consists of one or more input devices. Keyboard is one of the most commonly used input device. Other commonly used input devices are the mouse, floppy disk drive, magnetic tape, etc.

Output Unit: The output unit of a computer provides the information and results of a computation to outside world. The output of processing unit is in coded form. So, output unit converts these coded forms of data into human acceptable form. Commonly used output devices are Printers, Visual Display Unit (VDU), hard disk drive and etc.

Storage Unit: The storage unit of the computer holds data and instructions that are entered through the input unit, before they are processed. It preserves the intermediate and final results before these are sent to the output devices. It also saves the data for the later use. The various storage devices of a computer system are divided into two categories:

1. **Primary Storage:** It stores and provides the data very fast. This memory is generally used to hold the program being currently executed in the computer, the data being received from the input unit, the intermediate and final results of the program. The primary memory is temporary in nature. The data is lost, when the computer is switched off. In order to store the data permanently, the data has to be transferred to the secondary memory. The cost of the primary storage is more compared to the secondary storage. Therefore, most computers have limited primary storage capacity.

   RAM: Random access memory is a type of computer memory that can be accessed randomly; that is; any byte of memory can be accessed without touching the preceeding bytes. Random access memory is a volatile memory, meaning it loses its contents once power is cut-off. RAM is the most common type of memory found in computers and other devices, such as printers.

ROM: Read-only memory is a computer memory on which data has been prerecorded. Once data has been written onto a ROM chip, it cannot be removed and can only be read. Unlike main memory (RAM), ROM retains its contents even when the computer is turned off. ROM is referred to as being non-volatile. Most personal computers contain a small amount of ROM that stores critical programs such as the program that boots the computer.

Stack Memory: The stack is an area of memory for keeping temporary data. Stack is used by the CALL instruction to keep the return address for procedures. The return RET instruction gets this value from the stack and returns to that offset. The stack is a Last In First Out (LIFO) memory. Data is placed onto the stack with a PUSH instruction and removed with a POP instruction. The stack memory is maintained by two registers: the Stack Pointer (SP) and the Stack Segment (SS) register.

Cache Memory: Cache memory is extremely fast memory that is built into a computer's central processing unit (CPU). The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed. The advantage of cache memory is that the CPU does not have to use the system bus for data transfer. The CPU can process data much faster by avoiding the bottleneck created by the system bus.

2. **Secondary Storage:** Secondary storage is used like an archive. It stores several programs, documents, databases, etc. The program that you run on the computer is first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory. Some of the commonly used secondary memory devices are hard disk, CD, etc.

Central Processing Unit: The control unit and ALU of the computer are together known as the Central Processing Unit (CPU). The CPU is like brain which performs the following functions:

- It performs all calculations.
- It takes all decisions.
- It controls all units of the computer.

A PC may have microprocessors IC such as Intel 8088, 80286, 80386, 80486, Celeron, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium IV, Dual Core, AMD, etc.

Arithmetic and Logical Unit: All calculations are performed in the Arithmetic Logic Unit (ALU) of the computer. It also do the comparison and takes decision. The ALU can perform basic operations such as addition, subtraction, multiplication, division, etc. and does logic operations, viz., >, <, =, etc. Whenever calculations are required, the control unit transfers the data from storage unit to ALU. Once the computations are done, the results are transferred to the storage unit by the control unit and then it is send to the output unit for displaying results.

Control Unit: It controls all other units in the computer. The control unit instructs the input unit, where to store the data after receiving it from the user. It controls the flow of data and instructions from the storage unit to ALU. It also controls the flow of results from the ALU to the storage unit. The control unit is generally referred to as the central nervous system of the computer that controls and synchronizes its working.

1.3.2 Buses

In computer architecture a bus is a subsystem that transfers data between components inside a computer or between computers. It provides connection to various input-output devices and memory devices with CPU. Bus can logically connect several peripherals over the same set of wires. Each bus defines its set of connectors to physically plug devices, cards or cables together. A bus can be unidirectional or bidirectional depending upon the nature of data traveling through these bus lines. The buses are categorized depending on their tasks performed. There are three types of buses:

- Data bus,
- Address bus,
- Control bus.

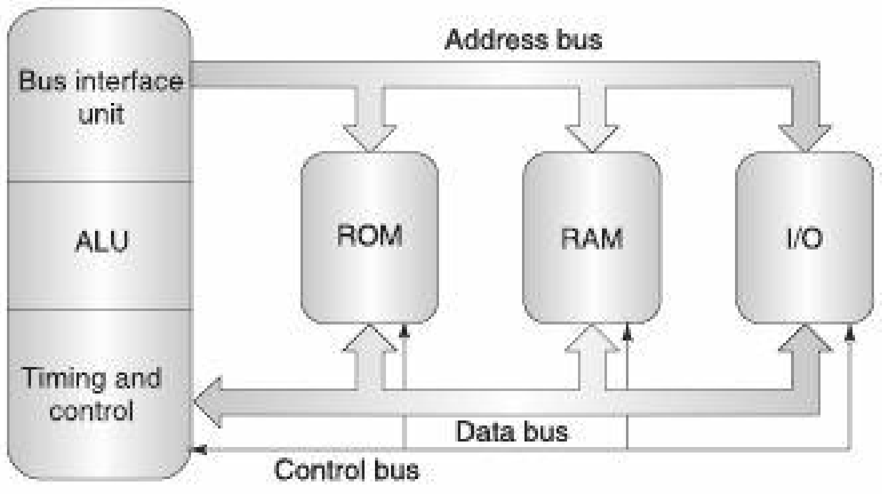


Fig. 1.2 Buses Connection with Input-Output, Memory and CPU.

Data Bus: The data bus transfers actual data. It is a collection of wires through which data is transmitted from one part of a computer to another. You can think of a bus as a highway on which data travels within a computer. This is a bus that connects all the internal computer components to the CPU and main memory.

Address Bus: The address bus transfers information about where the data should be placed. It is an internal channel from the CPU to memory across which the addresses of data (not the data) are transmitted. The number of lines in the address bus determines the memory addressing capability of a CPU. If CPU has n address lines, it can directly address 2^n memory locations. For example, a 16-line address bus can directly address 64 K bytes memory. A computer with a 32-bit address bus can directly address 4 GB of physical memory, while one with 36-bits can address 64 GB memory.

Control Bus: The physical connection that carries control information between the CPU and other devices within the computer is called control bus. The control bus carries signals that report the status of various devices. For example, one line of the control bus is used to indicate whether the CPU is currently reading from or writing to main memory.

Clock: The clock of a digital system is a periodic signal, usually a square wave, used to trigger memory latches simultaneously throughout the system. Square waves are used because the quick transitions between high and low voltages minimize the time spent at uncertain digital levels. The clock ideally reaches all parts of the system at the same time in order to prevent sections from getting out of sync. Clock signals are generally periodic because the user wants to run the system as fast as possible, but this is often not a necessary attribute. A faster clock rate means that you can process more instructions in a given amount of time at the cost of increased power consumption.

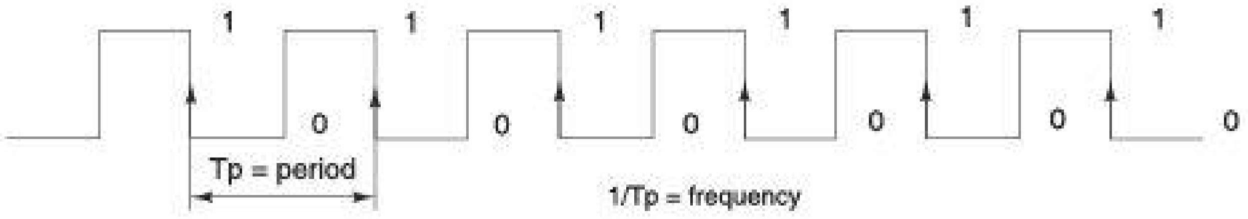


Fig. 1.3 Digital Clock Pulse.

1.4 PROCESSING SPEED OF A PROCESSOR

The processing speed of a computer is defined in terms of number of instructions executed by microprocessor per second and also called as "Throughput" of the microprocessor. Generally it is expressed in MIPS (Millions of instructions per second). The speed of processor depends on the clock frequency of the processor; higher the clock rate, higher will be the speed of processor.

A processor fetches, decodes and executes instructions at proper intervals which are timed by a specific number of clock cycles. One machine cycle is the time required to perform one operation such as moving a byte of data from one memory location to another memory location. Normally several clock cycles are required to fetch, decode and execute a single program instruction. For a faster processor its clock cycle should be small. Hence, the speed at which a computer executes an instruction directly depends on computers built-in clock speed, which is defined as the number of pulses produced per second.

The processor speed also depends upon various factors like circuit size, processor type, type of chip used, instruction set, etc. On observation, you will find that all these factors are interlinked to one another. Essentially, the processor design has its effective role to play, in which it has to specify the internal time requirements so that the maximum limit does not exceed the speed, which the chip can handle. Microprocessors that are advertised as x86 processors are also known as "32-bit" processors. These microprocessors are capable of interpreting instructions that are 32 bits (or binary digits) wide.

T State: One subdivision of an operation performed in one clock cycle is called a T-state.

1.4.1 Machine Cycle

A machine cycle is defined as the time required for completing the operation of accessing memory or I/O devices. It comprises 3 to 6 T-states. T-state is the one subdivision of an operation performed in one clock cycle. Operations like opcode fetch, memory read/write, I/O read/write are performed in machine cycles.

A machine cycle consists of a sequence of three steps that is performed continuously and at a rate of millions per second while a computer is in operation. They are fetch, decode and execute. A machine cycle can be of following types:

- Opcode Fetch
- Memory Read
- Memory Write
- Input/output Read
- Input/output Write

1.4.2 Instruction Cycle

An instruction cycle is the basic operation cycle of an instruction. The total time to execute the instruction is called instruction time and may consist of one or more machine cycles as mentioned above. The first machine cycle of any instruction is opcode fetch which consist of three steps:

1. **Fetching:** Before the CPU can execute an instruction, the control unit must retrieve or fetch a command or data from the computer's memory. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
2. **Decoding:** During this cycle the instruction inside the IR (instruction register) gets decoded and then a command can be executed.
3. **Execution cycle:** It is the time required to execute an instruction by processor after it is decoded. It may consist of memory read/ write or input/output read/write.

1.5 CLASSIFICATION OF MICROPROCESSORS

Depending upon the instruction type used in microprocessor, it can be classified as follows.

1. RISC
2. CISC

1.5.1 Reduced Instruction Set Computing (RISC)

Reduced Instruction Set Computing, or RISC is a CPU design strategy based on the insight that simplified instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a reduced instruction set computer also called RISC.

Various suggestions have been made regarding a precise definition of RISC, but the general concept is that a system that uses a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architecture, is a RISC system. RISC systems use the load-store architecture.

Popular RISC processor used in workstations are POWER (used in IBM workstations), SPARC (used in SUN workstation), and PA-RISC (used in HP workstation).

Features of RISC

- Uniform instruction format, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding.
- Identical general-purpose registers, allowing any register to be used in any context, simplifying compiler design (although there are separate floating point registers as well).
- Simple addressing modes. Complex addressing is performed via sequences of arithmetic and load-store operations.
- Few data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC.

1.5.2 Complex Instruction Set Computing (CISC)

A complex instruction set computer is a computer where single instructions can execute several low-level operations (such as load from memory, an arithmetic operation and memory store) or these are capable of multi-step operations of addressing within single instructions. These have generally complex hardware with complex instruction set. The term was retroactively coined in contrast to reduce instruction set computer (RISC). Examples of CISC instruction set architectures are System/through PDP-11, VAX, Motorola 68k and x86, etc.

Before the RISC philosophy became prominent, many computer architects tried to bridge the so called semantic gap, i.e., to design instruction sets that directly supported high-level programming constructs such as procedure calls, loop control, and complex addressing modes, allowing data structure and array accesses to be combined into single instruction. Instructions are also typically highly encoded in order to further enhance the code density. The compact nature of such instruction sets results in smaller program sizes and fewer and slow main memory accesses, which at the time (early 1960s and onwards) resulted in a tremendous savings on the cost of computer memory and disc storage, as well as faster execution. It also meant good programming productivity even in assembly language, as high-level languages such as Fortran.

Features of CISC

- Complex instruction set.
- More addressing modes.
- Highly pipelined. CISC chips are relatively slow.
- Low performance due to the use of more number of transistors.
- Comparatively costlier.

1.6 COMPUTER GENERATIONS

1.6.1 First Generation (1940-1955)

The Intel 8008 was an early byte-oriented microprocessor designed and manufactured by Intel and introduced in April 1972. It was an 8-bit CPU with an external 14-bit address bus that could address 16KB of memory. Originally known as the 1201, the chip was commissioned by Computer Terminal Corporation (CTC) to implement an instruction set of their design for their Data point 2200 programmable terminal. As the chip was delayed and did not meet CTC's performance goals, the 2200 ended up using CTC's own TTL based CPU instead. An agreement permitted Intel to market the chip to other customers after Seiko expressed an interest in using it for a calculator.

1.6.2 Second Generation (1956-1963)

The second generation used transistors instead of vacuum tubes. The transistors were highly reliable since they had no parts like a filament which could be burn out unlike in vacuum tubes. They are more rugged and easier to handle than vacuum tubes and consumed much smaller power for their operation and hence dissipated much less heat as compared to vacuum tubes. The high-level programming languages (like FORTRAN, COBOL, SNOBOL, etc.) and batch operating system emerged during the second generation. Some first generation representative systems are IBM 7090, Honeywell 400 UNIV AC LARC, etc.

1.6.3 Third Generation (1964-1974)

Integrated Circuits (ICs) were first introduced in 1958 by Jack St. Clair Kilby and Robert Noyce. Third generation computers were manufactured using ICs and it is known as 'microelectronics' technology because it makes possible to integrate larger number of circuit components into small surface of semiconductor called 'chip'. ICs were smaller, less expensive to produce, more rugged and faster in operation. Power consumed by ICs is much less as compared to other electronic devices. The concept of timesharing operating system enables multiple users to directly access and share a computer system simultaneously. Some first generation representative systems are IBM 360/91, CDC 6600, etc.

1.6.4 Fourth Generation (1975-1990)

The process of implementing a number of electronic components on a single silicon chip lead to era of LSI and then it was possible to integrate thousands of electronic components on a single chip, followed by VLSI, where it was possible to integrate over millions of electronic components on a single chip. During the fourth generation, semiconductor memories were replaced by magnetic core, memories resulting in large random access memories with fast access time. In addition to improve processing speed and storage capabilities of mainframe system, the fourth generation saw the advent of supercomputers based on parallel vector processing and symmetric multiprocessing technologies.

1.6.5 Fifth Generation (1991-Present)

The tremendous processing power and massive storage computers of the fifth generation also made them very useful and a popular tool for a wide range of applications. In operating system area, some new concepts that gained popularity are microkernels, multithreading and multicode operating system. microkernels technology enabled designers to model and design operating system in modular fashion and to allow user to implement their own service. Multithreading technology improves application performance through parallelism. Multicore operating system runs multiple programs at same time on a multicore chip with each core handling a separate program.

1.7 EVOLUTION OF MICROPROCESSORS

The first general-purpose programmable computer system was developed in 1946 at the University of Pennsylvania. The first generation of computers was a big computing machine called ENIAC (Electronic Numerical Integrator and Computer) and made up of over 17000 vacuum tubes. Being of vacuum tubes, it required a lot of maintenance as the vacuum tube components have short life. Prone to breakdowns could perform a limited operation.

The first microprocessor, Intel 4004, a 4-bit PMOS microprocessor was introduced in the year 1971 by Intel Corporation, U.S.A. In 1972, Intel produced the first 8-bit Intel 8008 which also use PMOS technology. The microprocessor using PMOS technology was slow and not compatible with TTL Logic.

The second generation computers used transistors and the size of computers became smaller. This generation caused a decrease in the computer size and increase in computing power. The microprocessor using NMOS technology is faster, compatible with TTL logic and provides high density.

S. No.	Processor	Year of Introduction	Transistors On-chip	Maximum Clock Frequency	On-chip Cache Memory	Maximum Addressable Memory	Data Bus Width
1	4004	1971	2300	750 kHz	–	1 KB	4
2	8008	1972	3500	800kHz	–	14/16 KB	8
3	8080	1974	6000	2 MHz	–	64 KB	8
4	8085	1976	6500	3 MHz	–	64 KB	8
5	8088	1980	29 K	8/5 MHz	–	1 MB	16/8
6	8086	1978	29 K	8 MHz	–	1 MB	16
7	80286	1982	134 K	12.5 MHz	–	16 MB	16
8	80386	1985	275 K	20 MHz	–	4 GB	32
9	80486	1989	1.2 M	25 MHz	8 K Level 1	4 GB	32
10	Pentium 80586	1993	3.1M	60 MHz	16 K Level 1	4 GB	
11	Pentium Pro 80586	1995	5.5M	200 MHz	16 K Level 1, 256/512 K Level 2	64 GB	64
12	Pentium II 80680	1988	7.5M	333 MHz	16 K Level 1, 256/512 K Level 2	64 GB	64
13	Pentium III 80680	2000	28M	1000 MHz	16 K Level 1, 256/512 K Level 2	64 GB	64
14	Pentium IV 80680	2000	43M	2000 MHz	16 K Level 1, 256/512 K Level 2	64 GB	64

The third generation of computers came up with the invention of integrated circuits (SSI, and MSI). In the late 1970s, third generation of computers came up. In 1979, Intel manufactured its first 16-bit data microprocessor that was 8086. This was the birth of Intel's 80. The invention of large-scale integration caused further miniaturization of microprocessor. This generation made another remarkable decrease in digital computer size and increase in computing power and overall improvement in the performance of computer. An advanced form of LSI technology, VLSI (Very Large-Scale Integration) has further miniaturized the microprocessor and increased the processing power of digital computer.

At the middle age of the fourth generation, microprocessor technology had improved tremendously. The RISC (Reduced Instruction Set Computer) technology has recently gained acceptance. Some other advanced microprocessors came up, such as Intel 486, Pentium, Pentium II, Pentium III, Pentium IV and Intel has recently manufactured Dual Core and Core2Duo. Other manufacturers have also made a very significant advancement in microprocessor technology such as the Advanced Micro Devices (AMD), has manufactured K5, K6 and K7 microprocessors. Similarly, there is a big series of advanced microprocessors from Motorola. RISC technology enhanced the processing speed, without the need for a significant advancement in integrated circuit technology. Today, advanced processors use many of architectural techniques employed by RISC machines.

1.8 INTEL MICROPROCESSOR FAMILY

1.8.1 Intel 4004 Microprocessor

The Intel 4004, a 4-bit microprocessor, was the world's first microprocessor. It addressed 4096 memory locations (each 4-bit wide). The instruction set of 4004 merely contained 45 instructions. It was fabricated on P-channel MOSFET technology and executed instructions at a very slow rate of 50 k-instructions per second. It was used in video game systems and small microprocessor based control systems. The speed, word length, and memory size were the main problems of this early microprocessor. The 4-bit microprocessor still survives in applications such as microwave ovens and calculators.

1.8.2 Intel 8008 Microprocessor

In 1972, Intel released the 8008 microprocessor, an extended 8-bit version of the 4004 microprocessor. An 8-bit word length and 14 address lines of 8008 expanded its memory size to 16 kilobytes (16 KB). A byte is an 8-bit long binary number and 1024 = 1 Kbits.

1.8.3 Intel 8080 Microprocessor

The 8008 has small memory size, slow speed and limited instruction set. To overcome these problems, Intel introduced the 8080 microprocessor in 1973. The 8080 was the first modern 8-bit microprocessor. The specialities of the 8080 were capability to address more memory and execute additional instructions much faster than the 8008. Also, the 8080 was compatible with TTL.

1.8.4 Intel 8085 Microprocessor

Intel Corporation introduced the 8085 microprocessor in 1977—an updated version of the 8080 microprocessor. The 8085 was the last 8-bit, 16 address lines general-purpose microprocessor, slightly more advanced than an 8080. The 8085 executed at an even higher speed. The main advantages of the 8085 microprocessor were its internal clock generator, internal system controller, and higher clock frequency.

1.8.5 Intel 8086 Microprocessor

Intel introduced the 8086 microprocessor—a 16-bit microprocessor having 20-bit address lines which can address 1 MB of memory, which was 16 times more faster than 8085. These modern microprocessors executed 2.5 millions of instructions per second. This higher execution speed and larger memory size allowed 8086 and 8088 to replace smaller microcomputers in many applications. Another enhanced feature found in the 8086/8088 was a small 4-byte/6-byte instruction stream queue or cache that fetched a few instructions before they were executed. The process of fetching the next instruction, once the current instruction is executed is called pipelining.

1.8.6 Intel 80286 Microprocessor

In 1983, Intel launched 80286 microprocessor which was also a 16-bit microprocessor almost identical to 8086 and 8088, except it addressed 16 MB of memory instead of 1 MB. The instruction set of 80286 was identical to the 8086 and 8088, except for a few additional instructions that managed the extra 15 MB of memory. The clock speed of 80286 was increased, so it executed more instructions in as little as 250 ns.

1.8.7 Intel 80386 Microprocessor

In 1986, Intel introduced 80386, which was a 32-bit microprocessor, had fast microprocessor speed, increased memory, and wider data bus. The 80386 was Intel's first 32-bit microprocessor that contained a 32-bit data bus and a 32-bit memory address. The 32-bit memory address bus increased the memory size to 4 GB. The instruction set of 80386 was upward compatible with the previous Intel's 8086, 8088 and 80286.

1.8.8 Intel 80486 Microprocessor

Intel produced the 80486 microprocessors in 1989, which incorporated 80386-like microprocessor, and 80387-like numeric coprocessor. The internal structure of the 80486 was modified from 80386 microprocessor, so that about half of its instructions got executed in one clock instead of two clocks. The average speed improvement for a typical mix of instructions was about 50 per cent more than the 80386 that operated at the same clock speed.

1.8.9 Intel Pentium Microprocessor

Intel produced two versions of the Pentium processor operating with a clock frequency of 60 MHz and 66 MHz. The memory system contained up to 4 GB, with the data bus increased from the 32 bits to a full 64 bits. The wider data bus width accommodated double precision floating point numbers used for high-speed and vector generated graphical displays.

In 1994, Intel released the Pentium Overdrive for older 80486 systems that operated at 63 MHz or 83 MHz frequency. The most genius feature of the Pentium is its dual integrated processors. The Pentium executes two instructions, which are not dependent on each other simultaneously because it contains two independent internal integer processors called superscalar architecture. This allows the Pentium to often execute two instructions per clock period. Another feature which enhances performance of a Pentium is a jump prediction technology that speeds up the execution of the program included loops.

1.8.10 Intel Pentium Pro Microprocessor

In 1995, Intel released the Pentium Pro microprocessor, which contains 21 million transistors, 3 integer units, and a floating point unit to increase the performance of most softwares. In addition to the internal 16 KB level-one (L1) cache (8 KB for data and 8 KB for instructions), the Pentium Pro processor also contains a 256 KB level two (L2) cache.

This processor uses three execution engines, so it can execute up to three instructions at a time, which may conflict but still execute in parallel. The Pentium Pro processor is optimized to efficiently execute 32-bit code; because of this, it is preferred with windows NT rather than with normal version of windows 95. The Pentium Pro processor can address either a 4 GB memory system or 64 GB memory system. This processor has a 36-bit address bus; if configured for a 64 GB memory system.

1.8.11 Intel Pentium II Microprocessor

In 1997, Intel released Pentium II processor; it is placed on a small circuit board. In 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 the speed of 133 MHz and stores 512 KB of information.

In mid-1998, Intel introduced a new version of Pentium II processors, called Xeon, which was specially designed for high-end workstation and server applications. Main difference between the Pentium II and the Pentium II Xeon is that the Xeon is available with L1 cache size of 32 KB and a L2 cache size of either 512 KB or 1 MB or 2 MB.

1.8.12 Intel Pentium III Microprocessor

In 1999, Intel released Pentium III processor, which uses a faster core than the Pentium II. It is 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. Pentium III is available with clock frequencies of 1 GHz. The slot 1 version contains a 512 KB cache and the flip-chip version contains a 256 KB cache.

1.8.13 Intel Pentium IV Microprocessor

In late 2000, Intel's Pentium 4 microprocessor started using P6 architecture. The Pentium 4 is available in a 1.3, 1.4 and 1.5 GHz speed version, and chipset which supports the Pentium 4 which uses the RAMBUS memory technology instead of SDRAM technology. These higher microprocessor speeds are made available by an improvement in the size of the internal integration.

The microprocessor technology may change to RISC technology, but the new technology will embody the CISC instruction set of the 80×86 family of microprocessor, so that software for the system will survive. The basic theme behind this technology is that microprocessors will communicate directly with each other, allowing parallel processing without any change in the instruction set or program.

1.9 MICROPROCESSOR BY VARIOUS MANUFACTURERS

The Intel was not only the manufacturer of the microprocessor; there were many vendors which also produced microprocessor comparable to Intel. Some microprocessors along with their vendors, name and specification are mentioned in table below:

Vendors/Processor	8-bit	16-bit	32-bit	64-bit
Intel	8008,8051	80286, 80186 8088, 8086	IAPX 432	IA 64
Zilog	Z180	Z8000	Z80000	Z380
Motorola	6800, MC6809		MC68000,MC6 8010	MC68000,MC68010
Freescale	68HC08, 68HC11	68HC12, 68HC16	MPC5125	Quad-Core Qor IQ P5040
Infineon	XC800 family	XE166,C166 family	XMC4000	MIPS-Based™ 64-bit Processor
AMD's		AMD K5,		AMD64
National Semiconductor	INS8080A	INS8900	NS 32032	
Western Digital design system	CMOS 65C02	MCP-1600,		
IBM	System/7, Series/1, System/36		IBM 7030 Stretch supercomputer	

1.10 COMPUTER ARCHITECTURE

It is the coordination of abstract levels of a processor under changing forces; involving design, measurement and evaluation. It also includes the overall fundamental working principle of the internal logical structure of a computer system.

It can also be defined as the design of the task-performing part of computer, i.e., how various gates and transistors are interconnected and are caused to function as per the instructions given by an assembly language programmer.

1.10.1 Von Neumann Architecture

The term "Von Neumann architecture", also known as the "Von Neumann model" or "The Princeton architecture". In the von Neumann Architecture, the Computer consisted of a CPU, memory and I/O devices. The program is stored in the memory. The CPU fetches an instruction from the memory at a time and executes it.

The instructions are executed sequentially, which is a slow process. Neumann microcontrollers or microprocessors are called control flow computers because instructions are executed sequentially as controlled by a program counter. To increase the speed, parallel processing of computer has been developed in which serial CPU's are connected in parallel to solve a problem. Even in parallel computers, the basic building blocks are Neumann processors.

The von Neumann architecture is a design model for a stored-program digital computer that uses a processing unit and a single separate storage structure to hold both instructions and data. It is named after mathematician and early computer scientist John von Neumann. Such a computer implements a universal turing machine, and the common "referential model" of specifying sequential architectures, in contrast with parallel architectures.

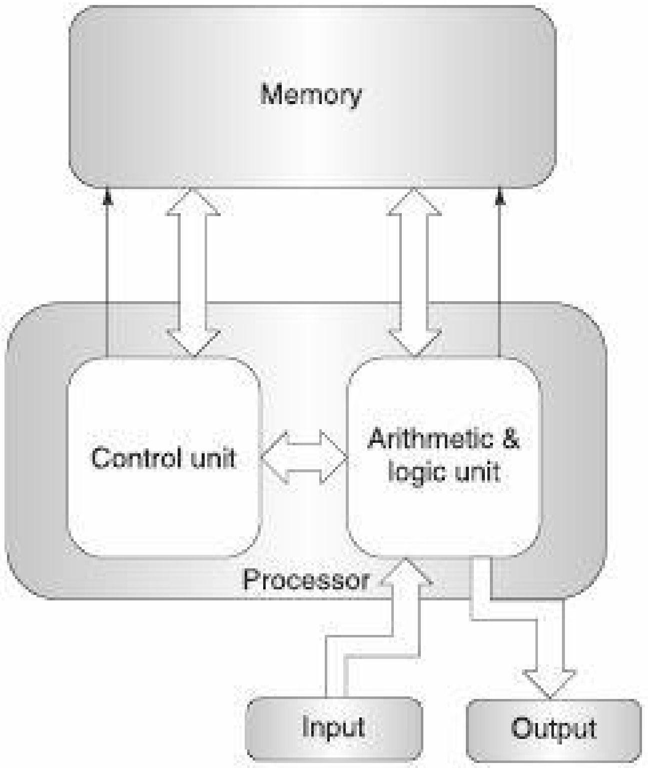


Fig. 1.4 Von Neumann Architecture of Computer.

1.10.2 Harvard Architecture

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electromechanical counters. These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator and the processor could not boot itself.

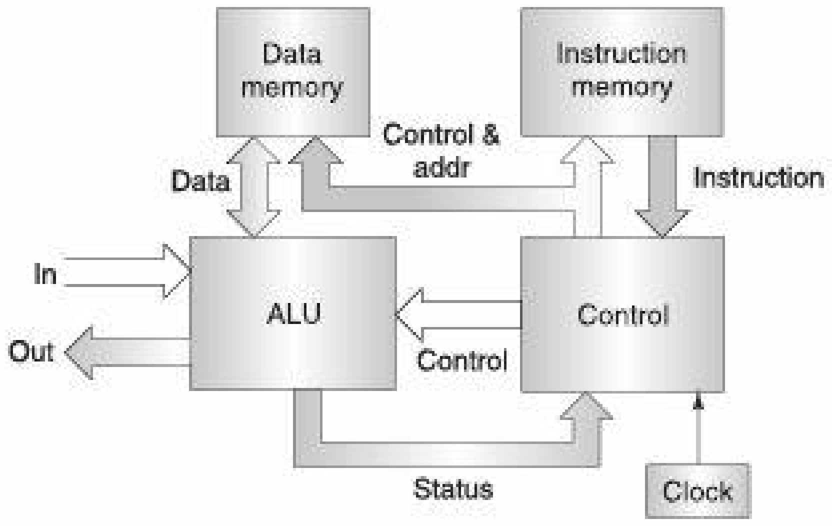


Fig. 1.5 Harvard Architecture of Computer.

Today, most processors implement such separate signal pathways for performance reasons but actually implement Modified Harvard architecture, so they can support tasks such as loading a program from disk storage as data and then executing it. Figure 1.5 shows the Harvard architecture of computer system.

1.10.3 Data Flow Architecture of Computers

Data flow architecture is a computer architecture that directly contrasts the traditional Von Neumann architecture or Control Flow architecture. Data flow architectures do not have a program counter, the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions. Instructions are executed depending upon the availability of data. Its fetch/decode unit fetches 20 to 30 instructions in advance, decodes them and their opcodes are placed in instruction pool. Execution unit then check these opcodes and executes them.

Although no commercially successful general-purpose computer hardware has used data flow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphics processing and more recently in data warehousing. It is also very relevant in many software architectures today including database engine designs and parallel computing frameworks.

Data flow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources. Hardware architecture for data flow was a major topic in computer architecture research in the 1970s and early 1980s.

1.10.4 Von Neumann vs. Harvard Architecture

In a Von Neumann architecture, program and data both are stored in the same memory and managed by the same information-handling subsystem. In the Harvard architecture, program and data are stored and handled by different subsystems. This is the essential difference between the two architectures.

In the original "Harvard computer", built in 1944 and for which the architecture is named, the program-handling task and the data-handling task were sufficiently different to result in two different storage technologies. Today, the vast majority of computers are Von Neumann architecture based, because of the efficiency gained in designing, implementing, and operating one memory system instead of two.

However, in some niches particularly in a certain embedded applications where the program is more or less hard wired, task requirements are such that the Harvard architecture can provide distinct operational advantages. Under certain conditions, a Harvard computer can be much faster than a Von Neumann computer because data and program do not contend for the same information pathway, and storing the program in an immutable read-only memory can result in vast reliability improvements.

1.11 COMPUTER LANGUAGES

Computers (or processors) understand and operate in binary numbers. These binary numbers are expressed by the bit ('0' and '1') as bit is smallest unit for any logical device. So a byte is plural of the bit that contains a cluster of bits. Mainly a byte consists of a group of 8 bits. One term also exists in companion to the byte, i.e., nibble. A nibble consists of a group of 4 bits and a byte consists of two nibbles. Next to byte comes a word. Sometimes a byte is also called a word, while it can vary from computer to computer or device to device. It may vary 4-bit for small computers to 32-bit for large computers. Its length defined as the number of bits the computer can identify and process at a time. For example, a 32-bit microprocessor has a word length of 4 bytes. If width of the registers in a processor is enlarged, the number of data and address pins may also be made superior so that the speed of the processor and memory capacity are enlarged. That means, increase in word length will improve the CPU's performance in the following manner.

- Increases the computational capability of the processor (or) the execution speed.
- Increases memory addressing capacity.
- Provides a powerful instruction set.
- Facilitates programming in high-level languages.

The number of bits in a word or word length for a given computer (or) processor is fixed and words are formed through various combinations of these bits. As we know, every device has its own set of instructions based upon the size of the ALU and also based upon the design of the processor. As we know that computer does not understand our own language, it only understands the binary form of language, i.e., Machine Language. But it is difficult to write the program in 0, s and 1, s so inventors proposed a language that is friendly to both the programmer and machine. This language is in English and presents a format of instructions that represent the binary instructions of the machine. Computer languages are of three types:

- Machine Language
- Assembly Language
- High-level Language

1.11.1 Machine Language

Machine code or machine language is a set of instructions and these instructions are executed directly by a computer's central processing unit (CPU). Instructions are patterns of bits with different patterns corresponding to different commands to the machine. Every CPU or a microprocessor has its own machine code, or instruction set. A machine code instruction set may have all instructions of the same length, or may have variable-length instructions.

1.11.2 Assembly Language

As this book presents, the different types of microprocessors; for a 8085 microprocessor, it contains 8-bit of ALU so its instruction is of eight bits, while instruction is a binary pattern that you write in assembly language by the input device in memory to perform a certain or a specific function. The 8085 microprocessor has 246 such bit patterns, amounting to 74 different instructions for performing various operations. These 74 different instructions are called its instruction set.

1.11.3 High-level Language

As the above languages illustrate the instruction set and depend upon the microprocessor while high-level language is independent of the given computer. In this language, program is written in user defined language means English-like words and these programs are executed on a microprocessor using a translator named as compiler or an interpreter.

1.11.4 Compiler

A compiler is a program that converts the instruction of a high-level language into machine language as a whole. A program written in high-level language is called source program. After the source program is converted into machine language by the compiler, it is called an object program. The compiler checks each statement in the source program and generates machine instructions. Compiler also checks syntax errors in the program. A source program containing an error cannot be compiled into an object program. A compiler can translate the programs of only that language for which it is written. For example, C++ compiler can translate only those programs, which are written in C++. Each machine requires a separate compiler for each high-level language.

1.11.5 Interpreter

An interpreter is a program that converts one statement of a program at a time. It executes this statement before translating the next statement of the source program. If there is an error in the statement, the interpreter will stop working and displays an error message. The advantage of interpreters over compilers is that an error is found immediately. So the programmer can make corrections during program development. The disadvantage of interpreter is that it is not very efficient. The interpreter does not produce an object program. It must convert the program each time it is executed. Visual basic uses interpreter.

1.11.6 Assembler

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. The programmer can write a program using a sequence of these assembler instructions. This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started. The assembler program takes each program statement in the source program and generates a corresponding bitstream or pattern (a series of 0's and 1's of a given length).

1.12 CLASSIFICATION OF COMPUTERS

Basically, computers are classified on the basis of their size, speed, etc. On these bases they are of three main types:

1.12.1 Mainframe Computer

These are the largest and most powerful computers; they are implemented using two or more central processing units. These are designed to work at very high speed with large data length typically 64-bits or greater. They have very high data storage capacity. These computers are most often used for complex scientific calculations, large data processing, military defence control and computer graphics displays for fiction movies. The fastest and the most powerful mainframe computers are called supercomputers.

1.12.2 Minicomputer

Minicomputers are the scaled down mainframes with slow speed. They work directly with the smaller data words and have smaller memory. They are used for scientific, research, data processing applications, etc.

1.12.3 Microcomputer

Microcomputers are smaller in size. They contain only one CPU, which is usually a single integrated circuit called microprocessor. Once we add ROM, RAM and ports to a microprocessor then it becomes microcomputer. Today, microcomputers are used in everything from smart sewing machines to computer-aided design systems. They are used for small industrial control and prices control. They are also used where storage and speed requirements are limited. As mentioned above, computers were classified as microcomputers, minicomputers, mainframes (large computers) and supercomputers. This classification is no longer used. Low-cost small digital computers are known as microcomputers. Portable computers, personal computers (PCs) (single user desktop computers), computers for dedicated applications such as industrial control, instrumentation, appliances control, etc. come under the category of microcomputers. Minicomputers were more powerful than microcomputers. They were multiuser systems. Mainframes are very powerful large computers. They are rarely used. They are being replaced by computer network (distributed computer system). Examples of some mainframe computers are: IBM's ES2000, ES9000, DEC1000, DECVAX9000, CDC Cyber 2000V, etc. The performance of mainframe computers are in the range of several hundred MFLOPS to 2.4 GFLOPS. The ES9000 has machine cycle time 9 ns, maximum memory 1 GB, extended memory 8 GB, peak performance 2.4 GFLOPS. DEC VAX 9000/400 VP has machine cycle time 16 ns, maximum memory 512 MB, extended memory 2 GB, peak performance 500 MFLOPS. The number of processors in mainframe varies from one to six. Supercomputers are the most powerful, very large computers. They are used for complex analysis and computations. Today, computers are classified as notebook computers (laptop computer), Personal Digital Assistant (PDA, also known as Palmtop Computer), desktop computers (PCs), workstations, servers, and supercomputers. Laptop or notebook computers are personal portable computers. They are used for word processing and spreadsheet computing while a person is traveling. Their power supply is provided from batteries. They consume less power. They use hard disks, floppy disks and flat LCD (liquid crystal display) screen. They can be connected to computer network. Wireless connection can be provided to laptops so that they can get information from large stationary computers. Notebooks use 32-bit processor specially designed for such computers.

Desktop Computers: These are single-user PCs (personal computers). They use 32-bit processors like Pentium III, Pentium IV, Celeron, PowerPC, etc. They use hard disks of capacity 40 GB to 80 GB. RAM capacity is 64 to 512 MB. They use optical disks and 3.12 inch floppy disks. Printer is optional. They use WINDOWS-XP or WINDOWS-98 operating system.

Workstations: Workstations are also desktop computers. They are more powerful than desktop machines. They are provided with graphics capability. They are used for numeric and graphic intensive applications. They have large colour video display unit (19-inch monitor). Their hard disk and RAM capacity is more than desktop computers. They use hard disks of more than

100 GB and RAM 512 MB. They use RISC processors such as DEC's Alpha processor, SUN's SPARC processor, HP'S PA-RISC, etc. Operating systems used are UNIX, SUN's Solaries, HP'S HP-UX, etc.

Servers: Servers are powerful computers. A number of PCs and terminals are connected to a server through a communicating network. A server has large disk and RAM capacity. A low-end server contains one microprocessor whereas a high-end server may contain more than one microprocessor. Microprocessors within the computer operate in parallel. A person working on a PC connected to server makes simple computation on the PC. For more complex computation he can connect his PC to the server through LAN, WAN or the Internet. He can utilize computing power, facilities and database available with the server. He can also use the facilities available at other PC connected to the server.

Supercomputers: Supercomputers are the most powerful and very large computers. They are used for complex computations. They use intensive parallelism. They contain vector processors. They are capable of performing vector computations. They handle large amount of data. A number of RISC processors are used in a supercomputer. They operate in parallel. Supercomputers are used for weather forecasting; in aerodynamics, seismology, atomic, nuclear and plasma analysis; for weapons research and development; sending rockets into space, etc. Examples of supercomputers are: Cray-1(1976), Cray-2 (1985), Cray X-MP (1983), Cray Y-MP(1988), Cray Y-MP/C-90 (1991), Cray T3D (1993), NEC SX-X/44 (1991), Fujitsu VP 2600/10 (1991), Hitachi 820/80 (1987), C-DEC's Param, etc. The Cray T3D/MPP is attached to Cray Y-MP as a back-end accelerator engine. Its performance is 150 GFlops peak in 1024 processor configuration, to 300 GFlops peak in a 2048 processor configuration. Recently, Cray has announced Cray T3E-900 high-end supercomputer with 1.8 Teraflops. It works on 450 MHz, it is successor to Cray T3E supercomputer. Cray Y-MP/C-90 used 16 processors, machine cycle 4.2 ns, maximum memory 256 M words (2 G bytes), Optional SSD memory 512 M, 1024 M or 2048 M words (16 Gbytes), 16 GFlops peak performance, I/O bandwidth 13.6 Gbytes/s NEC SX-X series uses 4 processors, 2.9 ns machine cycle, maximum memory 2 GB, 1024-way interleaving, optional SSD memory 16 GB with 2.75 GB/s transfer, 22 GFlops, 1 GB/s per I/O processor I/O bandwidth. Fujitsu VP-2000 series uses 2 processors, 3.2 ns machine cycle, 3 GB SRAM, 32 GB extended memory, 5 GFlops performance, I/O bandwidth 2 GB/s with 256 channels. C-DEC (Centre for Development of Advanced Computing) has developed Param series of supercomputers, Param 10,000 and Param Padma. The Param Padma is 10 times more powerful than Param 10,000. It is open frame architecture and it can scale up to the teraflop level. Param 10,000 has been designed using SUN's latest 160 UltraSparc II processors along with C-DAC's own designed communication processor and networks. Other Indian supercomputers are: C-DOTs (Centre for Development of Telematics) Chips-152, National Aeronautical Laboratories Flosolver Mark-3 and Bhabha Atomic Research Centre's Anupam.

Things to Remember

◊ Microprocessor is an integrated circuit containing the arithmetic, logic, and control circuitry required to interpret and execute instructions from a computer program.

◊ The MP requires supporting IC (Integrated Circuits) to perform the task at hand, this include external RAM, ROM, I/O interface, Clock circuitry, etc.

◊ The central processing unit (CPU) is the hardware within a computer system which carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system.

◊ A set of instructions written for the computer to perform a job is called a program and a collection of programs is called software.

◊ In computer architecture a bus is a subsystem that transfers data between components inside a computer or between computers. There are three types of buses: Data bus, Address bus and Control bus.

◊ The first general-purpose programmable computer system was developed in 1946 at the University of Pennsylvania.

◊ The Intel 4004, a 4-bit microprocessor was the world's first microprocessor.

◊ The first generation computers worked on principle of storing program instructions along with data in memory of computer so that they could automatically execute a program without human intervention. The second generation computers are based on transistor instead of vacuum tubes.

◊ Transistors proved to be better electronic switching devices than vacuum tubes. Third generation computers wear manufactured using ICs.

◊ Earlier ones used SSI technology and later ones used MSI technology. The fourth generation computers were made using LSI and VLSI techniques.

◊ The fifth generation provided some latest techniques such as operating system, microkernels and multithreading, etc.

◊ The processing speed of a computer is defined in terms of number of instructions executed by microprocessor per second and also called "Throughput" of the microprocessor. Generally, it is expressed in MIPS.

◊ A machine cycle is defined as the time required for completing the operation of accessing memory or I/O devices.

◊ An instruction cycle is the basic operation cycle of a computer. The step required by CPU to fetch and execute an instruction is called instruction cycle.

◊ Execution cycle is the time required to execute an instruction by processor.

◊ In a Von Neumann Architecture, program and data are stored in the same memory and managed by the same information-handling subsystem.

◊ In the Harvard architecture, program and data are stored and handled by different subsystems.

◊ Depending upon the instruction type used in microprocessor; it is classified into RISC and CISC.

◊ Machine code or machine language is a set of instructions and these instructions are executed directly by a computer's central processing unit (CPU).

◊ Assembly language consists of symbols or words called mnemonics to write the code for the processor. It requires the knowledge of the internal architecture for programing the processor.

◊ High-level languages are used for advance application where assembly becomes less efficient for programmer.

◊ An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.

◊ An interpreter is a program that converts one statement of a program at a time. It executes this statement before translating the next statement of the source program.

◊ A compiler is program that converts the instruction of a high-level language into machine language as a whole.

Questions and Answers

1. What is a microprocessor? Give its real world applications.

Microprocessor is an integrated circuit containing the arithmetic, logic, control circuitry and register arrays on a single chip required to interpret and execute instructions from program. It controls all of the computer's functions and data processing actions. When combined with other integrated circuits that provide storage for data and programs, input and output, it becomes the heart of a small computer, or microcomputer. Microprocessor can be considered silicon chip that processes data input by the user, and translates it into a binary code, which the device can understand. Microprocessors aren't just found in computers; they are part of most electronic products, everything from microwave ovens to cell phones.

2. What do you mean by MPU?

The MPU requires supporting ICs (Integrated Circuits) that include external RAM, ROM, I/O interface, Clock circuitry, etc. along with microprocessor. This forms a complete system which can be used to perform the general microprocessor based application. This complete setup is called Microprocessor Unit (MPU). The number of supporting chips depends upon the task to be performed, types of signals to be processed and the complexity involved.

3. What is a microcontroller?

A microcontroller is a small computer on a single integrated circuit containing microprocessor, memory, and programmable input/output peripherals. Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems.

4. Name the various integrated circuit technologies. Also give comparison between them.

(a) **SSI:** The first integrated circuits contained only a few transistors. Called "small-scale integration" (SSI), digital circuits containing transistors numbering in the tens provided a few logic gates for example, while early linear ICs such as the Plessey SL201 or the Philips TAA320 had as few as two transistors.

(b) **MSI:** In the late 1960s development of integrated circuits introduced devices which contained hundreds of transistors on each chip, called "medium-scale integration" (MSI).

(c) **LSI:** Further development, driven by the same economic factors, led to "large-scale integration" (LSI) in the mid-1970s, with tens of thousands of transistors per chip. Integrated circuits such as 1 K-bit RAMs, calculator chips, and the first microprocessors, that began to be manufactured.

(d) **VLSI:** The final step in the development process, starting in the 1980s and continuing through the present, was "very large-scale integration" (VLSI). The development started with hundreds of thousands of transistors in the early 1980s, and continues beyond several billion transistors as of 2009.

(e) **ULSI:** The term ULSI that stands for "ultra-large-scale integration". The ULSI chips have very high complexity. It contains more than 1 million transistors.

5. How the speed of a processor is defined?

The processing speed of a computer is defined in terms of number of instructions executed by microprocessor per second and also called "Throughput" of the microprocessor. Generally, it is expressed in MIPS (Millions of instructions per second).

6. Write a note on the following:

(a) **T State:** The machine cycles are divided into T states. One subdivision of an operation performed in one clock cycle is called a T-state.

(b) **Machine cycle:** A machine cycle is defined as the time required completing the operation of accessing memory or I/O devices. It includes of 3 to 6 T-states. T-state is the one subdivision of an operation performed in one clock cycle. Operations like opcode fetch, memory read/write, I/O read/write are performed in machine cycle.

(c) **Instruction cycle:** An instruction cycle is the basic operation cycle of an instruction. The total time to execute the instruction is called instruction time and may consist of one or more machine cycles.

(d) **Execution cycle:** It is the time required to execute an instruction by processor after it is decoded. It may consist of memory read/write or input/output read/write.

7. What is RISC and CISC?

RISC: Reduced Instruction Set Computing is a CPU design strategy based on the insight that simplified instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a reduced instruction set computer also called RISC. Popular RISC processor used in workstations are POWER (used in IBM workstations), SPARC (used in SUN workstation), and PA-RISC (used in HP workstation).

CISC: A Complex Instruction Set Computer is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation and memory store) are capable of multi-step operations of addressing within single instructions. The term was retroactively coined in contrast to reduce instruction set computer (RISC). Examples of CISC instruction set architectures are System/through PDP-11, VAX, Motorola 68k and x86 etc.

8. Differentiate between Harvard and Von-Neumann computer architectures.

In a Von Neumann Architecture, program and data are stored in the same memory and managed by the same information-handling subsystem. In the Harvard architecture, program and data are stored and handled by different subsystems. This is the essential difference between the two architectures.

In the original "Harvard computer", built in 1944 and for which the architecture is named, the program-handling task and the data-handling task were sufficiently different to result in two different storage technologies. Today, the vast majority of computers are Von Neumann architecture because of the efficiencies gained in designing, implementing, and operating one memory system instead of two.

However, in some niches, particularly certain embedded applications where the program is more-or-less hard wired, task requirements are such that the Harvard architecture can provide distinct operational advantages. Under certain conditions, a Harvard computer can be much faster than a Von Neumann computer because data and program do not contend for the same information pathway, and storing the program in an immutable read-only memory can result in vast reliability improvements.

9. What do you mean by machine language?

Machine code or machine language is a set of instructions and these instructions are executed directly by a computer's central processing unit (CPU). Instructions are patterns of bits with different patterns corresponding to different commands to the machine. Every CPU or a microprocessor has its own machine code, or instruction set. A machine code instruction set may have all instructions of the same length, or may have variable-length instructions.

10. What is the need of a compiler and assembler?

Compiler: A compiler is a program that converts the instruction of a high-level language into machine language as a whole. A program written in high-level language is called source program. After the source program is converted into machine language by the compiler, it is called an object program. The compiler checks each statement from the source program and generates machine instructions. Compiler also checks syntax errors in the program.

Assembler: An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can perform its basic operations. The programmer can write a program using a sequence of these assembler instructions. This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started. The assembler program takes each program statement in the source program and generates a corresponding bitstream or pattern (a series of 0's and 1's of a given length).

Exercise

1. List some applications of microprocessor.
2. What is microprocessor? What is the difference between microprocessor and personal computer?
3. What are the bases on which microprocessors are classified?
4. Define bit, byte, word, instruction and bus.
5. What are the advantages of assembly language as compared to high-level language?
6. What are the functional components of microprocessor based systems? Sketch block diagram.

7. List some of the manufacturers of microprocessor chips.
8. List applications of 8-bit/16-bit/32-bit microprocessors.
9. Write a note on the evolution of microprocessors.
10. What is the definition of a digital computer?
11. What is the difference between microprocessor and microcomputer?
12. Describe the terms – minicomputer, microcomputer and mainframe computer.
13. What are supercomputers?
14. Write a note on various computer languages.
15. Write a note on microprocessor's family and explain each of them in brief.
16. What do you mean by processing speed of a computer and in which units is it measured?
17. What are the main functions of a CPU?

Chapter 2

Digital Circuits & Memories

- Introduction
- Number System
- Binary Arithmetic
- Compliments
- BCD Codes
- Gray Code
- ASCII Code
- Flip-Flops and Introductory Sequential Logic
- Registers
- Memories
- Analog/Digital Conversion

2.1 INTRODUCTION

Digital electronics is the key to every microcomputer. It is necessary to have some basic knowledge of digital circuits and logics for clear understanding of topics to be discussed later in this book. This chapter discusses various number systems, memories, digital logic and concepts that are necessary to understand computer operation.

2.2 NUMBER SYSTEM

The concept of counting is as old as the evolution of man on this earth. The number system is used to quantify the magnitude of something. One way of quantifying the magnitude of something is by proportional values. This is called analog representation. The other way of representation of any quantity is numerical (digital). There are many number systems. The most frequently used number systems in digital computers are:

(a) Binary Number System
(b) Octal Number System
(c) Decimal Number System
(d) Hexadecimal Number System

Number system	*Base*	*Symbol Used*	*Assigned to Position Weight (a^i or a^{-f})*		*Example*
Binary	2	0,1	2^i	2^f	1011.11
Octal	8	0,1,2,3,4,5,6,7	8^i	8^f	3567.25
Decimal	10	0,1,2,3,4,5,6,7,8,9	10^i	10^f	3974.57
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	16^i	16^f	FA9.56

Base or radix (r) of a number system is defined as the number of different symbols (digits or characters) used in that number system. The radix of binary number system is 2, i.e., it uses two different symbols 0 and 1 to write the number sequence. The radix of octal number system is 8, i.e., it uses eight different symbols 0, 1, 2, 3, 4, 5, 6 and 7 to write the number sequence. The radix of decimal number system is 10, i.e., it uses ten different symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 to write the number sequence. The radix of hexadecimal number system is 16, i.e., it uses sixteen different symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F to write the number sequence. So, we have the following characteristics of any number system:

1. Base or radix is equal to the number of digits in the system.
2. The largest value of a digit is one (1) less than the radix.
3. Each digit is multiplied by the base raised to the appropriate power depending upon the position of the digit.

Any positional number system can be expressed as a sum of products of place value and the digit value, e.g., $756_{10} = 7 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$

- The place values or weights of different digits in a mixed hexadecimal number are as follows:

$$16^5\ 16^4\ 16^3\ 16^2\ 16^1 16^0\ .\ 16^{-1}\ 16^{-2}\ 16^{-3}$$

- The place values or weights of different digits in a mixed decimal number are as follows:

$$10^5\ 10^4\ 10^3\ 10^2\ 10^1 10^0 .\ 10^{-1}\ 10^{-2}\ 10^{-3}$$

- The place values or weights of different digits in a mixed octal number are as follows:

$$8^5\ 8^4\ 8^3\ 8^2\ 8^1 8^0 .\ 8^{-1}\ 8^{-2}\ 8^{-3}$$

- The place values or weights of different digits in a mixed binary number are as follows:

$$2^5\ 2^4\ 2^3\ 2^2\ 2^1 2^0 .\ 2^{-1}\ 2^{-2}\ 2^{-3}$$

2.2.1 Decimal to Binary Conversion

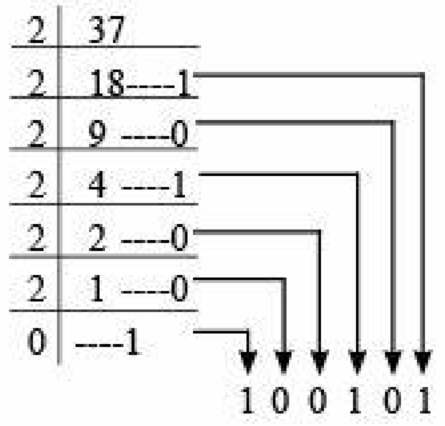
Conversion for integer number: Divide the given decimal number repeatedly by 2 and collect the remainder. This must continue until the integer quotient becomes zero.

Example:

$37_{10} =$

	Quotient	**Remainder**
37/2	18	1
18/2	9	0
9/2	4	1
4/2	2	0
2/2	1	0
1/ 2	0	1

= 100101 (in reverse order)



So, 37_{10}=100101

The conversion from decimal integer to any base r system is similar to the above example except that the division is done by r instead of 2.

Conversion for fractional number: The conversion of decimal fractional to binary is as follows:

Example:

0.6875 =

	Integer value
$0.6875 \times 2 = 1.3750$	1
$0.3750 \times 2 = 0.7500$	0
$0.7500 \times 2 = 1.5000$	1
$0.5000 \times 2 = 1.0000$	1

So, 0.6875 = 0.1011

First, 0.6875 is multiplied by 2 to give an integer and fraction. The new fraction is multiplied by 2 to give new integer and new fraction. This process is continued until the fraction becomes 0 or until the number of digits has sufficient accuracy. To convert a decimal fraction to a number expressed in base r, a similar procedure is used. Multiplication is by r instead of 2, and the coefficient found from the integers may range in value from 0 to (r − 1).

2.2.2 Binary to Decimal Conversion

Any binary number can be converted into its equivalent decimal number using the weights assigned to each bit position. Since only two digits are used, the weights are powers of 2. These weights are 2^0 (units), 2^1 (twos), 2^2 (fours), 2^3 (eights) and 2^4 (sixteens). If a longer binary number is involved, the weights continue in ascending powers of 2.

The decimal equivalent of a binary number equals the sum of all binary numbers equal to the sum of all binary digits multiplied by their weights.

Example 2.1: Find the decimal equivalent of binary number 11111.

Solution: The equivalent decimal number is,

$$= 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 16 + 8 + 4 + 2 + 1$$
$$= (31)_{10}$$

Example 2.2: Determine the decimal numbers represented by the following binary numbers.

(a) 101101. 10101 (b) 1001. 0101

Solution:

(a) $(101101.\ 10101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$$+ 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$$
$$= 32 + 0 + 8 + 4 + 0 + 1 + 1/2 + 0 + 1/8 + 0 + 1/32$$
$$= (45.65625)_{10}$$

(b) $(1001.0101)_2 = 8 + 0 + 0 + 1 + 0 + 0.25 + 0 + 0.0625$
$$= (9.3125)_{10}$$

2.2.3 Octal Number System

The number system with base (or radix) eight is known as the octal number system. In this system, eight symbols, 0, 1, 2, 3, 4, 5, 6, and 7 are used to represent the number. Similar to decimal and binary number systems, it is also a positional system and has, in general, two parts: integer and fractional, set apart by a radix point. For example, $(6327.4051)_8$ is an octal number. Using the weights, it can be written as:

$$(6327.4051)_8 = 6 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} + 0 \times 8^{-2} + 5 \times 8^{-3} + 1 \times 8^{-4}$$
$$= 3072 + 192 + 16 + 7 + 4/8 + 0 + 5/512 + 1/4096$$
$$= (3287.5100098)_{10}$$

Using the above procedure, an octal number can be converted into an equivalent decimal number. The conversion from decimal to octal is similar to the conversion procedure from decimal to binary. The only difference is that number 8 is used in place of 2 for division in the case of integers and for multiplication in the case of fractional numbers.

2.2.4 Octal to Binary

Octal numbers can be converted into equivalent binary numbers by replacing each octal digit by its 3-bit binary equivalent.

Numbers and their binary equivalents for decimal numbers 0 to 15

Octal	Decimal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
10	8	1000
11	9	1001
12	10	1010
13	11	1011
14	12	1100
15	13	1101
16	14	1110
17	15	1111

Example 2.3: Convert 736_8 into an equivalent binary number.

Solution: From the above table, the binary equivalents of 7, 3 and 6 are 111, 011 and 110 respectively

Therefore, $736_8 = 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0_2$

2.2.5 Binary to Octal

Binary numbers can be converted into equivalent octal numbers by making groups of three bits starting from LSB and moving towards MSB for integer part of the number and then replacing each group of three bits by its octal representation. For fractional part the groupings of three bits are made to start from the radix point.

Example 2.4: Convert 1001110_2 to its octal equivalent.

Solution:

$$\begin{aligned} 1001110_2 &= 001 \quad 001 \quad 110 \\ &= \ \ 1 \qquad 1 \qquad 6 \\ &= 116_8 \end{aligned}$$

2.2.6 Application of Octal Number System

It is highly inconvenient to handle long strings of binary numbers while entering into the digital systems. It may cause errors also. Therefore, octal numbers are used for entering binary data and displaying certain information.

2.2.7 Hexadecimal Number System

Hexadecimal number system is very popular among computer users. The base for hexadecimal number system is 16 which requires 16 distinct symbols to represent the number. These are numerals 0 through 9 and alphabets A through F. This is an alphanumeric number system because it uses both alphabets and numericals to represent a hexadecimal number. The table below gives hexadecimal numbers with their binary equivalents for decimal numbers 0 through 15.

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

2.2.8 Hexadecimal to Decimal Conversion

Hexadecimal numbers can be converted into their equivalent decimal numbers.

Example 2.5: Obtain decimal equivalent of hexadecimal number $3A.2F_{16}$.

Solution: $3A.2F = 3 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 15 \times 16^{-2}$

$$= 48 + 10 + 2/16 + 15/16^2$$

$$= 58.1836_{10}$$

2.2.9 Decimal to Hexadecimal Conversion

For conversion from decimal to hexadecimal, the procedure used in binary as well as octal system is applicable, using 16 as the dividing (n for integer part) and multiplying (for fractional part) factor.

2.2.10 Hexadecimal to Binary Conversion

Hexadecimal numbers can be converted into equivalent binary numbers by replacing each hex digit by its equivalent 4-bit binary numbers.

Example 2.6: Convert $2F \cdot 9A_{16}$ into its equivalent binary number. Find the binary equivalent of each hexdigit.

$$(2F \cdot 9A)_{16} = (0010\ 1111\ 1001\ 1010)_2$$
$$= (0010111110011010)_2$$

2.2.11 Binary to Hexadecimal Conversion

Binary numbers can be converted into the equivalent hexadecimal numbers by making groups of four bits starting from LSB and moving towards MSB for integer's part and then replacing each group of four bits by its hexadecimal representation. For the fractional part, the above procedure is repeated starting from the bit next to the binary point and moving towards the right.

Example 2.7: Convert the following binary numbers into their equivalent hexadecimal numbers.

(a) 10100110101111.

(b) 0.00011110101101.

Solution:

(a) $(10100110101111)_2 = (\underset{2}{0010}\ \underset{9}{1001}\ \underset{A}{1010}\ \underset{F}{1111})$

$\therefore\ 10100110101111_2 = 29AF_{16}$

(b) $0.00011110101101_2 = 0.\underset{1}{0001}\ \underset{E}{1110}\ \underset{B}{1011}\ \underset{4}{0100}$

$\therefore\ 0.00011110101101_2 = 0.1EB4_{16}$

2.3 BINARY ARITHMETIC

We all are familiar with the arithmetic operations such as addition, subtraction, multiplication and division of decimal numbers. Similar operations can be performed on binary numbers. Binary arithmetic is simpler than decimal arithmetic because here only two digits, 0 and 1 are involved.

2.3.1 Binary Addition

The rules of binary addition are given in the following table.

Augend	Addend	Sum	Carry in from previous stage	Carry out to next stage
0	+0	0	0	0
0	+1	1	0	0
1	+0	1	0	0
1	+1	0	0	1
1	+1	1	1	1

Example 2.8: Add the following binary numbers.

(i) 1011 and 1100 (ii) 0101 and 1111

Solution:

(i) 1011 (ii) 0101

```
       1  0  1  1                  0  1  0  1
   (+) 1  1  0  0              (+) 1  1  1  1
   ---------------             ---------------
    1  0  1  1  1               1  0  1  0  0
```

2.3.2 Binary Subtraction

The rules of binary subtraction are given in the following table.

Minuend	0	1	1	0
Subtrahend	–0	–1	–0	–1
Difference	0	0	1	1
Borrow	0	0	0	1

The first three rules are the same as in decimal subtraction. The last rule requires a borrow from the next most significant place. The minuend is then binary 10 and the subtrahend is 1 with a difference of 1.

Example 2.9: Perform the following subtraction.

(i) 1011 and 0110

(ii) 01010101 and 00111001

Solution:

```
   1  0  1  1           0  0  1  0  1  0  1  0  1
 – 0  1  1  0         – 0  0  1  1  1  0  0  1
 ------------         ------------------------
   0  1  0  1           0  0  0  1  1  1  0  0
```

2.4 COMPLIMENTS

For simplifying the subtraction operation and logic manipulations, compliments are used in digital computers. There are two types of compliments for each base-r system:

1. The r's compliment
2. The r − 1's compliment

2.4.1 The r's Compliment

For given positive number N in base r with integer part of n digits, the r's compliment of N is defined as

$r^n - N$ for $N \neq 0$.

1. for N = 0

   E.g. the 10's compliments of 47480_{10} is given as

$$10^5 - 47480$$
$$= 52520$$

2.4.2 The r −1's Compliment

For given positive number N in base r with integer part of n digits and fraction part of m digits, the r–1's compliment of N is defined as

$r^n - r^{-m} - N$

E.g. the 9's compliments of 65270_{10} is given as

$10^5 - 1 - 65270 = 99999 - 65270 = 34729$

2.4.3 Subtraction with r's Compliments

The subtraction of two positive numbers (M – N), both of base r may be done as follows:

1. Add the minuend M to r's compliment of subtrahend N.
2. Check the results of obtained in step 1 for an end carry.
3. If an end carry occurs, discard it.
4. If an end carry does not occur, take the r's compliment of the number obtained in step 1 and place a negative sign in front.

Example: Using 10's compliment, subtract 57232 – 3550.

Solution:

M = 57232 and N = 03550

10's compliments of N = 96540

$$\begin{array}{r} 57232 \\ +\ 96450 \\ \hline \mathbf{1\ 53682} \\ \hline \end{array}$$

Discard end carry.

Answer is 53682.

2.5 BCD CODES

Binary coded decimal, in this each digit of the decimal number is represented by its four-bit binary equivalent. It is also called Natural BCD or 8421 code. It is a weighted code. The standard BCD code uses the first ten numbers 0 to 9. To convert any number into its equivalent BCD number, just represent each of the decimal digit in the number by its four-bit BCD number.

Example: 462 to BCD

Solution:

Decimal	4	6	2
BCD	0100	0110	0010

2.6 GRAY CODE

This is also known as reflected code. In the reflected code, the number is changing by only 1 bit as it proceeds from one number to the next. e.g.

Reflected Gray Code	Decimal Equivalent
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9
1111	10
1110	11
1010	12
1011	13
1001	14
1000	15

2.7 ASCII CODE

The American Standard Code for Information Interchange is a character-encoding scheme, originally based on the English alphabets. ASCII codes represent text in computers, communication equipment, and other devices that use text.

Computer is used to store and manipulate text; this can be done by simply representing different alphabetic letters by specific numbers. By the late 1950s, computers were getting more common, and starting to communicate with each other. There was a pressing need for a standard way to represent text so it could be understood by different models and brands of computers. This was the impetus for the development of the ASCII table, first published in 1963 but based on earlier similar tables used by teleprinters. After several revisions, the modern version of the 7-bit ASCII table was adopted as a standard by the American National Standards Institute (ANSI) during the 1960s.

ASCII includes definitions for 128 characters: 33 are non-printing control characters (many of them are now obsolete) that affect how text and space is processed and 95 printable characters, including the space. It's a 7-bit character code where every single bit represents a unique character. The table below shows 7-bit ASCII characters.

Hex	0	1	2	3	4	5	6	7	8	9	q	b	c	d	e	f
00	NUL ^@ 0	SOH ^A 1	STX ^B 2	ETX ^C 3	EOT ^D 4	ENQ ^E 5	ACK ^F 6	BEL ^G 7	BS ^H 8	TAB ^I 9	LF ^J 10	VT ^K 11	FF ^L 12	CR ^M 13	SO ^N 14	SI ^O 15
10	DLE ^P 16	DC1 ^Q 17	DC2 ^R 18	DC3 ^S 19	DC4 ^T 20	NAK ^U 21	SYN ^V 22	ETB ^W 23	CAN ^X 24	EM ^Y 25	SUB ^Z 26	ESC ^[27	FS ^\ 28	GS ^] 29	RS ^^ 30	US ^? 31
20	32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
30	48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
40	64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
50	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	– 95
60	` 96	97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
70	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	DEL 127

2.8 FLIP-FLOPS AND INTRODUCTORY SEQUENTIAL LOGIC

We now turn to digital circuits which have states which change in time, usually according to an external clock. The *flip-flop* is an important element of such circuits. It has the interesting property of *memory*. It can be set to a state which is retained until explicitly reset.

2.8.1 The D-Type (Data) Flip-Flop

The Data flip-flop (DFF) is unique, in that it only has one external input along with a clock input. The logic symbol for this flip-flop is given in Figure 2.1. Two asynchronous inputs, PRESET and CLEAR enable the flip-flop to be set to a predetermined state, independent of the CLOCK. Note the invert bubble on these lines indicates that these lines are normally held at 1 and that the function (CLEAR or PRESET) is performed by taking the line to 0.

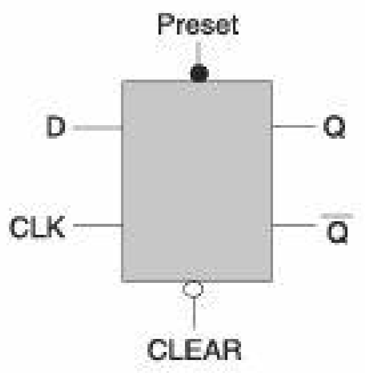


Fig. 2.1 D Flip-Flop.

The Data flip-flop transfers whatever is applied at the external input D to the output Q. This does not happen immediately, however, it only happens on the rising clock pulse (i.e., as CLK goes from 0 to 1). The input is thus delayed by a clock pulse before appearing at the output. This is illustrated in the timing diagram below. The DFF is an *edge-triggered device* which means that the change of state occurs on a clock transition (in this case the rising clock pulse as it goes from 0 to 1).

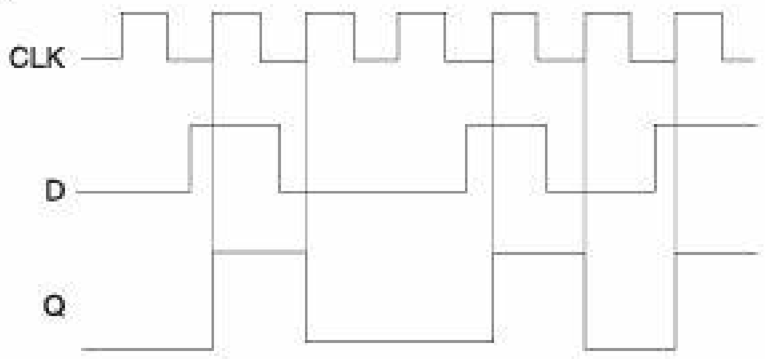


Fig. 2.2a Clock Plus Diagram of D Flip-Flop.

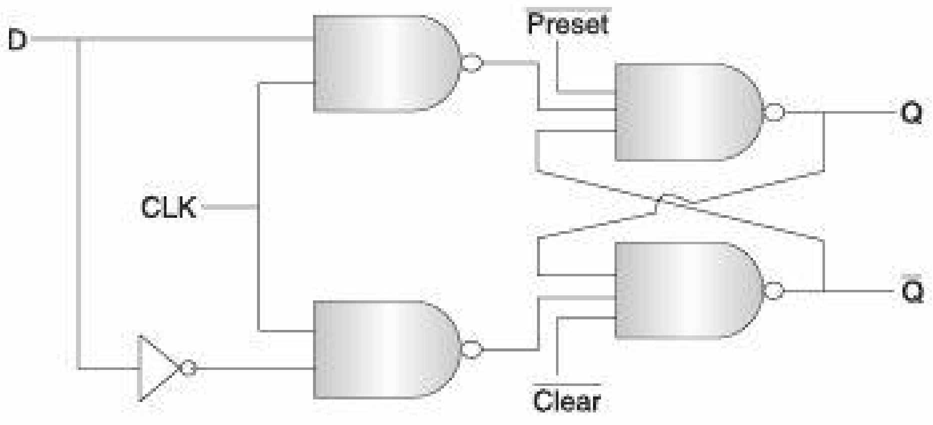


Fig. 2.2b Implementation of D Flip-Flop.

One implementation of the Data flip-flop is given below.

Here the function of the asynchronous inputs can clearly be seen, taking PRESET momentarily to 0 sets Q = 1 and taking CLEAR momentarily to 0 sets Q = 0. The Data flip-flop can also be configured from a J K flip-flop where the input is connected to J and the compliment of the input is connected to K.

The Data Flip-Flop as a Frequency Divider

If the DFF is configured as it is indicated below, then there is no external input (D has become an internal input) and only the clock pulse (CLK) is operated on. Note that for clarity the asynchronous inputs PRESET and CLEAR have been omitted from this logic diagram.

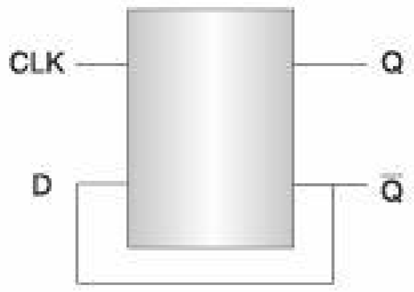


Fig. 2.3 Frequency Divider.

Assuming the initial state of CLK = 0 and Q = 0 then it follows that since D is connected to Q then D = 1. As seen from previous discussion that whatever is at D is transferred to Q at the next rising clock pulse. So, as CLK goes from 0 to 1, Q becomes 1 and so D becomes 0. At the next rising CLK pulse, the input at D (which is 0) is transferred to Q and so Q becomes 0 and hence D = 1, etc. This cycle is illustrated in the timing diagram below.

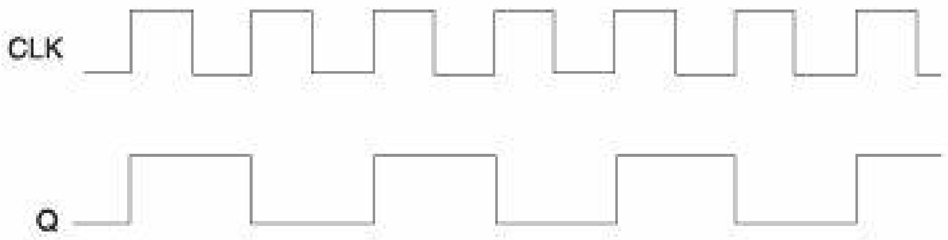


Fig. 2.4 Clock Pulse Diagram of Frequency Divider.

It can be seen that for every two clock pulses in, there is only one clock pulse out, the circuit is therefore performing division by 2.

It should be noted that this behavior only takes place when the clock pulses are reasonably short (but at least long enough for the output to change state). If the clock pulse is long, oscillation may occur.

2.8.2 Level Triggering and Edge Triggering in Flip-Flops

A clock signal is used in flip-flops as a control mechanism to make sure that unnecessary changes in inputs don't affect the output of the flip-flop. Traditionally, the inputs were bound to the clock signal using AND gates and thus were only registered in the flip-flop when the clock signal became high. This is known as level triggering.

One of the drawbacks of level triggering was that any unwanted change in the inputs while the clock was high got registered into the flip-flop and thus affected the output which caused a certain degree of unreliability in operation. Then there was also the problem of a certain 'race around' condition in J-K flip-flops that was caused because the outputs affected the inputs throughout the period the clock pulse remained high.

A new hardware configuration of flip-flops was thus devised which only responded to the inputs when the clock signal either went from high to low or from low to high. This is known as edge triggering and can be achieved by adding a little extra hardware logic to the flip-flop circuitry. But this logic is added in parallel and therefore the number of gating stages of the circuit remains the same as the original flip-flop before the output is achieved. This meant that not only the problem of the unwanted input changes was solved but the original speed of the flip-flop was also maintained.

If the inputs are registered when the clock goes from low to high, then it is known as positive or rising edge triggering. If they are registered when the clock goes from high to low, then it is known as negative or falling edge triggering. This is shown in the Figure 2.5. The usual practice is to use the edge triggering configuration in D flip-flops because of the high speed desired in data storage applications.

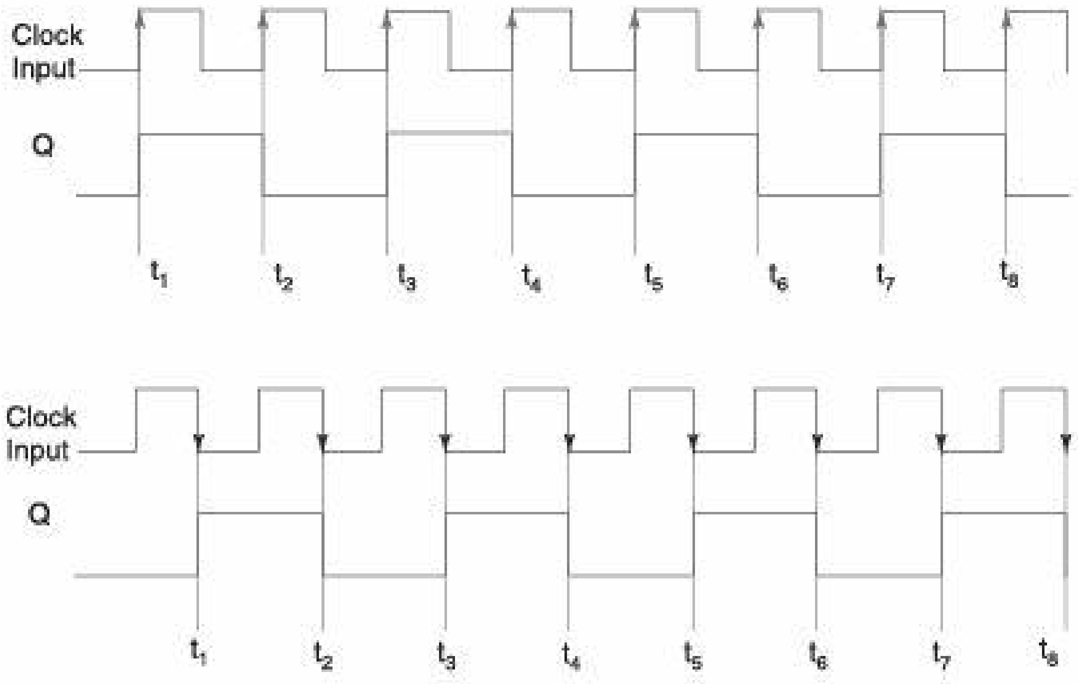


Fig. 2.5 Positive and Negative Edge Triggered Flip-Flop Timing Diagram.

For J-K flip-flops, Master-Slave configuration is the one that is commonly used in the applications like binary counters in which they are used that doesn't require that much speed.

2.9 REGISTERS

Figure 2.6 below represents a 4-bit memory. We can think of it as 4 individual D-type flip-flops. The important point about a data register of this type is that all of the inputs are latched into the memory synchronously by a single clock cycle.

2.9.1 Buffers

A buffer is a means of isolating a signal source circuit from the loading circuit. They are generally needed when the signal source does not have sufficient capacity to deliver the current demanded by the load circuit. If buffers are not used, the problem of input loading results and this may cause the circuit to malfunction or to become damaged. In digital circuits, the buffers reproduce

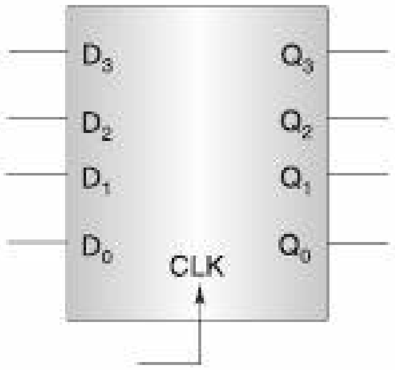


Fig. 2.6 4-bit Data Register.

the sequence of 1's and 0's received from one circuit and make them available to another circuit at a higher power level. A buffer is like a non-inverting amplifier with a gain of unity.

2.9.2 Data Buffer

A buffer is a region of a physical memory storage used to hold data temporarily while it is being moved from one place to another. Typically, the data is stored in a buffer as it is retrieved from an input device (such as a mouse) or just before it is sent to an output device (such as speakers). However, a buffer may be used when moving data between processes within a computer.

2.9.3 Tri-state Logic

In digital electronics, three-state, tri-state, or 3-state logic allows an output port to assume a high impedance state in addition to the 0 and 1 logic levels, effectively removing the output from the circuit. This allows multiple circuits to share the same output line or lines (such as a bus which cannot acknowledge to more than one device at a time).

Three-state outputs are implemented in many registers, bus drivers, and flip-flops in the 7400 and 4000 series as well as in other types, but also internally in many integrated circuits. Other typical uses are internal and external buses in microprocessors, memories, and peripherals. Many devices are controlled by an active-low input called $\overline{\text{OE}}$ (Output Enable) which dictates whether the outputs should be held in a high-impedance state or drive their respective loads (to either 0- or 1-level). A tri-state buffer can be thought of as a switch. If B is on, the switch is closed. If B is off, the switch is open as shown in Figure 2.7.

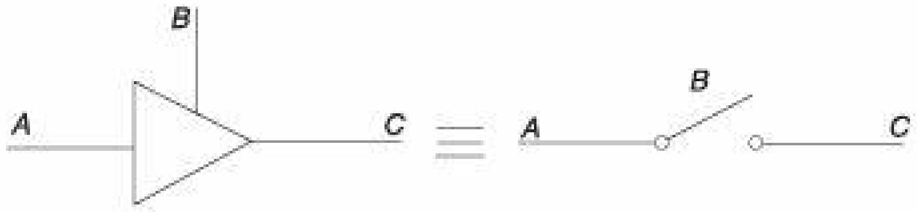


Fig. 2.7 Tri-state Buffer.

Input		Output
A	B	C
0	1	0
1	1	1
x	0	z

Application of Tri-state Logic

The whole concept of the third state (Hi-Z) is to effectively remove the device's influence from the rest of the circuit. If more than one device is electrically connected, putting an output into the Hi-Z state is often used to prevent short circuits, or one device driving high (logical 1) against another device driving low (logical 0).Three-state buffers can also be used to implement efficient multiplexers, especially those with large numbers of inputs. In particular, they are essential to the operation of a shared electronic bus. Three-state logic can reduce the number of wires needed to drive a set of LEDs.

2.10 MEMORIES

The basic goal of digital memory is to provide a means to store and access binary data: sequences of 1's and 0's. Memories are made up of registers. Each register is made up of flip-flops which are the basic building blocks of memory. Each register in memory is a storage location which can be used to hold the binary data. Each memory location is identified by an address. Each location can be used to store one or more bits. The total number of bits that a memory can store is called the capacity. The capacity of memories is generally specified in terms of bytes (8 bits).

Figure 2.8 shows a memory unit. It stores the binary information in groups of bits. To access the memory, we require address line, data line and control lines. The address lines are used to locate the position of the memory register where data is to be stored or retrieved. The data lines are used to carry data from device to the memory (in case of write operation) or from the memory to device (in case of read operation). The operation to be performed is decided by the control lines. The device interfaced with the memory asserts the read/write signal to perform the data operation. The capacity of memories depends upon the number of address and data lines. With M address line, the 2^M memory location can be addressed where each location can store N bits of data.

$$C = 2^M \times N$$

where C = Capacity of memory

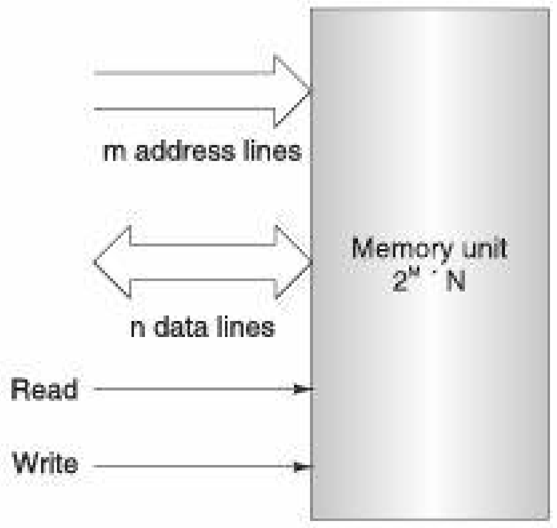


Fig. 2.8 Block Diagram of Memory Unit.

M = No. of address lines

N = No. of data lines

2.10.1 Classification of Memories

Computer memory refers to components, devices, chips and recording media that are used for temporary, semi-permanent and permanent storage of data. These include RAM, ROM, cache, flash memory, hard disk, floppy disk, CDs, and so on. Memory devices can be broadly classified into two types, namely, primary memory and secondary storage.

Figure 2.9 shows the various types of memory device present in a typical computer system. It may be mentioned here that, in computer terminology, 'memory' usually refers to RAM and ROM and the term 'storage' refers to hard disks, floppy disks and CDs. In general, the memories can also be classified as shown in Figure 2.10.

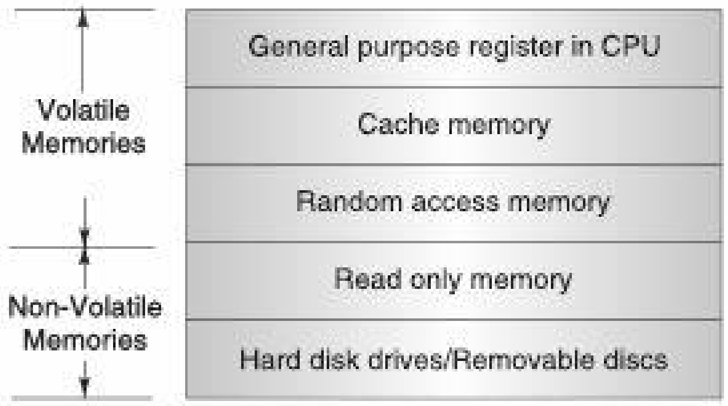


Fig. 2.9 Various Types of Memory Present in a Computer System.

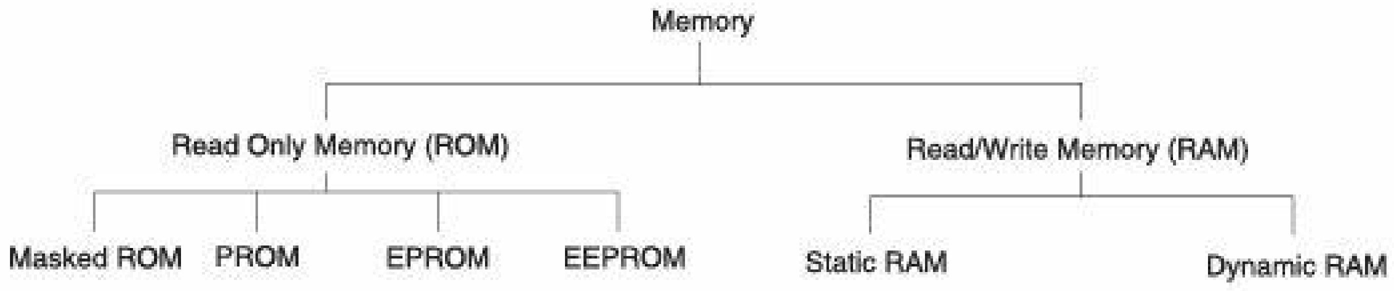


Fig. 2.10 Classification of Memories.

2.10.2 Random Access Memory

RAM is a read/write memory where the data can be read from or written into any of the memory locations regardless of the order in which they are arranged. Therefore, all the memory locations in a RAM can be accessed at the same speed. RAM is used to store data, program instructions and the results of any intermediate calculations during the execution of a program. Also, the same data can be read any number of times and different data can be written into the same memory location, with every fresh data item overwriting the existing one. It is typically used for short-term data storage as it cannot retain data when the power is turned off.

RAM has three basic building blocks, namely, an array of memory cells arranged in rows and columns with each memory cell capable of storing either a '0' or a '1', an address decoder and a read/write control logic.

Depending upon the nature of the memory cell used, there are two types of RAM, namely, static RAM (SRAM) and dynamic RAM (DRAM). In SRAM, the memory cell is essentially a latch and can store data indefinitely as long as the DC power is supplied. DRAM on the other hand, has a memory cell that stores data in the form of charge on a capacitor. Therefore, DRAM cannot retain data for long and hence needs to be refreshed periodically. SRAM has a higher speed of operation than DRAM but has a smaller storage capacity.

2.10.2.1 Types of RAM

Depending upon the principle of operation the RAM can be classified into two categories Static RAM and Dynamic RAM.

2.10.2.2 Static RAM

The basic element of SRAM is a latch memory cell. Figure 2.11 shows a basic SRAM memory cell. The memory cell is selected by setting the 'select' line active. The data bit is written into the cell by placing it on the 'data in' line and is read from the 'data out' line.

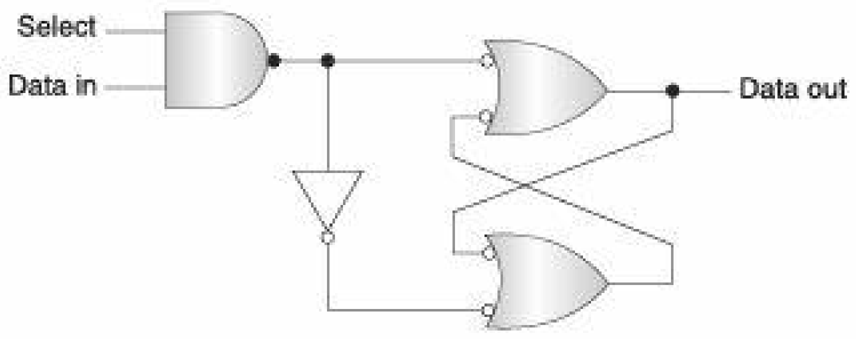


Fig. 2.11 Basic SRAM Memory Cell.

SRAMs can be broadly classified as asynchronous SRAM and synchronous SRAM. Asynchronous SRAMs are those, whose operations are not synchronized with the system clock, i.e., they operate independently of the clock frequency. 'Data in' and 'data out' in these RAMs are controlled by address transition. Synchronous SRAMs are those whose timings are initiated by clock edges. 'Address', 'data in', 'data out' and all other control signals are synchronized with the clock signal. Synchronous SRAMs normally have an address burst feature, which allows the memory to read and write at more than one location using a single address. While bipolar SRAM offers a relatively higher speed of operation, MOS technology offers higher capacity and reduced power consumption.

2.10.2.3 Dynamic RAM

The memory cell in the case of a DRAM comprises a capacitor and a MOSFET. The cell holds a value of '1' when the capacitor is charged and '0' when it is discharged. The main advantage of this type of memory is its higher density, or more bits per package, compared with SRAM.

This is because the memory cell is very simple compared with that of SRAM. Also, the cost per bit is less in the case of a DRAM. The disadvantage of this type of memory is the leakage of charge stored on the capacitors of various memory cells when they are storing a '1'. To prevent this from happening, each memory cell in a DRAM needs to be periodically read, its charge (or voltage) compared with a reference value and then the charge restored to the capacitor. This process is known as 'memory refresh' and is done approximately every 5–10 ms.

Figure 2.12 shows the basic memory cell of a DRAM and its principle of operation. The MOSFET acts like a switch. When in the 'write' mode ($R/\overline{W} = 0$), the input buffers are enabled while the output buffers are disabled. When '1' is to be stored in the memory, the 'data in' line must be in the HIGH state and the corresponding 'row line' should also be in the HIGH state so that the MOSFET is switched ON. This connects the MOSFET to the 'data in' line, and it charges the capacitor to a positive voltage level. When '0' needs to be stored, the 'data in' line is LOW and the capacitor also acquires the same level. When the 'row line' is taken to the LOW state, the MOSFET is switched OFF and is disconnected from the bit line. This traps the charge on the capacitor.

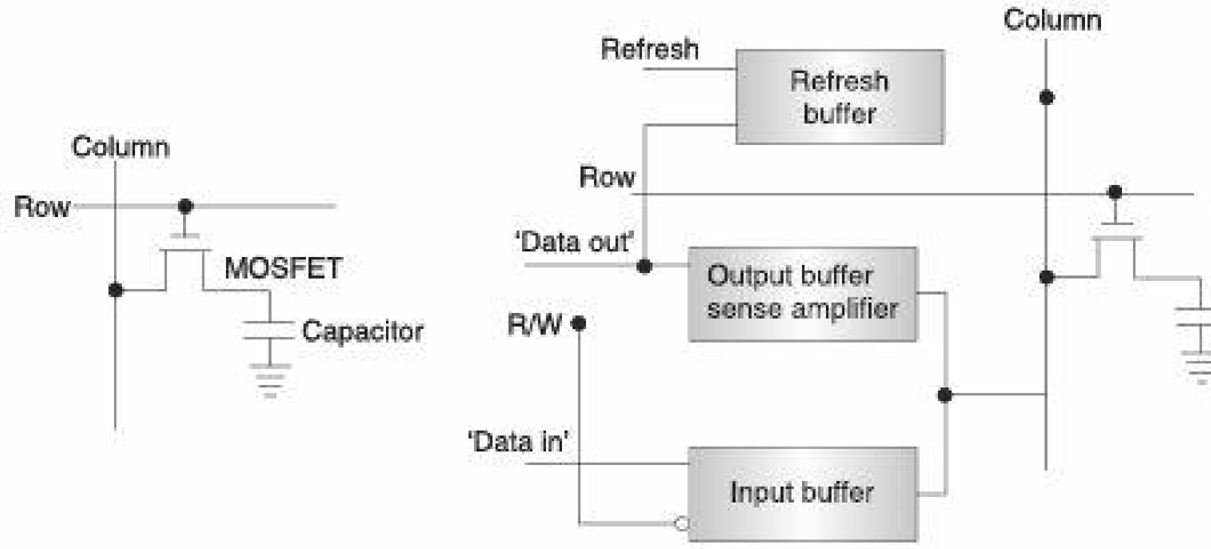


Fig. 2.12 The Basic Memory Cell of a DRAM.

In 'read' mode ($R/\overline{W} = 1$), the output buffers are enabled while the input buffers are disabled. When the 'row line' is taken to HIGH logic, the MOSFET is switched ON and connects the capacitor to the 'data out' line through the output buffer. The refresh operation is performed by setting $R/\overline{W} = 1$ and by enabling the refresh buffer. There are two basic modes of refreshing the memory, namely, the burst refresh and distributed refresh modes. In burst refresh mode, all rows in the memory array are refreshed consecutively during the refresh burst cycle. In distributed refresh mode, each row is refreshed at intervals interspaced between 'read' and 'write' operations. One of the major applications of RAM is its use in cache memories. It is also used as main memory to store temporary data and instructions in a computer.

2.10.2.4 Cache Memory

The computer's main memory stores program instructions and data that the CPU needs during normal operation. In order to get the maximum out of the system, this would normally require all of the system's main memory to have a speed comparable with that of the CPU. It is not economical for all the main memory to be high speed. Cache memory is a block of high-speed memory located between the main memory and the CPU. The cache memory block is the one that communicates directly with the CPU at high speed. It stores the most recently used instructions or data. When the processor needs data, it checks in the high-speed cache to see if the data are there. If they are there, called a 'cache hit', the CPU accesses the data from the cache. If they are not there, called a 'cache miss', then the CPU retrieves them from the relatively slower main memory. Cache memory mostly uses SRAM chips, but it can also use DRAM. There are two levels of cache memory. The first is the level 1 cache (L1 or primary or internal cache). It is physically a part of the microprocessor chip. The second is the level 2 cache (L2 or secondary or external cache). It is in the form of memory chips mounted external to the microprocessor. It is larger than the L1 cache. The L1 and L2 cache memories range from 2 to 64 kB and from 256 kB to 2 MB in size respectively. Some systems have higher-level caches (L3, L4, etc.), but L1 and L2 are the most common. Figure 2.13 shows the use of L1 and L2 cache memories in a computer system.

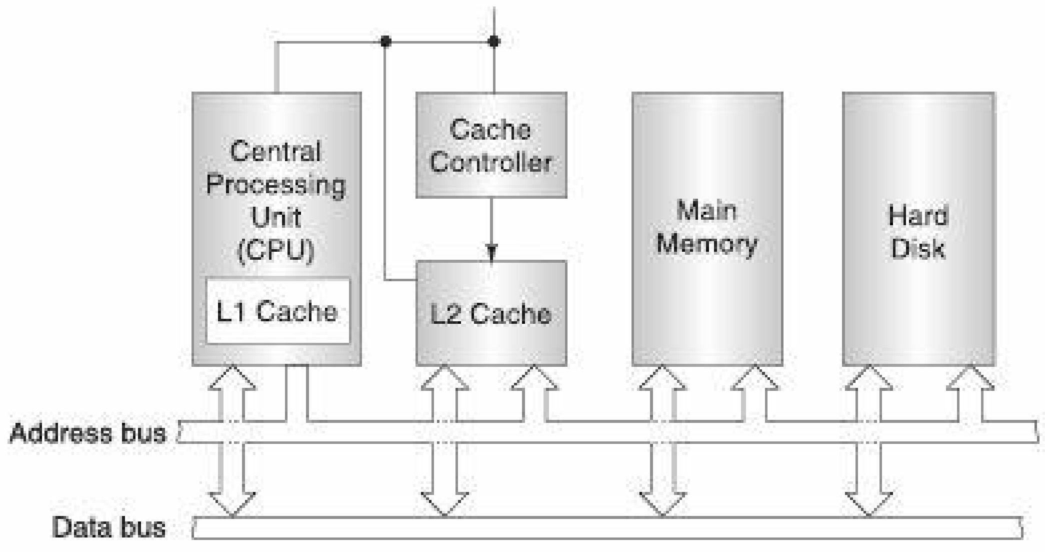


Fig. 2.13 Cache Memory in a Computer System.

2.10.3 Read Only Memory

ROM is a non-volatile memory that is used for permanent or semi-permanent storage of data. The contents of ROM are retained even after the power is turned off.

Working principle

The read only memory cell usually consists of a single transistor (ROM and EPROM cells consist of one transistor, EEPROM cells consist of one, one-and-a-half, or two transistors). The threshold voltage of the transistor determines whether it is a "1" or "0." During the read cycle, a voltage is placed on the gate of the cell. Depending on the programmed threshold voltage, the transistor will or will not drive a current. The sense amplifier transforms this current, or lack of current, into a "1" or "0." Figure 2.14 shows the basic principle of how a Read Only Memory works.

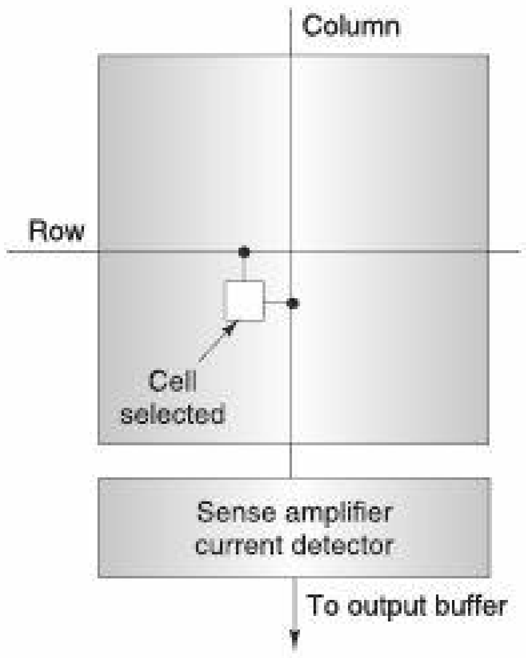


Fig. 2.14 Read Only Memory Schematic.

2.10.3.1 *Types of ROM*

Depending upon the methodology of programming, erasing and reprogramming information into ROMs, they are classified as mask-programmed ROMs, programmable ROMs (PROMs) and erasable programmable ROMs (EPROMs). EPROMs are further classified into Ultraviolet-erasable programmable ROMs (UV EPROMs) and electrically erasable programmable ROMs (EEPROMs).

2.10.3.2 *Mask Programmed ROMs*

In the case of a mask-programmed ROM, the ROM is programmed at the manufacturer's site according to the specifications of the customer. A photographic negative, called a mask, is used to store the required data on the ROM chip. A different mask would be needed for storing each different set of information. As preparation of a mask is an expensive proposition, mask-programmed ROM is economical only when manufactured in large quantities. The limitation of such a ROM is that, once programmed, it cannot be reprogrammed.

The basic storage element is an NPN bipolar transistor, connected in common-collector configuration, or a MOSFET in common drain configuration. Figures 2.15(a) and (b) show a MOSFET-based basic cell connection when storing a '1' and '0' respectively. As is clear from the figure, the connection of the 'row line' to the gate of the MOSFET stores '1' at the location when the 'row line' is set to level '1'. A floating-gate connection is used to store '0'. Figures 2.15(c) and (d) show the basic bipolar memory cell connection when storing a '1' and '0' respectively.

2.10.3.3 *Programmable ROM*

In the case of PROMs, instead of being done at the manufacturer's premises during the manufacturing process, the programming is done by the customer with the help of a special gadget called a PROM programmer. Since the data, once programmed, cannot be erased and reprogrammed, these devices are also referred to as one-time programmable ROMs.

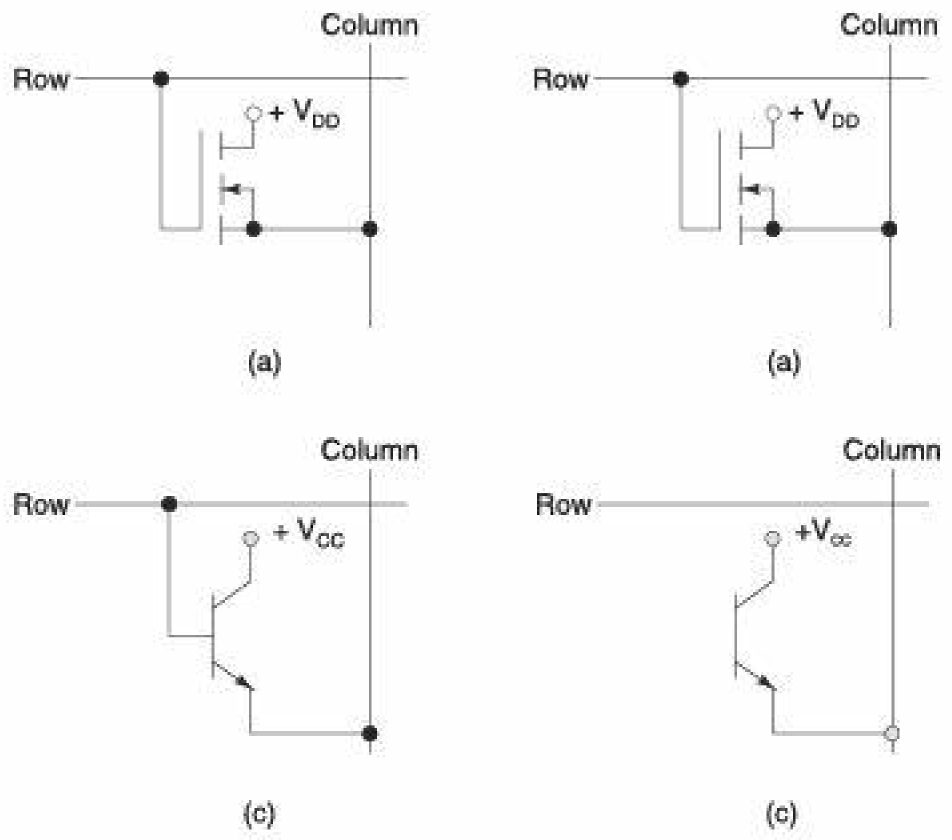


Fig. 2.15 Basic Cell Connection of a Mask-programmed ROM.

The basic memory cell of a PROM is similar to that of a mask programmed ROM. Figures 2.18(a) and (b) show a MOSFET-based memory cell and bipolar memory cell respectively. In the case of a PROM, each of the connections that was left either intact or open in the case of a mask-programmed ROM is made with a thin fusible link, as shown in Figure 2.16. Basic fuse technologies used in PROMs are metal links, silicon links and PN junctions. These fusible links can be selectively blown off to store the desired data. A sufficient current is injected through the fusible link to burn it open to store '0'.

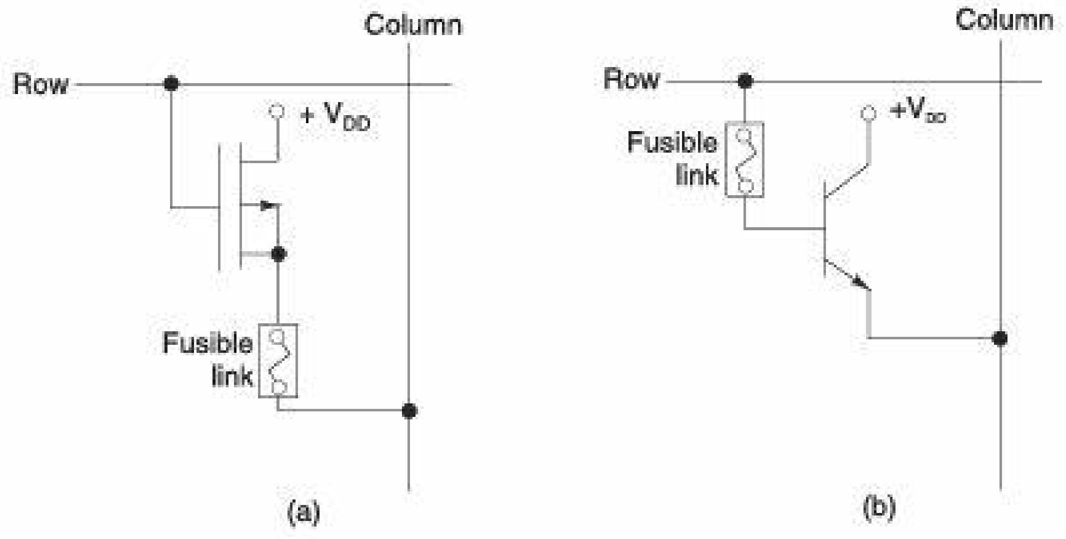


Fig. 2.16 Basic Memory Cell of a PROM.

The programming operation, as said earlier, is done with a PROM programmer. The PROM chip is plugged into the socket meant for the purpose. The programmer circuitry selects each address of the PROM one by one, burns in the required data and then verifies the correctness of the data before proceeding to the next address. The data are fed to the programmer from a keyboard or a disk drive or from a computer.

2.10.3.4 Erasable PROM

EPROM can be erased and reprogrammed as many times as desired. Once programmed, it is non-volatile, i.e., it holds the stored data indefinitely. There are two types of EPROM, namely, the ultraviolet-erasable PROM (UV EPROM) and electrically erasable PROM (EEPROM).

The memory cell in a UV EPROM is a MOS transistor with a floating gate. Figure 2.17 shows the cell used in a typical EPROM. The floating gate is where the electrical charge is stored. In the normal condition, the MOS transistor is OFF. It can be turned ON by applying a programming pulse (in the range 5 – 10 V) that injects electrons into the floating-gate region. These electrons remain trapped in the gate region even after removal of the programming pulse. This keeps the transistor ON once it is programmed to be in that state even after the removal of power. The stored information can, however, be erased by exposing the chip to ultraviolet radiation through a transparent window on the top of the chip meant for the purpose.

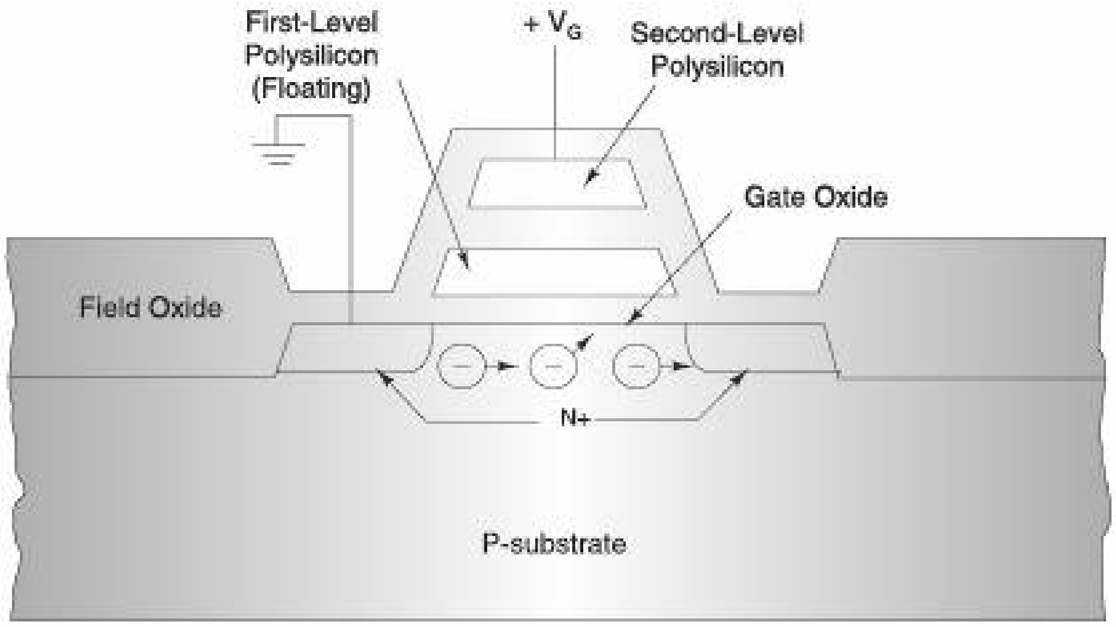


Fig. 2.17 Double-Poly Structure (EPROM/Flash Memory Cell).

The photocurrent thus produced removes the stored charge in the floating-gate region and brings the transistor back to the OFF state. The erasing operation takes around 15–20 min, and the process erases information on all cells of the chip. It is not possible to carry out any selective erasure of memory cells.

Intel's 2732 is 4K × 8 UV EPROM hardware implemented with NMOS devices. Type numbers 2764, 27128, 27256 and 27512 have capacities of 8K × 8, 16K × 8, 32K × 8 and 64K × 8 respectively. The access time is in the range 150–250 ns.

UV EPROMs suffer from disadvantages such as the need to remove the chip from the circuit if it is to be reprogrammed, the non-feasibility of carrying out selective erasure and the reprogramming process taking several tens of minutes. These are overcome in the EEPROMs and flash memories.

EEPROMs have another advantage – it is possible to erase and rewrite data in the individual bytes in the memory array. The EEPROMs, however, have lower density (bit capacity per square mm of silicon) and higher cost compared with UV EPROMs.

2.10.3.5 Flash Memory

Flash memories are high-density non-volatile read/write memories. Flash memory combines the low cost and high density features of an UV EPROM and the in-circuit electrical erasability feature of EEPROM without compromising the high-speed access of both. Structurally, the memory cell of a flash memory is like that of an EPROM. The basic memory cell of a flash memory is shown in Figure 2.18. It is a stacked-gate MOSFET with a control gate and floating gate in addition to drain and source.

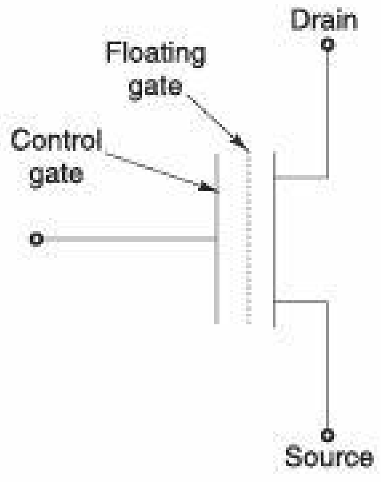


Fig. 2.18 Basic Cell of Flash Memory.

The floating gate stores charge when sufficient voltage is applied to the control gate. A ‘0’ is stored when there is more charge, and a ‘1’ when there is less charge. The amount of charge stored on the floating gate determines whether the MOSFET is turned ON or not. It is called flash memory because of its rapid erase and write times. Most flash memory devices use a ‘bulk erase’ operation in which all the memory cells on the chip are erased simultaneously. Some flash memory devices offer a ‘sector erase’ mode in which specific sectors of the memory device can be erased at a time. This mode comes in handy when only a portion of the memory needs to be updated.

To sum up, while PROMs are least complex and low cost, they cannot be erased and reprogrammed. UV EPROMs are a little more complex and costly, but then they can be erased and reprogrammed by being taken out of the circuit. Flash memories are in-circuit electrically erasable either sector wise or in bulk mode. The most complex and most expensive are the EEPROMs, but then they offer byte-by-byte electrical erasability in circuit.

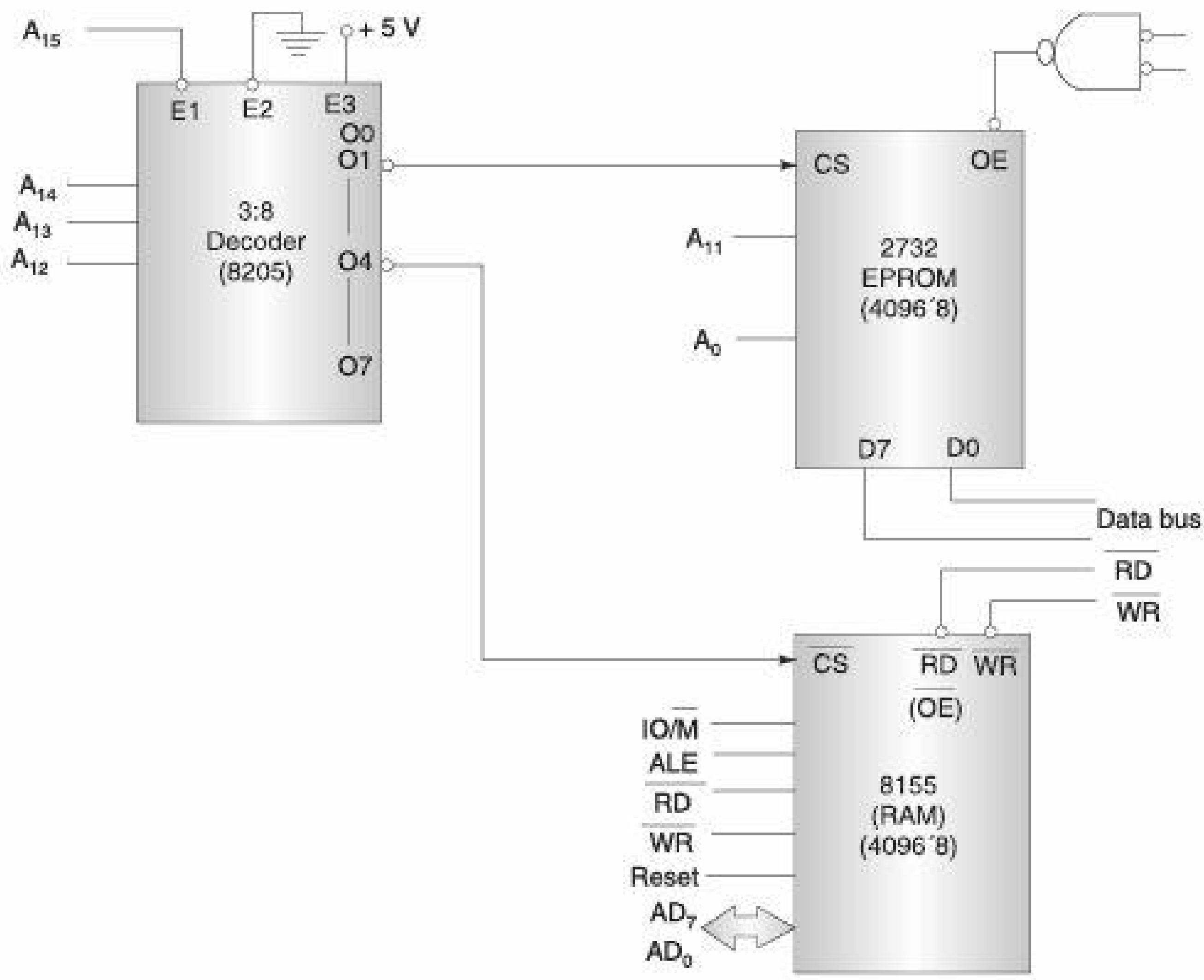


Fig. 2.19 Interfacing EPROM and RAM Memory Chips.

Note: In its strictest sense, ROM refers only to mask ROM (the oldest type of solid state ROM), which is fabricated with the desired data permanently stored in it, and thus can never be modified. However, more modern types such as EPROM and flash EEPROM can be erased and re-programmed multiple times; they are still described as "read-only memory" (ROM) because the reprogramming process is generally infrequent, comparatively slow, and often does not permit random access writes to individual memory locations. Despite the simplicity of mask ROM, economies of scale and field-programmability often make reprogrammable technologies more flexible and inexpensive, so mask ROM is rarely used in new products.

2.10.4 Interfacing EPROM, RAM with Memory and Memory Mapping

Example: Analyze the interfacing circuit shown in Figure 3.17 and find its memory address range.

Solution: As shown in figure, the EPROM (Electrically Programmable Read Only Memory) and RAM (Random Access Memory) memory chips are interfaced using 3:8 decoder. The 8205, is a 3 into 8 decoder that decodes address lines A_{15} –A_{12} and the output lines are O_0-O_7.

Out of these eight lines, O_1 line is used to activate the EPROM memory chip by making chip select ($\overline{CS}$) signal low. The address lines A_0-A_{11} are inputs of EPROM memory chip to decode 4096 (4 K) registers. Similarly, O_4 line of decoder is used to select the RAM memory. It needs 8 address lines as input to decode 256 K registers (to address any one of the 256 memory locations).

The control and status signals from 8085 are directly connected to the respective signals on the memory chips. Starting address is calculated by making all input address lines to 0 (low) and end address calculated by making all input address lines to 1 (high). The address lines A_{15} –A_{12} are configured as 1H to make O_1 line high to select the EPROM chip as:

A_{15}	A_{14}	A_{13}	A_{12}	
0	0	0	1	= 1H (To select the chip)

The memory address of the EPROM memory chip ranges from 1000H to 1FFFH as shown below as:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 1000H (Starting Address)

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	= 1FFFH (End Address)

The address lines A_{15} –A_8 are configured as 40H to make O4 line high to select the RAM chip, because O4 line is connected to the chip select line of the RAM chip. The address lines A11-A8 are don't care lines and should be kept low.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	
0	1	0	0	0	0	0	0	= 40H (To select the chip)

The memory address of the RAM memory chip ranges from 4000H to 40FFH as shown below as:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	= 4000H
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(Starting Address)

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	= 40FFH
0	1	0	0	0	0	0	01	1	1	1	1	1	1	1	1	(End Address)

2.11 ANALOG/DIGITAL CONVERSION

In this section, we discuss the important topic of analog to digital conversion (often written A/D), and digital to analog conversion (D/A). On the one hand, most electrical measurements are intrinsically analog. But to take advantage of the great capabilities available for digital data storage, processing, and computation, we have to convert it into analog to digital. Hence, analog to digital (A/D) conversion techniques have become extremely important. A great deal of technical effort has gone into producing A/D converters (ADCs) which are fast, accurate, and cheap. D/A converters (DACs) are also very important. For example, video monitors convert digital information generated by computers to analog signals which are used to direct the electron beam at a specified portion of the monitor screen. DACs are conceptually simpler than ADCs, although it is difficult to build a precise DAC.

2.11.1 A/D Resolution

First of all, we should keep in mind that there are several different schemes for encoding analog information as bits, depending upon what is required by a particular application. One extreme is that of encoding the complete analog signal in as much detail as possible. For example, a musical instrument produces an analog signal which is readily converted into an analog electrical signal using a microphone. If this is to be recorded digitally, one naturally would choose to digitize enough information so that when the recording is played back, the resulting audio is not perceived to be significantly different from the original. In this case, the analog signal is a voltage which varies with time, $V(t)$.

At any time t_0, $V(t_0)$ can be sampled and converted into digital. The analog signal must be sampled for a finite time, called the *sampling time*, Δt. One may guess that it is necessary to sample the analog signal continuously, with no gaps between consecutive samples. This turns out to be overkill. The *Nyquist Theorem* states that if the maximum frequency of interest in the analog input is f_{max}, then perfect reproduction only requires that the sampling frequency f_{samp} be slightly greater than twice f_{max}. That is,

$$f_{samp} > 2f_{max}$$

For example, for audio signals, the maximum frequency of interest is usually 20 kHz. In this case, the input analog must be sampled at a little over 40 kHz. In fact, 44 kHz is typically used.

Alternatively, it might not be of interest to represent the entire analog input digitally. Perhaps only one feature of the analog signal is useful. One example is "peak sensing," where one samples and digitizes the input only at the instant where an instrument's output achieves a maximum analog output-or one may average ("integrate") an input signal over some predefined time, retaining only the average value to be digitized.

For any of these sampling schemes, there remains the issue of how many bits are to be used to describe the sampled signal $V(t_0)$. This is the question of A/D *resolution*. We need a standard definition of resolution. Let's say, for example, that we choose to digitize the input using 12 bits. This means that we will try to match our analog input to 1 of $2^{12} = 4096$ possible levels. This is generally done by ascribing a number from 0 to 4095. So, assuming our ADC works correctly, the digital estimate of the analog input can, at worst, be wrong by the range of the LSB. On average, the error is half of this. This defines the resolution. Therefore, for our 12-bit example, the resolution is 1= (2 _ 4096), or a little worse than 0:01%.

2.11.2 D/A Conversion

The basic element of a DAC is the simplest analog divider: the resistor. First, we need to review the two important properties of an operational amplifier ("op-amp") connected in the inverting configuration. This is shown in Figure 2.33. The two important properties are:

1. The "–" input is effectively at ground. ("virtual ground")
2. The voltage gain is $G = V_{OUT}/V_{IN} = -R2/R1$

An equivalent statement is that for a current at the – input of $I_{IN} = V_{IN}=R_1$, the output voltage is $V_{OUT} = GV_{IN} = -R_2I = -V_{IN}R_2/R_1$. Sometimes this is written in the form $V_{out} = gI_{in}$, where g is the *transconductance*, and $g = -R_2$ in this case.

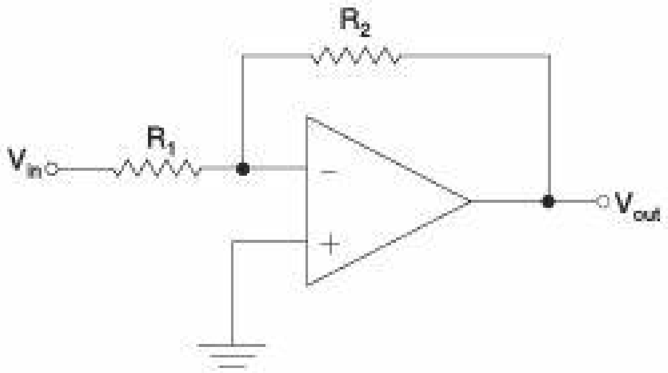


Fig. 2.20 Inverting Op-amp Configuration.

The basic idea of most DACs is then made clear by the 4-bit example illustrated in figure. The input 4-bit digital signal defines the position of the switches labeled a_0-a_3. A HIGH input bit would correspond to a switch connected to 1:0 V, whereas a LOW connects to ground. The configuration in the figure represents a binary input of 1010, or 1010. Since the virtual ground keeps the op-amp input at ground, then for a switch connected to ground, there can be no current flow. However, for switches connected to 1:0 V, the current presented to the op-amp will be 1:0 V divided by the resistance of that leg. All legs with HIGH switches then contribute some current. With the binary progression of resistance values shown in the figure, the desired result is obtained. So for the example shown, the total current to the op-amp is I = 1:0/R + 1:0/(4R) = 5/(4R). The output voltage is

$$V_{OUT} = -IR = -5/4 = -1.25 \text{ V}$$

When all input bits are HIGH (1111 = 1510), we find V_{out} = 15/8 V. A simple check of our scheme shows that (5/4)/ (15/8) = 2/3 = 10/15 = 1010/1111 as expected.

Things to Remember

◊ Bit is a binary digit that can have the value 0 or 1.

◊ A byte is defined as 8 bits, a nibble is half a byte, a word is two bytes, a double word is four bytes and a kilobyte is 2^10 bytes (1024 bytes). The abbreviation K is most often used.

◊ The control unit controls the flow of information between all the units.

◊ ALU (arithmetic logic control) unit performs arithmetic and all other data processing tasks.

◊ Base or radix (r) of a number system is defined as the number of different symbols (digits or characters) used in that number system.

◊ The radix of binary number system is 2, i.e., it uses two different symbols 0 and 1 to write the number sequence.
◊ The radix of octal number system is 8, i.e., it uses eight different symbols 0, 1, 2, 3, 4, 5, 6 and 7 to write the number sequence.
◊ The radix of decimal number system is 10, i.e., it uses ten different symbols 0, 2, 3, 4, 5, 6, 7, 8 and 9 to write the number sequence.
◊ The radix of hexadecimal number system is 16, i.e., it uses sixteen different symbols 0, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F to write the number sequence.
◊ The largest value of digit is one (1) less than the radix.
◊ Binary coded decimal is also called natural BCD or 8421 code.
◊ Excess three codes assignment is obtained from the corresponding value of BCD after the addition of 3.
◊ Gray code is also known as reflected code.
◊ Binary code hexadecimal code is represented by its four-bit binary equivalent.
◊ Multiplexing means transmitting large number of information units over a smaller number of channels or lines.
◊ A demultiplexer is a circuit that receives information on a single line and transmits that information on a one of 2n possible output lines.
◊ The counter is a frequency divider circuit that is driven by a clock signal and can be used to count the number of clock cycles.
◊ Shift register can be used as ring counter.
◊ Twisted ring counter is also called Johnson's ring counter.
◊ The basic logic gates are AND, OR, NAND, NOR, XOR, INV, and BUF. The last two are not standard terms; they stand for "inverter" and "buffer", respectively.
◊ The half adder adds to one-bit binary numbers (AB). Full adder is called "full" because it will include a "carry-in" bit and a "carry-out" bit. The carry bits will allow a succession of 1-bit full adders to be used to add binary numbers of arbitrary length.
◊ For simplifying the subtraction operation and logic manipulations, complements are used in digital computers.

Exercise

1. How many symbols can be represented with 4 bits?
2. Convert the following decimal numbers into binary.
   (a) 15 (b) 92 (c) 1024
3. Convert the following binary numbers into decimal.
   (a) 10101 (b) 1000000 (c) 1111111
4. Convert the following decimal numbers into hexadecimal.
   (a) 64 (b) 98 (c) 47
5. Convert the following hex numbers into binary directly without first converting them to decimal.
   (a) F8A (b) 144B

6. Convert 163.875_{10} into binary.
7. Convert 105.15_{10} into binary.
8. Divide 101101_2 by 110_2.
9. Multiply 1100_2 by 1011_2.
10. Convert 756.6038 into hex.
11. Convert $5C7_{16}$ into decimal.
12. Convert 756.6038 into hex.
13. Perform the following binary arithmetic:
    (a) 00110110 + 00110011 (b) 1101 + 0101
    (c) 00110101 – 00001010 (d) 0011 – 0111
14. Perform the following binary arithmetic:
    (a) 11011 + 1101 (b) 1011 + 1101 + 1001 + 1111
    (c) 10111.101 + 110111.01 (d) 11011 + 1101.10 + 1001.11 + 1111.11
15. Draw the logical diagram that implements:
    $A = (Y_1 + Y_2)(Y_3 + Y_4) + (Y_5 + Y_6 + Y_7)$
16. What is the main feature of Gray code which is different from ordinary binary code?
17. Write the truth tables for a 3-input NAND and a 3-input NOR function.
18. Write the truth table for the function $S = A \otimes B + C \otimes D$.
19. Rewrite A + B + C + D using only NOT functions and two-input NAND functions.
20. Show that $A \cdot C + AC + BC = B + C$.
21. Show that $A \cdot (B + A \cdot C)$ can be implemented using one 3-input AND gate.
22. Using a Karnaugh map, show that $AC + AB + ABC + A \cdot B \cdot C = A + B \cdot C$.
23. Use a Karnaugh map to reduce each expression to a minimum SOP form:
    (a) $X = A + BC + CD$
    (b) $X = ABCD + ABCD + ABCD + ABCD$
    (c) $X = AB(CD + CD) + AB(CD + CD) + ABCD$
    (d) $X = (AB + AB)(CD + CD)e)$ $X = AB + AB + CD + CD$
24. Given a two-input multiplexer, write down its truth table and use it to implement
    (a) an AND gate and (b) an OR gate.
25. For a multiplexer with control inputs A and B, derive the required data inputs to implement
    (a) the carry function of a full adder (b) $f = (A + B)(A + C)$.
26. What values do the following numbers have if they are (a) unsigned and (b) signed?
27. Design a half adder using only NAND gates
28. Design a full adder using 2 half adders and an OR gate.
29. What is the range of possible results from (a) adding and (b) multiplying two N-bit signed numbers? How many bits are required to represent the answer in each case?
30. Construct a Karnaugh map for each flip-flop expressing the next value of Q in terms of the present value of Q and the inputs. Hence, show how you could synthesise a T flip-flop and a JK flip-flop using D flip-flops and appropriate gates.

31. Why is it necessary for an output port to *latch* the data written to it by the CPU?
32. Show that a binary downcounter can be implemented using a binary upcounter. To do this, draw the transition table for a 4-bit upcounter and for a 4-bit downcounter and then consider the relationship between corresponding bits in the two tables.
33. Without reducing, convert the following expressions to NAND logic
    (a) $(A + B)(C + D)$ (b) $(A + C)(ABC + ACD)$
    (c) $(A + \overline{B}C)D$ (d) $(AB + CD)(A\overline{B} + CD)$
34. Prove that
    (a) If $A + B = A + C$ and $\overline{A} + B = \overline{A} + C$, then $B = C$
    (b) If $A + B = A + C$ and $AB = AC$, then $B = C$
35. Given $A\overline{B} + \overline{A}B = C$, show that $A\overline{C} + \overline{A}C = B$.

Chapter 3

Intel 8085 Microprocessor

- Introduction
- Microprocessor Architecture
- Pin Configuration
- The 8085 Machine Cycles and Bus Timings
- Instruction Execution in 8085
- Generating Control Signals
- Instructions Types in 8085
- Brief Introduction to 8085 Instruction Set
- Instruction Format and Assembly Language Programming of 8085
- Instructions, Hex Codes, Machine Cycle and T States of 8085
- Timing Diagram of Various Instructions
- Addressing Modes
- Instruction Set
- I/O Execution
- Stack and Subroutines
- Interrupts in 8085
- Serial Communication in 8085
- Direct Memory Access (DMA) with 8085
- Time Delay Generation
- Assembly Language Programs

3.1 INTRODUCTION

In this chapter, we will discuss the Intel 8085 microprocessor. As we know 8085 microprocessor is a programmable device which consists of register array, arithmetic, logical and control units. It is a 40-pin IC and the data bus consists of 8-bit, that is why, this microprocessor is called 8-bit microprocessor. The 8085 is an enhanced version of its predecessor, the 8080A, i.e., all instructions of 8080A can be executed in 8085 although these are not pin compatible.

3.2 MICROPROCESSOR ARCHITECTURE

Intel 8085 is an 8-bit, NMOS microprocessor. It is a 40-pin IC package fabricated on a single LSI chip. The Intel 8085 uses a single +5 V DC supply for its operation. Its clock speed is about 3 MHz and the clock cycle is of 320 ns. The time for the clock cycle of the Intel 8085AH-2, version is 200 ns. It has 74 basic instructions and 256 opcodes. It consists of three main sections: an arithmetic and logic unit, timing and control unit and a set of registers. These important sections are described in the subsequent sections. This process of data manipulation and communication is determined by the logic design of the microprocessor, called the architecture. The microprocessor can be programmed to perform functions on the given data by selecting necessary instructions from its set. These instructions are given to the microprocessor by writing them into its memory. Writing (or entering) instructions and data are done through an input device such as a keyboard. The microprocessor reads or transfers one instruction at a time, matches it with its instruction set, and performs the data manipulation, as indicated by the instruction. The result can be stored in memory or sent to output devices such as LEDs or a CRT terminal. In addition, the microprocessor can respond to external signals. It can be interrupted, reset, or asked to wait to synchronize with slower peripherals. To perform these functions, the microprocessor requires a group of logic circuits and a set of signals called control signals. However, early processors did not have the necessary circuitry on one chip; the complete units were made up of more than one chip. Therefore, the term microprocessing unit (MPU) is defined here as a group of devices, that can perform these functions with the necessary set of control signals.

3.2.1 Register Section

The register section includes all the register that are required during the execution of the programs by microprocessor. 8085 microprocessor contains several registers named as

(i) One 8-bit accumulator (ACC), i.e., register A
(ii) Six 8-bit general-purpose registers (B, C, D, E, H and L)
(iii) One 16-bit stack pointer, SP
(iv) One 16-bit program counter, PC
(v) Instructions register
(vi) Temporary register

Accumulator (ACC): The accumulator is an 8-bit register associated with the ALU. The register 'A' in the 8085 is an accumulator. It is used to hold one of the operands of an arithmetic or logical operation. It serves as one input to the ALU. The other operand for an arithmetic or

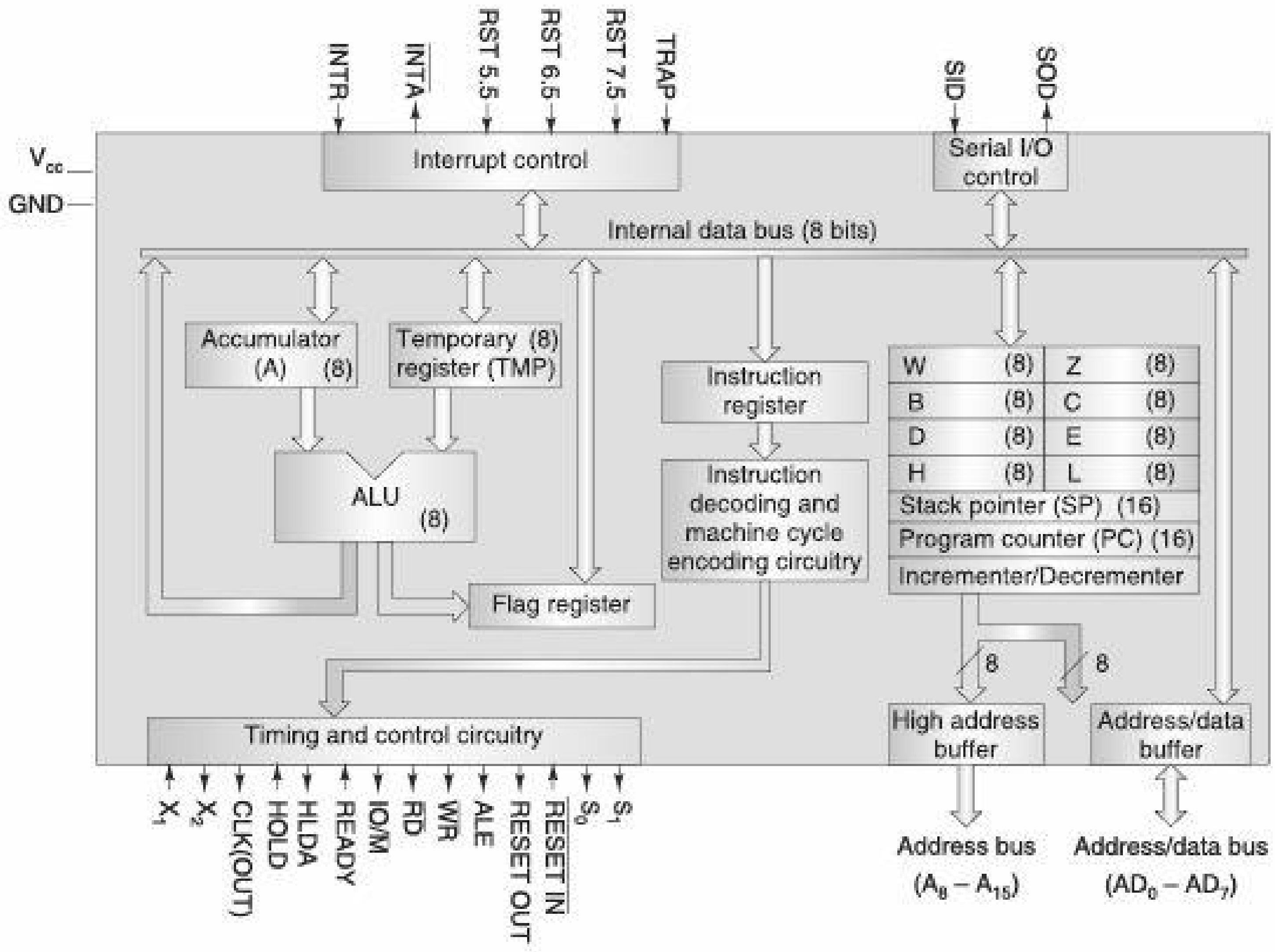


Fig. 3.1 Microprocessor Architecture.

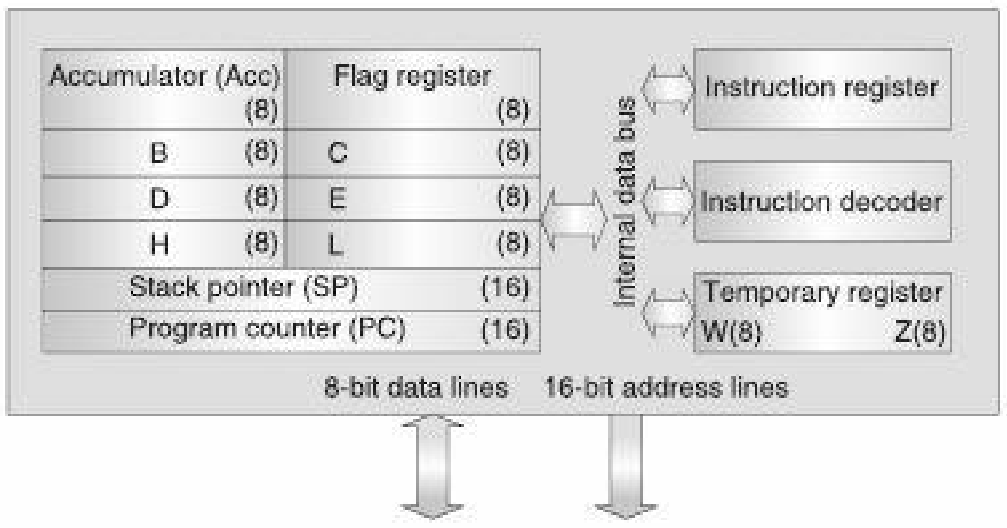


Fig. 3.2 Register Architecture.

logical operation may be stored either in the memory or in one of the general-purpose registers. The final result of an arithmetic or logical operation is placed in the accumulator. These above descriptions are true for general cases, not for some typical or exceptional cases. For example, there are some logical instructions which need only one operand. It is held in the accumulator.

The result is placed in the accumulator. Such instructions do not require any other register or memory location because there is no other operand.

General-Purpose Registers: The 8085 microprocessor contains six 8-bit general-purpose registers. They are: B, C, D, E, H and L registers. To hold 16-bit data, a combination of two 8-bit registers can be employed. The combination of two 8-bit registers is known as a register-pair. The valid register-pairs in the 8085 are: B-C, D-E and H-L. The programmer cannot form a register-pair by selecting any two registers of his choice. The H-L pair is used to act as memory pointer and for this purpose it holds the 16-bit address of a memory location. The general-purpose registers and the accumulator are accessible to the programmer. He can store data in these registers during writing his program.

Program Counter (PC): It is a 16-bit special-purpose register. It is used to hold the memory address of next instruction to be executed. It keeps the track of memory addresses of the instructions in a program while they are being executed. The microprocessor increments the content of the program counter during the execution of an instruction so that it points to the address of the next instruction in the program at the end of the execution of the current instruction.

Stack Pointer (SP): It is a 16-bit special-function register which holds the address of the topmost filled stack memory. The stack is a sequence of memory locations set aside by a programmer to store/retrieve the contents of accumulator, flags, program counter and general-purpose registers during the execution of a program. Usually the topmost memory location is initialized for stack. Since the stack works on LIFO (last-in-first-out) principle, its operation is faster compared to normal store/retrieve of memory locations. During the execution of a program, sometimes it becomes necessary to save the contents of some registers which are needed for some other operations in the subsequent the program. The contents of such registers are saved in the stack. Then the registers are used for some other operations. After completing the needed operations, the contents which were saved in the stack are brought back to the registers. The contents of only those registers are saved, which are needed in the later part of the program. The stack pointer (SP) controls addressing of the stack. The stack is defined and the stack pointer is initialized by the programmer at the beginning of a program which needs stack operation. Stack is also used by the microprocessor. For example, it stores the contents of program counter when it jumps to a subroutine using CALL instruction.

Instruction Register: The instruction register holds the opcode (operation code or instruction code) of the instruction which is being decoded and executed.

Temporary Register: There are two 8-bit registers associated with the ALU. They hold data during an arithmetic/logical operation temporarily. They are used by the microprocessor and are not accessible to programmer. These are named W and Z registers. These registers are used to hold 8-bit data during the execution of some instructions. However, because they are used internally, they are not available to the programmer.

Flags: In addition to the above-mentioned registers, the 8085 microprocessor contains a set of five flip-flops which serve as flags (or status flags). A flag is a flip-flop which indicates some condition which arises after the execution of an arithmetic or logical instruction. The five status flags of Intel 8085 are:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	X	AC	X	P	X	CY

S – Sign flag	Z – Zero flag
AC – Auxiliary flag	P – Parity flag
CY – Carry flag	X – Don't care

S-Sign flag: After the execution of an arithmetic or logic operation, if bit D_7 of the result (usually in the accumulator) is 1, the sign flag is set. This flag is used with signed numbers. In arithmetic operations with signed numbers, bit D_7 is reserved for indicating the sign, and the remaining seven bits are used to represent the magnitude of a number. In a given byte, if D_7 is 1, the number will be viewed as a negative number; if it is 0, the number will be considered positive. The sign flag has its significance only when signed arithmetic operation is performed.

For unsigned arithmetic operation, all the 8 bits are used to represent the magnitude of the number. After the execution of an arithmetic operation, all the 8 bits of the result represent its magnitude. Therefore, the sign flag has no significance in unsigned arithmetic operation. Also, for logical operation sign bit has no significance. Since the sign flag is set or reset according to the MSB of the result, it is set or reset on the value of MSB of the result of logical operation also.

Z-Zero flag: The zero flag is set if the ALU operation results in 0, and the flag is reset if the result is not 0. This flag is modified by the results in the accumulator as well as in the other registers.

AC-Auxiliary Carry flag: In an arithmetic operation, when a carry is generated by digit D_3 and passed on to digit D_4, the AC flag is set. The flag is used only internally for BCD (Binary Coded Decimal) operations and is not available for the programmer to use with conditional instructions.

P-Parity flag: After an arithmetic or logical operation, if the result has an even number of 1s, the flag is set. If it has an odd number of 1s, the flag is reset. (For example, the data byte 00000101 has even parity even if the magnitude of the number is odd.)

CY-Carry flag: If an arithmetic operation results in a carry, the carry flag is set; otherwise it is reset. The carry flag also serves as a borrow flag for subtraction. The bit positions reserved for these flags in the flag register are as follows:

STATUS FLAG REGISTER

PSW (Program status word): The PSW is a register pair of 16-bit. This includes accumulator register and flag register. The accumulator is used to store the result during the execution of the instructions and flag are affected by the current status of accumulator register. It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	S	Z	X	AC	X	P	X	CY
Accumulator								**Flag register**							

Program Status Word

3.2.2 Arithmetic and Logical Unit

The most important unit of a microprocessor is ALU. All the arithmetic and logical operations are performed by ALU on the operands for the implementation of data transformation. As already discussed that one operand that is to be used for the operation must be in accumulator and the result that comes out by certain processing by ALU is stored in the accumulator. The block diagram of ALU is shown in Fig. 3.3, the main components of ALU are accumulator, and other logic circuitry. It carries several operations such as binary addition, subtraction and logical operations, i.e., AND, OR, NOT, XOR, increment, decrement, shift, clear, etc. Other certain mathematical operations (division, multiplication) are also performed by the ALU but they are performed by making programs, i.e., for multiplication we use a counter and using addition repeatedly with the help of a counter.

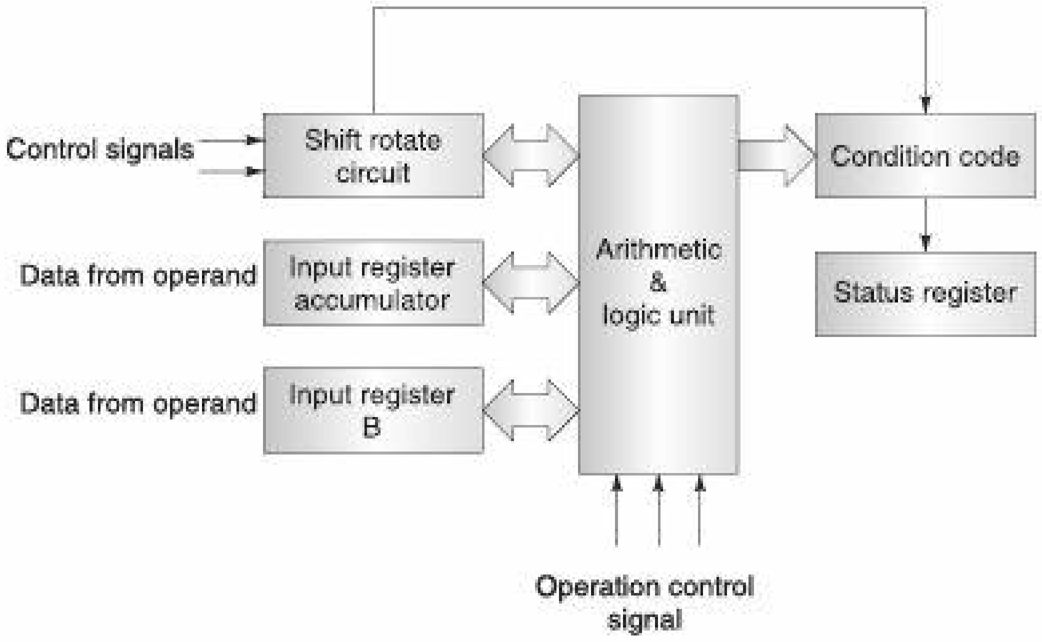


Fig. 3.3 Arithmetic and Logic Unit.

3.3 PIN CONFIGURATION

The architecture of 8085 indicates the internal circuitry of microprocessor but as it is a 40-pin IC, so each pin performs a certain function, on the basis of the pin diagram, we further classify pin into seven groups.

1. Data Pins
2. Address Pins
3. Interrupt and other Peripheral Signal Pins
4. Serial IN/OUT Signal Pins
5. Control and Status Signal Pins
6. Power Supply and GND Pins
7. Oscillator Pins

The description of various pins as shown in Figure 3.4 is as follows:

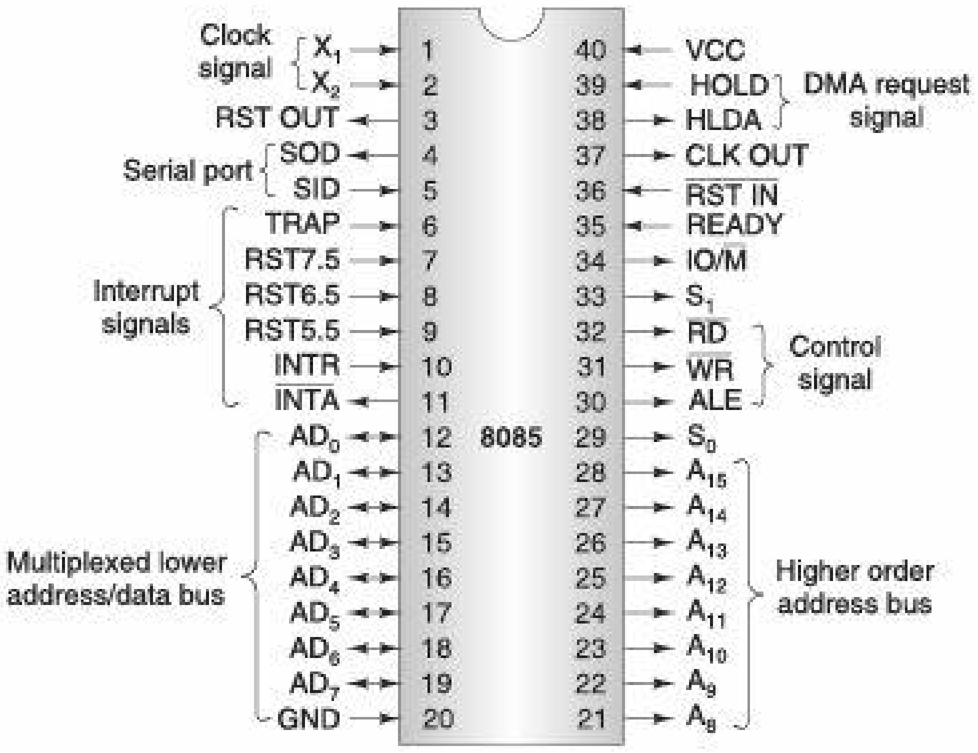


Fig. 3.4 Pin Diagram of 8085.

The pins can functions as either input pin or output pin but some pins perform both operations at different clock periods these pins are called bidirectional pins.

A_8-A_{15} (Output): These are address bus used for the most significant bits of the memory address or 8 bits of I/O address.

AD_0-AD_7 (Input/Output): These are the time multiplexed address/data buses, i.e., they serve dual purpose. They are used for the least significant 8 bits of the memory address or I/O address during the first clock cycle of a machine cycle. Again they are used for data during second and third clock cycles.

ALE: It is an address latch enable signal. It goes high during first clock cycle of every machine cycle and enables the lower 8 bits of the address to be latched either into the memory or external latch.

$IO/\overline{M}$: It is a status signal which distinguishes whether the address is for memory or I/O. When it goes high the address on the address bus is for an I/O device and when it goes low the address on the address bus is for a memory location.

$\overline{RD}$: It is a signal to control READ operation. When it goes low the selected memory or I/O device is read.

$\overline{WR}$: It is a signal to control WRITE operation. When it goes low the data on the data bus is written into the selected memory or I/O location.

S_0, S_1: These are the status signals sent by the microprocessor to distinguish the various types of operations given in Table 3.1.

READY: It is used by the microprocessor to sense whether a peripheral is ready to transfer the data or not. A slow peripheral may be connected to the microprocessor through READY line. If READY is high the peripheral is ready. If it is low the microprocessor waits till it goes high.

Table 3.1 Status of machine cycle

Machine Cycle	*IO/$\overline{M}$*	*S_1*	*S_0*	*Operations*
Opcode Fetch	0	1	1	Read
Memory Read	0	1	0	Read
Memory Write	0	0	1	Write
I/O Read	1	1	0	Read
I/O Write	1	0	1	Write
Interrupt Acknowledge	1	1	1	$\overline{INTA}$
Halt	Z	0	0	$\overline{RD}$ = Z
Hold	Z	X	X	$\overline{WR}$ = Z
Reset	Z	X	X	$\overline{INTA}$ = 1

HOLD: It indicates that another device is requesting for the use of the address and the data bus. After receiving a Hold request the microprocessor relinquishes the use of the buses as soon as the current machine cycle is completed. Internal processing may continue. The processor regains the bus after the removal of the Hold signal. When a Hold is acknowledged, address bus, Data bus, $\overline{RD}$, $\overline{WR}$ and IO/$\overline{M}$ are tri-stated. Hold is sampled in T_2 clock cycle.

HLDA: It is a signal for hold acknowledgement. It indicates that the Hold request has been received. After the removal of a Hold request the HLDA goes low. The CPU takes over the buses half clock cycle after the HLDA goes low.

INTR: It is an interrupt request signal. Among interrupts it has the lowest priority. When it goes high the program counter does not increment its content. The microprocessor suspends its normal sequence of instructions. After completing the instruction at hand it attends the interrupting device. The $\overline{INTR}$ line is sampled in the last state of the last machine cycle of an instruction. The microprocessor acknowledges the interrupt signal and issues INTA signal. The INTR is enabled or disabled by the software. An interrupt is used by I/O devices to transfer data to the microprocessor without wasting its time. If CPU is in Hold state or interrupt enable flip-flop is reset, an interrupt request is not processed.

$\overline{INTA}$: It is an interrupt acknowledgement sent by the microprocessor after INTR is received.

RST 5.5, 6.5, 7.5, and TRAP: These are interrupts. When an interrupt is recognized the next instruction is executed from a fixed location in the memory as given below:

RST 5.5: It is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 0X2CH (hexadecimal) address.

RST 6.5: It is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 0X34H (hexadecimal) address. It is only level triggered interrupt.

RST 7.5: It is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 0X3CH (hexadecimal) address. It is only edge triggered interrupt.

TRAP: It is a non-maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 24H (hexadecimal) address. It is both edge and level triggered interrupt.

The order of priority of interrupts is as follows:

Table 3.2 Various interrupts in 8085

Interrupt	*Priority*	*Vector address*	*Maskable/Non-maskable*
TRAP	1-Highest priority	24 H	Non-maskable
RST 7.5	2	3C H	Maskable
RST 6.5	3	34 H	Maskable
RST5.5	4	2C H	Maskable
INTR	5	Non Vectored	Maskable

$\overline{\text{RESET IN}}$: It resets the program counter to zero. It also resets interrupt enable and HLDA (Hold Acknowledgement) flip-flops. It does not affect any other flag or register except the instruction register. The CPU is held in reset condition as long as RESET is applied.

RESET OUT: It indicates that the CPU is being reset. It can be used to reset other devices.

X1, X2: These are the terminals to be connected to an external crystal oscillator which drives the internal circuitry of the microprocessor to produce a suitable clock for the operation of the microprocessor.

CLK: It is a clock output for user, which can be used for the other digital IC's. The frequency of clock is same at which the processor operates.

SID: It is a data line for serial input. The data on this line is loaded into the 7th bit of the accumulator when RIM instruction is executed.

SOD: It is a data line for serial output. The 7th bit of the accumulator is the output on SOD line when SIM instruction is executed.

Vcc: +5 Volts supply.

Vss: Ground Reference.

3.3.1 Demultiplexing the Address and Data Bus

The 8085 uses a multiplexed Data Bus. The address is split between the higher 8-bit Address Bus and the lower 8-bit Address/Data Bus. During the first cycle the address is sent out. The lower 8-bits are latched into the peripherals by the Address Latch Enable (ALE). During the rest of the machine cycle the Data Bus is used for memory or I/O data. The need for demultiplexing the bus AD_7-AD_0 becomes easier to understand after examining. Figure 3.5 shows the connection for multiplexed address and data buses.

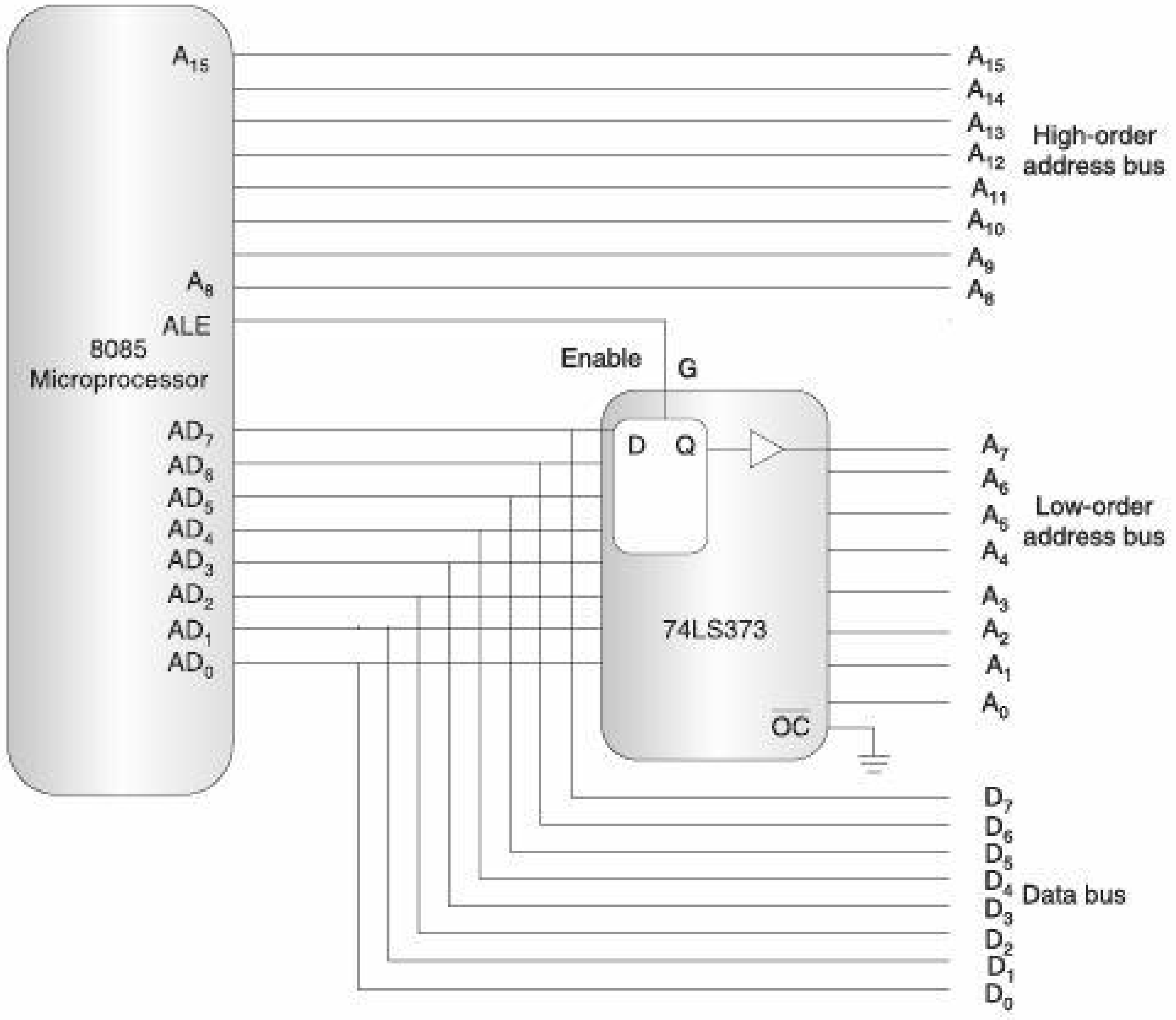


Fig. 3.5(a) Demultiplexing of Address and Data Bus.

Figure 3.5(b) shows that the address on the high-order bus (20 H) remains on the bus for three clock periods. However, the low-order address (05 H) is lost after the first clock period.

This address needs to be latched and used for identifying the memory address. If the bus AD_0-AD_5 is used to identify the memory location (2005 H), the lower order multiplexed address will change to 4FH data after the first clock period. Figure 3.5(a) shows a schematic that uses a latch and the ALE signal to demultiplexing the bus. The bus AD_0-AD_7 is connected as the input to the latch 74LS373.

The ALE signal is connected to the Enable (G) pin of the latch, and the Output Control ($\overline{OC}$) signal of the latch is grounded. The ALE goes high during T_1. When the ALE is high, the latch is transparent; this means that the output changes according to input data. During T_1 the out-put of the latch is 05 H. When the ALE goes low, the data byte 05 H is latched until the next ALE and the output of the latch represent the low-order address bus A_7-A_0 after the latching operation. Intel has circumvented the problem of demultiplexing the low-order bus by designing special devices such as the 8155 (256 bytes of R/W memory + I/O's), which is compatible with the 8085 multiplexed bus. These devices internally demultiplex the bus using the ALE signal. After carefully examining the Figure 3.5(b), we can make the following observations:

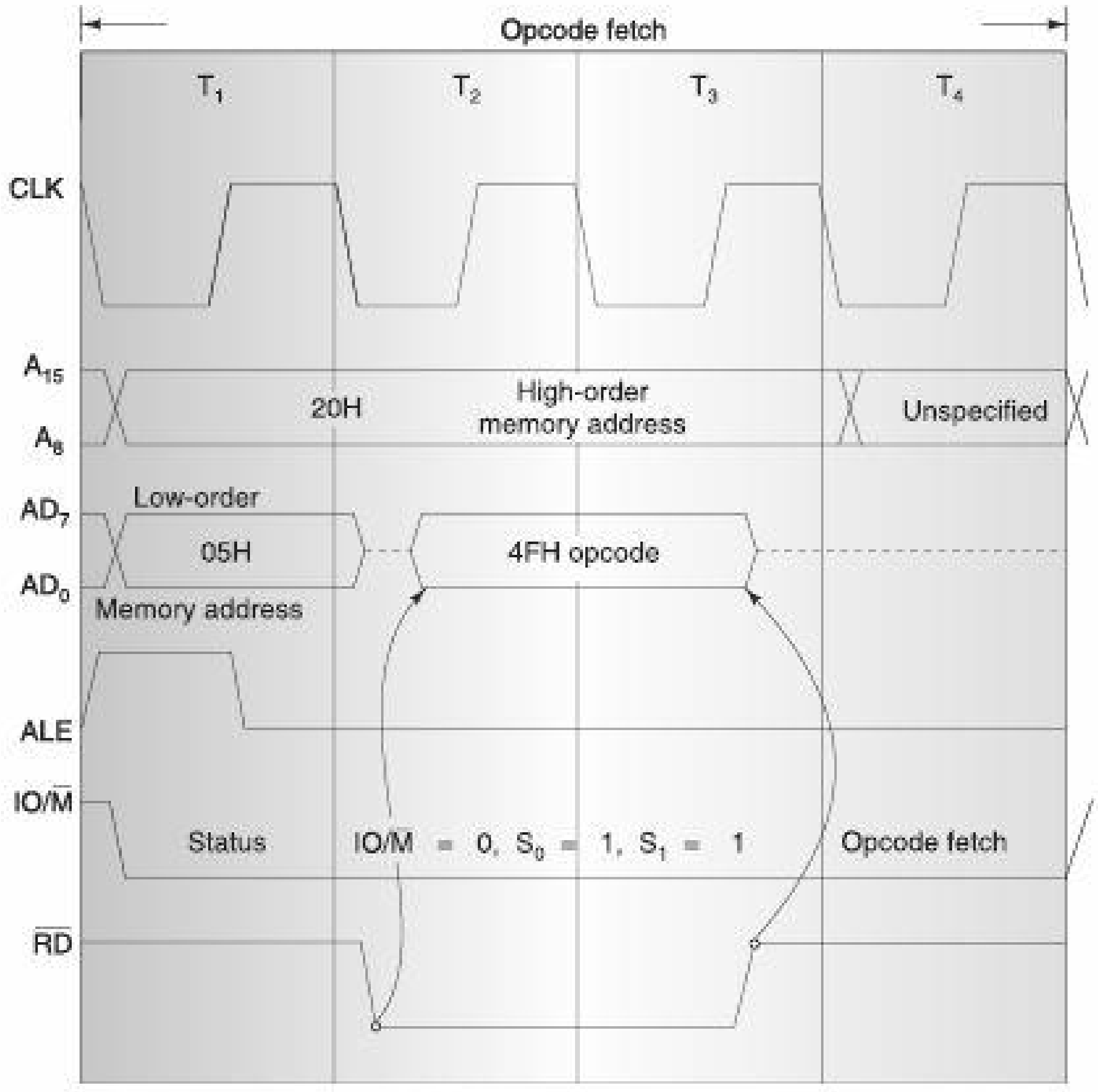


Fig. 3.5(b) Demultiplexing Timing Diagram.

1. The machine code 4FH (0100 1000) is a one-byte instruction that copies the contents of the accumulator into register C for instruction MOV C: A.
2. The 8085 microprocessor requires one external operation-fetching a machine code from memory location 2005 H.
3. The entire operation—fetching, decoding, and executing—requires four clock periods.

Decoding and executing an instruction after it has been fetched can be illustrated with the following example.

Example: Assume that the accumulator contains data byte 82 H, and the instruction MOV B, A (42H) is fetched. List the steps in decoding and executing the instruction.

Solution: In this example, we have the contents of the accumulator opcode 47. To decode and execute the instruction, the following steps are performed.

1. The contents of the data bus (42H) are placed in the instruction register and decoded.
2. The contents of the accumulator (82H) are transferred to the temporary register in the ALU.
3. The contents of the temporary register are transferred to register B.

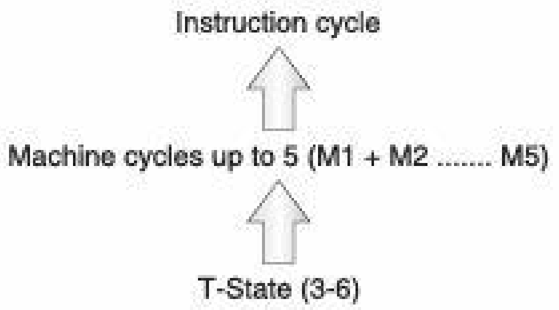
3.4 THE 8085 MACHINE CYCLES AND BUS TIMINGS

The 8085 microprocessor is designed to execute 74 different instruction types. Each instruction has two parts: operation code, known as opcode, and operand. The opcode is a command such as Add, and the operand is an object to be operated on, such as a byte or the contents of a register. Some instructions are 1-byte instructions and some are multi-byte instructions.

The first machine cycle is opcode fetch where the opcode of the instruction is fetched and decoded. After decoding, to execute an instruction, the 8085 needs to perform various operations, such as Memory Read/Write and I/O Read/Write etc. However, there is no direct relationship, between the number of bytes of an instruction and the number of operations the 8085 has to perform. All instructions are divided into a few basic machine cycles and these machine cycles are further divided into precise system clock periods. Basically, the microprocessor external communication functions can be divided into three categories:

1. Opcode fetch (memory read plus opcode decode)
2. Memory Read and Write
3. I/O Read and Write

Now we can define three terms; instruction cycle, machine cycle and T-state and use these terms later for examining timings of various 8085 operations. Instruction cycle is defined as the time required in completing the execution of an instruction. The 8085 instruction cycle consists of one to five machine cycles or one to five operations. Machine cycle is defined as the time required to complete one operation of accessing memory, I/O, or acknowledging an external request. This cycle may consist of three to six T-states. T-state is defined as one subdivision of the operation performed in one clock period. These subdivisions are internal states synchronized with the system clock, and each state is precisely equal to one clock period. The terms T-state and clock period are often used synonymously.



3.4.1 Opcode Fetch Machine Cycle

The first operation in any instruction is Opcode Fetch. The microprocessor needs to get (fetch) this machine code from the memory register where it is stored before the microprocessor can begin to execute the instruction. We discussed this operation in example. Figure 3.5(a) shows how the 8085 fetches the machine code, using the address and the data buses and the control signal, and also shows the timing of the Opcode Fetch machine cycle in relation to the system's clock. However, to differentiate an opcode from a data byte or an address, this machine cycle is identified as the Opcode Fetch cycle by the status signals ($IO/\overline{M} = 0$, $S_1 = 1$, $S_0 = 1$); the active low $IO/\overline{M}$ signal indicates that it is a memory operation, and S_1 and S_0 being high indicate that it is an Opcode Fetch cycle.

This Opcode Fetch cycle is called the M cycle and has 3-6 T-states. The 8085 uses the first three states T_1-T_3 to fetch the code and T_{4-6} to decode the opcode. In the 8085 instruction set, some instructions have opcode with six T-states, we may find that these two operations (Opcode Fetch and Memory/Read) are almost identical except that the Memory Read Cycle has three T-states.

3.4.2 Memory Read Machine Cycle

To illustrate the Memory Read machine cycle, we need to examine the execution of a 2-byte or a 3-byte instruction because in a 1-byte instruction the machine code is an opcode; therefore, the operation is always an Opcode Fetch. The execution of a 2-byte instruction is illustrated in the next example.

Example: Two machine codes—0011 1110 (3EH) and 001100 10 (32 H) are stored in memory locations 2000 H and 2001 H, respectively, as shown below. The first machine code (3EH) represents the opcode to load a data byte in the accumulator, and the second code (32 H) represents the data byte to be loaded in the accumulator. We have to calculate the time required to execute the Opcode Fetch and the Memory Read cycles and the entire instruction cycle if the clock frequency is 3 MHz.

Memory Location	Machine Code	Instruction	Comment
2000 H	3EH	MVI A, 32 H	Load Byte 32 H in the accumulator
2001 H	32H		

Solution: This instruction consists of two bytes; the first is the opcode and the second is the data byte. The 8085 needs to read these bytes first from memory and thus requires at least two machine cycles. The first machine cycle is Opcode Fetch and the second machine cycle is Memory Read, as shown in Figure 3.6; this instruction requires seven T-states for these two machine cycles. The timings of the machine cycles are described in the following paragraphs.

1. The first machine cycle M1 (Opcode Fetch) is identical in bus timings with the machine cycle illustrated in example, except for the bus contents. At T_1, the microprocessor identifies that it is an Opcode Fetch cycle by placing 011 on the status signals ($IO/\overline{M}$ = 0, S_1 = 1 and S_0 = 1). It places the memory address 2000 H) from the program counter on the address bus, 20 H on A_{15}-A_8, and 00 H on AD_7-AD_0 and increments the program counter to 2001 H to point to the next machine code. The ALE signal goes high during T_1, which is used to latch the low-order address 00 H from the bus AD_8-AD_0. At T_2, the 8085 asserts the $\overline{RD}$ control signal, which enables the memory, and the memory places the byte 3EH from location 2000 H on the data bus. Then the 8085 places the opcode in the instruction register and decodes the $\overline{RD}$ signal. The fetch cycle is completed in state T_3. During T_4, the 8085 decodes the opcode and finds out that a second byte needs to be read. After the T_3 state, the contents of the bus A_{15}-A_8 are unknown, and the data bus AD_8-AD_0 goes into high impedance.

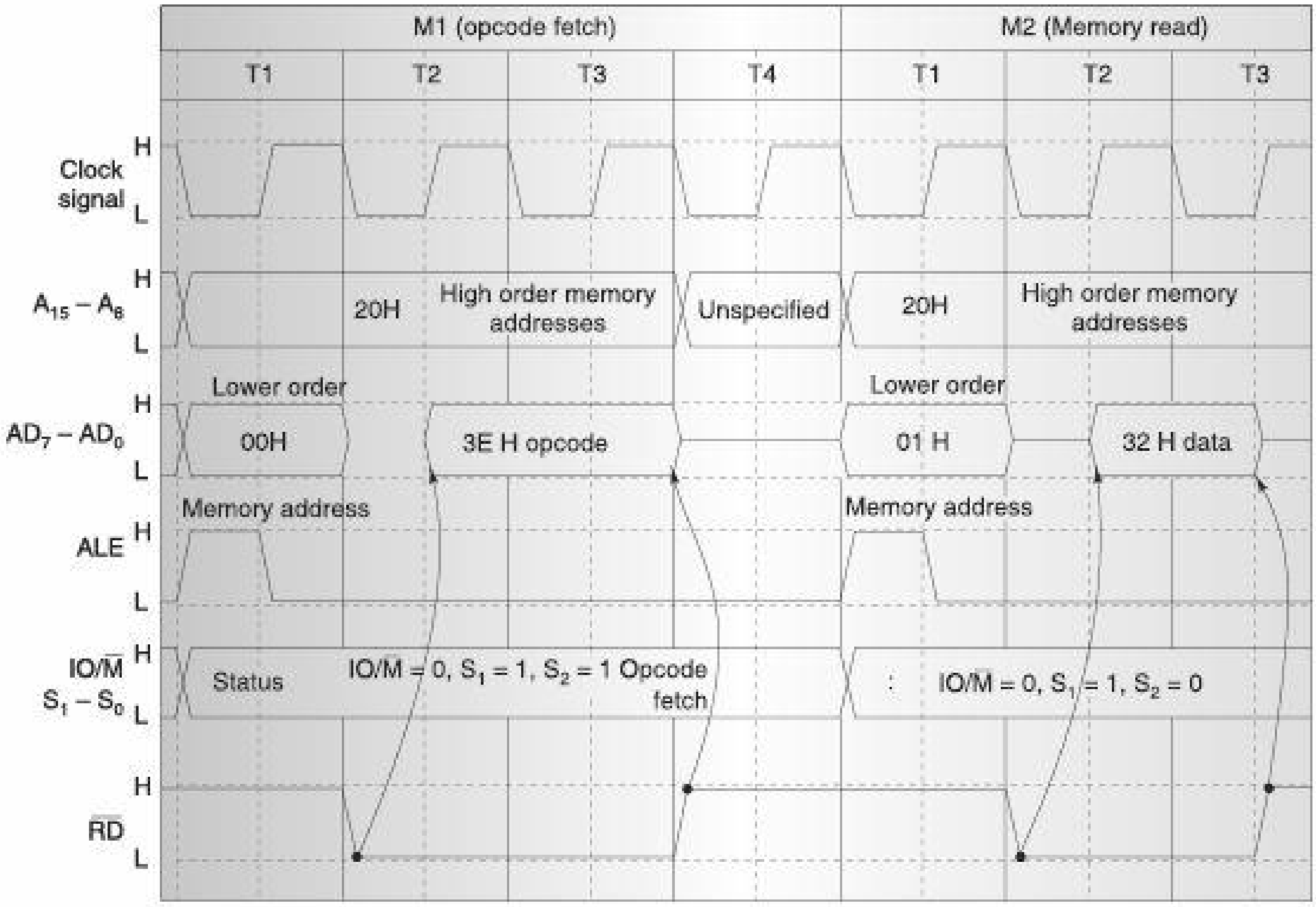


Fig. 3.6 Timing of the Instruction MVI A, 32 H.

2. After completion of the Opcode fetch cycle, the 8085 places the address 2001 H on the address bus and increments the program counter to the next address 2002 H. The second machine cycle M_2 is identified as the Memory Read cycle ($IO/\overline{M} = 0$, $S_1 = 1$, and $S_0 = 0$) and the ALE is asserted. At T_2, the $\overline{RD}$ signal becomes active and enables the memory chip.
3. At the rising edge of T_2, the 8085 activates the data bus as an input bus; memory places the data byte 32 H on the data bus, and the 8085 reads and stores the byte in the accumulator during T_3.

The execution times of the Memory Read machine cycle and the instruction cycle are calculated as follows:

- Clock frequency f = 3 MHz
- T-state = clock period (1/f) = 0.33 μs
- Execution time for Opcode Fetch: (4 T) × 0.33 = 1.3 3μs
- Execution time for Memory Read: (3 T) × 0.33 = 0.3267 μs
- Execution time for Instruction: (7 T) × 0.33 = 2.21 μs

3.5 INSTRUCTION EXECUTION IN 8085

To understand the functions of various signals of the 8085, we should examine the process of communication between the microprocessor and memory and the timings of these signals in relation to the system clock. The approach to designing an interfacing circuit for an I/O device is determined primarily by the instructions to be used for data transfer. An I/O device can be interfaced with the 8085 microprocessor either as a peripheral I/O or as a memory-mapped I/O. In the peripheral I/O, the instructions IN/OUT are used for data transfer, and the device is identified by an 8-bit address. In the memory-mapped I/O, memory-related instructions are used for data transfer, and the device is identified by a 16-bit address. However, the basic concepts in interfacing I/O devices are similar in both methods. Peripheral I/O and the memory-mapped I/O are described in the following section.

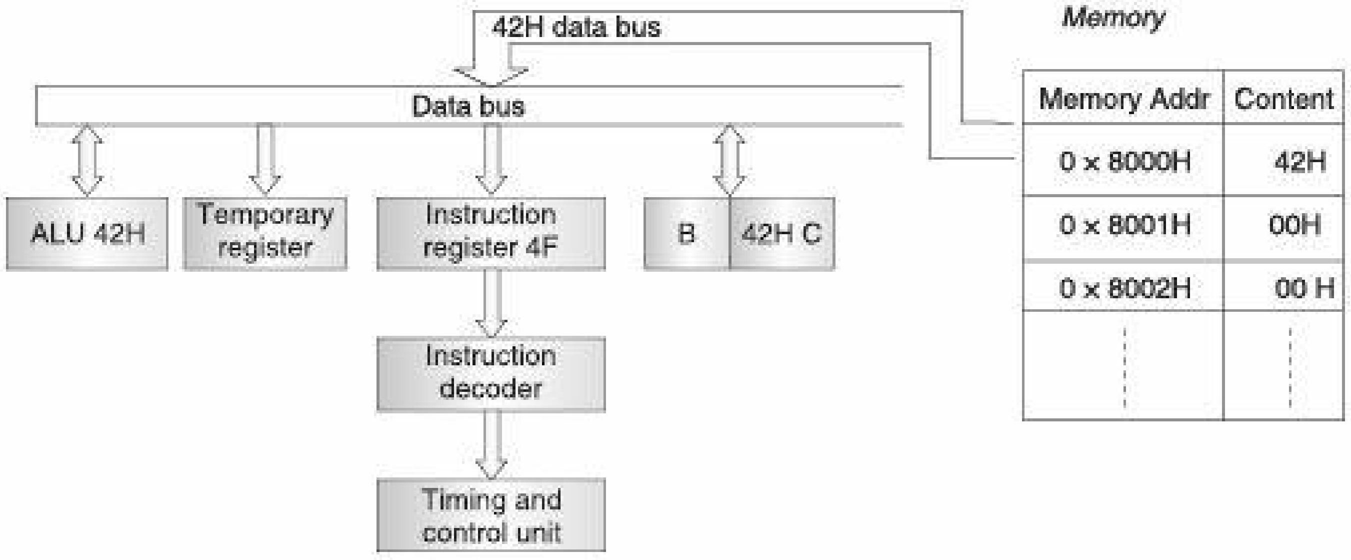


Fig. 3.7 Instruction Decoding and Execution.

Decoding and executing an instruction after it has been fetched can be illustrated with the following example.

Example: Assume that the accumulator contains data byte 82 H, and the instruction MOV B, A (42H) is fetched. List the steps in decoding and executing the instruction.

Solution: In this example, we have the contents of the accumulator opcode 47. To decode and execute the instruction, the following steps are performed.

1. The contents of the data bus (42H) are placed in the instruction register and decoded.
2. The contents of the accumulator (42H) are transferred to the temporary register in the ALU.
3. The contents of the temporary register are transferred to register B.

3.6 GENERATING CONTROL SIGNALS

Figure 3.8 shows the $\overline{\text{RD}}$ (Read) as a control signal. Because this signal is used both for reading memory and for reading an input device, it is necessary to generate two different Read signals:

one for memory and another for input. Similarly, two separate Write signals must be generated. Figure shows that four different control signals are generated by combining the signals $\overline{RD}$, $\overline{WR}$, and IO/$\overline{M}$. The signal IO/$\overline{M}$ goes low for the memory operation. This signal is ANDed with $\overline{RD}$ and $\overline{WR}$ signals by using the 74LS32 quadruple two input OR gates. The OR gates are functionally connected as negative NAND gates. When both input signals go low, the outputs of the gates go low and generate $\overline{MEMR}$ (Memory Read) and $\overline{MEMW}$ (Memory Write) control signals. When the IO/$\overline{M}$ signal goes high, it indicates the peripheral I/O operation. This signal is complemented using the Hex inverter 74LSO4 and ANDed with the $\overline{RD}$ and $\overline{WR}$ signals to generate $\overline{IOR}$ (I/O Read) and $\overline{IOW}$ (I/O Write) control signals.

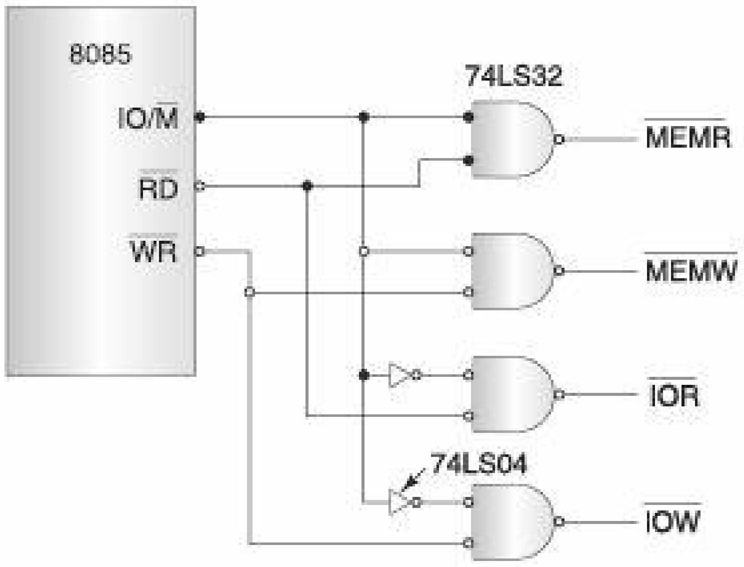


Fig. 3.8 Schematic to Generate Read/Write Control Signals for Memory and I/O.

3.6.1 Timing and Control Unit

This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communication between the microprocessor and peripherals. The control signals are similar to a sync pulse in an oscilloscope. The $\overline{RD}$ and $\overline{WR}$ signals are sync pulses indicating the availability of the data on the data bus.

3.6.2 Important Concepts

1. The 8085 microprocessor has a multiplexed bus AD_7-AD_0 used as the lower-order address bus and the data bus.
2. The bus AD_7-AD_0 can be demultiplexed by using a latch and the ALE signal.
3. The 8085 has a status signal IO/$\overline{M}$ and two control signals $\overline{RD}$ and $\overline{WR}$. By ANDing these signals, four control signals can be generated: $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, and $\overline{IOW}$. The 8085 MPU transfers data from memory locations to the microprocessor by using the control signal Memory Read ($\overline{MEMR}$-active low). This is also called reading from memory. The term data refers to any byte that is placed on the data bus; the byte can be an instruction code, data, or an address.

4. Transfer data from the microprocessor to memory by using the control signal memory write ($\overline{\text{MEMW}}$–active low). This is also called writing into memory.
5. Accept data from input devices by using the control signal I/O Read (IOR–active low).
6. Sends data to output devices by using the control signal I/O Write. This is also known as writing to an output port.

To execute an instruction, the MPU

- Places the memory address of the instruction on the address bus.
- Indicates the operation status on the status lines.
- Sends the $\overline{\text{MEMR}}$ control signal to enable the memory, fetches the instruction byte.
- Executes the instruction.

3.7 INSTRUCTION TYPES IN 8085

3.7.1 One-Byte Instructions

A 1-byte instruction includes the Opcode and the operand in the same byte. If the instruction does not contain any immediate byte (data), it is one byte instruction. For example:

Task	*Mnemonic*	*Operand*	*Hex code*
Copy the contents of the accumulator in register C	MOV	C,A	4F H
Add the contents of register B to the contents of the accumulator	ADD	B	80 H
Invert each bit in the accumulator	CMA		2F H

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both 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-bit binary format in memory; each requires one memory location.

3.7.2 Two-Byte Instructions

If the instruction contains 8 bits (data), it is 2-byte instruction. In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. For example, assume the data byte is 32 H. The assembly language instruction is written as **Mnemonics** MVI A, **Hex Code** 32H 3EH, 32 H. This instruction would require two memory locations to store in memory.

3.7.3 Three-Byte Instructions

If the instruction contains 16 bits (data), it is 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. For example:

Task	Mnemonic	Operand	Hex code
Transfer the program sequence to the memory location 2085 H	JMP	2085 H	C3 –First Byte 85- Second byte 20- Third Byte

3.8 BRIEF INTRODUCTION TO 8085 INSTRUCTION SET

The 8085 important instructions are explained in this section to make reader familiar with basic operation performed by 8085.

1. **Data Transfer Instructions**

   MOV regX, regY

   i.e. MOV B A

   Copy the content of regY into regX; content of regY remains unchanged.

   MVI reg, 8-bit data (immediate data)

   i.e. MVI A 32H

   Copy the immediate data given into reg.

   LDA 16-bit address

   i.e. LDA C0 05

   Load the Accumulator with the content of the memory location specified by 16-bit Address.

   STA 16-bit address

   i.e. STA C0 05

   Store the content of accumulator to the memory location specified by the 16-bit Address.

   LDAX Rp (Rp = Register pair)

   i.e. LDAX B

   (Load the accumulator with a data contained in a memory location where the 16-bit memory location is stored in the BC register pair)

   STAX Rp (Rp = Register pair)

   i.e. STAX B

   (Store the content of the accumulator to a memory location where the 16-bit memory location is stored in the BC register pair)

   IN portaddr

   i.e. IN 00 (Reads data from the Input Switch, 00 represents the port address of the input switch)

   OUT portaddr

   i.e. OUT 00 (Writes data to the Display device where 00 represents the Port address of the display)

2. Arithmetic Instructions

ADD reg

Add the content of given register to accumulator. Result is stored in accumulator.

i.e. ADD B

[A] = [A]+ [B]

ADI 8-bit data

Add the given 8-bit data to accumulator. Result is stored in accumulator.

i.e. ADI 47H

[A]= [A]+ 47H

SUB reg

Subtract the content of given register from accumulator. Result is stored in accumulator.

i.e. SUB C

[A] = [A] – [C]

SUI reg

Subtract the given 8-bit data from accumulator. Result is stored in accumulator.

i.e. SUI 5FH ; [A] = [A] – 5F

INR reg

Increment the Content of a Register specified by reg.

i.e: INR B

Increments the Content of Register B by 1

DCR reg

Decrement the Content of a Register specified by reg.

i.e: DCR D

Decrements the Content of Register D by 1

INX Rp

Increment the Content of a Register pair specified by Rp

i.e: INX B

Increments the Content of Register pair BC by 1

DCX Rp

Decrement the Content of a Register pair specified by Rp

i.e: DCX B

Decrement the Content of Register pair BC by 1

3. Logical Instructions

ORA reg

Performs the logical OR operation of given reg with accumulator. Result is stored in accumulator.

i.e. ORA B

Perform the OR operation of register B with accumulator.

ORI 8-bit data

Performs the logical OR operation of given 8-bit data with accumulator. Result is stored in accumulator.

i.e. ORA 12H

Perform the OR operation of 12H with Accumulator.

ANA reg

Performs the logical AND operation of given reg with accumulator. Result is stored in accumulator.

i.e. ANA B

Perform the AND operation of register B with accumulator.

ANI 8-bit data

Performs the logical AND operation of given 8-bit data with accumulator. Result is stored in accumulator.

i.e. ANA B

Perform the AND operation of register B with accumulator.

XRA reg

Performs the logical XOR operation of given reg with accumulator. Result is stored in accumulator.

i.e. XOR B

Perform the XOR operation of register B with accumulator.

XRI 8-bit data

Performs the logical XOR operation of given 8-bit data with accumulator. Result is stored in accumulator.

i.e. XOR 12H

Perform the XOR operation of 12H with accumulator.

CMA

Complements the Content of the accumulator (Performs the NOT Operation)

CMP reg

Compares the Content of the accumulator with the register specified by 'reg'

If (B) > (A) , then Carry flag is set

If (B) = (A) , then Zero flag is set

If (A) > (B) , then No flag is set

CPI 8-bit data

Compares the content of the accumulator with the immediate data specified by the 8-bit data.

4. Branching Instructions

JMP 16-bit address [Unconditional Jump]

The program sequence jumps unconditionally to a memory location defined by 16-bit address.

JC 16-bit address [Conditional Jump]

The program sequence jumps to a memory location defined by 16-bit address if the Carry Flag is set.

JNC 16-bit address [Conditional Jump]

The program sequence jumps to a memory location defined by 16-bit address if the Carry Flag is not set.

JZ 16-bit address [Conditional Jump]

The program sequence jumps to a memory location defined by 16-bit address if the Zero Flag is set.

JNZ 16-bit address [Conditional Jump]

The program sequence jumps to a memory location defined by 16-bit address if the Zero Flag is not set.

5. Machine Control Instructions

HLT

Stop program execution.

NOP

Do not perform any operation.

3.9 INSTRUCTION FORMAT AND ASSEMBLY LANGUAGE PROGRAMMING IN 8085

3.9.1 Assembler Instruction Format

The general format of an assembler instruction is

Memory Address	Opcode/ Hex code	Label	Mnemonics	Operands	;Comments

The inclusion of spaces between label, mnemonics, operands and comments are arbitrary, except that at least one space must be inserted. No space would lead to an ambiguity. There can be no space within a mnemonic or identifier. Each statement in the program is consists of fields.

(a) **Label:** It is an identifier that is assigned the address of the first byte of the instruction in which it appears. A label appears in a program to identify the name of a memory location storing data and for other purposes. The presence of a label in an instruction is optional, but if present, the label provides a symbolic name that can be used in branch instruction to branch to the instruction. A colon must be placed at the end of the label, if there is no label, then the colon must not be entered. All labels begin with a letter or one of the following special character: @, $, _ or ?. A label may be of any length from 1 to 35 characters.

(b) **Mnemonics:** Mnemonics is a name assigned to a machine function. The mnemonic specifies the operation to be executed. The assembler converts these mnemonics into actual processor instructions and associated data All instructions must contain a mnemonic.

(c) **Operands:** The presence of the operands depends on the instruction. Some instructions have no operands, some have one operand, and some have two. If there are two operands, they are separated by a comma.

(d) **Comments:** The comment field is for commenting the program and may contain any combination of the characters. It is optional in programming. A comment may appear on a line by itself provided that the first character on the line is a semicolon.

Example 1: Place 05 in register A; then move it to register B. Assume that the starting address of program in 0x8000H.

Program:

Address	**Opcode / Hex code**	**Mnemonics**	**Operands**	**Comments**
8000	3E, 05	MVI	A, 05	;Get 05 in register A.
8002	47	MOV	B, A	;Transfer 05 from A to B.
8003	76	HLT		;Stop the program execution.

Explanation: Here it is assumed that the program is loaded from memory location 8000H. The RAM memory on the kit on which this program is loaded should start from 8000H. Each opcode is loaded into memory one after another. After loading all the opcode the program can be executed.

3.9.2 Writing Assembly Language Program for 8085

To write a program for 8085 following steps are needed.

1. Analyze the problem
2. Develop program Logic
3. Write an Algorithm
4. Make a Flow Chart
5. Write program Instructions using Assembly language of 8085

Example: Program 8085 in Assembly language to add two 8-bit numbers and store 8-bit result in register C.

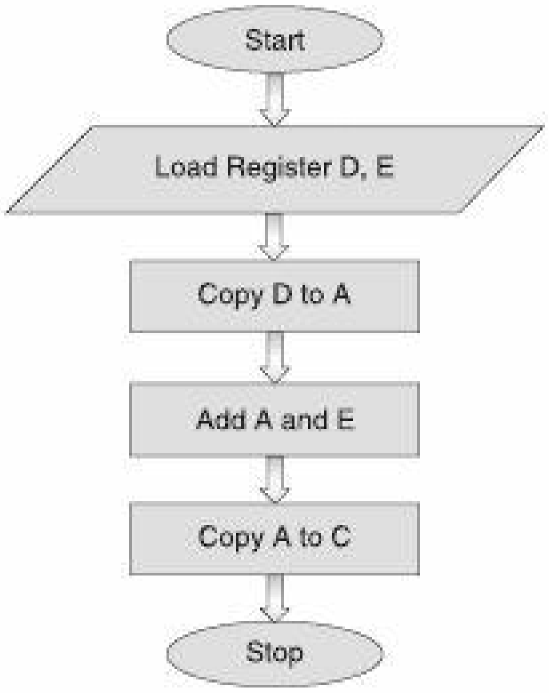
1. Analyze the problem
   - Addition of two 8-bit numbers to be done
2. Program Logic
   - Add two numbers
   - Store result in register C
   - Example

```
  10011001    (99H) A
+ 00111001    (39H) D
  11010010    (D2H) C
```

3. Algorithm

Operations	*Translation to 8085 operations*
1. Get two numbers	Load 1st no. in register D Load 2nd no. in register E
2. Add them	Copy register D to A Add register E to A
3. Store result	Copy A to register C
4. Stop	Stop processing

4. Make a Flow Chart



5. Assembly Language Program

S. No.	*Operations to be performed*	*Instructions*
1	Get two numbers	
	(a) Load 1st no. in register D (b) Load 2nd no. in register E	MVI D, 2H MVI E, 3H
2	Add them	
	(a) Copy register D to A. (b) Add register E to A	MOV A, D ADD E
3	Store result.	
	(a) Copy A to register C	MOV C, A
4	Stop	
	(a) Stop processing	HLT

3.10 INSTRUCTIONS, HEX CODES, MACHINE CYCLE AND T STATES OF 8085

The summary of 8085 instruction set, opcode, machine cycle and T state is given in the Table 3.3 shown below. This can be used for the obtaining hex code to perform experiment on 8085 trainer kit. The machine cycle is useful in calculated time delay associated with each instruction whereas T states are helpful in calculating exact time delay for each instruction.

Table 3.3 Mnemonics, Hex code, Machine cycle and T-state of 8085

Mnemonics	*Hex Code*	*Machine cycle*	*T State*
ACI 8-Bit	CE	2	7
ADC A	8F	1	4
ADC B	88	1	4
ADC C	89	1	4
ADC D	8A	1	4
ADC E	8B	1	4
ADC H	8C	1	4
ADC L	8D	1	4
ADC M	8E	2	7
ADD A	87	1	4
ADD B	80	1	4
ADD C	81	1	4
ADD D	82	1	4
ADD E	83	1	4
ADD H	84	1	4
ADD L	85	1	4
ADD M	86	2	7
ADI 8-Bit	C6	2	7
ANA A	A7	1	4
ANA B	A0	1	4
ANA C	A1	1	4
ANA D	A2	1	4
ANA E	A3	1	4
ANA H	A4	1	4
ANA L	A5	1	4
ANA M	A6	2	7
ANI 8-Bit	E6	2	7
CALL 16-Bit	CD	5	18
CC 16-Bit	DC	2 or 5	9 or 18
CM 16-Bit	FC	2	7
CPO 16-Bit	E4	2,8	9,21
CZ 16-Bit	CC	2,8	9,21
DAA	27	1	4
DAD B	9	1	3
DAD D	19	1	3
DAD H	29	1	3
DAD SP	39	1	3
DCR A	3D	1	4
DCR B	5	1	4
DCR C	0D	1	4
DCR D	15	1	4
DCR E	1D	1	4
DCR H	25	1	4
DCR L	2D	1	4
DCR M	35	3	10
DCX B	0B	1	6
DCX D	1B	1	6
DCX H	2B	1	6
DCX SP	3B	1	6
DI	F3	1	4
EI	FB	1	4
HLT	76	2 or more	5 or more
IN 8-Bit	DB	3	10
INR A	3C	1	4
INR B	4	1	4
INR C	0C	1	4
INR D	14	1	4
INR E	1C	1	4
INR H	24	1	4
INR L	2C	1	4

Contd...

Contd...

CMA	2F	1	4
CMC	3F	1	4
CMP A	BF	1	4
CMP B	B8	1	4
CMP C	B9	1	4
CMP D	BA	1	4
CMP E	BB	1	4
CMP H	BC	1	4
CMP L	BD	1	4
CMP M	BE	1	4
CNC 16-Bit	D4	2,5	9,18
CNZ 16-Bit	C4	2,6	9,19
CP 16-Bit	F4	2,7	9,20
CPE 16-Bit	EC	2,8	9,21
CPI 8-Bit	FE	2	7
LDAX B	0A	2	7
LDAX D	1A	2	7
LHLD 16-Bit	2A	5	16
LXI B,16-Bit	1	3	10
LXI D,16-Bit	11	3	10
LXI H,16-Bit	21	3	10
LXI SP,16-Bit	31	3	10
MOV A, A	7F	1	4
MOV A, B	78	1	4
MOV A, C	79	1	4
MOV A, D	7A	1	4
MOV A, E	7B	1	4
MOV A, H	7C	1	4
MOV A, L	7D	1	4
MOV A, M	7E	2	7
MOV B, A	47	1	4
MOV B, B	40	1	4
MOV B, C	41	1	4
MOV B, D	42	1	4
MOV B, E	43	1	4
MOV B, H	44	1	4
MOV B, L	45	1	4
MOV B, M	46	2	7

INR M	34	3	10
INX B	3	1	6
INX D	13	1	6
INX H	23	1	6
INX SP	33	1	6
JC 16-Bit	DA	2,5	7,10
JM 16-Bit	FA	2, 6	7, 11
JMP 16-Bit	C3	3	10
JNZ 16-Bit	D2	2,5	7,10
JNZ 16-Bit	C2	2,6	7,11
JP 16-Bit	F2	2,5	7,10
JPE 16-Bit	EA	2,6	7,11
JPO 16-Bit	E2	2,7	7,12
JZ 16-Bit	CA	2,8	7,13
LDA 16-Bit	3A	4	13
MOV E, M	5E	2	7
MOV H, A	67	1	4
MOV H, B	60	1	4
MOV H, C	61	1	4
MOV H, D	62	1	4
MOV H, E	63	1	4
MOV H, H	64	1	4
MOV H, L	65	1	4
MOV H, M	66	2	7
MOV L, A	6F	1	4
MOV L, B	68	1	4
MOV L, C	69	1	4
MOV L, D	6A	1	4
MOV L, E	6B	1	4
MOV L, H	6C	1	4
MOV L, L	6D	1	4
MOV L, M	6E	2	7
MOV M, A	77	2	7
MOV M, B	70	2	7
MOV M, C	71	2	7
MOV M, D	72	2	7
MOV M, E	73	2	7
MOV M, H	74	2	7

Contd...

Contd...

MOV C, A	4F	1	4
MOV C, B	48	1	4
MOV C, C	49	1	4
MOV C, D	4A	1	4
MOV C, E	4B	1	4
MOV C, H	4C	1	4
MOV C, L	4D	1	4
MOV C, M	4E	2	7
MOV D, A	57	1	4
MOV D, B	50	1	4
MOV D, C	51	1	4
MOV D, D	52	1	4
MOV D, E	53	1	4
MOV D, H	54	1	4
MOV D, L	55	1	4
MOV D, M	56	2	7
MOV E, A	5F	1	4
MOV E, B	58	1	4
MOV E, C	59	1	4
MOV E, D	5A	1	4
MOV E, E	5B	1	4
MOV E, H	5C	1	4
MOV E, L	5D	1	4
POP D	D1	3	10
POP H	E1	3	10
POP PSW	F1	3	10
PUSH B	C5	3	12
PUSH D	D5	3	12
PUSH H	E5	3	12
PUSH PSW	F5	3	12
RAL	17	1	4
RAR	1F	1	4
RC	D8	1	4
RM	F8	1,3	6,12
RNC	D0	1,3	6,12
RNC	C0	1,3	6,12
RP	F0	1,3	6,12
RPE	E8	1,3	6,12

MOV M, L	75	2	7
MVI A ,8-Bit	3E	2	7
MVI B, 8-Bit	6	2	7
MVI C, 8-Bit	OE	2	7
MVI D, 8-Bit	16	2	7
MOV E, 8-Bit	1E	2	7
MVI H, 8-Bit	26	2	7
MOV H, A	67	1	4
MVI L, 8-Bit	2E	2	7
MVI M, 8-Bit	36	3	10
NOP	0	1	4
ORA A	B7	1	4
ORA B	B0	1	4
ORA C	B1	1	4
ORA D	B2	1	4
ORA E	B3	1	4
ORA H	B4	1	4
ORA L	B5	1	4
ORA M	B6	2	7
ORI 8-Bit	F6	2	7
OUT 8-Bit	D3	3	10
PCHL	E9	1	6
POP B	C1	3	10
SUB C	91	1	4
SUB D	92	1	4
SUB E	93	1	4
SUB H	94	1	4
SBI 8-Bit	DE	2	7
SHLD 16-Bit	22	3	16
SIM	30	1	4
SPHL	F9	1	6
STA 16-Bit	32	4	13
STAX B	2	2	7
STAX D	12	2	7
STC	37	1	4
SUB A	97	1	4
SUB B	90	1	4
STAX B	2	2	7

Contd...

Contd...

RPO	E0	1,3	6,12
RRC	0F	1	4
RST 0	C7	3	12
RST 1	CF	3	12
RST 2	D7	3	12
RST 3	DF	3	12
RST 4	E7	3	12
RST 5	EF	3	12
RST 6	F7	3	12
RST 7	FF	3	12
RZ	C8	1,3	6,12
SBB A	9F	1	4
SBB B	98	1	4
SBB C	99	1	4
SBB D	9A	1	4
SBB E	9B	1	4
SBB H	9C	1	4
SBB L	9D	1	4
SBB M	9E	2	7
SBI 8-Bit	DE	2	7
SHLD 16-Bit	22	3	16
SIM	30	1	4
STAX D	12	2	7
STC	37	1	4
SUB A	97	1	4
SUB B	90	1	4
SUB C	91	1	4
SUB D	92	1	4
SUB E	93	1	4
SUB H	94	1	4
SUB L	95	1	4
SUB M	96	2	7
SUI 16-Bit	D6	2	7
XCHG	EB	1	4
XRA A	AF	1	4
XRA B	A8	1	4
XRA C	A9	1	4
XRA D	AA	1	4
XRA E	AB	1	4
XRA H	AC	1	4
XRA L	AD	1	4
XRA M	AE	2	7
XRI 8-Bit	EE	2	7
XTHL	E3	5	16

3.11 TIMING DIAGRAM OF VARIOUS INSTRUCTIONS

Timing diagram is the display of initiation of read/write and transfer of data operations under the control of 3-status signals IO/$\overline{M}$, S1, and S0. All actions in the microprocessor are controlled by either leading or trailing edge of the clock. The clock signal determines the time taken by the microprocessor to execute any instruction. The 3-status signals: IO/$\overline{M}$, S1, and S0 are generated at the beginning of each machine cycle. The unique combination of these 3-status signals identify read or write operation and remain valid for the duration of the cycle.

The execution of instruction always requires read and write operations to transfer data to or from the μP and memory or I/O devices. Each read/write operation constitutes one machine cycle (MC) as indicated in Table 3.4 shown below. Some instructions are explained with examples.

Table 3.4 Machine cycle status and control signals

Machine cycle		*Status*			*Controls*	
	$IO/\overline{M}$	S_1	S_0	$\overline{RD}$	$\overline{WR}$	$\overline{INTA}$
Opcode Fetch (OF)	0	1	1	0	1	1
Memory Read	0	1	0	0	1	1

Contd...

Memory Write	0	0	1	1	0	1
I/O Read (I/OR)	1	1	0	0	1	1
I/O Write (I/OW)	1	0	1	1	0	1
Acknowledge of INTR (INTA)	1	1	1	1	1	0
BUS Idle (BI) : DAD	0	1	0	1	1	1
ACK of RST, TRAP	1	1	1	1	1	1
HALT	Z	0	0	Z	Z	1
HOLD	Z	X	Z	Z	Z	1

1. ADC M

Function: Add the contents of memory location M whose address is in HL register pair in the contents of accumulator along with carry bit and result is stored in accumulator.

Example: The opcode for ADC M is 8E which is located at the address 8000H. Let HL contains the address memory 8001H. The data at 8000H is 23H and Accumulator has content 01H. Carry is zero.

Program:

Memory Address	Content
8000H	8E

(a) The first machine will fetch the opcode from memory address 8000H. In the second machine cycle the HL pair will leads to memory address 8001H whose contents will be added to accumulator along with carry. The result will be 32H+01H=24H will remain in accumulator.

2. HLT

Function: This instruction is used to stop the execution of the processor.

Example: Let assume that the opcode for HLT instruction is written at memory location 8000H. Timing Diagram

3. CALL 16-bit address

Function: This instruction is used for unconditional CALL to the subroutine identified by 16-bit operand. Before branching to subroutine, addresses of PC are saved on stack.

4. RET

Function: This instruction is used at the end of a subroutine. The execution of RET instruction brings back the saved address from the stack to program counter. Program jumps to the next instruction of the main program which is next to CALL instruction.

Example: We want to read the 8-bit data from a memory location and add 30H into given data and then want to store the result on a memory location. The CALL instruction is written at memory address 2013H.

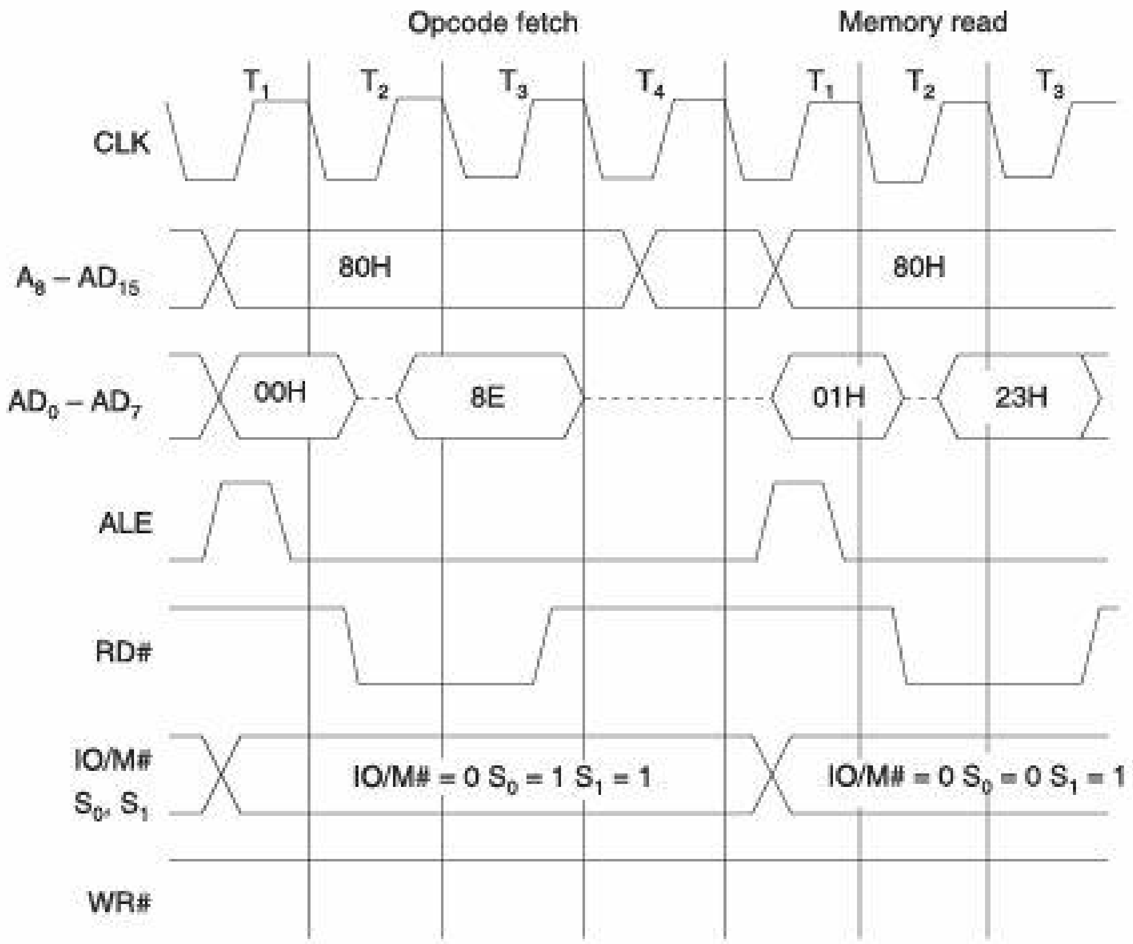


Fig. 3.9 Timing Diagram of ADC M Instruction.

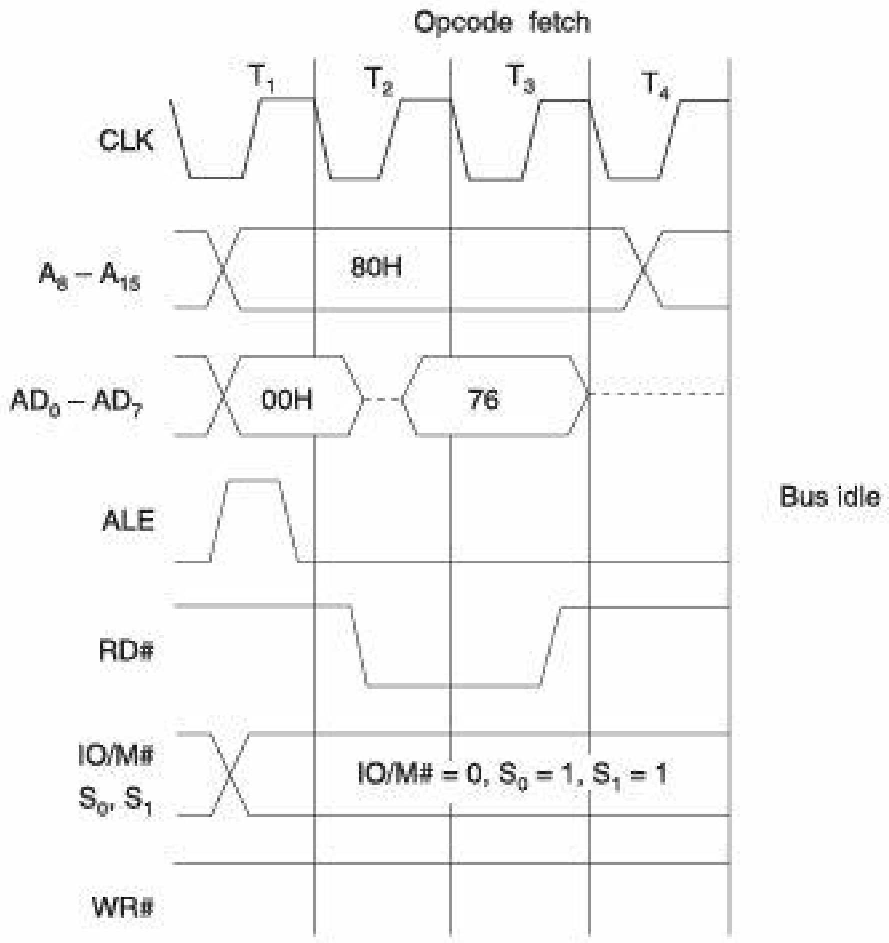


Fig. 3.10 Timing Diagram of HLT Instruction.

Main Program:

Address	Hex code	Label	Mnemonics	Operands	Comments
2009	3A, 08, 20		LDA	8200	;Get Hexa Data
2012	47		MOV	B, A	; Move data into accumulator
2013	CD,1A,20		CALL	ADD1	;Call subroutine to add 30H
2016	32, 0A, 20		STA	8200	;Store the result
2019	76		HLT		;Stop the execution

Subroutine to add 30H:

Address	Hex code	Label	Mnemonics	Operands	Comments
201A	FE, 0A	ADD1:	ADI	30H	;Add 30H to accumulator
201C	C9		RET		;return to main program

Timing Diagram of CALL instruction:

(a) In first machine cycle (M1), the contents of program counter (2013H) are placed on the address bus and instruction code CD is fetched using the data bus.

(b) In M2 and M3, memory read operation is done. In this 16 bit address (201A H) of CALL instruction is fetched.

(c) In machine cycle M4 and M5, storing of Program counter is done.

Timing Diagram of RET instruction:

(a) In first machine cycle (M1), the contents of program counter (2023H) are placed on the address bus and instruction code C9 is fetched using the data bus.

(b) In machine cycle M2 and M3, storing of Program counter is done. The program execution will start from the main program from the next instruction after the CALL instruction i.e. from memory location 2016H.

3.12 ADDRESSING MODES

The instructions consist of two parts opcode and operands. The opcode determines the operation to be performed and the operand determines the data to be operated on. The manner in which operand is specified in the instruction is called addressing mode. Intel 8085 uses the following addressing modes:

1. Direct addressing
2. Register addressing
3. Register indirect addressing
4. Immediate addressing
5. Implicit addressing

3.12.1 Direct Addressing

In this mode of addressing, the address of the operand (data) is given in the instruction itself.

Examples are:

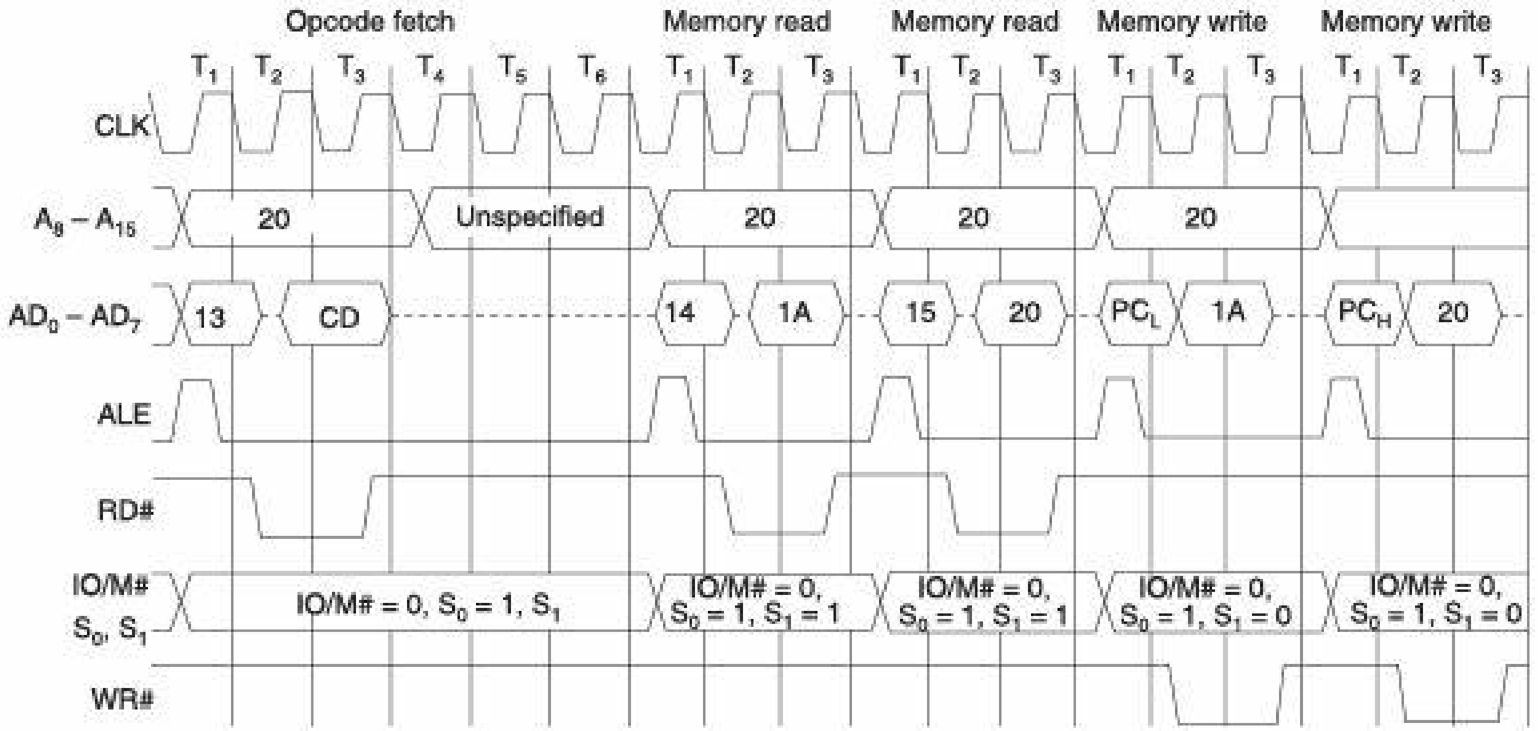

Opcode fetch
Memory read
Memory read
Memory write
Memory write
T_1 T_2 T_3 T_4 T_5 T_6
T_1 T_2 T_3
CLK
$A_8 - A_{15}$
20
Unspecified
20
20
20
$AD_0 - AD_7$
13
CD
14
1A
15
20
PC_L
1A
PC_H
20
ALE
RD#
IO/M#
S_0, S_1
IO/M# = 0, S_0 = 1, S_1
IO/M# = 0, S_0 = 1, S_1 = 1
IO/M# = 0, S_0 = 1, S_1 = 1
IO/M# = 0, S_0 = 1, S_1 = 0
IO/M# = 0, S_0 = 1, S_1 = 0
WR#

Fig. 3.11(a)

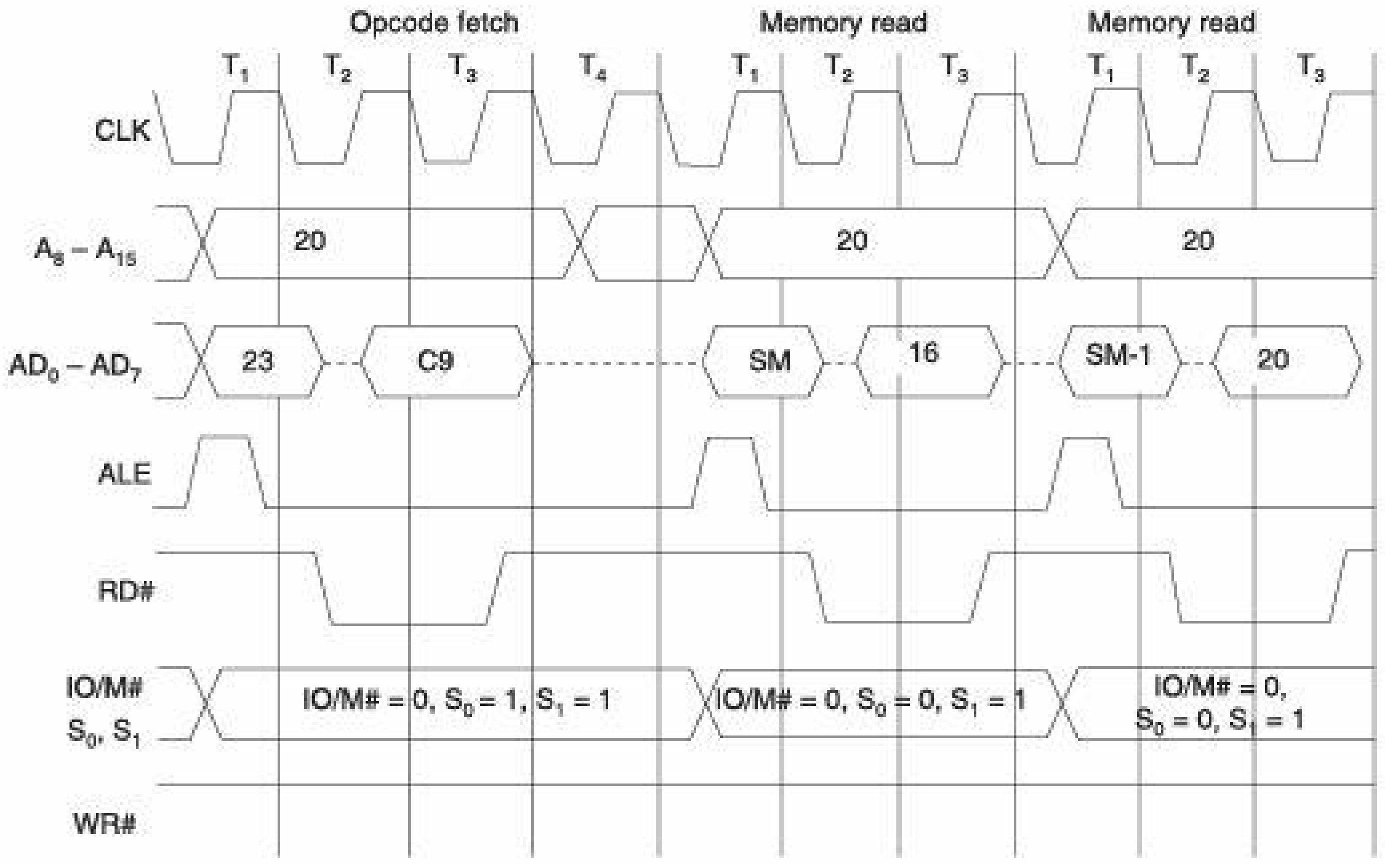


Fig. 3.11(b) Timing diagram of RET instruction.

Mnemonics	**Opcode**	**Function**
STA 2600 H	32, 00, 26	; Store the content of the accumulator in the memory location 2600 H.

In this instruction 2600 H is the memory address where data is to be stored. It is given in the instruction itself. The 2nd and 3rd bytes of the instruction specify the address of the memory location. Here, it is understood that the source of the data is accumulator.

Mnemonics	**Opcode**	**Function**
IN 04	DB, 04	Read data from the port C.

In this instruction 04 is the address of the port C of an I/O port from where the data is to be read. Here, it is implied that the destination is the accumulator. The 2nd byte of the instruction specifies the address of the port.

3.12.2 Register Addressing

In register addressing mode the operand is in one of the general-purpose registers. The opcode specifies the address of the register(s), in addition to the operation to be performed.

Examples are:

S. No.	Mnemonics	Opcode	Function
1	MOV A, B	88	Move the content of register B to register A.
2	ADD B	80	Add the content of register B to the content of register A

In Example 1 the opcode for MOV A, B is 88 H. Besides the operation to be performed, the opcode also specifies source and destination registers. The opcode 88 H can be written in binary form as 10001000. The first two bits, i.e., 10 are for MOV operation, the next three bits 001 are the binary code for register A, and the last three bits 000 are the binary code for register B.

In Example 2 the opcode for ADD B is 80 H. In this instruction one of the operands is register B (its content is one of the data) which is indicated in the instruction itself. In this type of instruction (arithmetic group) it is understood that the other operand is in the accumulator. The opcode 80 H in the binary form is 10000000. The first five bits, i.e., 10000 specify the operation to be performed, i.e., ADD. The last three bits 000 are the binary code for register B for 8085 microprocessor.

3.12.3 Register Indirect Addressing

In this mode of addressing the address of the operand is specified by a register pair. Examples are:

Mnemonics	Opcode	Function
LXI H, 2500 H		Load H-L pair with 2500 H.
MOV A, M		Move the content of the memory location, whose address is in H-L Pair (i.e., 2500 H) to the accumulator.
HLT		Stop the execution of the program

In the above program, the instruction MOV A, M is an example of register indirect addressing. For this instruction the operand is in the memory. The address of the memory is not directly given in the instruction. The address of the memory resides in H-L pair and this has already been specified by an earlier instruction in the program, i.e., LXI H, 2500 H.

Mnemonics	Opcode	Function
LXI H, 2500 H		Load the H-L pair with 2500 H
ADD M		Add the content of the memory location, whose address is in H-L pair (i.e., 2500 H) to the content of the accumulator
HLT		Stop the execution of the program

In this program, the instruction ADD M shows how register indirect addressing is used.

3.12.4 Immediate Addressing

In immediate addressing mode the operand is specified within the instruction itself, examples are:

S. No.	Mnemonics	Opcode	Function
1	MVI A, 05	3E, 05	Move 05 in register A
2	ADI 06	C6, 06	Add 06 to the content of the accumulator.

In these instructions the 2nd byte specifies data.

3.12.5 Implicit Addressing

If address of source of data as well as address of destination of result is fixed, then there is no need to give any operand along with the instruction. The instruction itself specifies the type of operation and location of data to be operated. This type of instruction does not have any address, register name, immediate data specified along with it. For example CMA is the operation (Complement accumulator) A is the source and A is the destination. Some other examples are RAR, RAL, RLC, RRC, etc.

3.13 INSTRUCTION SET

An instruction is a command given to the computer to perform a specified operation on given data. The instruction set of a microprocessor is the collection of the instructions that the microprocessor is designed to execute. The instructions described in this chapter are of INTEL 8085. The programmer can write a program in assembly language using these instructions. There are 74 valid instructions in the complete set of 8085 instructions. These instructions have been classified into the following groups.

1. Data Transfer Group
2. Arithmetic Group
3. Logical Group
4. Branch Control Group
5. I/O and Machine Control Group.

3.13.1 Data Transfer Group

Instructions which are used to transfer data from one register to another register, from memory to register or register to memory, come under this group. Examples are: MOV, MVI, LXI, LDA, STA, etc. When an instruction of data transfer group is executed, data is transferred from the source to the destination without altering the contents of the source. For example, when MOV A, B is executed the content of the register B is copied into the register A, and the content of register B remains unaltered. Similarly, when LDA 3500 is executed the content of the memory location 3500 is loaded into the accumulator, but the content of the memory location 3500 remains unaltered.

S. No	Instruction	Register Transfer Logic	Details	Addressing Mode	Instruction length
1	MOV r_1, r_2	$[r_1] \leftarrow [r_2]$	Contents of register r_2 are moved to register r1. Register r1/r2 may be any register out of six GPR*s A, B, C, D, E, H, L.	Register	1 Byte

Contd...

Contd...

2	MOV r, M	[r] ← [(HL)]	Contents of memory location whose address is in HL register pair are moved to register r.	Indirect Register	1 Byte
3	MOV M, r	[(HL)] ← [r]	Contents of register 'r' are moved to memory location whose address is in HL register pair.	Indirect Register	1 Byte
4	MVI r, data	[r] ← data	Move 8-bit data specified in the instruction to register r.	Immediate	2-Byte
5	MVI M, data(8)	[(Hl)] ← data	Move 8-bit data specified in the instruction to memory location whose address is in register pair HL.	Immediate	3-Byte
6	LXI rp, data16	[rp] ←data	Load register pair with 16-bit data.	Immediate	3-Byte
7	LDA addr	[A] ← [addr]	Load accumulator directly with contents of memory location whose address is specified in the instruction itself.	Direct	3-Byte
8	STA addr	[addr] ← [A]	Store contents of memory location whose address is specified in the instruction itself in accumulator.	Direct	3-Byte
9	LHLD addr	[L] ← [addr] [H] ← [addr + 1]	Load the contents of memory location whose address is specified in the instruction into register L and next memory location into register H.	Direct	3-Byte
10	SHLD addr	[addr] ← [L] [addr + 1] ←[H]	Store the contents of register L into memory location whose address is specified in the instruction and contents of register H into next memory location.	Direct	3-Byte
11	LDAX rp	[A] ← [(rp)]	Load accumulator indirect Contents of memory location whose address is in register pair rp is loaded into accumulator. Register pair used is either BC or DE but not HL.	Register Indirect	1 Byte
12	STAX rp	[(rp)] ← [A]	Store contents of accumulator into memory location whose address is in register pair rp. Register pair used is either BC or DE but not HL.	Register Indirect	2-Byte

Contd...

13	IN addr(8)	[A]←[port addr]	Read data byte from input device whose 8-bit port address is specified in instruction into accumulator.	Direct	3-Byte
14	OUT addr(8)	[port addr] ← [A]	Send data byte from accumulator to output device whose 8-bit port address is specified in instruction.	Direct	2-Byte
15	XCHG	[H]↔[D] [L]↔[E]	It exchanges the content of H-L register pair with that of D-E register pairs.	Register	1 Byte

3.13.2 Arithmetic Group

The instructions of this group perform arithmetic operations such as addition, subtraction; increment or decrement of the content of a register or memory. Examples are: ADD, SUB, INR, DAD, etc.

S. No.	Instruction	Register transfer logic	Arithmetic Group Details	Addressing Mode	Instruction length	Flags affected
1	ADD r	[A] ← [A] + [r]	Add the contents of register r in the contents of accumulator and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L.	Register	One Byte	All
2	ADD M	[A] ← [A] + [(HL)]	Add the contents of memory location M whose address is in HL register pair in the contents of accumulator and result is stored in accumulator.	Register Indirect	Two-Byte	All
3	ADC r	[A] ← [A] + [r] + [CY]	Add the contents of register r in the contents of accumulator along with carry bit and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L.	Register	Three-byte	All
4	ADC M	[A] ← [A] + [(HL)] + [CY]	Add the contents of memory location M whose address is in HL register pair in the contents of accumulator along with carry bit and result is stored in accumulator.	Register Indirect	Four-Byte	All
5	ADI data	[A] ← [A] + 8-bit data	Add the 8-bit data specified in the contents of accumulator and result is stored in accumulator.	Immediate	Two-Byte	All

Contd...

Contd...

6	ACI data	[A] ← [A] + 8-bit data + [CY]	Add the 8-bit data specified in the contents of accumulator along with carry bit and result is stored in accumulator.	Immediate	Two-Byte	All
7	DAD rp	[HL] ← [rp] + [HL]	Add the 16-bit data given in the specified register pair in the contents of HL register pair and the result is stored in HL register pair.	Register	One Byte	Only Carry flag
8	SUB r	[A]←[A]−[r]	Subtract the contents of register r in the contents of accumulator and result is stored from accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L.	Register	One Byte	All
9	SUB M	[A] ← [A] − [(HL)]	Subtract the contents of memory location M whose address is in HL register pair from the contents of accumulator and result is stored in accumulator.	Register Indirect	One Byte	All
10	SBB r	[A]←[A]−[r] −[CY]	Subtract the contents of register r from the contents of accumulator along with borrow bit and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L.	Register	One Byte	All
11	SBB M	[A] ← [A] – [(HL)] – [CY]	Subtract the contents of memory location M whose address is in HL register pair from the contents of accumulator along with borrow bit and result is stored in accumulator.	Register Indirect	One Byte	All
12	SUI data	[A] ← [A] − 8-bit data	Subtract the 8-bit data location M whose address is in HL register pair from the contents of accumulator along with borrow bit and result is stored in accumulator.	Immediate	Two-Byte	All
13	SBI data	[A] ← [A] − 8-bit data − [CY]	Subtract the 8-bit data specified from the contents of accumulator along with borrow bit and result is stored in accumulator.	Immediate	Two-Byte	All
14	INR r	[r] ← [r] + [1]	Increment the contents of register r by one and result is stored in register r. Register rmay be any register out of six GPRs A, B, C, D, E, H, L.	Register	Two-Byte	All Except Carry Flag

Contd...

Contd...

15	DAA	If [A0-3] > 9H, then [A0-3] =[A0-3] + 06H If [A4-7] > 9H, then [A4-7] = [A4-7] + 06H	If Lower or Higher nibble of A> 9 then DAA instruction automatically add 06 H to the lower nibble of Accumulator with making AC=1.	Implicit	One Byte	All
16	DCR r	[r]=[r]-1	Contents of register are decremented by 1and the result is stored in the same place.	Register	One Byte	All Except Carry Flag
17	DCX rp	[rp]=[rp]-1	Contents of registers pair (total 16 bit number) are decremented by 1	Register	One Byte	No Flag
18	INX rp	[rp]=[rp]+1	Contents of registers pair (total 16-bit number) are increment by 1	Register	One Byte	No Flag
19	INR M	[M]=[M]+1	Contents of memory location (whose address is specified by H-L register pair) are increment by 1	Register Indirect	One Byte	All Except Carry Flag
20	DCR M	[M]=[M]-1	Contents of memory location (whose address is specified by H-L register pair) are decremented by 1	Register	One Byte	All Except Carry Flag

DAA (Decimal Adjust Accumulator): Implicit Addressing Mode

This instruction is used to get result of addition of two BCD numbers in the BCD format. It is the only instruction that uses the Auxiliary carry flag. It is used after with ADD instruction.

The lower nibble of A (Accumulator) is greater than 9 or auxiliary carry (AC) = 1, then Lower nibble of A = Lower nibble of A + 06 H.

If Higher nibble of A > 9 or AC = 1, then Higher nibble of A = Higher nibble of A + 06 H.

For example:

```
MVI A, 12H
ADI 39H
DAA
```

A = 0001 0010 = 12 H

+ Data = 0011 1001 = 39 H

Result (In Accumulator) = 0100 1011 = 4B H

The lower nibble of addition is greater than 9 (B >9), so the result is not in BCD format. Hence we use DAA instruction. It automatically add 06 H to the lower nibble of Accumulator with making Auxiliary carry flag high (AC=1). After DAA instruction the result in Accumulator becomes 0101 0001 (51 H), which is in BCD format.

Restriction: Restriction on DAA instruction is condition of flags. If you execute any instruction that affect CY flag or AC flag after ADD instruction and then you try to adjust result in accumulator in BCD format, result will be wrong. As DAA instruction use AC, CY flags of previous instructions to take decisions.

3.13.3 Logical Group

The instructions under this group perform logical operation such as AND, OR, compare, rotate, etc. Examples are: ANA, XRA, ORA, CMP, and RAL, etc.

S. No.	Instruction	Register transfer logic	Logical Group Details	Addressing Mode	Instruction length	Flags affected
1	ANA r	[A]←[A] ∧ [r]	AND the contents of register r with the contents of accumulator and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L. AND operation is performed to the mask the bits or to check the status of particular bit.	Register	One Byte	All CY = 0 AC = 1
2	ANA M	[A]←[A]∧ [(HL)]	AND the contents of memory location whose address is in HL register pair with the contents of accumulator and result is stored in accumulator.	Register Indirect	One Byte	All CY = 0 AC = 1
3	ANI data	[A]←[A]∧ 8-bit data	AND the 8-bit data specified in the contents of accumulator and result is stored in accumulator.	Immediate	Two-Byte	All CY = 0 AC = 1
4	ORA r	[A]←[A]∨ r	OR the contents of register r with the contents of accumulator and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L. (OR operation is used to set the bits)	Register	One Byte	All CY = 0 AC = 0
5	ORA M	[A]←[A]∨ [(HL)]	OR the contents of memory location whose address is in HL register pair with the contents of accumulator and result is stored in accumulator.	Register Indirect	One Byte	All CY = 0 AC = 0
6	ORI data	[A]←[A]∨ 8-bit data	OR the 8-bit data specified in the contents of accumulator and result is stored in accumulator.	Immediate	Two-Byte	All CY = 0 AC = 0

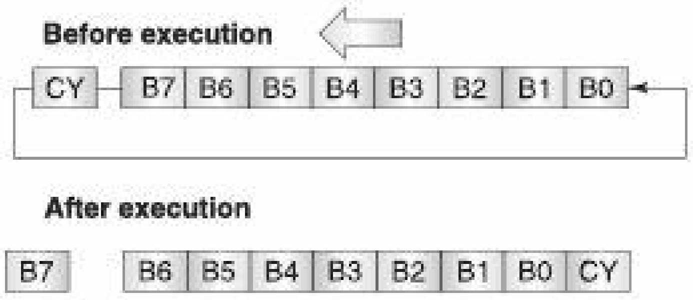
Contd...

Contd...

7	XRA r	[A]←[A]∀ [r]	Exclusive-OR the contents of register r with the contents of accumulator and result is stored in accumulator. Register r may be any register out of six GPRs A, B, C, D, E, H, L. (Exclusive-OR operation is performed to clear the register)	Register	One Byte	All CY = 0 AC = 0
8	XRA M	[A]←[A]∀ [(HL)]	Exclusive-OR the contents of memory location whose address is in HL register pair with the contents of accumulator and result is stored in accumulator.	Register Indirect	One Byte	All CY = 0 AC = 0
9	XRI data	[A]←[A]∀ 8-bit data	Exclusive-OR the 8-bit data specified in the contents of accumulator and result is stored in accumulator.	Immediate	Two-Byte	All CY = 0 AC = 0
10	CMA	$[\overline{A}]$	Complement the contents of accumulator.	Implicit	One Byte	None
11	CMC	$[\overline{CY}]$	Complement the status of carry flag	Implicit	One Byte	CY
12	STC	[CY] ← 1	Set carry flag.	Implicit	One Byte	CY
13	CMP r	[A] − [r]	Compare register with accumulator. The contents of register r is subtracted from the contents of accumulator and status flags are set according to the result of subtraction. But the result of subtraction is discarded. The content of accumulator remains unchanged.	Register	One Byte	All
14	CMP M	[A]−[(HL)]	Compare memory contents with acc. The contents of memory location whose address is in HL register pair is subtracted from the contents of accumulator and status flags are set according to the result of subtraction. But the result of subtraction is discarded. The contents of accumulator remain unchanged.	Register Indirect	One Byte	All
15	CPI data	[A] − 8-bit data	Compare immediate data with accumulator	Immediate	Two-Byte	All
16	RLC	[An+1] ← [An] [A0]←[A7] [CY]←[A7]	Contents of accumulator are rotated left by one bit. MSB of accumulator is copied into carry bit as well as to LSB of accumulator.	Implicit	One Byte	CY

Rotate Operations: These instructions address the accumulator only. They perform a shift left or shift right of the accumulator contents.

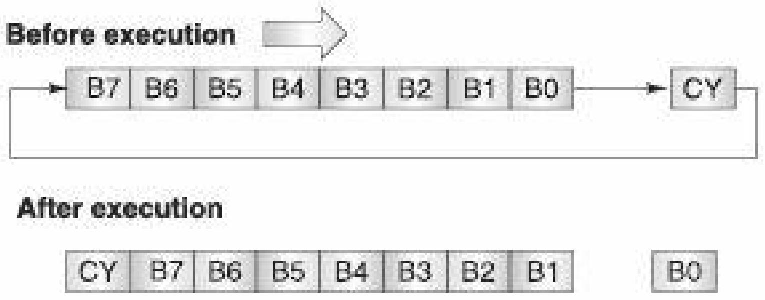
RAL: Each binary bit of the accumulator is rotated left by one position through the carry flag. Bit D_7 is placed in the bit in the carry flag and the carry flag is placed in the least significant position D_0.



Example:

CY=0		CY=1	
MVI A,8FH	;A=8FH	STC	;carry =1
RAL	;A=1EH	MVI A,8FH	;A=8FH
		RAL	;A=1FH, CY=1

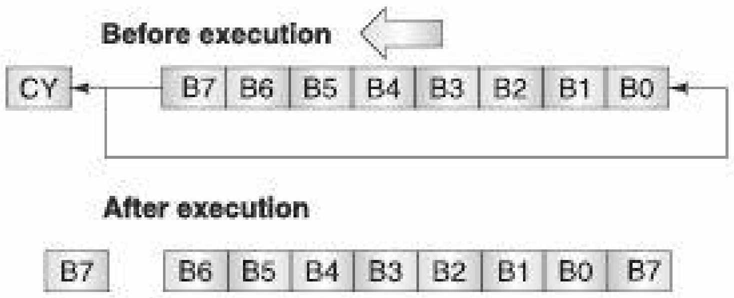
RAR: Each binary bit of the accumulator is rotated right by one position through the carry flag. Bit D_0 is placed in the carry flag and the bit in the carry flag is placed in the most significant position D_7.



Example:

CY=0		CY=1	
MVI A,1FH	;A=1FH	STC	;carry =1
RAR	;A=0FH, CY=1	MVI A,1FH	;A=1FH
		RAR	;A=8FH, CY=1

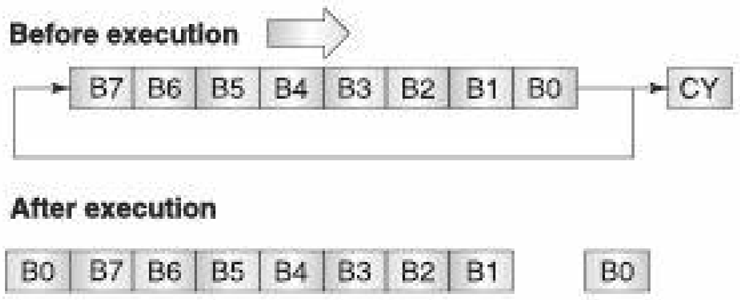
RLC: Each binary bit of the accumulator is rotated left by one position Bit D_7 is placed in the position of D_0 as well as in the carry flag.



Example:

CY=0		CY=1	
MVI A,1FH	;A=8FH	STC	;carry =1
RLC	;A=1FH, CY=1	MVI A,8FH	;A=8FH
		RLC	;A=1FH, CY=1

RRC: Each binary bit of the accumulator is rotated right by one position Bit D_0 is placed in the position of D_7 as well as in the carry flag.



Example:

CY=0		CY=1	
MVI A,1FH	;A=1FH	STC	;carry =1
RLC	;A=8FH, CY=1	MVI A,1FH	;A=1FH
		RLC	;A=8FH, CY=1

CMP R: In this instruction contents of the register are compared with the contents of the accumulator. Both contents are preserved. Comparison is done with help of subtraction method. The contents of register is subtracted from accumulator and following changes made in PSW(flags) register:

CMP R = [A]-[R]

- If A>R, it results carry and zero flags are reset (Z=0, CY=0)
- If A<R, carry flag is set and zero flag is reset (Z=0, CY=1)
- If A=R, zero flag is set and carry flag is reset (Z=1, CY=0)

3.13.4 Branch Control Group

This group includes the instructions for conditional and unconditional jump, subroutine call and return, and restart. Examples are: JMP, JC, JZ, CALL, CZ, RST, etc.

Sr. No.	Instruction	Register transfer logic	Branch Control Group Details	Addressing Mode	Instruction length	Flags affected
1	Jump adr (label)	[PC] ← label	Unconditional jump to the instruction specified by operand. Byte 2nd and Byte 3rd of the instruction give the address of label where the program branches.	Immediate	Three-Byte	None
2	Conditional Jump Addr Label	[PC] ← label	After the execution of a conditional jump instruction, the program sequence branches to the address specified as label in the instruction, if the specified condition is satisfied else the program follows the normal sequence.	Immediate	Three-Byte	None

Sr. No.	Instruction	Register transfer logic	Details	Addressing Mode	Instruction length	Flags affected
(i)	JZ addr	[PC] ← label	Jump to label address if Z = 1 (zero is set)	Immediate	Three-Byte	None
(ii)	JNZ addr	[PC] ← label	Jump to label address if Z = 0 (zero is reset)	Immediate	Three-Byte	None
(iii)	JC addr	[PC] ← label	Jump to label address if CY = 1 (Carry flag is set)	Immediate	Three-Byte	None
(iv)	JNC addr	[PC] ← label	Jump to label address if CY = 0 (carry flag is reset)	Immediate	Three-Byte	None
(v)	JP addr	[PC] ← label	Jump to label address if S = 0 (sign flag is reset, i.e., result is positive)	Immediate	Three-Byte	None
(vi)	JM addr	[PC] ← label	Jump to label address if S = 1 (sign flag is set, i.e. result is negative)	Immediate	Three-Byte	None
(vii)	JPE addr	[PC] ← label	Jump to label address if P = 1 (parity is even)	Immediate	Three-Byte	None
(viii)	JPO addr	[PC] ← label	Jump to label address if P = 0 (parity is odd)	Immediate	Three-Byte	None

Various conditions jumps are listed below:

S. No.	Instruction	Register transfer logic	Details	Addressing Mode	Instruction length	Flags affected
1	CALL addr (label)	[(SP) − 1]← [PCH] [(SP) −2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Unconditional CALL to the subroutine identified by operand (Before branching to subroutine, addresses of PC are saved on stack)	Immediate/ Register indirect	Three-Byte	None
2	Conditional CALL addr (label)	[(SP) − 1] ←[PCH] [(SP) − 2]← [PCL] [SP] ←[SP] − 2 [PC] ← addr (label)	After the execution of a conditional CALL instruction, the program sequence branches to the address specified as label in the instruction, if the specified condition is satisfied else the program follows the normal sequence.	Immediate	Three-Byte	None

Various conditional jump instructions are as follows:

(i)	CZ addr (label)	[(SP) −1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] −2 [PC] ← addr (label)	Call subroutine if Z = 1 (zero flag is set)	Immediate	Five-Byte/ Two-Byte	None
(ii)	CNZ addr (label)	[(SP) −1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if Z = 0 (zero flag is reset)	Immediate	Three-Byte	None
(iii)	CC addr (label)	[(SP) −1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if CY = 1 (carry flag is set)	Immediate	Three-Byte	None
(iv)	CNC addr (label)	[(SP) −1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if CY = 0 (carry flag is reset)	Immediate	Three-Byte	None
(v)	CP addr (label)	[(SP) − 1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if S = 0 (sign flag is reset, i.e., result is positive)	Immediate	Three-Byte	None

(vi)	CM addr (label)	[(SP) − 1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if S = 1 (sign flag is set, i.e., result is negative)	Immediate	Three-Byte	None
(vii)	CPE addr (label)	[(SP) − 1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if P = 1 (parity is even)	Immediate	Three-Byte	None
(viii)	CPO addr (label)	[(SP) − 1] ← [PCH] [(SP) − 2] ← [PCL] [SP] ← [SP] − 2 [PC] ← addr (label)	Call subroutine if P = 0 (parity is odd)	Immediate	Three-Byte	None

3	RET	[PCL] ←[(SP)] [PCH] ← [(SP) + 1] [SP] ← [SP] + 2	RET instruction is used at the end of a subroutine. The execution of RET instruction brings back the saved address from the stack to program counter. Program jumps to the next instruction of main program which is next to CALL instruction.	Register Indirect	One Byte	None
4	Conditional Return	[PCL] ← [(SP)] [PCH] ← [(SP) + 1] [SP] ← [SP] + 2	The execution of RET instruction brings back the saved address from the stack to program counter if the specified condition is satisfied. Program jumps to the next instruction of main program which is next to CALL instruction.	Register Indirect	One Byte	None

Various conditional return instructions are as follows:

(i)	RZ	[PCL] ← [(SP)] [PCH]← [(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if Z = 1 (zero flag is set)	Register Indirect	One Byte	None
(ii)	RNZ	[PCL] [(SP)][PCH] ← [(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if Z = 0	Register Indirect	One Byte	None
(iii)	RC	[PCL] [PCH] ← [(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if CY = 1 (carry flag is set)	Register Indirect	One Byte	None
(iv)	RNC	[PCL] ← [(SP)] [PCH] ←[(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if CY = 0 (carry flag is set)	Register Indirect	One Byte	None
(v)	RP	[PCH] ← [(SP)], [PCL] [PCH] ← [(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if S = 0 (sign flag is reset, i.e., result is positive)	Register Indirect	One Byte	None

(vi)	RPE	[PCL] ← [(SP)] [PCH] ← [(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if P = 1 Register Indirect/ (parity is even)	Register Indirect	One Byte	None
(vii)	RPO	[PCL] ←[(SP)] [PCH] ←[(SP) + 1] [SP] ← [SP] + 2	Return from subroutine if P = 0 (parity is odd)	Register Indirect	One Byte	None
(viii)	RST n	[(SP) − 1] ←[PCH] [(SP) − 2] ← [PCL] [SP] ←[SP] − 2 [PC] ← 8 times n	Restart interrupt subroutine. Register Indirect/ This is one word call instruction. One Byte The program jumps to the instruction starting at restart location. The address of restart location is 8 times integer n.	Register Indirect	One Byte	None

5	PCHL	[PC] ← [HL] [PCH] ← H [PCL] ← L	Contents of register pair HL are moved to program counter, such that contents of H register are moved to high order 8 bits of register PC and contents of register L are moved to low order 8 bits of register program counter.	Implicit	One Byte	None

Vector Address Calculation of RST n

Vector of RST n = hexadecimal equivalent of(8 * n)

For example RST 2 = hexadecimal equivalent of(8 * 2=16) = 0X010

Software interrupt	Vector address
RST 0	0x0000H
RST 1	0x0008H
RST 2	0x0010H
RST 3	0x0018H

3.13.5 I/O and Machine Control Group

This group includes the instructions for input/output ports, stack and machine control. Examples are: IN, OUT, PUSH, POP, and HLT, etc.

S. No.	Instruction	Register transfer logic	I/O and Machine Control Group Details	Addressing Mode	Instruction length	Flags affected
1	PUSH rp	[(SP) − 1] ← [rh] [(SP) − 2] ← [rl] [SP] ← [SP] − 2	Push the contents of register pair rp to stack.	Register Indirect	One Byte	None

2	PUSH PSW	[(SP) − 1] ← [A] [(SP) − 2] ← [PSW] [SP] ← [SP] − 2	Push the contents of accumulator and status register on stack.	Register Indirect	One Byte	None
3		[rl] ← [(SP)] [rh] ← [(SP) + 1] [SP] ← [SP] + 2	Pop the contents of register pair rp from stack	Register Indirect	One Byte	None
4	POP PSW	[PSW] ← [(SP)] [A] ← [(SP) + 1] [SP] ← [SP] + 2	Pop the contents of accumulator and status register from stack.	Register Indirect	One Byte	None
5	HLT	Halt	The execution of this instruction stops the microprocessor operation. The register and status flags remain unaffected.	Implicit	One Byte	None
6	XTHL	[L] ↔ [(SP)] [H] ↔ [(SP) + 1]	Exchange stack top with HL. The contents of register L are exchanged with the byte of stack top. The contents of register H are exchanged with byte below stack top.	Register Indirect	One Byte	None
7	SPHL	(SP) ← (HL)	Move the contents of HL pair to stack pointer.	Register	One Byte	None
8	EI	Enable Interrupt	With the execution of this instruction, interrupts are enabled.	Implicit	One Byte	None
9	DI	Disable Interrupt	With the execution of this instruction, interrupts are disabled.	Implicit	One Byte	None
10	NOP	No Operation	The execution of this instruction performs no operation. The register and status flags remain unaffected.	Implicit	One Byte	None
11	SIM	Set Interrupt Masks	The execution of this instruction loads the accumulator with 8-bit data.	Implicit	One Byte	None
12	RIM	Read Interrupt Mask	It reads the status of interrupts and to read the serial data input bit.	Implicit	One Byte	None

When **SIM** instruction is executed, an 8-bit data is loaded in accumulator and according to data bits we can mask the hardware interrupts and also do serial output data transmission

XCHG: This instruction is used to exchange the content of H-L register pair with that of D-E register pairs. The contents of register H are exchanged with the contents of register D and the contents of register L are exchanged with the contents of register E.

For example: Suppose the content of register D = 10H, E = 15H and the content of H = 20H, L = 40H. Then execute following instruction:

HLT: HLT is the mnemonics for 'Halt the microprocessor' instruction. It is 1-byte instruction. When this instruction is executed, the 8085 halts further processing and enter Halt state. This is indicated by S_1 and S_0 output signals becoming 00. The 8085 comes out of the Halt state when a valid interrupt occurs. In such a case, it executes the corresponding interrupt service subroutine and then continues with the instruction after the HLT instruction, however in most programs the HLT instruction is used for terminating the program. Also activation of reset in cause 8085 to comes out of Halt state.

XCHG

After this instruction the contents of DE register pair and HL register pair becomes:

D = 20H, E = 40H and H = 10H, L =15H.

The 8085 microprocessor has two instructions for data transfer between the processor and I/O device: IN and OUT. The instruction IN (Code DB) inputs data from an input device (such as keyboard) into the accumulator, and the instruction OUT (Code D3) sends the contents of the accumulator to an output device such as an LED display. These are 2-byte instructions, with the second byte specifying the address or the port number of an I/O device. For example, the OUT instruction is described as follows.

Opcode	**Operand**	**Description**
OUT	8-bit Port Address	This is a two-byte instruction with the hexadecimal opcode D3 H and the second byte is the port address of an output device. This instruction transfers (copies) data from the accumulator to the output device.

3.14 I/O EXECUTION

The execution of I/O instructions can best be illustrated using the example of the OUT instruction. The 8085 executes the OUT instruction in three machine cycles, and it takes ten T-states (clock periods) to complete the execution.

3.14.1 OUT Instruction

In the first machine cycle, M1 (Opcode Fetch), the 8085 places the high-order address 20 H on A_{15}-A_8 and the low-order address 50 H on AD_7-AD_0. At the same time, ALE goes high and IO/$\overline{M}$ goes low. The ALE signal indicates the availability of the address on AD_7-AD_0, and it can be used in demultiplexing the bus. The IO/$\overline{M}$, being low indicates that it is a memory related operation. At T_2, the microprocessor sends the $\overline{RD}$ control signal, which is combined with IO/$\overline{M}$ (externally) to generate the $\overline{MEMR}$ signal; and the processor fetches the instruction code D_3 H using the data bus. When the 8085 decodes the machine code D3 H, it finds out that the instruction is a 2-byte instruction and that it must read the second byte. In the second machine cycle, M2 (Memory Read), the 8085 places the next address, 2051 H, on the address bus and gets the device address 01 H via the data bus. In the third machine cycle, M3 (I/O Write), the 8085 places the device address 01 H on the low-order (AD7-AD0) as well as the high-order

(A15-A8) address bus. The IO/$\overline{M}$ signal goes high to indicate that it is an I/O operation. At T_2, the accumulator contents are placed on the data bus (AD_7-AD_0), followed by the control signal $\overline{WR}$. By ANDing the IO/$\overline{M}$ and $\overline{WR}$ signals, the $\overline{IOW}$ (see Fgure 3.12) signal can be generated to enable an output device.

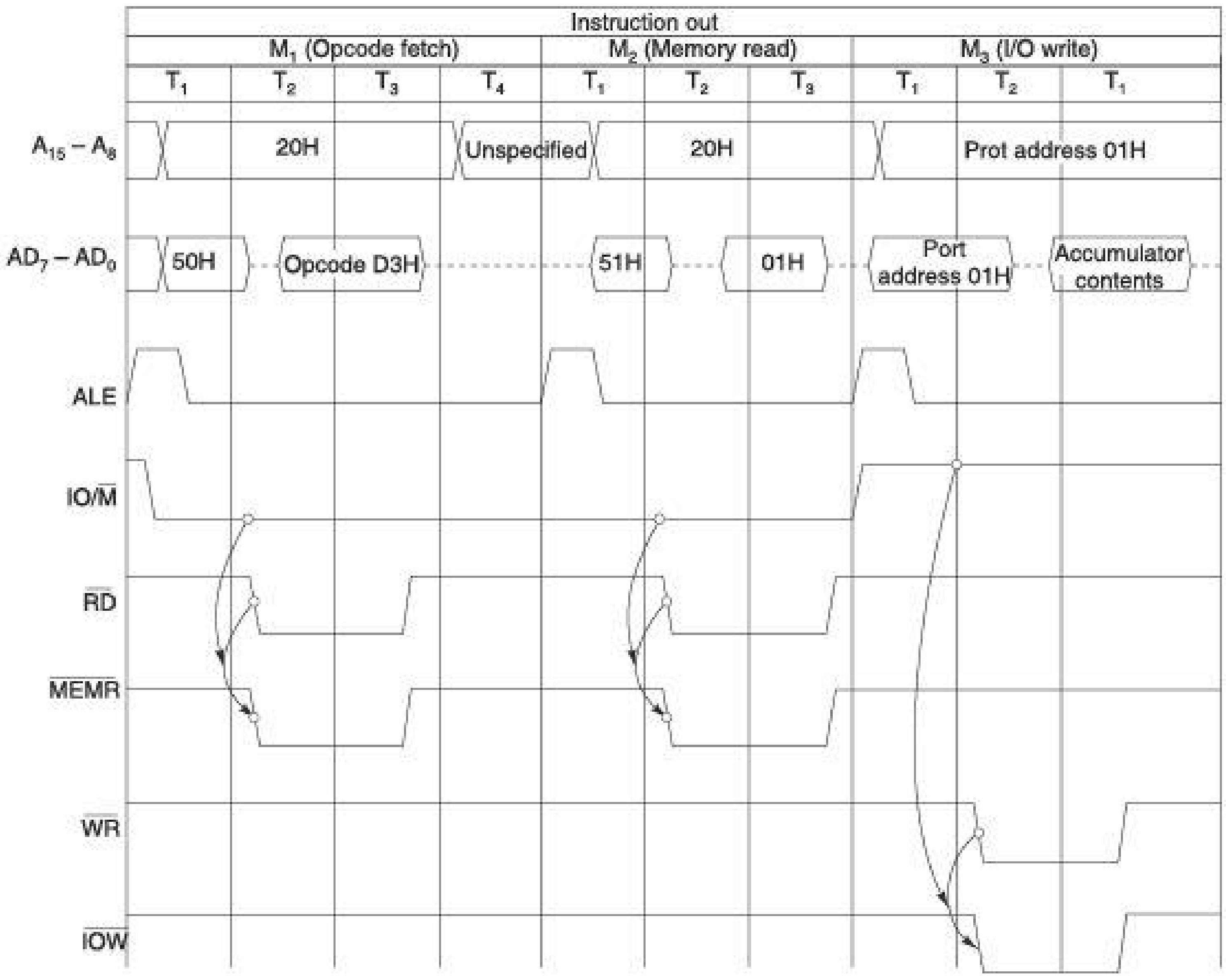


Fig. 3.12 Timing Diagram of the OUT Instruction.

To display the contents of accumulator at an output device (such as LEDs) with the address, for example 01H, the instruction will be written and stored in memory as follows:

Memory Address	Machine code	Mnemonics
2050	D3	OUT 01H
2051	01	

If the output port with the address 01H is designed as LED display the instruction OUT will display the contents of the accumulator at the port. The second byte of this OUT instruction can be any of the 256 combination of eight bits from 00H to FFH.

Figure 3.12 shows the execution timing of the OUT instruction. The necessary information for interfacing an output device is available during T_2 and T_3 of the M_3 cycle. The data byte to be displayed is on the data bus, the 8-bit device address is available on the low-order as well as high-order address bus, and availability of the data byte is indicated by the $\overline{WR}$ control signal.

The availability of the device address on both segments of the address bus is redundant information; in peripheral I/O, only one segment of the address bus (low or high) is sufficient for interfacing. The data byte remains on the data bus only for two T-states, then the processor goes on to execute the next instruction. Therefore, the data byte must be latched now, before it is lost, using the device address and the control signal.

3.14.2 IN Instruction

The 8085 instruction set includes the instruction IN to read (copy) data from input devices such as switches, keyboards, and A/D data converters. This is a two-byte instruction that reads an input device and places the data in the accumulator as shown in Fig. 3.13. The first byte is the opcode and the second byte specifies the port address. Thus, the addresses for input devices can manage from 00 H to FFH. The instruction is described as

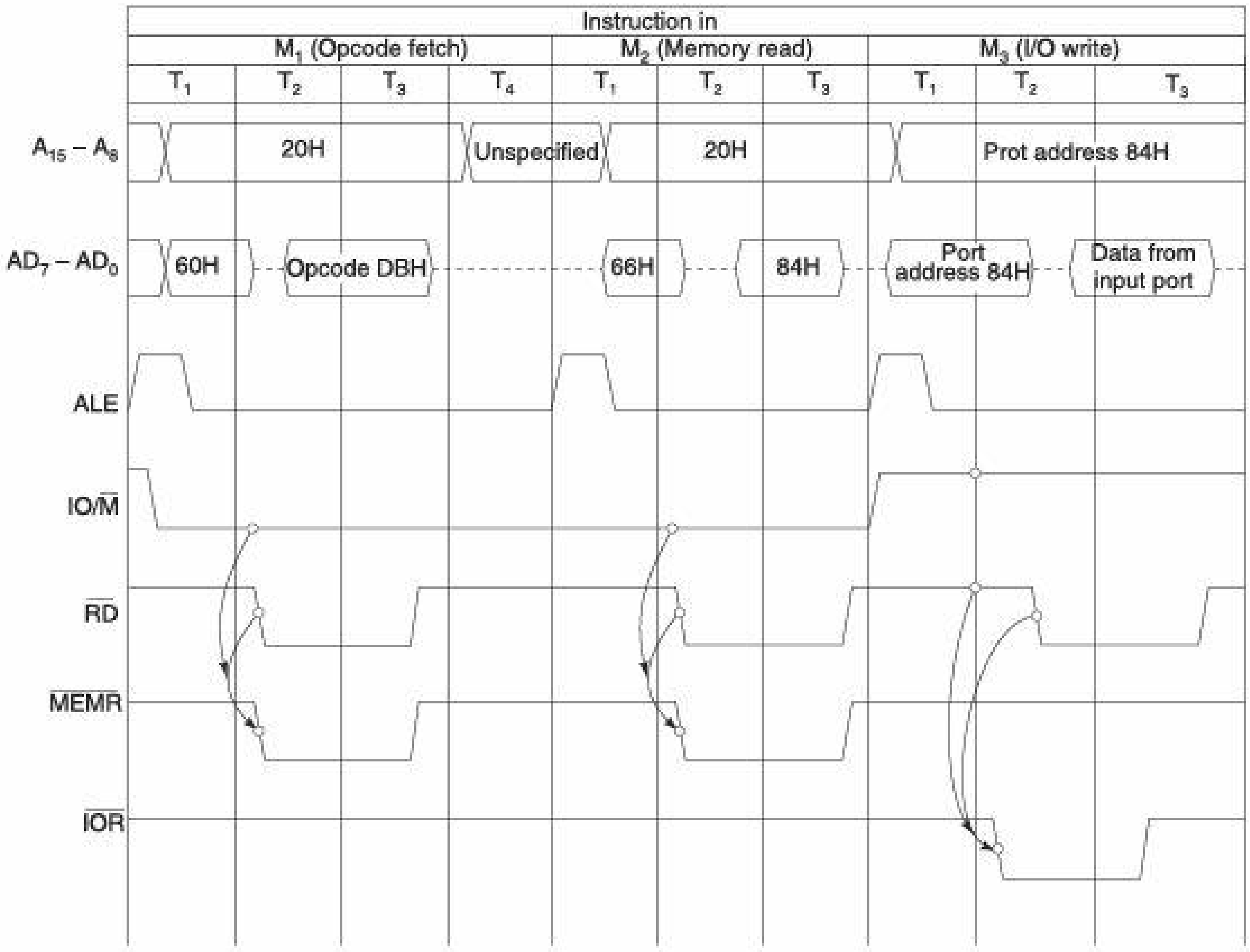


Fig. 3.13 Timing Diagram of IN Instruction.

Instruction	Operands	Functions
IN	8-bit	This is a two-byte instruction with the hexadecimal opcode DB, and the second byte is the port address of an input device.
Byte IN		This instruction reads (copies) data from an input device and places the data in the accumulator.

For example to read switch positions from an input port with the address 84H, the instructions will be written and stored in memory and follows:

Memory Address	Machine code	Mnemonics
2065	DB	IN 84H
2066	84	

3.14.3 Device Selection and Data Transfer

The objective of interfacing an output device is to get information or a result out of the microprocessor and store it or display it. The OUT instruction serves that purpose; during the M3 cycle of the OUT instruction the microprocessor places that information (accumulator contents) on the data bus. If we connect the data bus to a latch, we can catch that information and display it via LEDs or a printer. Now the questions are:

1. When should we enable the latch to catch that information?
2. What should be the address of that latch?

The answers to both the questions can be found in the M_3 cycle. The latch should be enabled when $IO/\overline{M}$ is high and $\overline{WR}$ is active low. Similarly, the address of an output port is also on the address bus during M_3 (it is 01 H). Now the task is to generate one pulse by decoding the address bus (A_7-A_0 or A_{15}-A_8), to indicate the presence of the port address, generate a timing pulse by combining $IO/\overline{M}$ and $\overline{WR}$ signals to indicate that the data byte we are looking for is on the data bus, and use these pulses (by combining them) to enable the latch. These steps are summarized as follows. (For all subsequent discussion, the bus A_7-A_0 is assumed to be the demultiplexing bus AD_7-AD_0.)

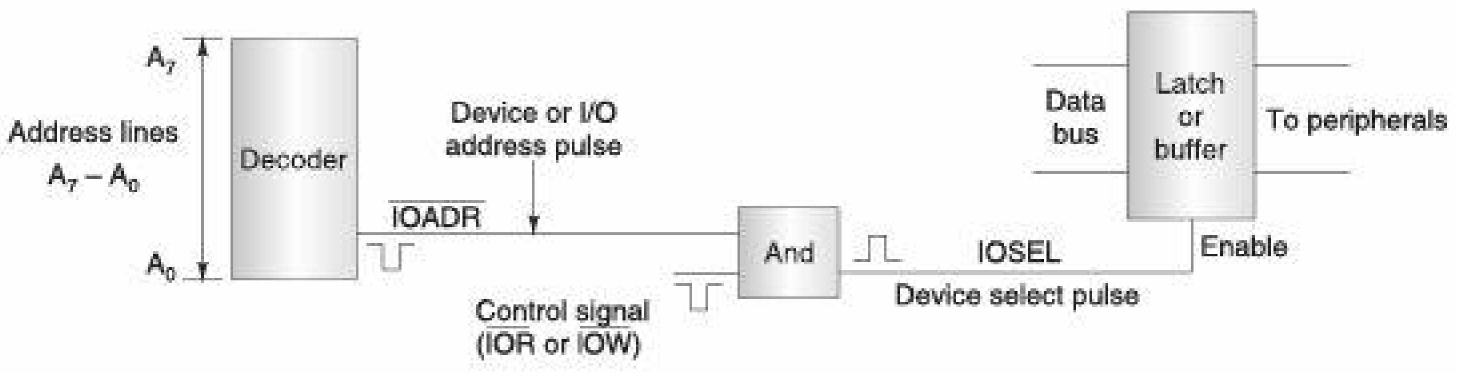


Fig. 3.14(a) Block Diagram of I/O Interface.

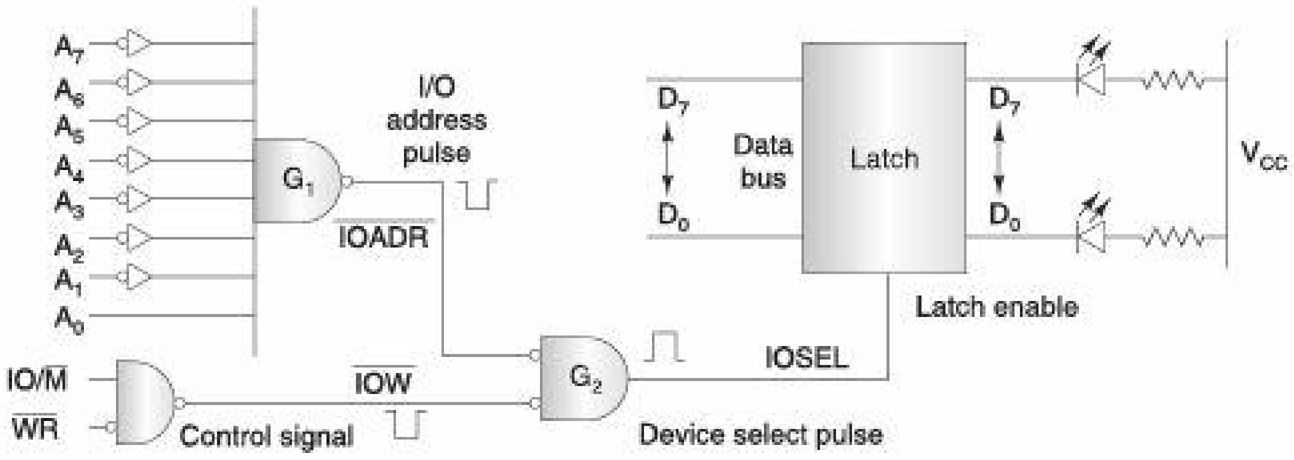


Fig. 3.14(b) Decode Logic for LED Output Port.

- Decode the address bus to generate a unique pulse corresponding to the device address on the bus; this is called the device address pulse or I/O address pulse.
- Combine (AND) the device address pulse with the control signal to generate a device select (I/O select) pulse that is generated only when both signals are asserted.
- Use the I/O select pulse to activate the interfacing device (I/O port).

The block diagram in Figure 3.14(a) illustrates these steps for interfacing an I/O device, address lines A_7-A_0 are connected to a decoder, which will generate a unique pulse corresponding to each address on the address lines. This pulse is combined with the control signal to generate a device select pulse, which is used to enable an output latch or an input buffer. Figure 3.14(b) shows a practical decoding circuit for the output device with address 01 H. Address lines A_7-A_0 are connected to the 8-input NAND gate that functions as a decoder. Line is connected directly, and lines A_7-A_1 are connected through the inverters. When the address bus carries address 01 H, gate G_1, generates a low pulse; otherwise, the output remains high Gate G_2 combines the output of G_1 and the control signal $\overline{IOW}$ to generate an I/O select pulse when both input signals are low.

Meanwhile (as was shown in the timing diagram, machine cycle M_3), the contents of the accumulator are placed on the data bus and are available on the data bus for a few microseconds and, therefore, must be latched for display. The I/O select pulse clocks the data into the latch for display by the LEDs.

3.14.4 Absolute vs. Partial Decoding

If all eight address lines are decoded to generate one unique output pulse; the device will be selected only with the address, 01 H. This is called absolute decoding and is a good design practice. However, to minimize the cost, the output port can be selected by decoding some of the address lines, this is called **partial decoding**. As a result, the device has multiple addresses (similar to fold back memory addresses).

3.14.4.1 Memory-Mapped I/O

In memory-mapped I/O, the input and output devices are assigned and identified by 16-bit addresses. To transfer data between the MPU and I/O devices, memory-related instructions (such as LDA, STA, etc.) and memory control signals ($\overline{MEMR}$ and $\overline{MEMW}$) are used. The microprocessor communicates with an I/O device as if it was one of the memory locations.

The memory-mapped I/O technique is similar in many ways to the peripheral I/O technique. To understand the similarities, it is necessary to review how a data byte is transferred from the 8085 microprocessor to a memory location or vice versa. For example, the following instruction will transfer the contents of the accumulator to the memory location 4000 H.

Memory	Machine	Mnemonics	Comment Address
2050	32	STA 8000H	Store contents of accumulator in memory location
2051	00		
2052	40		

The STA is a three-byte instruction; the first byte is the opcode, and the second and third bytes specify the memory address. The 16-bit address 8000 H is entered in the reverse order; the low-order byte l is stored on location 2051, followed by the high order address 80 H. In this example, if an output, instead of a memory register, is connected at this address, the accumulator contents will be transferred to the output device. This is called the memory-mapped I/O technique.

On the other hand, the instruction LDA (Load Accumulator Direct) transfers the memory location to the accumulator. The instruction LDA is a 3-byte instruction; the second and third bytes specify the memory location. In the memory-mapped I/O technique, an input device (keyboard) is connected instead of a memory. The input device will have the 16-bit address specified by the LDA instruction. When the microprocessor executes the LDA instruction, the accumulator receives data from the input rather than from a memory location. To use memory-related instructions for data, the control signals Memory Read ($\overline{MEMR}$) and Memory Write ($\overline{MEMW}$) should be connected to I/O devices instead of $\overline{WR}$ and $\overline{IOW}$ signals and the 16-bit address bus should be decoded.

3.14.4.2 Execution of Memory-Related Data Transfer Instructions

The execution of memory-related data transfer instructions is similar to the execution of IN or OUT instructions, except that the memory-related instructions have 16-bit addresses. The microprocessor requires four machine cycles (13 T-states) to execute the instruction STA. The machine cycle M_4 for the STA instruction is similar to the machine cycle M_3 for the OUT instruction. For example, to execute the instruction STA 8000 H in the fourth machine cycle (M_4), the microprocessor places memories address 8000 H on the entire address bus (A_{15}-A_0). The accumulator contents are sent on the data bus, followed by the control signal Memory Write $\overline{MEMW}$ (active low). On the other hand, in executing the OUT instruction (Figure 3.15), the 8-bit device address is repeated on the low-order address bus (A_0-A_7) as well as on the high-order bus, and the $\overline{IOW}$ control signal is used. To identify an output device, either the low-order or the high order bus can be decoded. In the case of the STA instruction, the entire bus must be decoded. Device selection and data transfer in memory-mapped I/O require three steps that are similar to those required in peripheral I/O:

1. Decode the address bus to generate the device address pulse.
2. AND the control signal with the device address pulse to generate the device selects (I/O select) pulse.
3. Use the device select pulse to enable the I/O port.

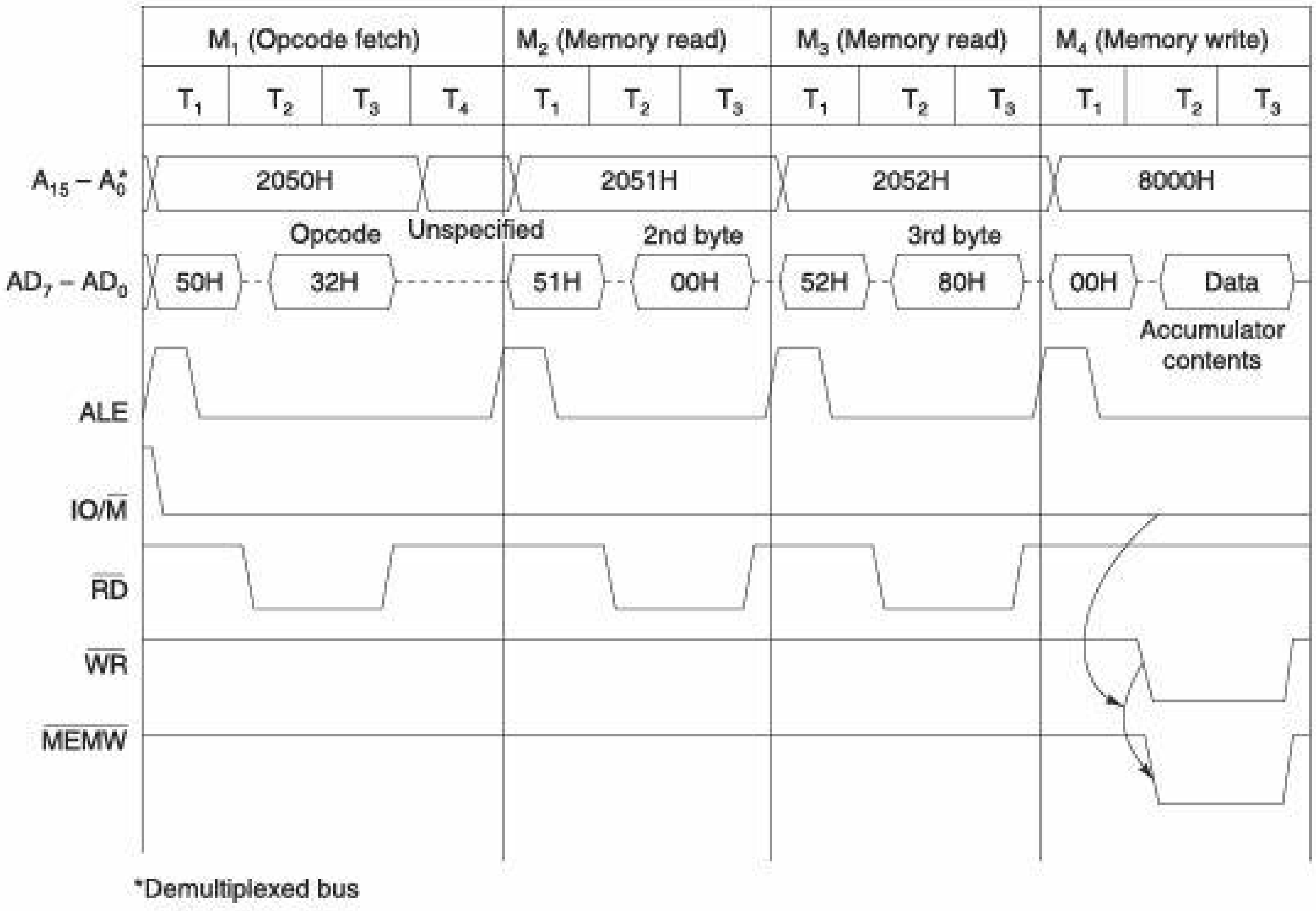


Fig 3.15 Data Transfer Timing Diagram.

To interface a memory-mapped input port, we can use the instruction LDA 16-bit, which reads data from an input port with the 16-bit address and places the data in the accumulator. The instruction has four machine cycles; only the fourth machine cycle differs from M_4 in Figure 3.15. The control signal will be RD rather than $\overline{WR}$, and data flow from the input port to the microprocessor.

3.14.4.3 Output Port and its Address

The various process control devices are connected to the data bus through the latch 74LS373 and solid state relays. If an output bit of the 74LS373 is high, it activates the corresponding relay, and turns on the process; the process remains on until the bit stays high. Therefore, to control these safety processes, we need to supply an appropriate bit to the latch

The 74LS373 is a latch followed by a tri-state buffer, as shown in Figure 3.16. The latch buffer is controlled independently by the Latch Enable (LE) and Output Enable ($\overline{OE}$). When LE is high, the data enter the latch, and when LE goes low, data are latched. The latched data are available on the output lines of the 74LS373 if the buffer is enabled by $\overline{\text{OE}}$ (active low). If $\overline{\text{OE}}$ is high, the output lines go into the high impedance state. Figure 3.16 shows that the $\overline{\text{OE}}$ is connected to the ground; thus, the latched data will keep the relays on /off according to the bit pattern. The LE is connected to the device select pulse, which is asserted when the output 00 of the decoder and the control signal $\overline{\text{MEMW}}$ go low.

Therefore to assert the I/O select pulse the output port address should ne FFF8H as shown below:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	= FFF8H

$\underbrace{\text{A15 – A8}}_{\text{To 8-input NAND gate to enable } \overline{\text{E2}}}$ $\underbrace{\text{A7 – A4}}_{\text{To 4-input NAND gate to enable } \overline{\text{E1}}}$ $\underbrace{\text{A3}}_{\text{To enable E3}}$ $\underbrace{\text{A2 – A0}}_{\text{Decoder input}}$

3.14.4.4 *Input Port and its Address*

The DIP switches are interfaced with the 8085 using the tri-state buffer 74LS244. The switches are tied high, and they are turned on by grounding, as shown in Figure 3.16. The switch positions can be read by enabling the signal $\overline{\text{OE}}$, which is asserted when the output O_1 of the decoder and the control signal $\overline{\text{MEMR}}$ go low. Therefore to read the input port the port address should be as follows:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	= FFF9H

$\underbrace{\text{A15 – A8}}_{\text{To 8-input NAND gate to enable } \overline{\text{E2}}}$ $\underbrace{\text{A7 – A4}}_{\text{To 4-input NAND gate to enable } \overline{\text{E1}}}$ $\underbrace{\text{A3}}_{\text{To enable E3}}$ $\underbrace{\text{A2 – A0}}_{\text{Decoder input}}$

3.15 STACK AND SUBROUTINES

The stack is a group location in the R/W memory that is used for temporary storage of binary information during the execution of a program. The starting memory location of the stack is defined in the main program, and space is reserved usually at the high end of the memory map. The method of information storage resembles a stack of books. The contents of each memory location are; in a sense, "stacked"—one memory location above another—and information is retrieved starting from the top. Hence, this particular group of memory locations is called the stack. This chapter introduces the stack instructions in the 8085 set. The latter part of this chapter deals with the subroutine technique, which is frequently used in programs.

A subroutine is a group of instructions that performs a subtask (e.g., time delay or arithmetic operation) of repeated occurrence. The subroutine is written, as a separate unit, apart from the

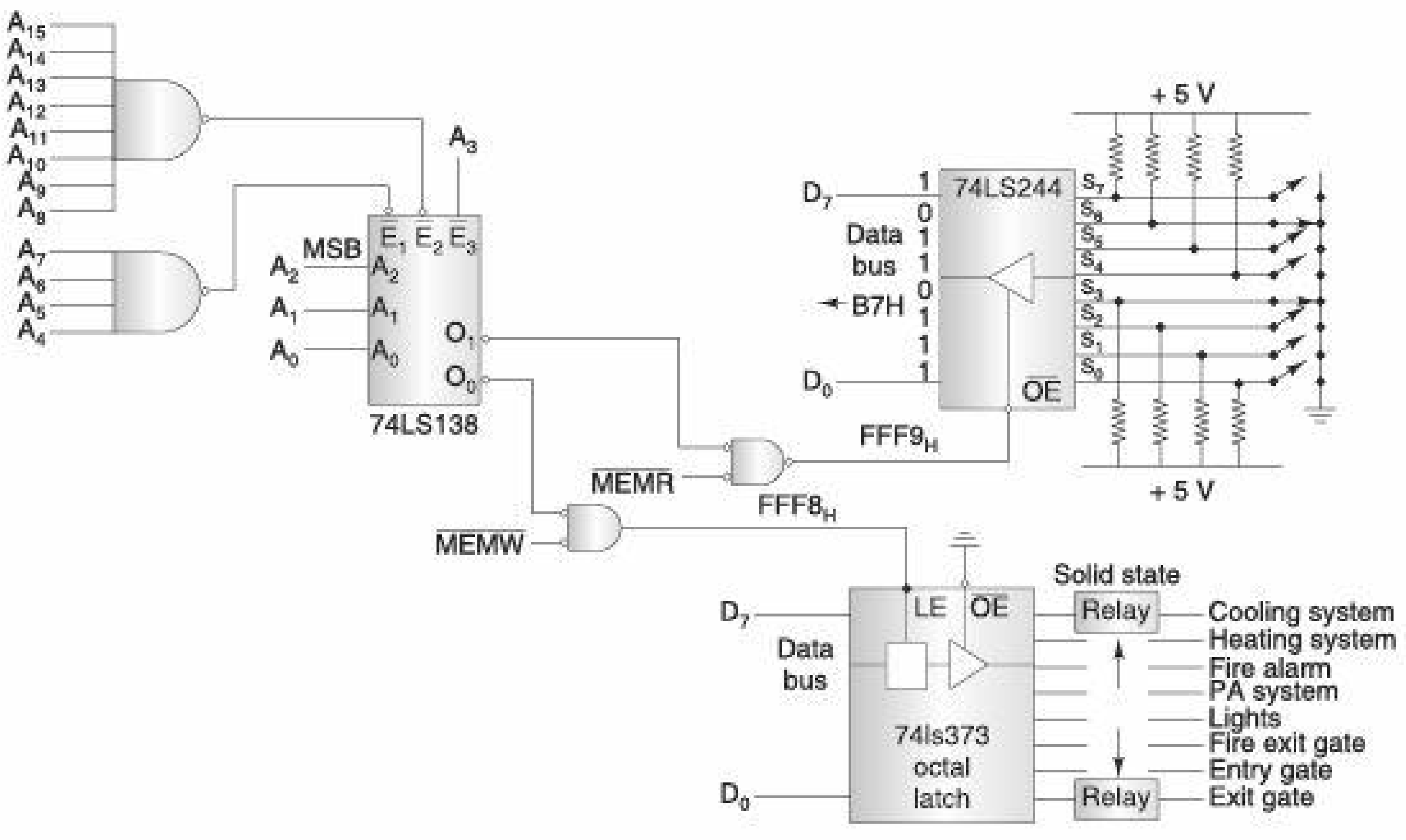


Fig. 3.16 Memory Mapped I/O Interfacing

main program, and the microprocessor transfers the program execution from the main program to the subroutine whenever it is called to perform the task. After the completion of the subroutine task, the microprocessor returns to the main program. The subroutine technique eliminates the need to write a subtask repeatedly, thus, it uses memory more efficiently.

Before implementing the subroutine technique, the stack must be defined the stack is used to store the memory address of the instruction in the main program that follows the subroutine call. The stack and the subroutine offer great deal of flexibility in writing programs. A large software project is usually divided into subtasks called modules. These are developed independently as subroutines by different programmers. Each programmer can use all the microprocessor registers to write a subroutine without affecting the other parts of the program. At the beginning of the subroutine module, the register contents of the main program are stored on the stack, and these register contents are retrieved before returning to the main program.

3.15.1 Stack

The stack in an 8085 microcomputer system can be described as a set of memory locations in the R/W memory, specified by a programmer in a main program. These memory locations are used to store binary information (bytes) temporarily during the execution of a program. The beginning of the stack is defined in the program by using the instruction LXI SP which loads a 16-bit memory address in the stack pointer register of the microprocessor. Once the stack-location is defined, storing of data bytes begins at the memory address that is one less than the address in the stack pointer register. For example, if the stack pointer register is loaded with the memory address 2099 H (LXI SP, 2099 H), the storing of data bytes begins at 2098 H and continues in reversed numerical order (decreasing memory addresses such as 2098 H, 2097 H, etc.). Therefore, as a general practice, the stack is initialized at the highest available memory location to prevent the program from being destroyed by the stack information. The size of the stack is limited only by the available memory. Data bytes in the register pairs of the microprocessor can be stored on the stack (two at a time) in reverse order (decreasing memory address) by using the instruction PUSH. Data bytes can be transferred from the stack to respective registers by using the instruction POP. The stack pointer register tracks the storage and retrieval of the information. Because two data bytes are being stored at a time, the 16-bit memory address in the stack pointer register is decremented by two; when data bytes are retrieved; the address is incremented by two. An address in the stack pointer register indicates that the next two memory locations (in descending numerical order) can be used for storage.

Opcode	**Operand**	**Comments**
LXI SP	16-bit address	Load the stack pointer register with a 16-bit address.
PUSH	Rp	Store register pair on stack, this is a 1 byte instruction.
PUSH	B/D/H	The operands B, D, H represent register BC, DE, and HL respectively. It copies the contents of the specified register pair on the stack. The stack pointer register is decremented, and the contents high order register (e.g. register B) are copied in the location. The stack pointer register is again decremented, and the contents low order register (e.g. register C) are copied in that location.
PUSH	PSW	The operand PSW represents Program Status Word, meaning the contents of the accumulator and the flags.
POP	Rp	Retrieve register pair from stack
POP	B/D/H	It copies the contents of the top two memory locations of the stack into the specified register pair. The contents are copied from memory location indicated by stack pointer to the low order register (e.g. register L) and pointer will be incremented by 1. The contents are again copied from the next memory location into the high order register (e.g. register H) and stack pointer will be again incremented by 1.

Example: In the following set of instructions, the stack pointer is initialized, and the contents of register pair HL are stored on the stack by using the PUSH instruction. Register pair HL is used for the delay counter. Actual instructions are not at the end of the delay counter, the contents of H-L are retrieved by using the instruction POP. Assuming the available user memory ranges from 2000 H to 20 FFH, illustrate the contents of various registers when PUSH and POP instructions are executed.

Solution: In this example, the first instruction LXI SP, 2099 H loads the stack pointer register with the address 2099 H. This instruction indicates to the microprocessor that memory space is reserved in the R/W memory as the stack and that the locations beginning at 2098 H and moving upward can be used for temporary storage. This instruction also suggests that the stack can be initialized anywhere in the memory; however, the stack location should not interfere with a program. The next instruction LXI H loads data in the HL register pair. When instruction

PUSH H is executed, the following sequence of data transfer takes place. After the execution, the contents of the stack and the register are as shown.

1. The stack pointer is decremented by one to 2098 H, and the contents of the H register are copied to memory location 2098 H.
2. The stack pointer register is again decremented by one to 2097 H, and the contents of the L register are copied to memory location 2097 H.
3. The contents of the register pair HL are not destroyed; however, HL is made available for the delay counter.

After the delay counter, the instruction POP H restores the original contents of the register pair HL, as follows. The contents of the stack and the registers following the POP instruction are:

1. The contents of the top of the stack location shown by the stack pointer are copied in the L register, and the stack pointer register is incremented by one to 2098 H.
2. The contents of the top of the stack (now it is 2098 H) are copied in the H register, and the stack pointer is incremented by one.
3. The contents of memory locations 2097 H and 2098 H are not destroyed until some other data bytes are stored in these locations.

3.15.2 Subroutine

A subroutine is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program. For example, if a time delay is required between three successive events; three delays can be written in the main program. To avoid repetition of the same delay instructions, the subroutine technique is used. Delay instructions are written once, separately from the main program, and are called by the main program when needed. The 8085 microprocessor has two instructions to implement subroutines: CALL (call a subroutine), and RET (return to main program from a subroutine). The CALL instruction is used in the main program to call a subroutine, and the RET instruction is used at the end of the subroutine to return to the main program. When a subroutine is called, the contents of the program counter, which is the address of the instruction following the CALL instruction, is stored on the stack and the program execution is transferred to the subroutine address. When the RET instruction is executed at the end of the subroutine, the memory address stored on the stack is retrieved, and the sequence of execution is resumed in the main program.

3.15.2.1 Subroutine Nesting and Processor Stack

A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost. Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the processor stack. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC. Figure 3.17 shows the Subroutine call with stack memory contents.

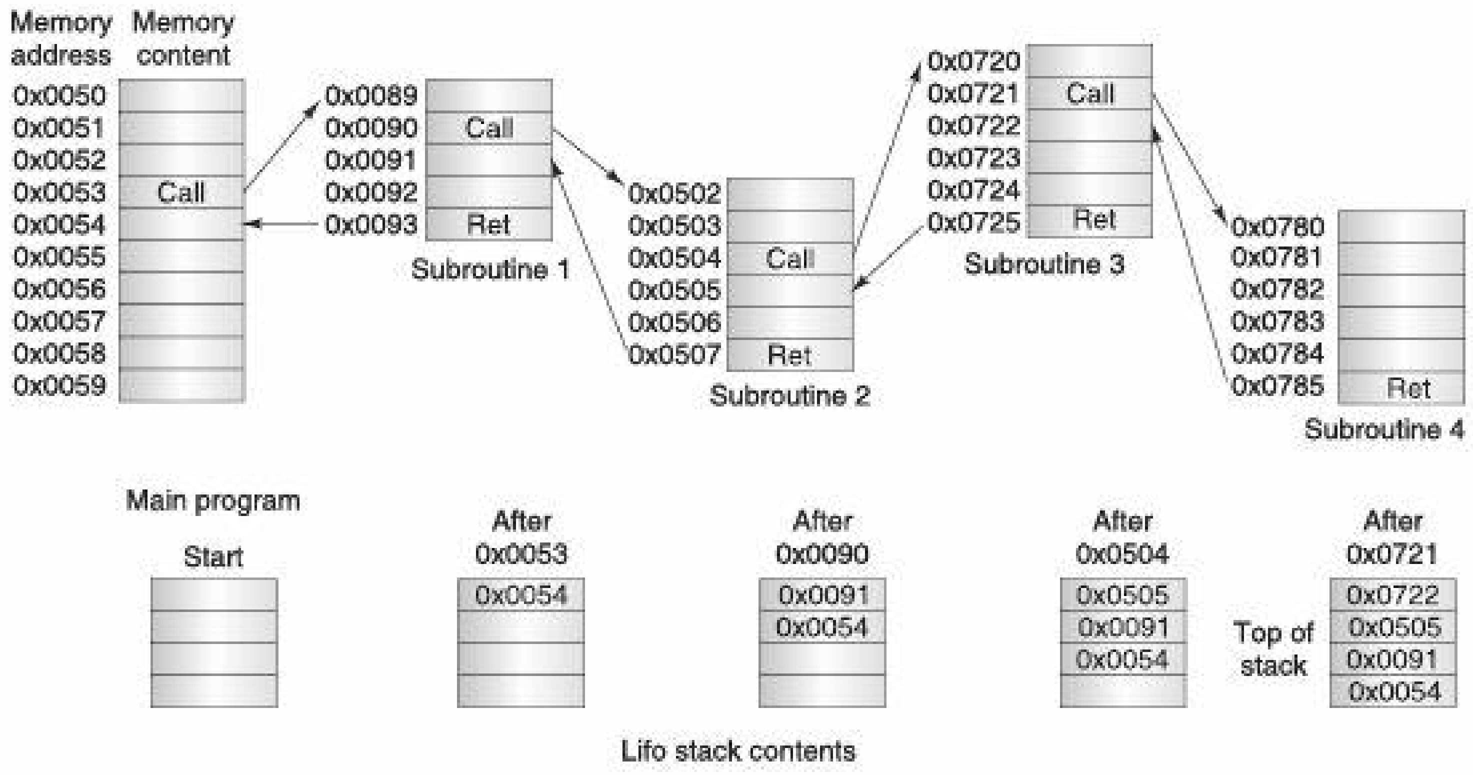


Fig. 3.17 Subroutine Call with Stack Memory Contents.

Example: Let us suppose we have a hexadecimal number and we want to convert it into its equivalent ASCII number using HEX-to-ASCII subroutine.

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	3A, 08, 20		LDA	8200	; Get Hexa Data
2003	47		MOV	B, A	
2004	E6, 0F		ANI	0F	; Mask Upper Nibble
2006	CD, 1A, 20		CALL	SUB1	; Get the ASCII code
2009	32, 09, 20		STA	8201	
200C	78		MOV	A, B	
200D	E6		ANI	F0	; Mask Lower Nibble
200F	07		RLC		
2010	07		RLC		
2011	07		RLC		
2012	07		RLC		
2013			CALL	SUB1	;Get ASCII code for lower nibble
2016	32, 0A, 20		STA	8202	
2019	76		HLT		

HEX-to-ASCII Subroutine					
201A	FE, 0A	SUB1:	CPI	0A	;compare the number with 10
201C	DA, 21, 00		JC	SKIP	
201F	C6, 07		ADI	07H	;add 07 to number
2021	C6, 1E	SKIP:	ADI	30H	;add 30H to number
2023	C9		RET		;return to main program

Explanation: The main make use of a subroutine program which is used twice in the main program. The subroutine program can be directly called by its starting address or it may be called by any label assigned to it. Once the PC reaches to CALL it comes to know that, the next execution will start from new memory location (subroutine program). The address of subroutine is already given in operand in the form of label. Before going to subroutine the next memory address is stored on the stack so that execution in the main program can be resumed from the memory location where it is left. The last instruction of subroutine program is RET instruction which restore the content of PC from the stack memory.

A subroutine can be called many times as needed. In this example it is called twice. This results in smaller program code. Another advantage of using subroutine program is that if any changes are required in the subroutine, it can be easily done. Because the value has to be change at one place only, no need to change at each and every place which happens in case programs without subroutine.

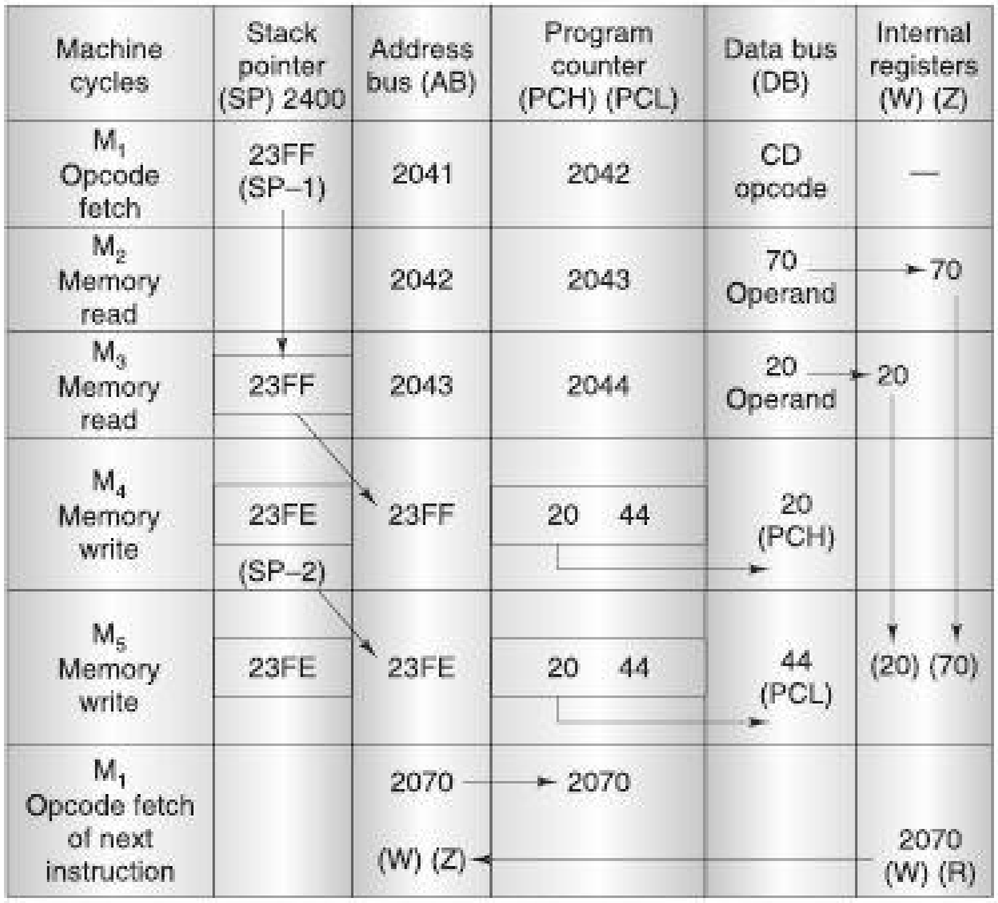
3.15.2.2 CALL Execution

Memory Address	**Mnemonics**	**Machine Code**	**Comments**	**CALL Address**
2041	CALL 2070 H	CD	Call subroutine located at the memory location 0x2070 H	2070 H
2042		70		
2043		20		
2044	Instruction	NEXT		

Figure 3.19 given below show that CALL instruction requires 5 machine cycles. The explanation of every event in each machine cycle is as follows:

In first machine cycle (M1), the contents of program counter (2041H) are placed on the address bus and instruction code CD is fetched using the data bus. Then PC is increment to next memory address 2042 H. After the instruction is decoded and executed, the stack pointer is decremented by one to 23FF H.

In M2 and M3, memory read operation is done. In this 16-bit address (2070 H) of CALL instruction is fetched. Then it is loaded in temporary register. The low-order address 70H is fetched first and placed in Z register. Then higher-order address 20H is fetched and stored in W register



Machine cycles	Stack pointer (SP) 2400	Address bus (AB)	Program counter (PCH) (PCL)	Data bus (DB)	Internal registers (W) (Z)
M_1 Opcode fetch	23FF (SP–1)	2041	2042	CD opcode	—
M_2 Memory read		2042	2043	70 Operand	70
M_3 Memory read	23FF	2043	2044	20 Operand	20
M_4 Memory write	23FE (SP–2)	23FF	20 44	20 (PCH)	
M_5 Memory write	23FE	23FE	20 44	44 (PCL)	(20) (70)
M_1 Opcode fetch of next instruction		2070 (W) (Z)	2070		2070 (W) (R)

Fig. 3.18 Operation Performed in Call Execution.

In machine cycle M4 and M5, storing of Program counter is done. In beginning of M4 cycle, the contents of the Program counter on address bus is suspended; instead the contents of SP register (23FF H) are placed on address bus. The higher-order byte of Program counter (PCH = 20 H) is placed on data bus and stored in stack location 23FF H. At the same time the stack pointer is decrement to 23FE H. In M5 the content of stack pointer 23FE H are placed on address bus. The lower-order byte of program counter (PCL = 44H) is placed on data bus and stored in stack location 23FE H.

After CALL instruction the next instruction cycle starts. In this program execution sequence is transferred to CALL location 2070 H by placing contents of W and Z register (2070 H) on address bus. Then the program counter is incremented to location 2071 H. The timing diagram of CALL instruction is shown in Figure 3.19.

3.15.2.3 RET Execution

At the end of the subroutine, when the instruction RET is executed, the program execution sequence is transferred to the memory location 2044 H. The address 2044 H was stored in the top two locations of the stack (23FEH and 23FFH) during the CALL instruction. Figure 3.20 shows the sequence of events that occurs as the instruction RET is executed. Example is given below.

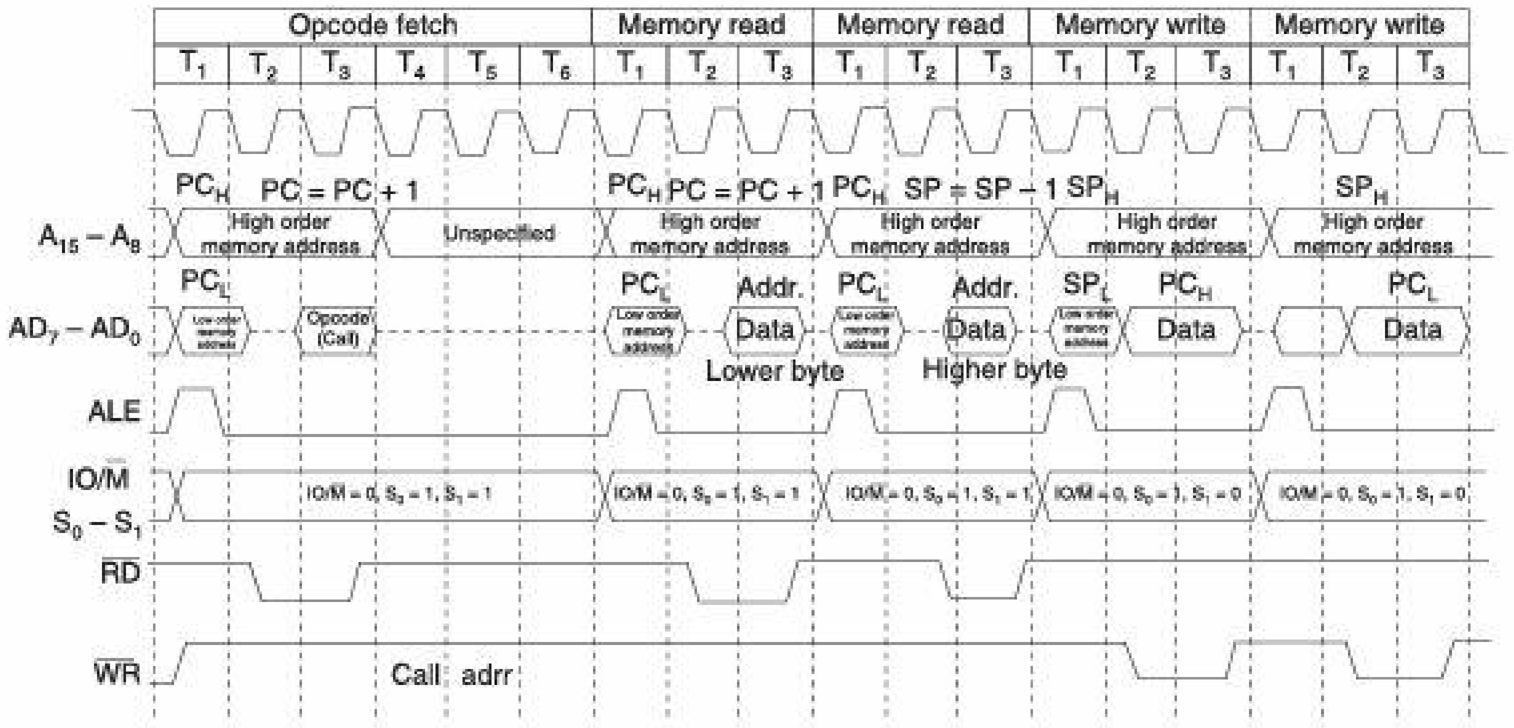


Fig. 3.19 Timing Diagram of CALL Instruction.

Main program

Memory Address	Mnemonics	Machine Code	Comments	CALL Address
2030	Main program starts			
2040	CALL 2070 H	CD	Call subroutine located at the memory location 0x2070 H	2070 H
2041		70		
2042		20		
2043	Instruction	NEXT		

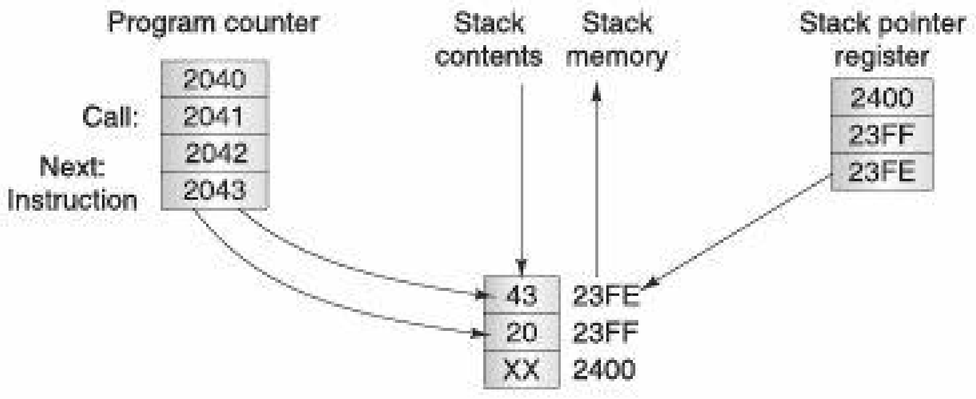
Subroutine

Memory Address	Mnemonics	Machine Code	Comments	Return Address
2070	Subroutine starts			
2071				
2072				
207F	RET	C9	Returns back to main program	2043

In first machine cycle, opcode C9 is fetched.

- In M2, the memory read cycle takes place and contents of top of stack 43H placed at stack memory location 23FE are stored in temporary register Z.
- In M3, the memory read cycle takes place and contents of next top of stack 20H placed at stack memory location 23FF are stored in temporary register W.

- The next machine cycle is opcode fetch of next instruction where the contents of temporary registers (W and Z registers) are placed on address bus 2043H and program returns from the subroutine to the main program..



Machine cycles	Stack pointer (23FE)	Address bus (AB)	Program counter	Data bus (DB)	Internal registers (W) (Z)
M_1 Opcode fetch	23FE	207F	2080	C9 Opcode	
M_2 Memory read	23FF	23FE		43 (Stack)	43
M_3 Memory read	2400	23FF		20 (Stack-1)	20
M_1 Opcode fetch of next instruction		2043 (W) (Z)	2044		2043 (W) (Z)

Fig. 3.20 Call and RET Execution.

3.15.2.4 *Restart, Conditional Call, and Return Instructions*

In addition to the unconditional CALL and RET instructions, the 8085 instruction set includes eight Restart instructions and eight conditional Call and Return instructions.

(a) Restart (RST) Instructions: RST instructions are 1-byte Call instructions that transfer the program execution to a specific location on page 00H. They are executed the same way as Call instructions. When an RST instruction is executed, the 8085 stores the contents of the program counter (the address of the next instruction) on the top of the stack and transfer the program to the Restart location. These instructions are listed here to emphasize that they are Call instructions and not necessarily always associated with the interrupts. The list of eight RST instructions is as follows:

RST 0	Call 0000 H	RST 4	Call 0020 H
RST 1	Call 0008 H	RST 5	Call 0028 H
RST 2	Call 0010 H	RST 6	Call 0030 H
RST 3	Call 0018 H	RST 7	Call 0038 H

(b) Conditional Call and Return Instructions: The conditional Call and Return instructions are based on four data conditions (flags): Carry, Zero, Sign, and Parity. The conditions are tested by checking the respective flags. In case of a conditional Call instruction, the program is transferred to the subroutine if the condition is met; otherwise, the main program is continued. In case of a conditional Return instruction, the sequence returns to the main program if the condition is met; otherwise, the sequence in the subroutine is continued. If the Call instruction in the main program is conditional, the Return instruction in the subroutine can be conditional or unconditional. The conditional Call and Return instructions are listed for reference.

(c) Conditional Call

CC-Call subroutine if Carry flag is set (CY = 1)

CNC-Call subroutine if Carry flag is reset (CY = 0)

CZ-Call subroutine if Zero flag is set (Z = 1)

CNZ-Call subroutine if Zero flag is reset (Z = 0)

CM-Call subroutine if Sign flag is set (S = 1, negative number)

CP-Call subroutine if Sign flag is reset (S = 0, positive number)

CPE-Call subroutine if Parity flag is set (P = 1, even parity)

CPO-Call subroutine if Parity flag is reset (P = 0, odd parity)

3.16 INTERRUPTS IN 8085

Interrupts can be seen as a number of functions. These functions make the programming much easier; it will transfer the program execution flow to new memory location based on the type of interrupt. A microprocessor is interrupted by either an external hardware device or software. The interrupt I/O is a data transfer an external device or a peripheral can inform the processor that it is ready for communication and it request attention. Before this polling method was used by processor for I/O data transfer, it would check each I/O device periodically whether they had data for transfer or not. This would made most of time processor busy in checking. This drawback is removed by interrupt, here when the device has data for transfer, it interrupts the processor.

Various interrupts are classified below.

3.16.1 Hardware and Software Interrupts

The interruption caused by I/O devices is called hardware interrupt. The abnormal internal conditions or special instructions can also interrupt the normal operation of microprocessors. Such an interrupt is called a software interrupt. RST n instructions of the 8085 are used for software interrupt. When RST n instruction is inserted in a program, the program is executed up to the point where RST n has been inserted. This is used in debugging of a program. The internal abnormal or unusual conditions which prevent the normal processing sequence of a microprocessor are also called exceptions. For example, divide by zero will cause an exception. Intel literatures do not use the term exception, whereas Motorola literatures use the term exception. Intel includes exception in software interrupt. When several I/O devices are connected to INTR interrupt line, an external hardware is used to interface I/O devices.

3.16.2 On the Basis of Call Location

When an interrupt occurs the program is transferred to a specific memory location, then the monitor transfers the program from the specific memory location to a memory location in RAM, from where the user can write the program for interrupt service subroutine (ISS).

(a) **Vectored interrupt:** The address of the subroutine is already known to the Microprocessor or the address of the service routine is hard-wired.

(b) **Non-vectored interrupt:** The device will have to supply the address of the subroutine to the Microprocessor or the address of the service routine needs to be supplied externally by the device.

3.16.3 On the Basis of Enabling and Disabling the Interrupt

(a) **Maskable:** These interrupts can be cleared or rejected.

(b) **Non-maskable:** These interrupts cannot be cleared or rejected.

The Intel 8085 has five interrupt inputs, namely, TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. The TRAP has the highest priority, followed by RST 7.5, RST 6.5 and RST 5.5 and the INTR has the lowest priority. When interrupts are to be used they are enabled by the software using the instruction EI (Enable Interrupt) in the main program. Figure 3.21 shows the schematic diagram of interrupts of Intel 8085. The instruction EI sets the interrupt enable flip-flop to enable the interrupts. The use of the instruction EI enables all the interrupts. The instruction DI (Disable Interrupt) is used to disable interrupts. In certain situations it may be desired to prevent the occurrence of interrupts while a particular task is being performed by the microprocessor. This can be done using DI instruction. The DI instruction resets the interrupt enable flip-flop and disables all the interrupts except non-maskable interrupt TRAP. The system RESET also resets the Interrupt Enable flip-flop. When an interrupt line goes high the processor completes its current instruction and saves program counter on the stack. It also resets Interrupt Enable flip-flop before taking up ISS so that the occurrence of further interrupts by other devices is prevented during the execution of ISS. All the interrupts except TRAP are disabled by resetting the Interrupt Enable flip-flop.

Interrupt name	Maskable	Vectored
INTR	Yes	No
RST 5.5	Yes	Yes
RST 6.5	Yes	Yes
RST 7.5	Yes	Yes
TRAP	No	Yes

3.16.4 The 8085 Interrupt Process

1. The interrupt process should be enabled using the EI instruction.
2. The 8085 checks for an interrupt during the execution of every instruction.
3. If INTR is high, MP completes current instruction, disables the interrupt and sends INTA (Interrupt acknowledge) signal to the device that interrupted.

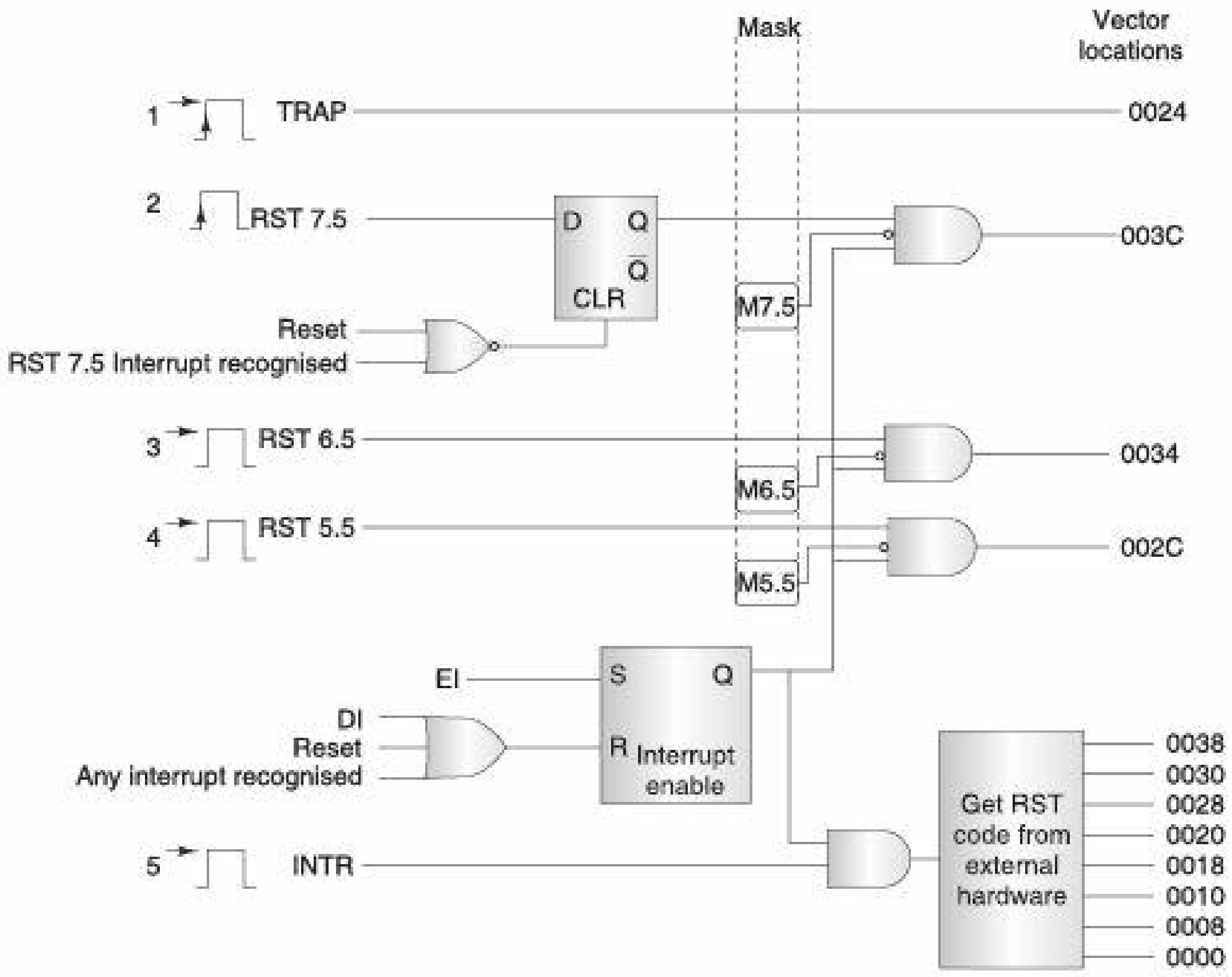


Fig. 3.21 Schematic Diagram of 8085 Interrupts.

4. INTA allows the I/O device to send a RST instruction through data bus.
5. Upon receiving the INTA signal, MP saves the memory location of the next instruction on the stack and the program is transferred to 'call' location (ISR Call) specified by the RST instruction.
6. Microprocessor performs the ISR.
7. ISR must include the 'EI' instruction to enable the further interrupt within the program.
8. RET instruction at the end of the ISR allows the MP to retrieve the return address from the stack and the program is transferred back to where the program was interrupted.

There are three ways to RESET the interrupt flip-flop: by the software using instruction DI, system reset and by recognition of an interrupt request. Before the program returns back from ISS to the main program all the interrupts are to be enabled again. This is done using instruction EI in ISS before using the instruction RET. At many occasions the programmer may like to prevent the occurrence of a few of several interrupts, while the microprocessor is performing certain tasks. This is done by masking off those interrupts which are not required to occur when certain task is being performed. The interrupts which can be masked off (i.e., made ineffective) are called maskable interrupts. Masking is done by software.

The Intel 8085 has two categories of interrupts: maskable and non-maskable. The TRAP is a non-maskable interrupt. It need not be enabled. It cannot be disabled. It is not accessible to the user. It is used for emergency situation such as power failure and energy shut-off. RST 7.5, RST 6.5 and RST 5.5 are maskable interrupts.

3.16.4.1 Interrupts Call-Locations

Vector of RST n = hexadecimal equivalent of(8 * n)

For example RST 7.5 = hexadecimal equivalent of(8 * 7.5 = 60) = 0X03C

n = 8.5 for trap

For TRAP, RST 7.5, 6.5 and 5.5 the program is automatically transferred to specific memory locations without any external hardware. The necessary hardware is already provided within 8085. The specific memory locations for these interrupts are as follows (Table 3.5):

Table 3.5

Interrupt	*Call-Location in Hex*
TRAP	0X024
RST 7.5	0X03C
RST 6.5	0X034
RST 5.5	0X02C

The external hardware circuit generates RST n codes to implement the multiple interrupt scheme. The various call locations for various RST-n instructions are shown in Table 3.6 below.

Table 3.6 Call location and hex code for RST n

RST n	*Hex Code*	*Call Location*
RST 0	C7	0000
RST 1	CF	0008
RST 2	D7	0010
RST 3	DF	0018
RST 4	E7	0020
RST 5	EF	0028
RST 6	F7	0030
RST 6	FF	0038

The microprocessor samples the INTR line in the last state of the last machine cycle of each instruction. When INTR is high, the microprocessor saves the contents of the program counter on the stack and then sends an interrupt acknowledge signal, INTA to the external hardware. In response to INTA the external hardware generates a RST n code. When microprocessor receives this code, it transfers program to the corresponding CALL-location. Up to 8 number of I/O devices can be connected to INTR through an external hardware. Priority can be assigned to I/O devices connected to INTR through external hardware or interrupt controller. The external hardware recognizes which I/O device has interrupted and it generates proper RST code that causes the microprocessor to take up ISS for that particular I/O device. An external hardware can be built employing priority encoder and a tri-state buffer, Alternatively, Priority Interrupt Controller 8214 can be used. Programmable interrupt controller 8259 is more versatile, and the present trend is to use programmable interrupt controller. INTA is used to activate 8259 or some other hardware. The 8259 has been described later in this book.

3.16.5 SIM Instruction

RST 7.5, RST 6.5 and RST 5.5 are maskable interrupts. These interrupts are enabled by software using instructions EI and SIM (Set Interrupt Mask). The 'EI' instruction is a one byte instruction and is used to enable the non-maskable interrupts. SIM Instruction: It is the abbreviation of Set Interrupt Mask. It is used to mask the hardware interrupts

It is a multipurpose interrupt used to implement the 8085 interrupts (RST 7.5, 6.5, 5.5) and serial data output. Firstly it masks the interrupts (bit 2, 1, 0), then reset RST 7.5 (bit 4) and implement serial O/P (bit 7, 6). If bit 6 = 1 is used to enable serial I/O and bit 7 is used to transmit serial output data bit.

The execution of the instruction SIM enables/disables interrupts according to the bit pattern of the accumulator. Figure 3.22 (a) shows accumulator contents for SIM instruction. Bits 0 -2 set/rest the mark bits of interrupt mask register for RST 5.5, 6.7 and 7.5. Bit 0 is for RST 5.5 mask, bit 1 for RST 6.5 mask and bit 2 for RST 7.5 mask. If a bit is set to 1 the corresponding interrupt is masked off (disabled).

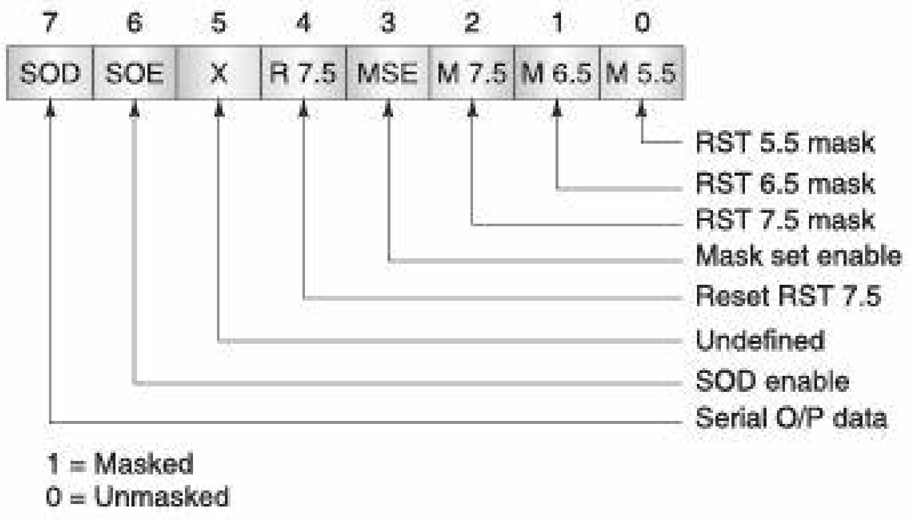


Fig. 3.22(a) Accumulator Content for SIM.

If it is set to 0, the corresponding interrupt is enabled. Bit 3 is set to 1 to make bits 0 -2 effective. Bit 4 is an additional control for RST 7.5. If it is set to 1, the flip-flop for RST 7.5 is reset. Thus RST 7.5 is disabled regardless of whether bit 2 for RST 7.5 is 0 or 1 Bits 6 and 7 are for serial data output. The instruction SIM is also used for serial data transmission. Bit 6 is to enable SOD. Serial Output Data (SOD) is single wire I/O connection usable with the SIM instructions. It can be used for serial data communications. If SIM instruction is executed the content of the 7th bit of the accumulator is output on SOD line of the microprocessor. The content of bit 7 may be either high (1) or low (0). The instruction DI disables all the interrupts.

This is not always desired. Occurrence of several interrupts is prevented when particular task is performed by processors. In such a situation the interrupts which are not desired to occur may be masked off using SIM instruction.

3.16.6 RIM Instruction

This is the abbreviation of Read Interrupt Mask. It checks weather interrupt is mask or not. RIM is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and to read serial input data bit. When RIM instruction is executed an 8-bit data is loaded in accumulator, which can be interpreted as shown in Figure 3.22(b). It reads the interrupt mask (bit 2, 1, 0) firstly. Then it identifies the pending interrupts (bit 6, 5, 4) and finally receives serial input data bit (bit 7). The Serial Input Data (SID) is also single wire I/O connections usable with the RIM instructions. Also, it can be used for serial data communication.

D7	D6	D5	D4	D3	D2	D1	D0
SID	**I7.5**	**I 6.5**	**I 5.5**	**IE**	**M7.5**	**M6.5**	**M5.5**

SID-	Serial Input Data
I 7.5-	Interrupt pending status of RST 7.5
I 6.5-	Interrupt pending status of RST 6.5
I 5.5	Interrupt pending status of RST 5.5
	if, 1= Interrupts are enable 0 = Interrupt are disable
M 7.5-	Mask status of RST 5.5
M 6.5-	Mask status of RST 6.5
M 5.5-	Mask status of RST 7.5

Triggering Levels: TRAP is edge as well as level triggered. This means that TRAP should go high and stay high until it is acknowledged. In this way false triggering caused by noise and transients is avoided. The flip-flop is cleared when interrupt is acknowledged so that future interrupt may be entertained. RST 7.5 is positive edge triggered interrupt. It can be triggered with a short duration pulse. RST 6.5 and 5.5 are level triggered interrupts. Triggering level for RST 6.5 and 5.5 has to be kept high until the microprocessor recognizes these interrupts. If the processor does not recognize these interrupts immediately, their triggering level should be held by external hardware.

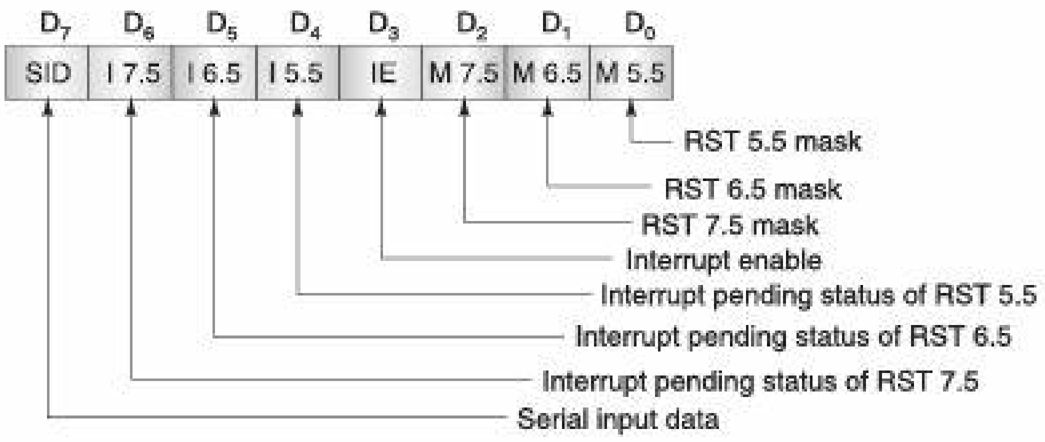


Fig. 3.22(b) Accumulator Content for RIM.

Interrupt Name	Maskable	Masking Method	Vectored	Memory	Triggering Method
INTR	Yes	DI / EI	No	No	Level Sensitive
RST 5.5 / RST 6.5	Yes	DI / EI , SIM	Yes	No	Level Sensitive
RST 7.5	Yes	DI / EI , SIM	Yes	Yes	Edge Sensitive
TRAP	No	None	Yes	No	Level & Edge Sensitive

Examples

1. Write a program to enable interrupt 5.5 and mask other interrupts.

Program:
```
MVI A, 0EH
SIM
```

Detail: The first instruction makes bit D3 = 1 and D0 = 0 of the accumulator. Second instruction SIM enables the interrupt 5.5.

A TTY receiver line is connected to the SOD pin of the 8085. Write a program to disable all the interrupts and send Stop bit (logic 1) to TTY without affecting interrupts mask.

Program:
```
MVI A, C0H
SIM
```

Detail: The first instruction makes D7 and D6 bits of accumulator high. (At logic 1). The D7 bit is the STOP bit and D6 is the Serial data enable bit of SIM. D3 is = 0 means it doesn't affect the interrupts.

The microprocessor is completing an RST 5.5 interrupts request. You have to check whether the RST 7.5 is pending or not. If it is pending, enable the RST 7.5 without affecting any other interrupts; otherwise, return to the main program.

Program:
```
RIM
MOV B, A
ANI 40H
JNZ NEXT
```

```
EI
RET
NEXT: MOV A, B
ANI 0BH
ORI 08H
SIM
JMP SERV
```

Detail: The instruction RIM checks for a pending interrupt. Instruction ANI 40h masks all the bits except D6 to check pending RST 6.5 interrupt. If D6 = 0, the program control is transferred to main program. D6 = 1 indicates that RST 7.5 is pending. Instruction ANI 0BH sets D1 = 0 (RST 7.5 bit for SIM), instruction ORI sets D3 = 1(this is necessary for SIM to be effective) and instruction SIM enable RST 7.5 without affecting any other interrupts. The JMP instruction transfer the program to the service routine (SERV) written for RST 7.5.

3.17 SERIAL COMMUNICATION IN 8085

8085 Microprocessor has two Serial Input /Output pins that are used to read/write one bit data to and from peripheral devices.

3.17.1 SID (Serial Input Data) Line

There is one bit input line inside the 8085 CPU (Pin number 5). This SID pin is used to read and store one bit data from external peripherals. The data that is read is stored in the A7th bit of the Accumulator. RIM instruction is used to read the SID line.

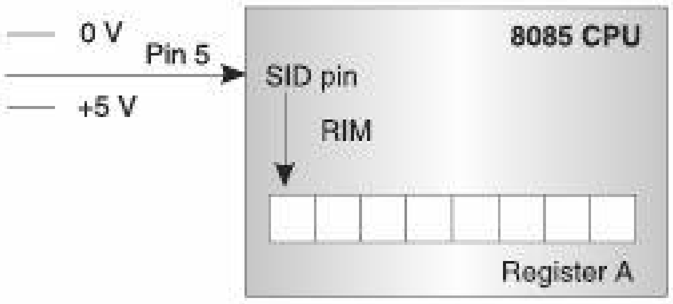


Fig. 3.23(a) Read and Store from SID Pin.

Example Pseudo code:

```
RIM
A7 (SID)
```

As we can see from Fig. 3.23(a), if the SID line is connected with +5V and RIM instruction is executed, then the Accumulator's MSB bit will be loaded with a Logic 1; if the SID line is connected with 0V (GND) and RIM instruction is executed, then the Accumulator's MSB bit will be loaded with a Logic 0

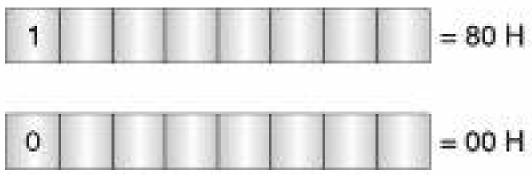


Fig. 3.23(b) Accumulator Content After Execution of RIM Instruction.

3.17.2 SOD (Serial Output Data) Line

This is one bit output port in the 8085 CPU (Pin number 4). This pin is used to write one bit data to external peripheral. To write data into this port, SIM instruction is used. The data that is to be written in this port must be stored in the A7th bit of the accumulator. Bit A6 of the Accumulator is known as SOE (Serial Output Enable). This bit must be set to 1 to enable Serial data output.

Example Pseudo code: To write a logic 1 in this SOD line, Load the accumulator with C0H.

```
A = C0H
SIM
SOD = (A7)
```

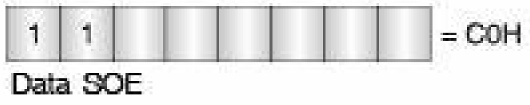


Fig. 3.23(c) Accumulator Content after the RIM Instruction is Executed Pseudo Code.

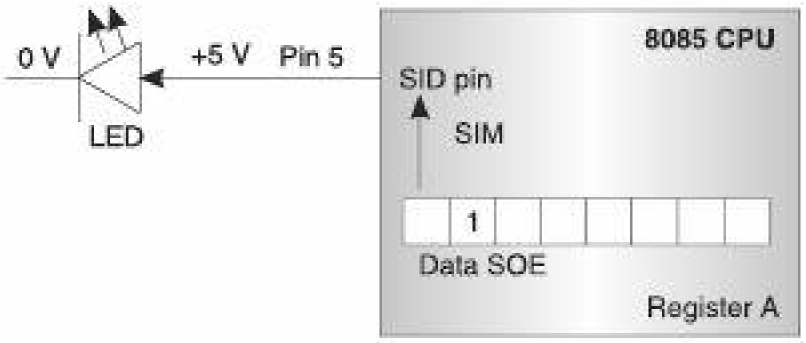


Fig. 3.23(d) Writing on Peripheral using SOD Pin of 8085.

3.18 DIRECT MEMORY ACCESS (DMA) WITH 8085

DMA stands for Direct Memory Access. The 8085 consists of 2 pins, namely, HOLD and HLDA. The former of which indicates the processor that either a peripheral or any IO device, is requesting the processor to hold its current activities and give the control of buses to IO devices. HLDA is the acknowledgement from the microprocessor to the concerned IO device sending the request.

During any given bus cycle, one of the system components connected to the system bus is given control of the bus. This component is said to be the master during that cycle and the component it is communicating with is said to be the slave. The CPU with its bus control logic is normally the master, but other specially designed components can gain control of the bus by sending a bus request to the CPU. After the current bus cycle is completed the CPU will return a bus grant signal and the component sending the request will become the master.

Taking control of the bus for a bus cycle is called cycle stealing. Just like the bus control logic, a master must be capable of placing addresses on the address bus and directing the bus activity during a bus cycle. The components capable of becoming masters are processors (and their bus control logic) and DMA controllers. Sometimes a DMA controller is associated with a single interface, but they are often designed to accommodate more than one interface.

3.18.1 DMA and 8085

The 8085 microprocessor receives bus requests through its HOLD pin and issues grants from the hold acknowledge (HLDA) pin. A request is made when a potential master sends a 1 to the HOLD pin. Normally, after the current bus cycle is complete the 8085 will respond by putting a 1 on the HLDA pin. When the requesting device receives this grant signal it becomes the master. It will remain master until it drops the signal to the HOLD pin, at which time the 8085 will drop the grant on the HLDA pin. One exception to the normal sequence is that if a word, which begins at an odd address, is being accessed, then two bus cycles are required to complete the transfer and a grant will not be issued until after the second bus cycle.

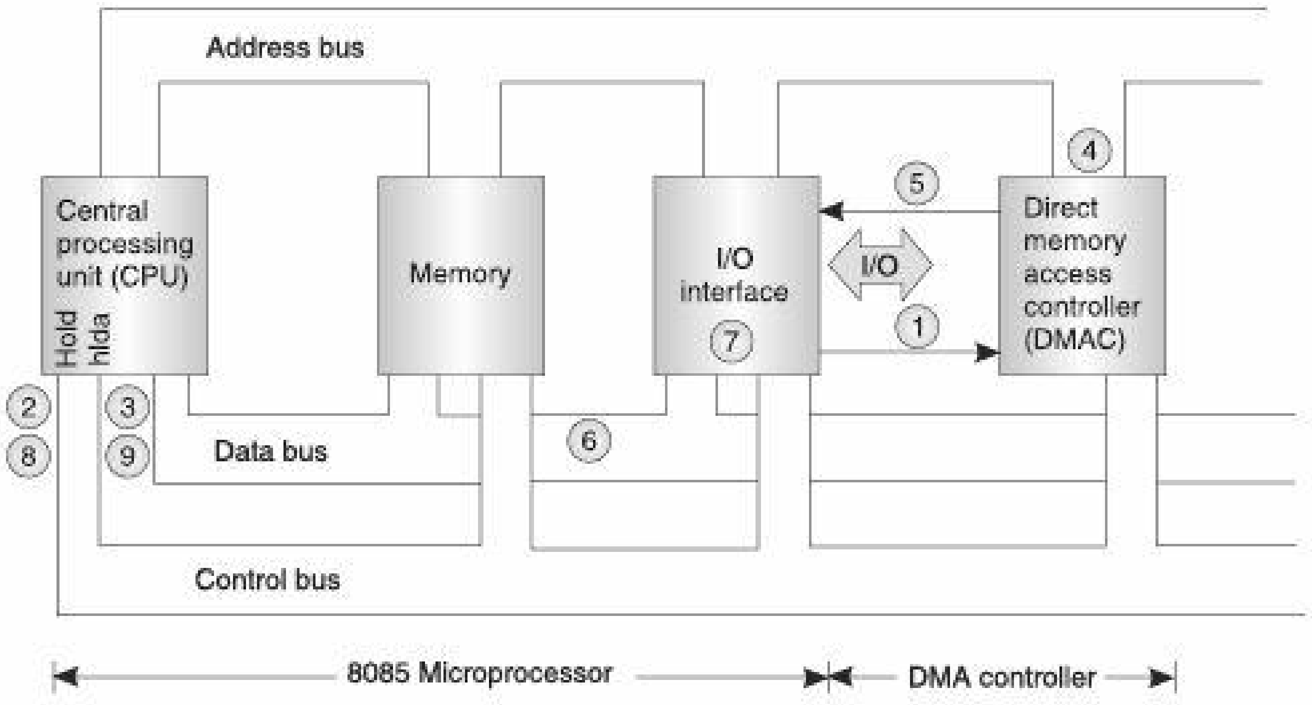


Fig. 3.24 DMA and 8085 Interfacing.

When a DMA controller becomes the master it places an address on the address bus and sends the interface the necessary signals to cause it to put data on, or receive data from, the data bus. Since the DMA controller determines when the bus request is dropped, it can return control to the CPU after each data byte is transferred and then request control again when the next data byte is ready, or it can retain control until the entire block is moved. The former is the usual case because this allows the CPU to continue its work until the next data byte is available.

3.18.2 DMA Block Transfer

During a block input byte transfer, the following sequence occurs as the data byte is sent from the interface to the memory:

1. The interface sends the DMA controller a request for DMA service.
2. A Bus request is made to the HOLD pin (active High) on the 8086 microprocessor and the controller gains control of the bus.
3. A Bus grant is returned to the DMA controller from the Hold Acknowledge (HLDA) pin (active High) on the 8086 microprocessor.
4. The DMA controller places contents of the address register onto the address bus.
5. The controller sends the interface a DMA acknowledgment, which tells the interface to put data on the data bus. (For an output it signals the interface to latch the next data placed on the bus.)
6. The data byte is transferred to the memory location indicated by the address bus.
7. The interface latches the data.
8. The Bus request is dropped, the HOLD pin goes Low, and the controller relinquishes the bus.
9. The Bus grant from the 8086 microprocessor is dropped and the HLDA pin goes Low.
10. The address register is incremented by 1.
11. The byte count is decremented by 1.
12. If the byte count is non-zero, return to step 1, otherwise stop.

3.19 TIME DELAY GENERATION

There are times when we are required to generate time delays. This time delay is required in various cases, for example, to synchronize the 8085 with slow peripheral, to implement time delay, etc. The time delays can be generated by 8085. Execution of each instruction needs some clock cycles. Thus, to generate the required time delay, a microprocessor will have to execute number of instruction. This is the principle of time delay generation.

3.19.1 NOP Instruction

The simplest way to generate a time delay is to use NOP instruction. Each NOP instruction uses four clocks for fetching, decoding and executing. Let us assume that 8085 is working with a crystal frequency of 6 MHz, and as such with an internal frequency of 3 MHz. Thus, each clock period is one-third of a microsecond. In such a case, a NOP instruction needs only 1.33 us for execution. Even if we use 64K NOP instruction, which is a maximum possible in memory, the time delay would only be 64K x 1.33 us = about 8.69 ms. Also the program size has become too much, for too little work done. Thus, we use a series of NOP instructions only when our interest is to generate small time delays of few microseconds.

3.19.2 Register Counter for Time Delay Generation

A much better way to generate a larger rime delay with a smaller program size, is to execute a series of instructions repetitively in a loop. An example is shown below.

```
MVI D, FFH     ; Uses 7 T states
REP: DCR D     ; Uses 4T states
JNZ REP        ; Uses 10 T states in a case of jump and 7 T state in case of no jump
RET            ; Uses 10 T states
```

In this example register D is loaded with maximum possible value which is FFH. Then reg. D is decremented. If the result is non zero, the decrement operation is repeated. In this case the delay generated can be calculated as follows:

Total T states are

=7+[(4+10)xFF]-3+10

=14+14xFF

=3584

Now the time delay in microseconds is

T=3584x1/3= 1194.666 us

=1.195 ms

The program is quite short and generates a delay of 1.195 ms.

If we need delay smaller than this we can load the count register with a proportionately small value. IF we need larger delay than 1196, we have to use a loop inside.

3.19.3 Register Pair for Time Delay Generation

Instead of using an 8-bit register we can use a register pair as counter in a loop, to generate much larger time delays using smaller program size. This is the method used for larger delays. The calculation for register pair is as shown

```
DELAY:  LXI B, FFFFH  ; Uses 10 T states
AGAIN:  DCX B         ; Uses 6 T states
        MOV A, B      ; Uses 4 T states
        ORC C         ; Uses 4 T states
        JNZ AGAIN     ; Uses 10 T state for jump and 7 T states for no jump
        RET           ; Uses 10 T states
```

Number of T states, N = 10+[(6+4+4+10)xFFFF]-3+10

=17+24xFFFF

=1572857

Thus time delay = 1572857x1/3 us = 0.52428 s

This delay is generally taken as half-second delay. However, if we use microprocessor to generate the time delays, the microprocessor will not be available for any other operation for the entire delay time. So it restricts the time delay generation by the microprocessor to only few micro or milliseconds in practical application. For large delay time, it is better to use dedicated time chips like Intel 8253 etc.

3.20 ASSEMBLY LANGUAGE PROGRAMS

Introduction

To learn assembly language programming, the beginner should write simple programs given in this chapter and try to execute them on Intel 8085 based microprocessor kit. The memory addresses given in the program are for a particular microprocessor kit. These addresses can be changed to suit the microprocessor kit available in the laboratory. Before writing assembly language program, one should learn some important Intel 8085 instructions such as MOV, MVI, ADD, SUB, LXI, LDA, INX, INR, HLT, etc.

Simple Examples

The use of some important instructions is described below with very simple examples.

Example 1 Place 04 in register B.

Program:

Address	Machine codes	Mnemonics	Operands	Comments
FC00	06, 04	MVI	B, 04	Get 04 in register B
FC02	76	HLT		Stop

The instruction MVI B, 04 moves 04 to register B. HLT halts the program. A program is fed to the microprocessor kit in machine codes. The machine code for the instruction MVI B, 04 is 06, 04. The first byte of the instruction is 06 which is the machine code for the instruction MVI B. The second byte of the instruction, 04 is the data, 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 the above program the memory addresses from F000 to FC02 have been used. The machine code 06 is entered in the memory location FC00 H; 05 in FC01 H and 76 in FC02 H. This program can be executed on a microprocessor kit and the B can be examined. After the execution of the program the register B will contain 04. The address can be changed to suit the microprocessor kit available in the laboratory. The address and data used for a microprocessor based system are in hexadecimal system. The symbol H digit denotes that it is in hexadecimal system.

Example 2: Place 05 in register A; then move it to register B

Program:

Address	Machine codes	Mnemonics	Operands	Comments
FC00	3E, 05	MVI	A, 05	Get 05 in register A
FC02	47	MOV	B, A	Transfer 05 from A to B
FC03	76	HLT		Stop

The instruction MVI A, 05 will move 05 to register A. In the code from it is written as 3E, 05. The 1st byte of the instruction is 3E. This code is for MVI A. The second byte 05 is data which is to be placed in A. The instruction MOV B, A transfers the content of register A to register B. After the execution of the above program, register B will contain 05. The program can be executed on a microprocessor kit and register B can be examined.

Example 3: Add 49 H and 56 H.

Solution:

The 1st 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 system.

Address	Machine codes	Mnemonics	Operands	Comments
2000	21, 01, 25	LXI	H, 2501H	Get address of 1st number in H-L pair
2003	7E	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-	49H
2502-	56H

Result

2503-	9FH

2501 H is the address of memory location for the 1st number. 2501 is placed in H-L pair by the instruction LXI H, 2501 H. The next instruction is MOV A, M which 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 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.

Example 4: Aim: Write a program for subtractions of two 8-bit numbers.

Solution:

The 1st number 49 H is in the memory location 2501 H.

The 2nd number 32 H is in the memory location 2502 H.

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

Address	Machine codes	Mnemonics	Operands	Comments
2000	21, 01, 25	LXI	H, 2501 H	Get address of 1st number in H-L pair
2003	7E	MOV	A, M	1st number in accumulator
2004	23	INX	H	Content of H-L pair increases from 2501 to 2502 H
2005	96	SUB	M, A	1st number − 2nd number
2006	23	INX	H	Content of H-L pair becomes 2503 H
2007	77	MOV	M, A	Store result in 2503 H
2008	76	HLT		Stop

The 1st number is in the memory location 2501 H. 2501 is placed in H-L pair by the execution of the instruction LXI H, 2501 H. The instruction MOV A, M moves the content of the memory location addressed by H-L pair to the accumulator. Thus, the 1st number 49 H (Example 1) which is in 2501 H is placed in the accumulator. INX H increases the content of H-L pair from 2501 to 2502 H. The instruction SUB M subtracts the content of the memory location addressed by H-L pair from the accumulator. The 2nd number which is in the memory location 2502 H is subtracted from the 1st number which is in the accumulator. The result resides in the accumulator. The instruction INX H increases the content of H-L pair from 2502 to 2503 H. The instruction MOV M, A transfers the content of the accumulator to the memory location addressed by H-L pair. So the result which is in the accumulator is stored in the memory location 2503 H. The instruction HLT ends the program.

Example 5: Add 98 H and 9A H.

Solution:

SUM = 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 2504 H.

Numbers are represented in hexadecimal.

Program:

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	21, 01, 25		LXI	H, 2500 H	Get address of 1st number in H-L pair
2003	0E, 00		MVI	C, 00	MSBs of sum in register C
2005	7E		MOV	A, M	1st number in accumulator
2006	23		INX	H, 2500 H	Content of H-L pair increases from 2501 to 2502 H

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, 00	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		Stop

Example 1	**Example 2**
DATA	DATA
2501-98 H	2501-F5 H
2502-9A H	2502-8A H

Result is stored in the Memory location 2503 H

Result 1	**Result 2**
2503-32 H, LSBs of sum	2503-7F H, LSBs of sum
2504-01 H, MSBs of sum	2504-01 H, MSBs of sum

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.

The 1st instruction of the program LXI H, 2501 H gets the address of the 1st number in H-L pair. The MSBs of the sum is placed in register C. The initial value of the MSBs of the sum is kept 00. MOV A, M transfers the 1st number from the memory location 2501 H to the accumulator. INX H increases the content of H-L pair from 2501 to 2502 H. The 2nd number is in 2502 H. ADD M adds 1st and 2nd numbers. After the execution of the instruction JNC the instruction INR C is executed if the addition of two numbers produces a carry. In binary system carry is equal to one and, therefore, the content of the register C is increased from 00 to 01. The value 01 is nothing but the MSBs of the sum. Now register C contains the MSBs of the sum. The accumulator contains the LSBs of the sum. The instruction STA 2503 H places the LSBs of the sum in memory location 2503 H. The instruction MOV A, C moves the MSBs of the sum from register C to the accumulator. The instruction STA 2504 H stores the MSBs of the sum in 2504 H. After the execution of the instruction JNC the program jumps to label AHEAD and executes STA 2503 H, if the addition of two numbers does not produce a carry. The LSBs of the sum is placed in the memory location 2503 H. Register C contains 00, so the execution of the instruction MOV A, C transfers 00 in the accumulator. The instruction STA 2504 H stores 00 in the memory location 2504 H. This is the MSB of the sum. Such a situation will arise when the sum of two numbers is of 8 bits, and hence there is no carry.

Example 6: Write a program for addition of two 16-bit numbers and the result should be stored in 16-bit or more.

Solution:

1st number is in the memory locations 2501 H and 2502 H.

The 2nd number is in the memory locations 2503 H and 2504 H.

The sum is stored in the memory locations 2505 H to 2507 H.

Program:

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	2A, 01, 25		LHLD	2501 H	1st 16-bit number in H-L pair
2003	EB		XCHG		Get 1st number in D-E pair
2004	2A, 03, 25		LHLD	2503 H	2nd 16-bit number in H-L
2007	OE, 00		MVI	C, 00	MSBs of the sum in register C, Initial value = 00
2009	19		DAD	D	1st number + 2nd number
200A	D2, OE, 20		JNC	AHEAD	Is carry? No, go to the label AHEAD
200D	OC		INR	C	Yes, increment C
200E	22, 05, 25	AHEAD:	SHLD	2505 H	Store LSBs of sum in 2505 and 2506 H
2011	79		MOV	A,C	MSBs of sum in accumulator
2012	32, 07, 25		STA	2507 H	Store MSBs of the sum in 2507 H
2015	76		HLT		Stop

Example 1

2501-98 H, LSBs of 1st number

2502-5B H, MSBs of 1st number.

2503-4C H, LSBs of 2nd number.

2504-8E H, MSBs of 2nd number.

Result

2505-E4, LSBs of sum.

2306-E9, LSBs of sum.

2507-00, MSBs of sum.

Example 2

2501-45 H, LSBs of 1st number.

2502-A6 H, MSBs of 1st number.

2503-23 H, LSBs of 2nd number.

2504-9B H, MSBs of 2nd number.

Result

2505-68 H, LSBs of sum.

2506-41 H, LSBs of sum.

2507-01 H, MSBs of sum.

The instruction LHLD 2501 H transfers the content of 2501 H to register L, and the content of 2502 H to register H. Thus, the LSBs of the 1st number are placed in register L and the MSBs of the 1st number in register H. Now H-L pair contains 1st number which is a 16-bit number. The instruction XCHG exchanges the content of D-E pair with H-L pair. Therefore, the 1st number is transferred from H-L pair to D-E pair. Again the instruction LHLD 2503 H places the 2nd number in H-L pair. MVI C, 00 places the MSBs of the sum in register C, and the initial value is 00. DAD D adds the content of D-E and H-L pairs and places the sum in H-L pair. This instruction is for 16-bit addition. If there is a carry after 16-bit addition, the content of the register C is incremented. If there is no carry, the program jumps after JNC to the Table AHEAD and executes SHLD 2505 H. The LSBs of the sum is stored in the memory location 2505 and 2506. The MSBs of the sum is transferred to the accumulator from register C and then it is stored in the memory location 2507 H. In case of the carry, the MSBs of the sum is 01. In case of no carry it is 00.

Example 7: Perform the subtraction of given 8-bit numbers; -96 D 1st number-38 D 2nd number

DAA instruction cannot be used after SUB or SBB instruction for decimal subtraction. It is used, after ADD, ADC, etc. Therefore, for decimal subtraction the number which is to be subtracted is converted into 10's complement. In the above example 38 D is to be subtracted. 10's complement of 38 is first obtained and then it is added to 96 D.

$$96 - 38 = 96 + (-38)$$
$$= 96 + 10\text{'s complement of } 38.$$

98 are stored in the memory location 2501 H and 38 in 2502 H. The result is to be stored in 2503 H.

Address	Machine codes	Mnemonics	Operands	Comments
2000	21, 02, 25	LXI	H, 2502 H	Get address of 2nd number in H-L pair
2003	3E, 99	MVI	A, 99	Place 99 in accumulator
2005	96	SUB	M	9's complement of 2nd number
2006	3C	INR	A, 99	10's complement of 2nd number
2007	2B	DCX	H, 2502 H	Get address of 1st number
2008	86	ADD	M	Add 1st number and 10's complement of 2nd number
2009	27	DAA		Decimal adjustment
200A	32, 03, 25	STA	2503 H	Store result in 2503 H
200D	76	HLT		Stop

Example 1	**Example 2**	**Example 3**
DATA	DATA	DATA
2501-96	2501-99	2501-50
2502-38	2502-48	2502-10
Result	**Result**	**Result**
2503-58	2503-51	2503-40

Instruction LXI H, 2502 H places the address of the 2nd number in H-L pair. MVI A, 99 places 99 D in the accumulator. SUB M subtracts the 2nd number (placed in 2502 H) from 99D to get 9's complement of the 2nd number. INR A increments the content of the accumulator. Thus, 10's complement of the 2nd number is obtained. DCX H decrements the content of H-L pair to obtain the address of the 1st number, i.e., 2501 H. ADD M adds 10's complement of the 2nd number to the 1st number. After addition the result is in hexadecimal. The instruction DAA is used to get the result in decimal system. The result is stored in the memory location 2503 H by the instruction STA 2503 H. The instruction HLT ends the program.

Example 8: Find one's complement of 96 H.

Solution:

The number in the binary form is represented as follows:

To obtain one's complement of a number, its 0 bits are replaced by 1 and 1 by 0.

The number is placed in the memory location 2501 H.

The result is stored in the memory location 2502 H.

Program:

Address	**Machine codes**	**Mnemonics**	**Operands**	**Comments**
2000	3A, 01, 25	LDA	2501 H	Get data in accumulator
2003	2F	CMA		Take its complement
2004	32, 02, 25	STA	2502 H	Store result in 2502 H
2007		HLT		Stop

Example 1	**Example 2**
DATA	DATA
2501-96 H	2501-E4 H
Result	**Result**
2502-69 H	2502-1BH

The 8 LSBs of the number are in the memory location 2501 H. The address 2501 is placed in H-L pair. The 8 LSBs of the number are transferred from 2501 H to the accumulator. The instruction CMA takes one's complement of 8 LSBs. The 8 LSBs of the result are stored in the memory location 2503 H. The address of 8 MSBs of the number is 2502 H, and it is placed in H-L pair. The 8 MSBs of the number, we transferred from 2502 H to the accumulator. The instruction CMA takes one's complement of 8. The 8 MSBs of the result are stored in memory location 2504 H.

Example 9: Find one's complement of 5485 H.

Solution: The number in the binary form can be represented as follows:

5485	0101	0100	1000	0101
	5	4	8	5
1's complement	1010	1011	0111	1010
	A	B	7	A

The number is in the memory locations 2501 H and 2502 H.

The result is to be stored in the memory locations 2503 H and 2504 H.

Program:

Address	Machine codes	Mnemonics	Operands	Comments
2000	21, 01, 25	LXI	H, 2501 H	Address of LSBs of the number
2003	7E	MOV	A, M	8 LSBs of the number in accumulator
2004	2F	CMA		Complement of 8 LSBs of the number
2005	32, 03, 25	STA	2503 H	Store 8 LSBs of result
2008	23	INX	H	Address of 8 MSBs of the number
2009	7E	MOV	A, M	8 MSBs of the number in accumulator
200A	2F	CMA		Complement of 8 MSBs of the number
200B	32, 04, 25	STA	2504 H	Store 8 MSBs of the result
200E	76	HLT		Stop

Example 1	Example 2
DATA	DATA
2501-7E H, LSBs of the number	2501-85 H, LSBs of the number.
2502-89 H, MSBs of the number	2502-54 H, MSBs of the number.
Result	**Result**
2503-81 H, LSBs of the result.	2503-7A H, LSBs of the result.
2504-76 H, MSBs of the result	2504-AB H, MSBs of the result

Example 10: Find two's complement of 96H.

Solution: Two's complement of a number is obtained by adding 1 to the 1's complement of the number.

The number is placed in the memory location 2501 H.

The result is to be stored in the memory location 2502 H.

Program:

Address	Machine codes	Mnemonics	Operands	Comments
2000	3A, 01, 25	LDA	2501 H	Get data in accumulator
2003	2F	CMA		Take its 1's complement
2004	3C	INR	A	Take 2's complement
2005	32, 02, 25	STA	2502 H	Store result in 2502 H
2008	76	HLT		Stop

Example 1	Example 2
DATA	DATA
2501-96 H	2501- E4 H

Result	**Result**
2502- 6A H	2502- 1C H

Example 11: Find the 2's complement of 5B8C.

Solution: Two's complement of a number is obtained by adding 1 to the one's complement of the number.

The number is stored in the memory locations 2501 H and 2502 H.

The result is to be stored in the memory locations 2503 H and 2504 H

Address	**Machine codes**	**Label**	**Mnemonics**	**Operands**	**Comments**
2000	21, 01, 25		LXI	H, 2501 H	Address of 8 LSBs of the number
2003	06, 00		MVI	B, 00	Uses register B to store carry
2005	3E		MOV	A, M	8 LSBs in accumulator
2006	2F		CMA		1's complement of 8 LSBs of the number
2007	C6, 01		ADI	1	2's complement of 8 LSBs of the number
2009	32, 03, 25		STA	2503 H	Store 8 LSBs of the result
200C	D2, 10, 20		JNC	GO	
200F	4		INR	B	Store carry
2010	23	GO:	INX	H	Address of 8 MSBs of the number
2011	7E		MOV	A, M	8 MSBs in accumulator
2012	2F		CMA		1's complement of 8 MSBs of the number
2013	80		ADD	B	Add carry
2014	32, 04, 25		STA	2504 H	Store 8 MSBs of the result
2017	76		HLT		Stop

Example 1

DATA

2501-8C, LSBs of the number.

2502-5B, MSBs of the number.

Result

2503-74, LSBs of the result.

2504-A4, MSBs of the result.

Example 12: Shift 65 left by one bit

Solution:

The number is placed in memory 2501 H.

The result is to be stored in memory 2502 H.

Address	Machine codes	Mnemonics	Operands	Comments
2000	3A, 01, 25	LDA	2501 H	Get data in accumulator
2003	87	ADD	A	Shift it left by one bit
2004	32, 02, 25	STA	2502 H	Store result in 2502 H
2007	76	HLT		Stop

Example

DATA

2501- 65 H

Result

2502- CA H

The instruction LDA 2501 H transfers the number from memory location 2501 H to the accumulator. ADD A adds the contents of the accumulator to itself. The result is twice the number and thus the number is shifted left by one bit. This program does not take carry into account after ADD instruction. If numbers to be handled are likely to produce carry, the program may be modified to store it.

12 (a) Shifting of a 16-Bit Number Left by 2 Bits

The number is stored in the memory locations 2501 and 2502 H.

The result is to be stored in the memory locations 2503 and 2504 H.

Address	Machine codes	Mnemonics	Operands	Comments
2000	3A, 01, 25	LDA	2501 H	Get data in accumulator
2003	87	ADD	A	Shift it left by one bit
2004	32, 02, 25	STA	2502 H	Store result in 2502 H
2007	76	HLT		Stop

Example

DATA

2501- 65 H

Result

2502- CA H

12 (b) Mask Off Least Significant 4 Bits of an 8-Bit Number

We want to mask off the least significant 4 bits of a given number. The LSD of the given number A6 is 6. It is to be cleared (masked off), i.e., it is to be made equal to zero. The MSD of the number A6 is A. In the binary form it is 1010. It is not to be affected. If this number is ANDed with 1111, i.e., F, it will not be affected. Similarly, the LSD of the number is 6. In the binary form it is represented by 0110. If it is ANDed with 0000, it becomes 0000, i.e., it is cleared. Thus, if the number A6 is ANDed with F0, the LSD of the number is masked off.

Main Program:

Address	Machine codes	Mnemonics	Operands	Comments
2000	3A, 01, 25	LDA	2501 H	
2003	E6, F0	ANI	FO	
2005	32, 02, 25	STA	2502 H	
2008	76	HLT		

Example

DATA

2501- A6

Result

2502- A0

The instruction LDA 2501 H transfers the content of memory location 2501 H, i.e., the given number to the accumulator. ANI F0 logically ANDS the content of the accumulator with F0 to clear the least significant 4 bits of the number. STA 2502 H stores the result in memory location 2502 H. HLT stops the program.

Example 13: Find the larger of 98 H and 87 H.

Solution:

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

The second number 87 H is placed in the memory location 2502 H.

The result is stored in the memory location 2503 H.

Program:

Address	Machine codes	Label	Mnemonics	Operands	Comments
2003	7E		MOV	A, M	1st number in accumulator
2004	23		INX	H	Address of 2nd number in H-L pair
2005	BE		CMP	M	Compare 2nd number with 1st number. Is the 2nd number > 1st?

2006	D2, OA, 20		JNC	AHEAD	No, larger number is in accumulator. Go to AHEAD
2009	7E		MOV	A, M	Yes, get 2nd number in accumula-tor
200A	32, 03, 25	AHEAD:	STA	2503 H	Store larger number in 2503 H
200D	76				Stop

Example 1	**Example 2**
DATA	DATA
2501-98 H	2501-A9 H
2502-87 H	2502–EB H
Result	**Result**
2503-98 H	2503–BH

The numbers are represented in hexadecimal system. The 1st number is moved from its memory location to the accumulator. It is compared with 2nd number. The larger of the two is then placed in accumulator. From the accumulator the larger number is moved to the desired memory location.

The 1st step of the program is LXI H, 2501 H. The address of the memory location of the 1st is 2501 H. This address, i.e., 2501 is placed in H-L pair. The next instruction of the program is MOV A, M. The execution of this instruction moves the content of the memory (which has been previously defined, i.e., 2501) to the accumulator. Now the 1st number, i.e., 98 (in Example 1) is in the accumulator.

Example 14: Write a program to find the largest number from the series given as: 98, 75 and 99.

Solution: As there are three numbers in the series, count = 03.

The count is stored in the memory location 2500 H.

The numbers are stored in the memory location 2501 H to 2503 H.

The result is to be stored in the memory location 2450 H.

Program:

Address	**Machine codes**	**Label**	**Mnemonics**	**Operands**	**Comments**
2400	21, 00, 25		LXI	H, 2500 H	Address for count in H-L pair
2403	4E		MOV	C, M	Count in register C
2404	97		SUB	A	Get 00 in accumulator
2405	23	LOOP:	INX	H	Address of the next number of the series
2406	BE		CMP	M	Compare next number with previous maximum. Is next number > previous maximum?

2407	D2, OB, 24		JNC	AHEAD	No, larger number is in the accumulator. Go to the label AHEAD
240A	7E		DCR	C	Yes, get larger number in the accumulator
			JNZ	LOOP	Decrement count
240B	OD	AHEAD:	STA	2450 H	
240C	C2, 05, 24		HLT		Stop
240F	32, 50, 24				
2412	76				

Example 1	**Example 2**
DATA	DATA
2500-03	2500–06
2501-98	2501–38
2502-75	2502-94
2503-99	2503–EB
	2504–AB
	2505–B5
	2506–FB
Result	**Result**
2450–99	2450–FB

The memory location 2500 H contains the count of the series. 2500 H is placed in H-L pair. The instruction MOV C, M places the count in register C. In example 1 count is 03. The instruction SUB makes the content of the accumulator 00. As 00 is the smallest 8-bit number, it is taken as initial value for comparison. INX H points out the 1st number of the series. CMPM compares 1st number with 00. As the 1st number is greater than 00, there is a carry, and after the execution of JNC, the instruction MOV A, M is executed. The first number, i.e., 98 is now transferred to the accumulator. Then count in register C is decremented and the program is transferred to the label LOOP. Then X points out the address of the 2nd number of the series (75). The 1st number (98) which is in the accumulator is compared with the 2nd number. As the 1st number (98) is larger than the 2nd number there is no carry, and after the execution of the instruction JNC the program jumps to the id AHEAD. Thus, the 1st number being larger number is retained in accumulator. Then the count is decremented and program jumps to LOOP again to process the third number of the series. In this way at every step of comparison, if the next number of the series is larger one, it is transferred to the accumulator. If the next number is smaller one, it is ignored and the previous larger number which remains in accumulator is retained. Thus, the larger number of the series is obtained and stored in memory location 2450 H.

Example 15: Arrange 54, EB, 85,913 and A8 in descending order.

Solution

These numbers are stored in the memory locations 2501 to 2505 H.

The count = 05 and it is stored in 2500 H.

Results are to be stored in 2601 to 2605 H.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	11, 01, 26		LXI	D, 2601 H	Memory locations to store result
2003	21, 00, 25		LXI	H, 2500 H	Count address in H-L pair
2006	46		MOV	B, M	Count in register B to check whether all numbers have been arranged in descending order
2007	CD, 00, 22	START	CALL	2200 H	Call subroutine-1 to find the largest number
200A	12		STA	DE	Store result
200B	CD, 50, 20		CALL	2050 H	Call subroutine-2 to check which number is the largest
200E	13		INX	D	
200F	5		DCR	B	Have all the numbers been arranged in descending order?
2010	C2, 07, 20		JNZ	START	No, repeat process
2013	76		HLT		Stop

First of all, the largest number is selected from the given series of numbers and it is stored in 2601 H. This number is deleted from the series and 00 is stored in its position. Again the largest number is selected second time from the modified series. Since 00 is the smallest 8-bit number it does not affect the result while selecting the largest number second time. The largest number is now stored in 2602 H. Again this number is also replaced by 00 and from the modified series of numbers the largest number is selected again and stored in 2603 H. This process is repeated till all the numbers of the string are arranged in descending order.

SUBROUTINE-1 is to find the largest number in a data array.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2200	21, 00, 25		LXI	H, 2500 h	Load address in H-L pair
2203	4E		MOV	C, M	Count in register C to check whether all numbers have been compared to find the largest number

2204	97		SUB	A	Make content of accumulator zero
2205	23	LOOP	INX	H	
2206	BE		CMP	M	Compare next number with previous largest. Is next number > previous largest?
2207	D2, OB, 22		JNC	AHEAD	No, larger number is in the accumulator. Go to label AHEAD
220A	7E		MOV	A, M	Yes, get larger number in accumulator
220B	OD	AHEAD	DCR	C	
220C	C2, 05, 22		JNZ	LOOP	
220F	C9		RET		

SUBROUTINE-2 is to check which number is the largest, and to replace it by zero.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2050	21, 00, 25		LXI	H, 2500 H	Load address in H-L pair
2053	4E		MOV	C, M	Count to check which number was selected as the largest one
2054	23		INX	H, 2500 H	Get next number
2055	BE	BEHIND	CMP	M	Compare the next number with the largest number which is in accumulator
2056	CA, 5D, 20		JZ	FORWARD	Is the present number larger one? Yes, go to FORWARD
2059	0D		DCR	C	Decrement count
205A			JNZ	BEHIND	No, jump to take up next number
205D		FORWARD	MVI	A, 00	
205F	77		MOV	M.A	Replace largest by 00
2060	C9		RET		

Example 1	**Example 2**
DATA	DATA
2500–05	2500–06
2501–54	2501–99
2502–EB	2502–B4
2503–85	2503–34
2504–9B	2504–FF
2505–A8	2505–A6
	2506–E8
Result	**Result**
2601–EB	2601–FF
2602–A8	2602–E8
2603–9B	2603–B4
2604–85	2604–A6
2605–54	2605–99
	2606–34

Program 15 (B) Alternate method.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	21, 00, 25		LXI	H, 2500 H	Address for count
2003	4E		MOV	C, M	Count for number of passes in register C
2004	21, 00, 25	BACK	LXI	H, 2500 H	
2007	56		MOV	D, M	Count for number of comparisons in register D
2008	23		INX	H, 2500 H	
2009	7E		MOV	A, M	1st number in accumulator
200A	23	LOOP	INX	H, 2500 H	Address of next number
200B	46		MOV	B, M	Next number in Register B
200C	B8		CMP	B	Compare next number with the previous greatest number
200D	D2, 16, 20		JNC	AHEAD	If the previous greatest number > next number, go to AHEAD
2010	2B		DCX	H, 2500 H	
2011	77		MOV	M, A	Place smaller of the two compared numbers in memory
2012	78		MOV	A, B	Place greater of the two numbers in accumulator

2013	C3, 18, 20		JMP	GO	
2016	2B	AHEAD	DCX	H, 2500 H	
2017	70		MOV	M, B	Place smaller of the two compared numbers in memory
2018	23	GO	INX	H, 2500 H	
2019	15		DCR	D	Decrease the count for comparisons
201A	C2, OA, 20		JNZ	LOOP	
201D	77		MOV	M, A	Place the greatest number after a pass in the memory
201E	OD		DCR	C	Decrease the count for passes
201F	C2, 04, 20		JNZ	BACK	
2022	76		HLT		Stop

Data	**Result**
2500–Count (04)	
2501–60	15
2502–40	25
2503–50	40
2504–15	50
2505–25	60

Note: When the instruction JNC at the memory location 200D is replaced by JC (code: DA), the above program becomes a program for arranging an array of data in descending order.

In this program, the microprocessor first compares the first two numbers of the data array. It keeps the larger of the two numbers in the accumulator and stores the smaller number in the memory. Then it proceeds further and takes up the third number and compares it with the number which is in the accumulator. If the number in the accumulator is smaller than the 3rd number, it stores 3rd number in the accumulator. It stores the smaller number (which was in the accumulator) in the memory. In this way it processes all the numbers of the array. Finally, it stores the largest number in the last memory location. In the example given above the last memory location was 2505 H. This is all in the 1st pass. Now the processor again goes in the loop for the 2nd pass. In 2nd pass again it selects the largest number from the remaining numbers and it stores it in the last but one memory location (in above example, the memory location is 2504 H. The processor goes through several passes and arranges numbers in ascending order. Both the number of passes and the number of comparisons will be less than the number of data in the array by one. In the above example, there are five data and hence both counts are 04. When the processor arranges numbers in descending order, it selects the smallest number from the data array and stores it in the last memory location. In this way, it proceeds further.

To Find Smaller of the Two Numbers

Example 16: Find the smaller of 84 H and 99 H.

Solution:

The first number 84 H is placed in the memory location 2501 H.

The 2nd number 99 H is placed in the memory location 2502 H.

Store the result in the memory location 2503 H.

Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	21, 01, 25		LXI	H, 2501 H	Address of 1st number in H-L pair
2003	7E		MOV	A, M	1st number in accumulator
2004	23		INX	H	Address of 2nd number in H-L pair
2005	BE		CMP	M	Compare 2nd number with 1st. Is 1st number < 2nd number
2006	DA, OA, 20		JC	AHEAD	Yes, smaller number is in accumulator. Go to AHEAD
2009	7E		MOV	A, M	No, Get 2nd number in accu-mulator
200A	32, 03, 25	AHEAD	STA	2503 H	Store smaller number in 2503 H
200D	76			HLT	Stop

Example 1	**Example 2**
DATA	DATA
2501-84 H	2501-B9 H
2502-99 H	2502-8AH
Result	**Result**
2503-84 H	2503-8AH

2501 H is the address of the memory location for the 1st number. The instruction LXI H, 2501H, result in storing 2501H in H-L pair. The next instruction MOV A, M moves the content of the memory location to the accumulator. Thus, the 1st number 84 H (Example 1) is now in the accumulator. The instruction INX H increases the content of H-L pair from 2501 to 2502 H. The instruction CMP M compare the content of the accumulator (1st number) with the content of the memory location 2502. The instruction CMP M sets the flags as if the content of the memory location addressed by H-L pair has been subtracted from the contents of the accumulator. However, the contents of the accumulator remain unchanged. If the 1st number (which is in the accumulator) is smaller than the 2nd number (which is in the memory), there is a carry (see Example 1). The instruction JC is executed and the program jumps to STA 2503 H. The smaller number (84 H) is in the accumulator and the instruction STA 2503 H stores it in the memory location 2503 H.

The instruction HLT ends the program. If the 1st number (which is in the accumulator) is greater than the 2nd number (which is in the memory), there is no carry (see Example 2).

In this case after the execution of instruction JC the program does not jump to the instruction STA 2503 H. The program takes up the next instruction MOV A, M. Since the 2nd number is smaller and it is in the memory, it is moved to the accumulator. Now the instruction STA 2503 H is executed and the smaller number is stored in the memory location 2503 H. The instruction HLT will end the program.

To Find the Smallest Number in a Data Array

Example 17: The numbers of a series are: 86, 58 and 75 As there are three numbers in the series, count = 03.

Solution:

The count is placed in the memory location 2500 H.

The numbers are placed in the memory location 2501 to 2503 H.

The result is to be stored in the memory location 2450 H.

Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2000	21, 00, 25		LXI	H, 2500 H	Get address for count in H-L pair
2003	4E		MOV	C, M	Count in register C
2004	23		INX	H	Get address of 1st number in H-L pair
2005	7E		MOV	A. M	1st number in accumulator
2006	OD		DCR	C	Decrement count
2007	23	LOOP	INX	H	Address of next number in H-L pair
2008	BE		CMP	M	Compare the next number with previous smallest. Is the previous smallest less than the next number?
2009	DA, OD,20		JC	AHEAD	Yes, smaller number in accu-mulator. Go to AHEAD
200C	7E		MOV	A, M	No, get the next number in accumulator
200D	OD	AHEAD	DCR	C	Decrement count
200E	C2, 07, 20		JNZ	LOOP	
2011	32, 50, 24		STA	2450 H	Store the smallest number in 2450H
2014	76		HLT		Stop

Example 1	**Example 2**
DATA	DATA
2500-03H	2500-0514
2501-86H	2501-EB H

The 1st number of the series is placed in the accumulator and it is compared with the 2nd number which is in the memory. The smaller of the two is placed in the accumulator. Again this number which is in the accumulator is compared with the 3rd number of the series and the smaller number is placed in the accumulator. This process of comparison is repeated till all the numbers of the series are compared and the smallest number is stored in the desired memory location. The memory location 2500 H contains the count of the series. In Example 1, count is 03. 2500 is placed in H-L pair. The instruction MOV C, M places the count in register C. INX H increases the content of H-L pair from 2500 to 2501 H. The 1st number of the series resides in the memory location 2501 H. MOV A, M moves the 1st number (86) in the accumulator. DCR C decreases the count from pair 2501 to 2502 H. The 2nd number of the series is in 2502 H. CMP M compares the content of the accumulator (1st number) with the content of memory location 2502 H (2nd number). Since the 1st number 86 is greater than the 2nd number 58, there will be no carry. As there is no carry the instruction MOV A, M will be executed. The smaller number 58 will be placed in the accumulator. DCR C will decrease count from 02 to 01. It indicates that there is one more number left in the series. As the content of register C is not zero, the program will jump to the label LOOP. The INX H will change the content of H-L pair from 2502 to 2503 H. The 3rd number of the series is in 2503 Ht. Again CMP M will compare the content of the accumulator (58) with the content of 2503H.

The content of 2503 H is 75 which is greater than the content of the accumulator. In this case there will be a carry. After the execution of the instruction JC AHEAD, the program will jump to the label AHEAD. The content of the accumulator will remain as it is. DCR C will reduce the count from 01 to 00. As the content of register C is zero, the program will not jump. It will proceed further to execute STA 2450 H. The smallest number (58) which is in the accumulator will be stored in the memory location 2450 H. The instruction HLT will end the program.

8-Bit Division

The computer performs division by trial subtractions. The divisor is subtracted from the 8 most significant bits of the dividend. If there is no borrow, the bit of the quotient is set to 1; otherwise 0. To lie up the dividend and quotient properly, the dividend is shifted to left by one bit before each trial of subtraction. The dividend and quotient share a 16-bit register. Due to shift of dividend one bit of the register falls vacant in each step. The quotient is stored in vacant bit positions.

Example 18: Divide 489B by 1A.

Solution: These are hexadecimal numbers. The dividend is a 16-bit number and the divisor an 8-bit number. If the dividend of a problem is an 8-bit number, it is extended to a 16-bit number by placing zeros in MSBs positions.

The dividend is placed in the memory locations 2501 and 2502 H.

The divisor is placed in the memory location 2503 H.

The results are stored in the memory locations 2504 and 2505 H.

The quotient is stored in the memory location 2504 H.

The remainder is stored in the memory location 2505 H

Program:

Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2400	2A, 01, 25		LHLD	2501 H	Get dividend in H-L pair
2403	3A, 03, 25		LDA	2503 H	Get divisor from 2503 H
2406	47		MOV	B, A	Divisor in register B
2407	OE, 08		MVI	C, 08	Count = 08 in register C
2409	29	LOOP	DAD	H	Shift dividend and quotient to left by one bit
240A	7C		MOV	A, H	Most significant bits of dividend in accumulator
240B	90		SUB	B	Subtract divisor from most significant bits of dividend
240C	DA, 11, 24		JC	AHEAD	Is most significant part of dividend > divisor? No, go to AHEAD
240F	67		MOV	H, A	Most significant bits of dividend in register H
2410	2C		INR	L	Yes, add 1 to quotient
2411	OD	AHEAD	DCR	C	Decrement count
2412	C2, 09, 24		JNZ	LOOP	Is count = 0? No, jump to LOOP
2415	22, 04, 25		SHLD	2504 H	Store quotient in 2504 and remainder in 2505 H.
2418	76		HLT		Stop

Example 1

DATA

2501-9B H	LSBs of dividend
2502-48H	MSBs of dividend
2503-1AH	Divisor

Result

2504-F2	Quotient
2505-07	Remainder

Example 2

Divide 54 H by 09.	The dividend 54 is extended to 16 bits. 54 = 0054 H
DATA	
2501-54H	LSBs of dividend
2502-00	MSBs of dividend
2503-09	Divisor
Result	
2504-09	Quotient
2505-03	Remainder

The count in register C is kept 08. The trial subtraction is done 8 times and an 8-bit quotient is obtained. The instruction DAD H shifts dividend and quotient to left by one bit. Due to shift of dividend, the bit positions in register L fall vacant. In the vacant bit positions, quotient is stored. Note that the dividend is shifted prior to trial subtraction. The MSB of the dividend should be zero, otherwise it will be shifted to carry bit. If a problem contains MSB not equal to zero, it will be solved by splitting it into two parts. Shifting of dividend before subtraction is not done in ordinary division by pen and paper, but the computer method gives correct result as the numbers are represented in binary coded hexadecimal system.

Multi-byte Addition

Example 19: A byte consists of 8 bits. In the above example, two multi-byte hex numbers are to be added. Each number consists of 4 bytes. An 8-bit microcomputer takes one byte of the numbers at a time and adds them with carry. A counter is initiated to count the byte. In Example 1, the count = 4.

The count is placed in the memory location 2500 H.

The 1st number is placed in the memory locations 2501 to 2504 H.

The 2nd number is placed in the memory locations 2601 to 2604 H.

The sum is placed in the memory locations 2501 to 2504 H.

Program:

Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2400	21, 00, 25		LXI	H, 2500 H	Address of byte count in H-L pair
2403	4E		MOV	C, M	Byte count in register C
2404	23		INX	H	Address of 1st byte of 1st number
2405	11, 01, 26		LXI	D, 2601 H	Address of 1st byte of 2st number

2408	B7		ORA	A	Clear carry
2409	1A	LOOP	LDAX	D	Get byte of 2nd number in accumulator
240A	8E		ADC	M	Byte of 2nd number + byte of 1st number + carry
240B	77		MOV	M, A	Store sum in memory addressed by H-L pair
240C	23		INX	H	Increment the content of H-L pair
240D	13		INX	D	Increment the content of D-E pair
240E	OD		DCR	C	Decrement count
240F	C2, 09, 24		JNZ	LOOP	Is count = 0? No, jump to LOOP
2412	76		HLT		Stop

The bytes of 1st number are placed in the memory locations 2501 to 2504 H. The bytes of the 2nd number are in 2601 to 2604 H. H-L pair has been initiated to contain the address for the byte of the 1st number and D-E pair to contain the address for the byte of the 2nd number. The instruction ORA A clears the carry bit. Any other logical operation can also be used for this purpose. LDAX D gets the content of the memory addressed by D-E pair, i.e., the byte of the 2nd number. ADC M adds contents of the accumulator, contents of the memory addressed by H-L pair (the byte of the 1st number) and the carry of the previous byte addition. MOV M, A stores the sum in memory location addressed by H-L pair. Thus, the 1st number is destroyed after the execution of the program, and the sum is stored in its place. If there is any carry after the addition of the last byte, it can be taken into consideration. This can be done by extending the numbers to 5th byte. The 5th byte will be 00. The count will be 5. The 5th byte of the sum will be 01. It is better to extend numbers one byte more.

Example 1

DATA

2500-04	
2501-67	2601–8B
2502–8A	2602–6C
2503–9C	2603- 47
2504-3A	2604–9B

Result

2501–F2

2502–F6

2503–E3

2504–D5

Example 20: Write a program to find the square root of a number.

Suppose that X is the square root of number N. To find a suitable equation to be used by the computer for iteration, the following manipulation is done.

$$X^2 = N$$

$$2X^2 = N + X^2$$

$$X^2 = \frac{N + X^1}{2}$$

$$X = \frac{N + X^1}{2X}$$

$$X = \frac{\frac{N}{X} + X}{2}$$

To find the square root of a given number, we provide an initial value of the root, which may be very approximate. In the above equation, X is the initial value of the root given by the programmer. The computer calculates Xnew and compares it with X. When Xnew = X, it gives the result = Xnew. A program is given below. To find the square root within certain tolerance, limits for comparison may also be provided and approximate square root can be obtained. The program given below is for integer number.

Address	Machine Codes	Labels	Mnemonics	Operands	Comments
2600	3E, X		MVI	A, X	X is the first approximation
2602	57	BACK:	MOV	D, A	
2603	2A, 00, 25		LHLD	2500H	N in H-L pair
2606	CD, 06, 24		CALL	2406H	Division Subroutine, N/X in register L
2609	7A		MOV	A, D	X in accumulator
260A	85		ADD	L	Get (X + N/X)
260B	6F		MOV	L, A	
260C	26, 00		MVI	H, 00	
260E	3E, 02		MVI	A, 02	
2610	CD, 06, 24		CALL	2406H	(N/X + X)/2 in L = Xnew
2613	7D		MOV	A, L	
2614	BA		CMP	D	Compare Xnew with X
2615	CA, 1B, 26		JZ	BELOW	Jump to BELOW, if Xnew = X
2618	C3, 02, 26		JMP	BACK	Jump to BACK, if Xnew * X

261B	32, 50, 25	BELOW	STA	2550H	
261E	76		HLT		Stop

Note: This program is valid for numbers upto 3F01 hex or 16129 decimal, i.e., N ≤ 3F01

DIVISION SUBROUTINE

Address	Machine Codes	Labels	Mnemonics	Operands
2406	4F		MOV	C, A
2407	06, 08		MVI	B, 08
2409	29	DIV	DAD	H
240A	7C		MOV	A, H
240B	91		SUB	C
240C	DA, 11, 24		JC	AHEAD
240F	67		MOV	H, A
2410	2C		INR	L
2411	05	AHEAD	DCR	B
2412	C2, 09, 24		JNZ	DIV
2415	22, 03, 25		SHLD	2503 H
2418	C9		RET	

Example 1

DATA

2500 – 10 H

2501 – 00 H

Result

2550 – 04 H

Example 21: A set of 10 readings is stored in memory starting at 8050H. Sort the readings in ascending order.

PROBLEM ANALYSIS

The technique used here is bubble sort, i.e. comparing two bytes at a time and placing them in proper sequence. We compare the first two bytes, and if the first byte is larger than the second byte, we exchange their memory locations to arrange them in ascending order otherwise we keep them in same locations. We follow the same procedure for the second and third bytes. The number of comparisons necessary is always N-1 where N is the number of data bytes. Here we need a counter for one complete comparison. The microprocessor should continue these comparisons until no exchanges occur. Another register (in this program, D) is used as a reminder for the processor to recognize that no exchanges occur in a given pass. We write 1 in register D when an exchange takes place; otherwise, register D is cleared.

ALGORITHM

1. Start
2. Set up HL as a memory pointer for bytes.
3. Clear register D to set up as a flag and set register C to 09 for comparison count.
4. Get data byte and then increment HL to point to next byte.
5. Compare the two bytes.
6. If A is less than second byte then go to step 11. If A is greater than second byte then go to step 7.
7. Move second byte for exchange in B and store the first byte in the second location.
8. Decrement HL to point to first location and store second byte in first location.
9. Increment HL to get ready for next comparison.
10. Load 1 in D as a remainder for exchange
11. Decrement comparison count.
12. If comparison count = 0 go to step 13. If comparison count is not equal to zero then go to step 4.
13. Get flag bit (contents o D) in A.
14. Place flag bit D0 in carry.
15. If flag =1 hen exchange occurred and start next pass and go to step 2. If flag is not equal to zero then stop.

Address	**Hex code**	**Mnemonics**	**Operands**	**Comments**
8000	21, 50, 80	LXI	H, 8050	Set up HL as a memory pointer for bytes
8003	16, 00	MVI	D, 00	Clear (D) to set up a flag
8005	0E, 09	MVI	C, 09	Set (C) for comparison count
8007	7E	MOV	A, M	Get data byte in (A)
8008	23	INX	H	Point to next byte
8009	BE	CMP	M	Compare the bytes. IS A < second byte
800A	DA, 14, 80	JC		If yes, do not exchange jump to memory location 80
800D	46	MOV	B, M	Get second byte for exchange
800E	77	MOV	M, A	Store the first byte in second location
800F	2B	DCX	H	Point to first location
8010	70	MOV	M,B	Store second byte in first location
8011	23	INX	H	Get ready for next comparison

8012	16, 01	MVI	D, 01	Load 1 in D as a reminder for exchange
8014	0D	DCR	C	Decrement compare count
8015	C2, 07, 80	JNZ		If comparison count is not equal to 0 then go back to memory location 8007H
8018	7A	MOV	A, D	Get flag bit in A
8019	0F	RRC		Place flag bit D0 in carry

Example 22: Write a program to multiply two unsigned numbers stored in memory locations 8050H and 8051H using repeated addition method.

ALGORITHM

1. Load Accumulator with one of the numbers (multiplicand).
2. Copy contents of A to E
3. Take a counter register with value = multiplier-1
4. Add contents of E to A
5. Decrement counter register D
6. If D=0 then go to step 7, otherwise go to step 4
7. Move the contents of A to E

Address	Hex code	Label	Mnemonics	Operands	Comments
8000	01, 50, 80		LXI	B, 8050	Load BC with the memory location at which multiplicand is stored
8003	21, 51, 80		LXI	H, 8051	Load HL with next location at which multiplier is stored
8006	0A		LDAX	B	Load A with multiplicand
8007	5F		MOV	E, A	Copy A's contents to E
8008	56		MOV	D, M	Load D with multiplier
8009	15		DCR	D	Decrement register D
800A	83	PRO:	ADD	E	Add contents of E to A
800B	15		DCR	D	Decrement register D. Is D=0?
800C	C2, 0A, 80		JNZ	PRO	If no, jump back to memory location 800A
800F	5F		MOV	E, A	Move contents of accumulator to E
8010	EF		RST5		End

Example 23: Write a program to multiply two unsigned numbers stored in memory locations 8050H and 8051H using shift and add method.

SHIFT AND ADD METHOD

Repeated addition method is an inefficient technique for a large multiplier. By following the model of manual multiplication of decimal numbers, a more efficient technique is devised.

For example:

```
                        108
                       x 15
Step 1:  (108x5) =      540
Step 2:  (108x1) =    + 108
                       1620
```

In this example, the multiplier multiplies each digit of the multiplicand, starting from the farthest right and adds the product by shifting to the left. The same process can be applied in binary multiplication.

ALGORITHM

1. Place the contents of 8050 (multiplicand) in L register and that of 8051 (multiplier) in H register
2. Place multiplier in D and multiplicand in E
3. Transfer multiplier to A
4. Clear D (to use in DAD instruction) and clear HL
5. Set up register B to count 8 rotations and rotate accumulator right through carry.
6. If multiplier bit is equal to 1 then go to step 7. If not then go to step 8
7. Add multiplicand to HL and place partial result in HL
8. Place the multiplicand in HL and shift left
9. Retrieve shifted multiplicand in and decrement counter
10. If B is equal to 0 then go to step 11, otherwise go to step 5
11. Store the product in location 8090H and 8091H

Main Program:

Address	Hex code	Label	Mnemonics	Operands	Comments
8000	2A, 50, 80		LHLD	8050H	Place the contents of 8050H in L register and contents of 8051H in H register
8003	EB		XCHG		Place multiplier in D and multiplicand in E
8004	7A		MOV	A, D	Transfer multiplier to A

8005	16, 00		MVI	D, 00H	Clear D to use in DAD instruction
8007	21, 00, 00		LXI	H, 0000H	Clear HL
800A	06, 08		MVI	B, 08H	Setup register B to count 8 rotations
800C	1F	NXTBIT:	RAR		Check if multiplier bit is 1
800D	D2, 11, 80		JNC	NOADD	If not, skip adding multiplicand
8010	19		DAD	D	If multiplier is 1, add multi-plicand to HL and place partial result in HL
8011	EB	NOADD:	XCHG		Place multiplicand in HL
8012	29		DAD	H	Shift left
8013	EB		XCHG		Retrieve shifted multiplicand
8014	05		DCR	B	One operation is complete, decrement counter
8015	C2, 0C, 80		JNZ	NXTBIT	Go back to next bit
8018	22, 90, 80		SHLD	8090H	Store the product in locations 8090H and 8091H
801B	EF		RST5		End

The product is stored in locations 8090 and 8091H.

Example 24: Write a program to count the number of 1's in any given number.

ALGORITHM

1. Take 2 registers B and C as counters. B is set to 08H for the 8 bits of the and C to 00H to count the number of 1's
2. Load the accumulator with the 8-bit number
3. Rotate each bit in A to the left through carry
4. If the carry flag is set then go to step 5, if not then go to step 6
5. Increment the contents of register C by 1
6. Decrement the contents of register B by 1
7. Check is zero flag is set. If set then end the program and if not set then go to step 3.

Address	**Hex code**	**Label**	**Mnemonics**	**Operands**	**Comments**
8000	06, 08		MVI	B, 08	Load register B with 08H
8002	0E, 00		MVI	C, 00	Load register C with 00H

8004	3E, XX		MVI	A, XX	Load register A with hex number
8006	17		RAL		Rotate accumulator left through carry
8007	D2, 0B, 80		JNC	LOOP	If carry not generated, jump to memory location 800B
800A	0C		INR	C	Increment contents of register C by 1
800B	05	LOOP:	DCR	B	Decrement contents of register B by 1
800C	CA, 12, 80		JZ		If zero flag is set jump to memory location 8012H
800F	C3, 06, 80		JMP		If zero flag not set, jump to memory location 8006
8012	EF		RST5		End

Here we have to check for the 8 bits of the number, whether a bit is 0 or 1. So, register B is used as a counter with contents of 08H. The result i.e. the number of 1's is store in register C which is initialized as 00H.

Example 25: To find the largest number in an array of data using 8085 instruction set.

ALGORITHM

1. Load the address of the first element of the array in H-L pair
2. Move the count to B – reg.
3. Increment the pointer
4. Get the first data in A – reg.
5. Decrement the count.
6. Increment the pointer
7. Compare the content of memory addressed by H-L pair with that of A - reg.
8. If Carry = 0, go to step 10 or if Carry = 1 go to step 9
9. Move the content of memory addressed by H-L to A – reg.
10. Decrement the count
11. Check for Zero of the count. If ZF = 0, go to step 6, or if ZF = 1 go to next step.
12. Store the largest data in memory.
13. Terminate the program.

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	21, 00, 82		LXI	H, 8200H	;Set pointer for array
2003	46		MOV	B, M	;Load the Count
2004	23		INX	H	
2005	7E		MOV	A, M	; Set 1st element as largest data
2006	05		DCR	B	; Decrement the count
2007	23	LOOP:	INX	H	
2008	BE		CMP	M	; If A- reg> M go to AHEAD
2009	D2, 0D, 00		JNC	AHEAD	
200C	7E		MOV	A, M	; Set the new value as largest
200D	05	AHEAD:	DCR	B	
200E	C2, 07, 00		JNZ	LOOP	; Repeat comparisons till count = 0
2011	32, 00, 83		STA	8300	; Store the largest value at 8300H
2014	76		HLT		

Input

8200	05(array size)
8201	12
8202	AB
8203	D3
8204	88
8205	EE

Output:

8300	EE

Example 26: To find the smallest number in an array of data using 8085 instruction set.

ALGORITHM

1. Load the address of the first element of the array in H-L pair
2. Move the count to B – reg.
3. Increment the pointer
4. Get the first data in A – reg.
5. Decrement the count
6. Increment the pointer
7. Compare the content of memory addressed by H-L pair with that of A - reg.
8. If carry = 1, go to step 10 or if Carry = 0 go to step 9

9. Move the content of memory addressed by HL to A – reg.
9. Decrement the count
10. Check for Zero of the count. If ZF = 0, go to step 6, or if ZF = 1 go to next step.
11. Store the smallest data in memory.
12. Terminate the program.

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	21, 00, 82		LXI	H, 8200H	; Set pointer for array
2003	4E		MOV	B, M	; Load the Count
2004	23		INX	H	
2005	7E		MOV	A, M	; Set 1st element as largest data
2006	0D		DCR	B	; Decrement the count
2007	23	LOOP:	INX	H	
2008	BE		CMP	M	; If A- reg< M go to AHEAD
2009	DA, 0D, 20		JC	AHEAD	
200C	7E		MOV	A, M	; Set the new value as smallest
200D	0D	AHEAD:	DCR	B	
200E	C2, 07, 20		JNZ	LOOP	; Repeat comparisons till count = 0
2011	32, 00, 83		STA	8300	; Store the largest value at 8300
2014	76		HLT		

OBSERVATION

Input:	05 (8200) ----- Array Size
	0A (8201)
	F1 (8202)
	1F (8203)
	26 (8204)
	FE (8205)
Output:	0A (8300)

Example 27: To write a program to arrange an array of data in ascending order

ALGORITHM

1. Initialize H-L pair as memory pointer
2. Get the count at 4200 into C – register
3. Copy it in D – register (for bubble sort (N-1) times required)

4. Get the first value in A – register
5. Compare it with the value at next location.
6. If they are out of order, exchange the contents of A –register and Memory
7. Decrement D –register content by 1
8. Repeat steps 5 and 7 till the value in D – register become zero
9. Decrement C –register content by 1
10. Repeat steps 3 to 9 till the value in C – register becomes zero.

Address	**Hex code**	**Label**	**Mnemonics**	**Operands**	**Comments**
2000	21, 00, 82		LXI	H, 8200	; Set pointer for array
2003	4E		MOV	C, M	; Load the Count
2004	0D		DCR	C	; Decrement the count
2005	51	REPEAT:	MOV	D, C	
2006	21, 01, 82		LXI	H, 8201	; Load the next input data
2009	E	LOOP:	MOV	A, M	; Move in accumulator
200A	23		INX	H	; Point to next memory location
200B	BE		CMP	M	; Compare next number with previous number
200C	DA, 14, 00		JC	SKIP	; If carry = 1, jump to skip
200F	46		MOV	B, M	; Load the next input data
2010	77		MOV	M, A	; Move the content of accumulator to the memory location
2011	2B		DCX	H	; Decrement memory pointer address
2012	70		MOV	M, B	; Move the content of register B to the memory location
2013	23		INX	H	; Increment memory pointer
2014	15	SKIP:	DCR	D	; Decrement the D register value
2015	C2, 09, 00		JNZ	LOOP	; Jump to Loop if register D value not equal to zero.
2018	0D		DCR	C	; Decrement count value
2019	C2, 05, 00		JNZ	REPEAT	; If c not equal to 0, jump to REPEAT
201C	76		HLT		; End of program

OBSERVATION

Input:	8200	05 (Array Size)
	8201	05
	8202	04
	8203	03
	8204	02
	8205	01
Output:	8200	05(Array Size)
	8201	01
	8202	02
	8203	03
	8204	04
	8205	05

Example 28: To write a program to arrange an array of data in descending order.

ALGORITHM

1. Initialize H-L pair as memory pointer
2. Get the count at 8200 into C – register
3. Copy it in D – register (for bubble sort (N-1) times required)
4. Get the first value in A – register
5. Compare it with the value at next location.
6. If they are out of order, exchange the contents of A –register and Memory
7. Decrement D –register content by 1
8. Repeat steps 5 and 7 till the value in D – register become zero
9. Decrement C –register content by 1
10. Repeat steps 3 to 9 till the value in C – register becomes zero

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	21, 00, 82		LXI	H, 8200 H	; Set pointer for array
2003	4E		MOV	C, M	; Load the Count
2004	0D		DCR	C	; Decrement the count
2005	51	REPEAT:	MOV	D, C	; Save the content of register C into D
2006	21, 01, 82		LXI	H, 8201H	; Load the H-L pair with 8201H
2009	7E	LOOP:	MOV	A, M	; Get the data from memory
200A	23		INX	H	; Increment the counter

200B	BE		CMP	M	; Compare the data with previous data
200C	D2, 18, 00		JNC	SKIP	; If carry = 0, jump to skip
200F	46		MOV	B, M	; Save contents of M in register B
2010	77		MOV	M, A	; Move the content of accu-mulator to the memory loca-tion
2011	2B		DCX	H	; Decrement the D register value
2012	70		MOV	M, B	; Move the content of register B to the memory location
2013	23		INX	H	; Increment memory pointer
2014	15		DCR	D	;Decrement the D register value
2015	C2, 09, 00		JNZ	LOOP	; Jump to Loop if register D value not equal to zero.
2018	0D	SKIP:	DCR	C	; Decrement count value
2019	C2, 05, 00		JNZ	REPEAT	; If c not equal to 0, jump to REPEAT
201C	76		HLT		; End of program

OBSERVATION

Input:	8200	05(Array size)
	8201	01
	8202	02
	8203	03
	8204	04
	8205	05
Output:	8200	05(Array Size)
	8201	05
	8202	04
	8203	03
	8204	02
	8205	01

Example 29: To convert two BCD numbers in memory to the equivalent HEX number using 8085 instruction set.

ALGORITHM

1. Initialize memory pointer to 8150 H
2. Get the Most Significant Digit (MSD)
3. Multiply the MSD by ten using repeated addition
4. Add the Least Significant Digit (LSD) to the result obtained in previous step
5. Store the HEX data in Memory

Address	Hex code	Mnemonics	Operands	Comments
2000	21, 50, 81	LXI	H, 8150H	
2003	7E	MOV	A, M	; Initialize memory pointer
2004	87	ADD	A	; MSD X 2
2005	47	MOV	B, A	; Store MSD X 2
2006	87	ADD	A	; MSD X 4
2007	87	ADD	A	; MSD X 8
2008	80	ADD	B	; MSD X 10
2009	23	INX	H	; Point to LSD
200A	86	ADD	M	; Add to form HEX
200B	23	INX	H	
200C	77	MOV	M, A	; Store the result
200D	76	HLT		

OBSERVATION

Input:	8150 : 02 (MSD)
	8151 : 09 (LSD)
Output:	8152 : 1D H

Example 30: To convert the given hexadecimal number into its equivalent BCD number using 8085 instruction set.

ALGORITHM

1. Initialize memory pointer to 8150 H
2. Get the hexadecimal number in C – register
3. Perform repeated addition for C number of times
4. Adjust for BCD in each step
5. Store the BCD data in Memory

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	21, 50, 81		LXI	H, 8150H	; Initialize memory pointer
2003	16, 00		MVI	D, 00	; Clear D – reg for Most significant Byte
2005	AF		XRA	A	; Clear accumulator
2006	4E		MOV	C, M	; Get HEX data
2007	C6, 01	LOOP2:	ADI	01	; Count the number one by one
2009	27		DAA		; Adjust for BCD count
200A	D2, 0E, 00		JNC	LOOP1	
200D	14		INR	D	
200E	0D	LOOP1:	DCR	C	
200F	C2, 07, 00		JNZ	LOOP2	
2012	32, 51, 81		STA	8151	; Store the least significant Byte
2015	7A		MOV	A, D	
2016	32, 51, 81		STA	8152	; Store the most significant Byte
2019	76		HLT		

OBSERVATION

Input:	8150 : FF
Output:	8151 : 55 (LSB)
	8152 : 02 (MSB)

Example 31: To convert given hexadecimal number into its equivalent ASCII number using 8085 instruction set.

ALGORITHM

1. Load the given data in A – register and move to B – register
2. Mask the upper nibble of the Hexadecimal number in A – register
3. Convert the nibble into ASCII using subroutine by adding 30 or 37 (if no is greater the 0A then 30 is added otherwise 37 is added).
4. Store it in memory
5. Move B – register to A – register and mask the lower nibble
6. Rotate the upper nibble to lower nibble position
7. Call subroutine to get ASCII of upper nibble
8. Store it in memory
9. Terminate the program.

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	3A, 08, 20		LDA	8200	; Get Hexa Data
2003	47		MOV	B, A	
2004	E6, 0F		ANI	0F	; Mask Upper Nibble
2006			CALL	SUB1	;Get ASCII code for upper nibble
2009	32, 09, 20		STA	8201	
200C	78		MOV	A, B	
200D			ANI	F0	; Mask Lower Nibble
200F	07		RLC		
2010	07		RLC		
2011	07		RLC		
2012	07		RLC		
2013			CALL	SUB1	;Get ASCII code for lower nibble
2016	32, 0A, 20		STA	8202	
2019	76		HLT		
201A	FE, 0A	SUB1:	CPI	0A	;compare the number with 10
201C	DA, 21, 00		JC	SKIP	;Jump if num<10
201F	C6, 07		ADI	07H	;add 07 to number
2021	C6, 1E	SKIP:	ADI	30H	;add 30H to number
2023	C9		RET		;return to main program

OBSERVATION

Input:	8200	E4(Hexa data)
Output:	8201	34(ASCII Code for 4)
	8202	45(ASCII Code for E)

Example 32: To transfer data 29H and 35H to output-ports PORT 1 and PORT 0 respectively.

Program Description: In this program the given data bytes are to be stored in any two registers (here we use register A and C). The data is transmitted to output-port only from accumulator. So, first the data byte stored in accumulator is transferred to output-port (PORT 1), then we have to move 2nd data byte into accumulator from register C, to transfer it to PORT 0.

Address	Machine codes	Mnemonics	Operands	Comments
2000	0E, 35	MVI	C, 35H	; Load 35H data into register C
2001	3E, 29	MVI	A, 29H	; Load 29H data into register A

2002		OUT	PORT 1	; Send 29H data to output PORT 1
2003	79	MOV	A, C	; Copy data (35H) from register C to register A
2004		OUT	PORT 0	; Send 29H data to output PORT 0
2005	76	HLT		; End of program

Example 33: Load the bit pattern 19H in register B and 82H in register D. Mask all the bits except D0 from register B and register D. If D0 is at logic 1 in both registers, Turn ON the light connected to D0 position of output-port PORT1; otherwise Turn OFF the light.

Program Description: In this program, we have to check the logic levels of D0 bit in both given data bytes. This is done by using 'And operation' of both data byte with 01H data. This will mask all the bits of data except D0 bit. Now we use And operation with D0 bits of both data bytes and the resultant logic (0 or 1) is transfer to output-port (PORT1), according to which either the light will Turn ON (in case of resultant is at logic 1) or Turn OFF (in case of resultant is at logic 0).

Address	**Machine codes**	**Mnemonics**	**Operands**	**Comments**
2000	06, 19	MVI	B, 19H	;Load 19H data into register B
2002	16, 82	MVI	D, 82H	; Load 82H data into register D
2004	78	MOV	A, B	; Copy data (19H) from register B to register A
2005	E6, 01	ANI	01H	; Mask all bits of 19H except D_0
2007	4F	MOV	C, A	; Save D_0 (from A register) into register C
2008	7A	MOV	A, D	; Copy data (82H) from register B to register A
2009	E6, 01	ANI	01H	; Mask all bits of 82H except D_0
200B	A1	ANA	C	; AND bits D_0 of 19H and 82H
200C		OUT	PORT 1	; Turn ON/OFF light connected to D_0
200E	76	HLT		; End of program

Example 34: To count number of one's in the contents of register C and store the result in register B.

Program Description: Load the given data into register C. To count the number of one's in the data, we have to rotate the content of accumulator eight times. For this, we use register D as a counter. The content of register is move to register A for applying 'Rotate accumulator right through carry' operation on the data. Now we check the generation of carry. If carry is not generated, we decrease the counter by one and then again repeat the rotate operation and carry checking. If carry generated the content of register B are incremented by one. In the end of eight rotations the number of one's in given data byte are stored in register B.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	06, 00		MVI	B, 00H	; Initialize register B
2002	0E, 12		MVI	C, 12H	; Load 12H data into register C
2004	16, 08		MVI	D, 08H	; Load register D (as a counter) with 08H
2006	79		MOV	A, C	; Copy data (08H) from register C to register A
2007	1F	Back:	RAR		Rotate accumulator right through carry
2008	D2, 0C, 00		JNC	Skip	; Jump to skip if carry not generated
200B	04		INR	B	; Increment register B content
200C	15	Skip:	DCR	D	; Decrement counter D
200D	C2, 07, 00		JNZ	Back	; Jump to back if D is not equal to zero
2010	76		HLT		; End of program

DATA	C reg. – 12
RESULT	B reg. – 02

Example 35: To convert unpacked BCD numbers into ASCII number.

Program Description: Any unpacked BCD number is converted into ASCII number by using OR operation. The content of given data byte is ORed with 30H (this is equivalent to ASCII '0').

Address	Machine codes	Mnemonics	Operands	Comments
2000	3E, 19	MVI	A, 19H	;Load 19H data into register B
2002	F6, 30	ORI	0	; This is ASCII '0'= 30H
2004	32, 00, 25	STA	2500H	; Store contents of accumulator at memory location 2500H
2007	76	HLT		; End of program

Example 36: To convert an ASCII character into its equivalent Hexadecimal number.

Program Description: The input ASCII character is loaded into accumulator. Then subtract 30H data from it and compare it with 0AH. If carry is generated, store the resultant (equivalent hexadecimal number of input number) in any memory location and if carry not generated then subtract 07H from it and then store it to memory location.

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	3A, 00, 42		LDA	4200H	; Get data from memory location 4200H
2003	D6, 30		SUI	30H	; Subtract 30H from register A
2005	FE, 0A		CPI	0AH	; Compare 0AH data with accumulator contents
2007	DA, 0C, 00		JC	SKIP	; Jump to SKIP if carry is set
200A	D6, 07		SUI	07H	; Subtract 07H from register A
200C	32, 01, 42	SKIP:	STA	4201H	; Store content of accumulator in 4501H memory location
200F	76		HLT		; End of program

DATA: 4500H-AC

RESULT: 4501H- 75

Example 37: To convert Binary to ASCII Hex code number.

Program Description: First the input binary number is loaded into accumulator. Then separate the byte into two nibbles, to convert them into their equivalent ASCII numbers. The program of conversion is stored in subroutine CONVERT. After masking higher order nibble (by ANDed accumulator with 0FH), the lower nibble is compared with 0AH. If lower nibble is less than 0AH, add 30H to it and if it is more than 0A, add 07H to it and store resultant ASCII number into memory location pointed DE register pair. Same procedure is repeated for higher nibble input binary number.

Main program:

Address	Machine codes	Label	Mnemonics	Operands	Comments
2000	31, 00, 20		LXI	SP, 2000H	; Initialize stack pointer
2003	21, 00, 25		LXI	H, 2500H	; Point index where binary number is stored
2006	11, 05, 25		LXI	D, 2505H	; where resultant ASCII number is stored
2009	7E		MOV	A, M	; Copy input data from Memory into Accumulator
200A	47		MOV	B, A	; Save the content of register A into B
200B	0F		RRC		; Shift high order nibble to lower nibble position
200C	0F		RRC		
200D	0F		RRC		

200E	0F		RRC		
200F	CD, 1A, 00		CALL	CONVERT	; Call subroutine
2012	12		STAX	D	; Stored first ASCII Hex number in location pointed by D-E reg. pair
2013	13		INX	D	; Point to next memory location
2014	78		MOV	A, B	; Get input number again in accumulator
2015	CD, 1A, 00		CALL	CONVERT	;Call subroutine
2018	12		STAX	D	;Stored 2nd ASCII Hex number in location pointed by D-E reg. pair
2019	76		HLT		; End of program
Subroutine CONVERT					
201A	E6, 0F	CONVERT:	ANI	0FH	; Mask higher nibble
201C	FE, 0A		CPI	0AH	; Compare it with 0AH
201E	DA, 21, 00		JC	CODE	; Jump to CODE if carry generated
2021	C6, 07	CODE:	ADI	07H	; Add 07H to accumulator contents to obtain code for digit A to F
2023	C6, 30	RET:	ADI	30H	; Add 30H
2025	C9		RET		; Return to main program

Example 38: Write a program for 2 X 2 MATRIX MULTIPLICATION

ALGORITHM

1. Load the 2 input matrices in the separate address and initialize the HL and the DE register pair with the starting address respectively.
2. Call a subroutine for performing the multiplication of one element of a matrix with the other element of the other matrix.
3. Call a subroutine to store the resultant values in a separate matrix.

Main Program:

Address	Hex code	Label	Mnemonics	Operands	Comments
2000	0E, 00		MVI	C, 00	;Clear C reg.
2002	21, 00, 85		LXI	H, 8500	;Initialize HL reg. to 8500
2005	11, 00, 86	LOOP2:	LXI	D, 8600	;Load D-E register pair

2008	CD, 31, 00		CALL	MUL	;Call subroutine MUL
200B	47		MOV	B,A	;Move A to B reg.
200C	23		INX	H	;Increment H-L register pair
200D	13		INX	D	;Increment D-E register pair
200E	13		INX	D	;Increment D-E register pair
200F	CD, 31, 00		CALL	MUL	;Call subroutine MUL
2012	80		ADD	B	;Add [B] with [A]
2013	CD, 42, 00		CALL	STORE	;Call subroutine STORE
2016	2B		DCX	H	;Decrement H-L register pair
2017	1B		DCX	D	;Decrement D-E register pair
2018	CD, 31, 00		CALL	MUL	;Call subroutine MUL
201B	47		MOV	B, A	;Transfer A reg. content to B reg.
201C	23		INX	H	;Increment H-L register pair
201D	13		INX	D	;Increment D-E register pair
201E	13		INX	D	;Increment D-E register pair
201F	CD, 31, 00		CALL	MUL	;Call subroutine MUL
2022	80		ADD	B	;Add A with B
2023	CD, 42, 00		CALL	STORE	;Call subroutine MUL
2026	79		MOV	A, C	;Transfer C register content to Acc.
2027	FE, 04		CPI	4	;Compare with 04 to check whether all elements are multi-plied
2029	CA, 30, 00		JZ	LOOP1	;If completed, go to loop1
202C	23		INX	H	;Increment H-L register Pair
202D	C3, 05, 00		JMP	LOOP2	;Jump to LOOP2
2030	76	LOOP1:	HLT		;Stop the program
2031	1A	MUL:	LDAX	D	;Load acc from the memory location pointed by D-E pair
2032	57		MOV	D, A	;Transfer acc. content to D register
2033	66		MOV	H, M	;Transfer from memory to H register
2034	25		DCR	H	;Decrement H register
2035	C2, 38, 00		JZ	LOOP3	;If H is zero go to LOOP3

2038	82	LOOP4:	ADD	D	;Add Acc with D reg
2039	25		DCR	H	;Decrement H register
203A	C2, 38, 00		JNZ	LOOP4	;If H is not zero go to LOOP4
203D	26, 55	LOOP3:	MVI	H, 85	;Transfer 85 TO H register
203F	16, 56		MVI	D, 86	;Transfer 86 to D register
2041	C9		RET		;Return to main program
2042	06, 57	STORE:	MVI	B, 87	;Transfer 87 to B register
2044	02		STAX	B	;Load A from memory location pointed by BC pair
2045	0C		INR	C	;Increment C register
2046	C9		RET		;Return to main program

Things to Remember

◊ The 8085 microprocessor signal can be classified in six groups: address bus, data bus, control bus and status signals, externally initiated signals and their acknowledgement, power and frequency, serial I/O signals.

◊ The data bus and the lower order address bus are multiplexed; they can be demultiplexed by using the ALE (Address Latch Enable) signal and a latch.

◊ The IO/$\overline{M}$ is a status signal–when it is high, it indicates an I/O operation; when it is low, it indicates a memory operation.

◊ The $\overline{RD}$ and $\overline{WR}$ are control signals; the $\overline{RD}$ is asserted to read from an external device, and the $\overline{WR}$ is asserted to write into an external device.

◊ The $\overline{RD}$ and $\overline{WR}$ signals are logically ANDed with the IO/$\overline{M}$ signal to generate four active low control signals: $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ and $\overline{IOW}$.

◊ Each instruction of the 8085 microprocessor can be divided into a few basic operations called machine cycles, and each machine cycle can be divided into T-states.

◊ The frequently used machine cycles are Opcode Fetch, Memory Read and Write, and I/O Read and Write.

◊ When the 8085 performs any of the operations; it asserts the appropriate control signal and status signal.

◊ Most Opcode Fetch operations consist of four T-states, and the subsequent Memory Read or Memory Write cycles require three T-states. Some Opcode Fetch operations require six T-states.

◊ To read from memory, the address of the register should be placed on the address lines and the $\overline{RD}$ signal must be asserted low to enable the output buffer.

◊ To interface a memory chip with 8085, the necessary low order address lines of the 8085 address bus are connected to the address lines of the memory chip. The high order address lines are decoded to generate $\overline{CS}$ signals to enable the chip.

◊ Peripheral Mapped I/O:

- ◊ The OUT is a two-byte instruction. It copies data from the accumulator to the addressed port.
- ◊ When the 8085 executes the OUT instruction, in the third machine cycle, it places the output port address on the low-order bus, duplicates the same port address on the high-order bus, places the contents of the accumulator on the data bus and asserts the control signal WR.
- ◊ A latch is commonly used to interface output devices.
- ◊ The IN instruction is two-byte instruction. It copies data from an input port and places the data into the accumulator.
- ◊ When the 8085 executes the IN instruction in the third machine cycle, it places the input port address on the lower order bus, as well as on the high-order bus, asserts the control signal $\overline{RD}$ and transfers data from the port to the accumulator.
- ◊ A tri-state buffer is commonly used to interface input devices.
- ◊ To interface an output or an input device, the low-order address bus A_7-A_0 needs to be decoded to generate the device address pulse, which must be combined the control signal $\overline{IOR}$ to select the device.
- ◊ Memory-Mapped I/O:
- ◊ Memory related instructions are used to transfer data.
- ◊ To interface I/O devices, the entire bus must be decoded to generate the device address pulse, which must be combined with the control signal $\overline{MEMR}$ to generate the I/O pulse. This pulse is used to enable the I/O device and transfer the data.

Questions and Answers

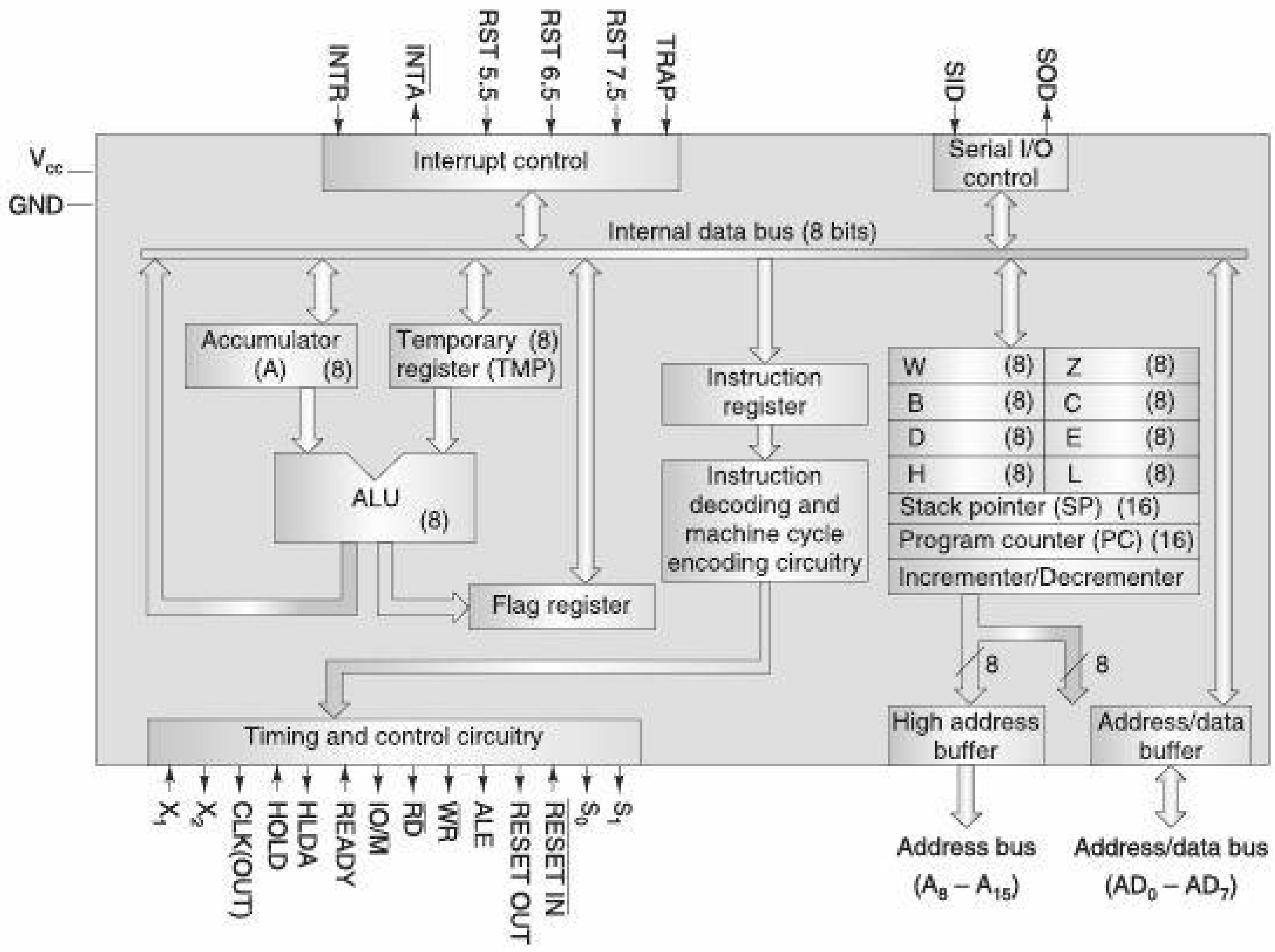
Microprocessor Architecture and Operation

1. Describe the functions of general-purpose and special-purpose registers of 8085, also write the flag register format and explain the function of each flag.

General-purpose Registers

- The 8085 contains 6 general-purpose registers of 8 bits each, named as B, C, D, E, Hand L.
- These registers are available to the user. They are used to hold data, results of arithmetic and logical operations and address of data memory.
- The main use is to hold data which is frequently used. It increases the speed of program execution. The main reason is as follows. The data in the microprocessor can be stored in memory or general-purpose registers. If the data is present in the memory the microprocessor has to perform an operation of memory read. This data is taken by microprocessor, the required operation is performed and the result is stored back in memory.

To store the result in memory the microprocessor has to perform one more operation of memory write. But if the data is present in general-purpose registers there is no operation involved as the registers are part of microprocessor architecture, the microprocessor doesn't have



to perform an external read and write operation. Thus, the time required to execute programs using general-purpose registers is very less compared to execution using memory.

- These registers can be used for general applications such as counters, memory pointers, etc.· B, C, D E, H and L can be used to store 8 bits of data or can be used to form a register pair to store 16 bits of data. The register pairs available are BC, DE, and HL. These registers are programmable by the user. The user can store data and perform different operations.
- The flag register indicates the status of the result after an execution. This register is 8-bit wide. The bit configuration of flag register is as shown.

Digit	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Flag	S	Z		AC		P		CY

- When the result of an operation is –ve then the sign flag is set, i.e. S = 1. The status of the sign flag also indicates the condition of overflow when signed arithmetic operations are performed. The sign flag status is meaningful only for signed operations. While performing arithmetic or logical operations on unsigned data the status of sign flag may indicate wrong status of the result.
- When the contents of the accumulator is zero after the execution of arithmetic or logical instruction, the zero flag is set, i.e. Z = 1, otherwise Z = 0.

- During an arithmetic operation when a carry is generated from the bit 3 to 4, the auxiliary carry is set, i.e. AC = 1.
- When the 8-bit data in the accumulator after an operation contains even number ones then the parity flag is set, i.e. = 1.
- When a carry is generated from the MSB of the accumulator during an operation then the carry flag is set.

Architecture: The ALU receives information from the accumulator and temporary registers. The temporary registers are not accessible to the users.

- To facilitate the data transfer within the microprocessor an 8-bit internal bus is used.
- The data transfer between accumulator and other registers are bidirectional. Therefore, the accumulator and temporary registers communicate with internal bus in bidirectional manner.
- The output of ALU is sent back to the accumulator through internal bus.
- The flag register communicates with ALU in bidirectional manner.
- The instruction register IR receives the instruction to be executed. This instruction is decoded by the instruction decoder and is given to ALU for generating the necessary control signals.
- W & Z registers are used for storing the address associated with an instruction.
- The register select will send command signal to suitable internal registers of the microprocessor depending on the instruction executed.
- The increment register increments the contents of PC after each instruction are executed.
- Buffers are used for sending the address from microprocessor to the address bus and the transfer of data bus the buffers increase the driving capability of the processor.

Timing and Control Unit

This unit synchronizes all the operations of the processor with the clock and also generates the control signals necessary for communication with processor and peripheral.

AD_0-AD_7: Data bus of the microprocessor is 8-bit wide. The pins labeled AD0-AD7 carries 8-bit data and this bus is multiplexed with lower 8 bits of address. The data transfer is bidirectional.

A_8-A_{15}: Address bus is 16-bit wide. Pins A_8 to A_{15} will carry most significant byte of the address.

ALE signal: Address enable latch is used to enable the latch, latches the lower b-bit address on AD_0-AD_7

IO/$\overline{M}$ signal: The address sent by the microprocessor could be either for an I/O device or for memory. This signal is used to distinguish whether the address is for I/O device or memory. If this signal is high the address is meant for I/O device otherwise the memory is addressed.

RD and WR signal: When $\overline{RD}$ is 0 the microprocessor reads the data either from I/O device or the memory depending on IO/$\overline{M}$ signal. When $\overline{WR}$ is 0 the microprocessor sends data to the I/O device or memory depending upon IO/$\overline{M}$.

INTA*: When μp is interrupted on INTA* signal, this signal is activated. In response to this signal, the suitable external device will provide the vector address to the routine.

HOLD and HLDA: For fast and bulk transfer of data between two peripherals DMA operation is used. For this a DMA controller is used. For this a DMA controller is required. Whenever DMA operation is to be initiated, the DMAC activates HOLD signal. In response to this signal, the processor completes the execution of the present instruction and relinquishes the control on the BUS, enters into HOLD state and then activates HLDA signal. In response to HLDA signal, the DMAC becomes the master and performs the data transfer on the BUS. At the end of data transfer operation, HOLD signal is made low, then the microprocessor comes out of HOLD state, HLDA is made low.

READY: This signal is used to synchronize a slower peripheral with μp.

RESET IN: This signal is used to RESET the processor. When the signal on this pin goes low, the PC is set to zero, and the buses are tri-stated, interrupt enable and HLDA flip-flops are also reset. The data and address buses are three-stated during RESET operation. The signal RESET is asynchronous in nature. All registers and flag may be altered by RESET with unpredictable results. This signal must remain low for at least 10 ms after minimum vcc has been reached. For proper RESET operation after power-up duration, RESET signal must be low at least for 3 clock periods.

RESET OUT: This signal indicates that the processor is being reset. This signal can be used to reset other devices. This signal is synchronized to the processor clock and lasts for integral number of clock cycles.

SID and SOD: These pins are used for the processor to do serial communication. On SOD pin the processor can transmit data serially after executing SIM instruction. Similarly, the processor receives data serially on SID pin by executing RIM instruction. SID pin is connected to an external transmitter and the SOD is connected to a receiver.

2. Explain the functions of the following registers.

(a) **H-L Pair:** This is a register pair which is used as a memory pointer. The higher order bits are used in H register and lower order bits are stored in L register. It is used to store 16-bit data.

(b) **Stack pointer:**

- Stack is a reserved portion of memory where register pair information can be stored or taken back under software control. This memory area is referred to as stack area.
- This is a 16-bit register used to define the stack starting address. It always points at top of the stack.
- It is used to keep track of data stored on stack.
- The stack pointer is decremented after each stack write operation and incremented after each stack read operation.
- The stack pointer is loaded with an initial value by means of a transfer type instruction. This initial value must be the highest address of an assigned stack in memory.
- Stack should be always initialized in data memory.

(c) **Flag register:**

- The flag register indicates the status of the result after an execution. This register is 8-bit wide. The bit configuration of flag register is as shown.
- When the result of an operation is –ve then the sign flag is set, i.e. S = 1. The status of the sign flag also indicates the condition of overflow when signed arithmetic operations are performed. The sign flag status is meaningful only for signed operations. While performing arithmetic or logical operations on unsigned data the status of sign flag may indicate wrong status of the result.
- When the contents of the accumulator is zero after the execution of arithmetic or logical instruction, the zero flag is set, i.e. Z = 1, otherwise Z = 0.
- During an arithmetic operation when a carry is generated from the bit 3 to 4, the auxiliary carry is set, i.e. AC = 1.
- When the 8-bit data in the accumulator after an operation contains even numbers then the parity flag is set, i.e. = 1.
- When a carry is generated from the MSB of the accumulator during an operation then the carry flag is set.

3. Explain the pin configuration of 8085.

The 8085 is a general-purpose microprocessor with 40 pins. The 8085 is an 8-bit device capable or addressing 64 K memory as it employs 16 address lines. This requires a single supply of +5 V and a single phase clock of 3 MHz frequency. All the signals of 8085 can be classified into 6 groups.

1. Data bus
2. Address bus
3. Control and status
4. Power supply and frequency
5. Interrupts and peripheral initiated
6. Serial I/O ports

AD_0-AD_7: Data bus of the microprocessor is 8-bit wide. The pins labeled AD_0-AD_7 is carries 8-bit data and this bus is multiplexed with lower 8-bits of address. The data transfer is bidirectional.

AD_8-AD_7: Address bus is 16-bit wide. Pins A_8 to A_{15} will carry most significant byte of the address.

ALE signal: Address enable latch is used to enable the latch, latches the lower b-bit address on AD_0-AD_7

$IO/\overline{M}$ signal: The address sent by the microprocessor could be either for an I/O device on for memory. This signal is used to distinguish whether the address is for I/O device or memory. If this signal is high the address is meant for I/O device otherwise the memory is addressed.

$\overline{RD}$ and $\overline{WR}$ signal: When $\overline{RD}$ is 0 the microprocessor reads the data either from I/O devices or the memory depending on $IO/\overline{M}$ signal. When $\overline{WR}$ is 0 the microprocessor sends data to the I/O device or memory depending upon $IO/\overline{M}$.

$IO/\overline{M}$	$\overline{RD}$	$\overline{WR}$	**REMARKS**
0	0	1	Reading from memory
0	1	0	Writing into memory
1	0	1	Reading from IO device
1	1	0	Writing into IO device
x	1	1	No communication with peripheral

4. Explain how the control signals are generated in 8085.

The four control signals are generated by combining the signals $\overline{RD}$, $\overline{WR}$, and $IO/\overline{M}$. The signal $IO/\overline{M}$ goes low for the memory operation. This signal is ANDed with $\overline{RD}$ and $\overline{WR}$ signals by using the 74LS32 quadruple two input OR gates. The OR gates are functionally connected as negative NAND gates. When both input signals go low, the outputs of the gates go low and generate $\overline{MEMR}$ (Memory read) and $\overline{MEMW}$ (Memory write) control signals. When the $IO/\overline{M}$ signal goes high, it indicates the peripheral I/O operation. This signal is complemented using the Hex inverter 74LS04 and ANDed with the $\overline{RD}$ and $\overline{WR}$ signals to generate $\overline{IOR}$ and $\overline{IOW}$ control signals.

5. Why do you think it is necessary for 8085 to have 2 status signals, s_0 and s_1?

These are output status signals used to give information performed by the microprocessor. These are generally used in small systems but can be used to generate advanced control signals for large systems.

6. Give the functions of the following pins of 8085.

(a) **RST 6.5:** It is a level sensitive interrupts. It has priorities then the INTR interrupt.

(b) **HOLD:** This signal indicates that a peripheral such as the DMA controller is requesting the use of address and data bus.

(c) **READY:** This signal is used to synchronize a slower peripheral with µp. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.

7. Explain the purpose of each pin

(a) **ALE:** Address enable latch is used to enable the latch, latches the lower b-bit address on AD_0-AD_7.

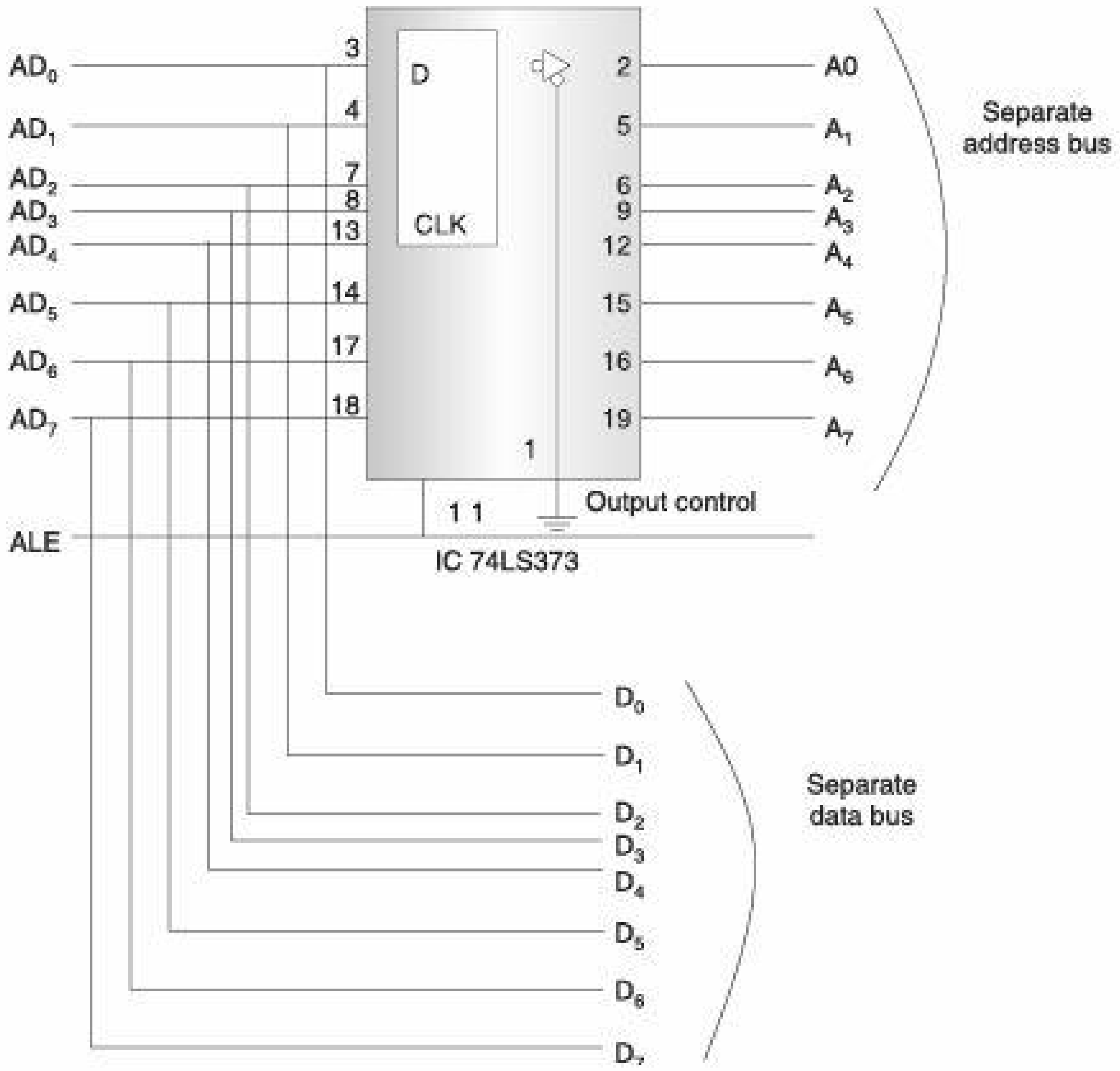
(b) **IO/M*:** The address sent by the microprocessor could be either for an I/O device or for memory. This signal is used to distinguish whether the address is for I/O device or memory. If this signal is high the address is meant for I/O device otherwise the memory is addressed.

(c) **READY:** This signal is used to synchronize a slower peripheral with µp. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.

(d) **HOLD:** This signal indicates that a peripheral such as the DMA controller is requesting the use of address and data bus.

(e) **RESET IN:** This signal is used to RESET the processor. When the signal on this pin goes low, the PC is set to zero, and the buses are tri-stated, interrupt enable and HLDA flip-flops are also reset. The data and address buses are three-stated during RESET operation. The signal RESET is asynchronous in nature. All registers and flags may be altered by RESET with unpredictable results. This signal must remain low for at least 10 ms after minimum Vcc has been reached. For proper RESET operation after power-up duration, RESET* signal must be low at least for 3 clock periods.

8. With block schematics, explain how the multiplexed address/data bus is de-multiplexed in 8085.



Step 1: Address will appear on AD_0–AD_7 lines.

Step 2: ALE will go high. This makes 74LS373 latch transparent. That is, whatever will be input that will be output. Presently, input is address (AD_0–AD_7). Therefore, output is AD_0–AD_7.

Step 3: Before address disappears, ALE = 0. Due to this the contents will get latched. Even though input toggles, output will not change. Thus, till next cycle, i.e., unless next ALE appears, AD_0–AD_7 status is latched and will not change.

Step 4: AD_0–AD_7 will be continued as data bus. A_{15}-A_8, i.e., higher order address bus is not multiplexed. Thus, output of latch (AD_0–AD_7) and upper order address in total will provide 16-bit address. Thus, at this stage we are ready with AD_{15} –AD_0 and D_7–D_0 data bus.

9. Differentiate between power on reset and manual reset.

When power is first applied to the microprocessor, the various registers and flip-flops assume random states, and the operation of the processor is unpredictable. Therefore, the processor must be reset when it is first powered up in order to fetch the first instruction. The processor is not guaranteed to work until 10 ms after Vcc reaches minimum operating voltage of 4.5 V. An RC circuit is used to get automatic power on reset operation. When the power is first applied to the circuit, the voltage across the capacitor is. The capacitor now charges through the resistor R with a time constant RC to a final voltage Vcc. The voltage across the capacitor is the input signal at RESETIN* PIN. The values of R and C are selected such that the voltage across the capacitor remains at logic 0 for the required time. Once the system is operating, the push button switch can manually reset it. Pressing this switch shorts the capacitor and discharges it. Releasing this switch, causes the capacitor charge back to Vcc bringing RESETIN* to logic 1. For proper resetting, the RESETIN* input must remain low for three clock pulses.

What are the various addressing modes of 8085? Explain each of them with examples.

Register Addressing Mode: Number of instruction come under this group. With some of these instructions the accumulator is implied as second operand. Ex: CMP D means the contents of D are compared with that of accumulator. All data transfer instructions come under this group. Ex: MOV A, B

Immediate Addressing Mode: Instructions that use the second byte of the instruction as the data come under this group. Since the data required for the execution is directly available it is called immediate mode of addressing. Ex: MVI B, 09 H is a two-byte instruction, the second byte being the data.

Direct Addressing Mode: In certain instructions the Opcode is followed by an address. The instructions are said to have direct addressing since the address of the memory location is directly mentioned in the instruction itself. Ex: LDA 1C05 H.

Indirect Addressing Mode: The data transfer instructions between internal register of the microprocessor and the memory belongs to this category. Since the address is not specified directly in the instruction it is indirect addressing mode. Ex: MOV M,C

Implied Addressing Mode: The addressing mode of certain instructions is implied by the instruction's function. For such instructions the operand is not specified. Ex: The instruction STC deals with only the carry flag. Similarly, DAA deals with the accumulator.

10. How do you classify the instruction set of 8085? Give example for each?

Data transfer operations: Ex: MOV A, M

Arithmetic instructions: Ex: ADD M

Logical instructions: Ex: XRA M

Branching operations: Ex: JUMP, CALL

11. Explain the effect of executing the following instructions.

1. MVI M, 08: This instruction moves immediate data to memory. That is, the H-L pair is loaded with 08 H.
2. LDA 8850 H: The accumulator is directly loaded from memory. That is, the accumulator is loaded with the contents of the memory location 8850 H.
3. STAX D: This copies the contents of the accumulator to a memory location pointed by a register pair. In this the accumulator contents are copied to memory location pointed by D-E register pair.
4. XCHG: This instruction exchanges the contents of H register with D register and L register with E register.
5. DAD: This instruction adds the contents of the specified register pair with the contents of H-L pair.
6. DAA(Decimal Adjust Accumulator): The contents of accumulator (after addition of two BCD numbers) are changed from binary format to BCD format.
7. CMP M: This instruction compares the contents of accumulator with the contents of memory pointed by H-L pair.
8. RAL: This instruction rotates the contents of the accumulator left by 1-bit and the carry is transferred to the LSB.
9. XTHL: This instruction exchanges H-L pair contents with the top of the stack.
10. SPHL: The contents of H-L pair are transferred to stack pointer register. H register contents to higher order bits and L register contents to low order 8 bits.
11. NOP: No operation is performed. This is used to achieve a time delay.

12. What is a subroutine? What are the different subroutine instructions available in 8085?

In the course of execution of a program it may be necessary that certain portions of the programs be executed repeatedly. Instead of writing these repeatedly, this can be written separately from the main program. These groups of instructions are called **subroutines**. The subroutine instructions available are CALL and RETURN.

13. Explain the steps involved in executing CALL instruction.

When a CALL instruction is encountered in the main program, the contents of the program counter, which is the RETURN ADDRESS of the main program, are saved on the stack and the address of the required subroutine is loaded in the program counter. With this, the program branches to the subroutine.

14. What is the effect of execution of the following?

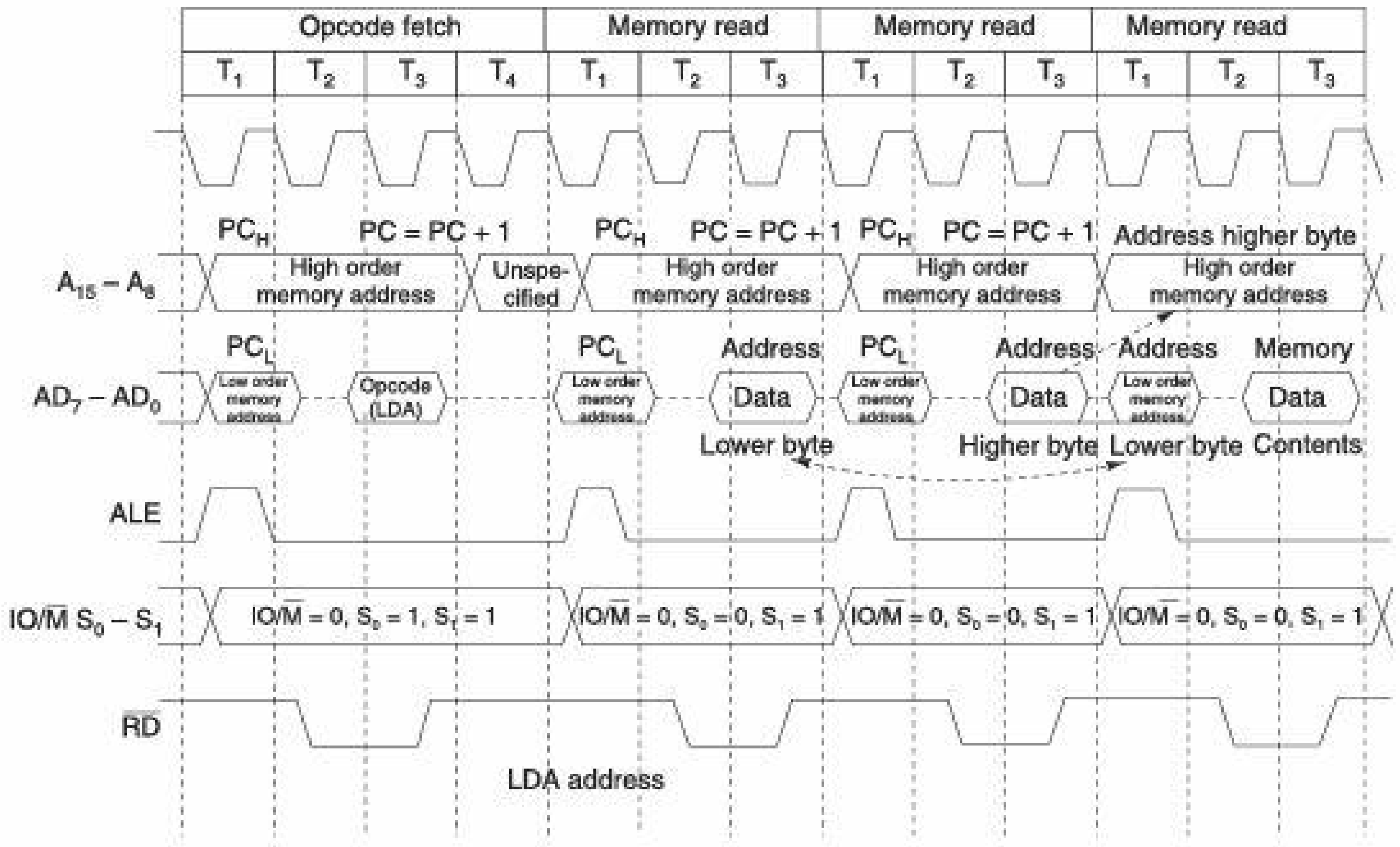
(a) POP H: The contents of the memory location pointed by the stack pointer (SP) register are copied to the low order register (in this case L) of the register pair. The SP is incremented by one and the contents of that memory location are copied to the higher order register (in this case H).

(b) IN 80 H: The data of input device having address 80 H is read by the accumulator. It is a 2-byte instruction.

(c) OUT 83 H: The data present in the accumulator is written into the output device having address 83 H. It is a 2-byte instruction.

(d) DAD: This instruction adds the contents of specified register pair with the contents of H-L pair.

15. Draw the timing diagram for: LDA XX, MVI A, 08

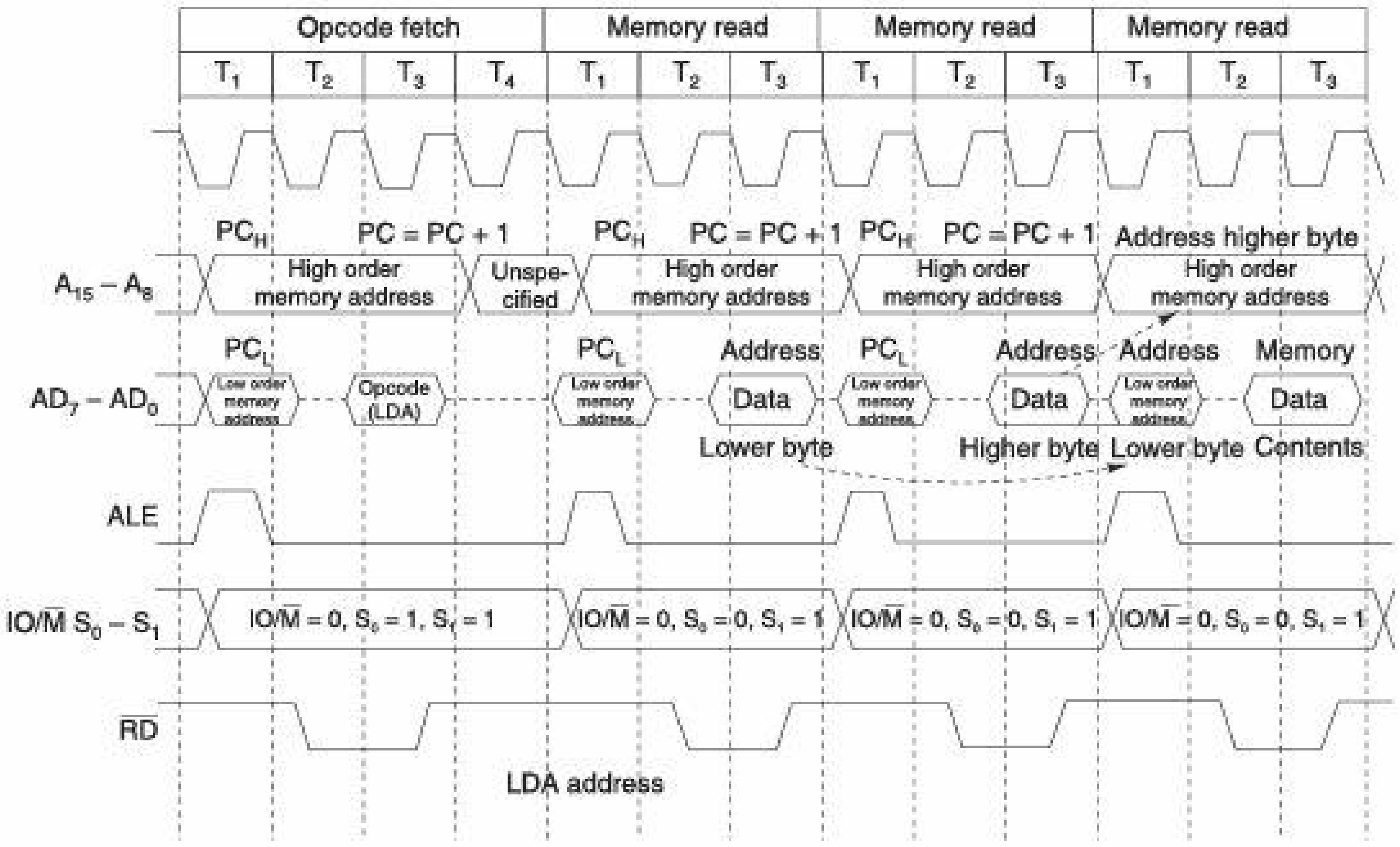


Timing diagram is for MVI A data

16. Write an ALP to generate FIBONACCI numbers up to N where N is also a Fibonacci number. Include comments.

```
        MVI D, n      ; Initialize a counter
        MVI B, 00     ; (B) = 00 H (Initialize variable to store previous number)
        MVI C, 01     ; (C) = 01 H (Initialize variable to store current number)
BACK:   MOV A, B      ; (A) + (B) --- (A)
        ADD C         ; (Add two numbers)
        MOV B, C      ; (C) --- (B) (Current number is now previous number)
        MOV C, A      ; (A) --- (C) (Save result as a new current number)
        DCR D         ; (D)-(01) --- (D) (Decrement the counter)
        JNZ BACK      ; if count ≠ 0 go to back
        HLT           ; stop
```

17. Draw the timing diagram for: CALL 8875 instruction.



Timing diagram for CALL instructions

18. Explain the following instructions indicating their addressing modes, flags affected and length of the instruction.

1. XRA A:

   Operation: The contents of specified register are EXORed with accumulator and the result is placed in the accumulator.

   Addressing mode: Indirect

   Number of bytes: 1 Byte

   Flags: S, Z, P are affected to reflect of operation, AC and CY are reset.

2. SUB M:

   Operation: The memory location contents pointed by H-L pair are subtracted from accumulator contents and the result is placed in the accumulator.

   Addressing mode: Indirect

   No. of bytes: 1 Byte

   Flags: ALL flags are affected.

19. Explain the following instructions indicating their addressing modes, flags affected and length of the instruction.

1. DAA: The contents of the accumulator are changed from a binary value to a 4-bit BCD no.

   No. of bytes: 1 byte

Addressing mode: Implied

Flags: ALL flags are affected

2. LXI H, 2540: This instruction loads 16-bit data to the H-L pair.

   No. of bytes: 3 bytes

   Addressing mode: Register

   Flags: None of the flags are affected.

3. CALL 9540: The program sequence is transferred to the address 9540 as specified in the instruction.

   No. of bytes: 3

   Addressing mode: Indirect

   Flags: None of the flags are affected.

4. LDAX B: This instruction copies the contents of the memory location to the accumulator. The address of memory location is given B-C pair specified along with the instruction.

   No. of bytes: 1 byte

   Addressing mode: Indirect

   Flags: None of the flags are affected

5. CMP R = [A]-[R]
   - If A>R , It results carry and zero flags are reset (Z=0, CY=0)
   - If A<R , carry flag is set and zero flag is reset (Z=0, CY=1)
   - If A=R, zero flag is set and carry flag is reset (Z=1, CY=0)

20. Write an ALP to transfer block of 10 bytes of data, which starts from memory address X100 into some other location Y100.

```
MVI C, 0A H          ; C = 0A H (Initialize a counter)
LXI H, X100 H        ; (H) (L) = X100 H (Initialize source memory pointer)
LXI D, Y100 H        ; (D) (E) = Y100 H (Initialize destination pointer)
MOV A, M             ; (M) --- (A) (Get the contents of memory to Accumulator)
STAX D               ; (A) --- ((D) (E)) (Store the contents of Accumulator
```

INX H	; Increment source memory pointer
INX D	; Increment destination memory pointer
DCR C	; Decrement the counter
JNZ BACK	; If counter ≠ 0 repeat
HLT	; Stop
Before execution:	After execution:
(X100) 00	(X100) 00
(X101) 01	(X101) 01
(X102) 02	(X102) 02
(X103) 03	(X103) 03

(X104) 04	(X104) 04
(X105) 05	(X105) 05
(X106) 06	(X106) 06
(X107) 07	(X107) 07
(X108) 08	(X108) 08
(X109) 09	(X109) 09
(Y100) XX	(Y100) 00
(Y101) XX	(Y101) 01
(Y102) XX	(Y102) 02
(Y103) XX	(Y103) 03
(Y104) XX	(Y104) 04
(Y105) XX	(Y105) 05
(Y106) XX	(Y106) 06
(Y107) XX	(Y107) 07
(Y108) XX	(Y108) 08
(Y109) XX	(Y109) 09

Exercise

1. What is the function of an accumulator?
2. Why are the program counter and the stack pointer called 16-bit registers?
3. What is the function of the WR signal on the memory chip?
4. How many address lines are used to identify an I/O port in the peripheral I/O and in memory-mapped I/O methods?
5. While executing a program, when the 8085 MPU completes the fetching of the machine code located at the memory address 2057 H, what is the content of the program counter?
6. Explain the need to demultiplex the bus AD_7-AD_0.
7. Explain the functions of ALE and IO/M signals of the 8085 microprocessor.
8. If the clock frequency is 5 MHz, how much time is required to execute an instruction of 18 T-states?
9. Explain why a latch is used for an output port, but a tri-state buffer can be used for an input port.
10. What are the control signals necessary in the memory-mapped I/O?
11. Can the microprocessor differentiate whether it is reading from a memory-mapped input port or from memory?
12. Write a program using the ADI instruction to add two hexadecimal numbers 3BH and 67 H and to display the answer at an output port.
13. What operation can be performed by using the instruction ADD A?

14. Write instructions to
    (a) Load 15 H in the accumulator
    (b) Decrement the accumulator
    (c) Display the answer

    Specify the answer you would except at the output port.
15. Write a program to
    (a) Clear the accumulator
    (b) Add 47 H
    (c) Subtract 92 H
    (d) Add 64 H (using ADI instruction)
    (e) Display the results after subtracting 92 H and after adding 47 H.
16. What operation can be performed by using the instruction XRA A? Specify the status of Z and CY.
17. Write instructions to clear the CY flag, to load number FFH in register B, and increment (B). If the CY flag is set, display 01 at the output port; otherwise, display the contents of register B. Explain your results.
18. Specify the address of the output port, and explain the type of numbers that can be displayed at the output port.

```
         MVI A, BYTE1        ; Get a data byte
         ORA A               ; Set flags
         JP OUTPRT           ; Jump if the byte is positive
         XRA A
OUTPRT:  OUT F2H
         HLT
```

19. Explain the interrupts of 8085 with suitable diagram. Also explain INTR CALLLocation in 8085.
20. What are the main maskable and non-maskable interrupts.
21. Compare RST 7.5, 6.5, 5.5 and TRAP and INTR.
22. Explain the main function of STACK and Subroutine in 8085.
23. Write a short note on:
    (a) Generating the control signal.
    (b) Demultiplexing and data signal bus.
24. What is microprocessor and what are its basic units?
25. What is a bus? Why data bus is bidirectional?
26. Define the terms—machine cycle, T-state, instruction cycle, fetch and execute cycle.
27. What does memory mapping means?
28. What are the instructions used to control interrupts?
29. What are different types of interrupts and what are the types of hardware interrupts?
30. Difference between memory-mapped I/O and I/O mapped I/O?

31. Describe the function of the following pins in 8085?
    (a) READY (b) HOLD (c) SID and SOD
32. Comparison between full address decoding and partial address decoding?
33. What is the need of timing diagrams?
34. Give some examples of port devices used in 8085 microprocessor based system.
35. What operation is performed during first T-state of every machine cycle in 8085?
36. What is interrupt acknowledge cycle?
37. What are vectored and non-vectored interrupt?
38. List software and hardware interrupts of 8085. What is TRAP?
39. How clock signals are generated in 8085 and what is the frequency of internal clock?
40. When the 8085 checks for an interrupt?
41. Why interfacing is needed for I/O devices?

Chapter 4

Intel 8086 Microprocessor

4.1 INTRODUCTION

In the last chapter, we discussed 8-bit microprocessor and also studied the basic idea of microprocessors and associated devices. In this chapter, we will discuss in detail Intel's 8086 microprocessor. The Intel's 8086 microprocessor is a 16-bit, N-channel, HMOS microprocessor. It has a 20-bit address bus, so it can address any one of $2^{20} \cong 1$ MB memory locations. Its clock frequencies for its different versions are: 5,8 and 10 MHz. It was introduced in 1978. It is built on a single semiconductor chip and packaged in a 40-pin DIP(Dual In-Line Package). In this chapter, we will study the internal organization of 8086, bus interface unit, execution unit, register organization, memory organization and its bus cycle.

4.2 INTERNAL ARCHITECTURE (8086)

The architecture of 8086 microprocessor provides a number of improvements over 8085 architecture. It supports a 16-bit ALU, a set of 16-bit registers and provides segmented memory addressing capability, a rich instruction set, powerful interrupt structure, fetched instruction queue for overlapped fetching and execution, etc.

The internal block diagram, shown in Figure 4.1, describes the overall organization of different units inside the chip.

The complete architecture of 8086 microprocessor can be divided into two parts, the first part is Bus Interface Unit (BIU) and the second part is Execution Unit (EU). The bus interface unit contains the circuit for physical address calculations and a pre-decoding instruction byte queue (6 bytes long). The bus interface unit makes the system bus signals available for external interfacing of the devices. In other words, this unit is responsible for establishing communications with external devices and peripherals including memory via the bus. The complete physical address which is 20-bit long is generated using segment and offset registers, each 16-bit long. Thus, the segment addressed by the segment value 1005H can have offset values from 0000H to FFFFH within it, i.e., maximum 64 K locations may be accommodated in the segment. The segment register indicates the base address of a particular segment, while the offset indicates the distance of the required memory location in the segment from the base address. Since the offset is a 16-bit number, each segment can have a maximum of 64 K(2^{16}=64k) locations as shown in Figure. 4.2. The bus interface unit has a separate adder to perform this procedure for obtaining a physical address. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP or an immediate 16-bit value, depending upon the addressing mode.

Addresses within a segment can range from address 0000H to address 0FFFFH. This corresponds to the 64 K-byte length of the segment. An address within a segment is called an offset or logical address. A logical address gives the displacement from the address base of the segment to the desired location within it, as opposed to its "real" address, which maps directly anywhere into the 1 MB memory space. This "real" address is called the physical address. Figure 4.3 shows the generation of physical address from offset address and segment address.

In case of 8085 microprocessor, once the opcode is fetched and decoded, the external bus remains free for some time, while the processor internally executes the instruction. This time slot is utilized in 8086 to achieve the overlapped fetch and execution cycles. While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next

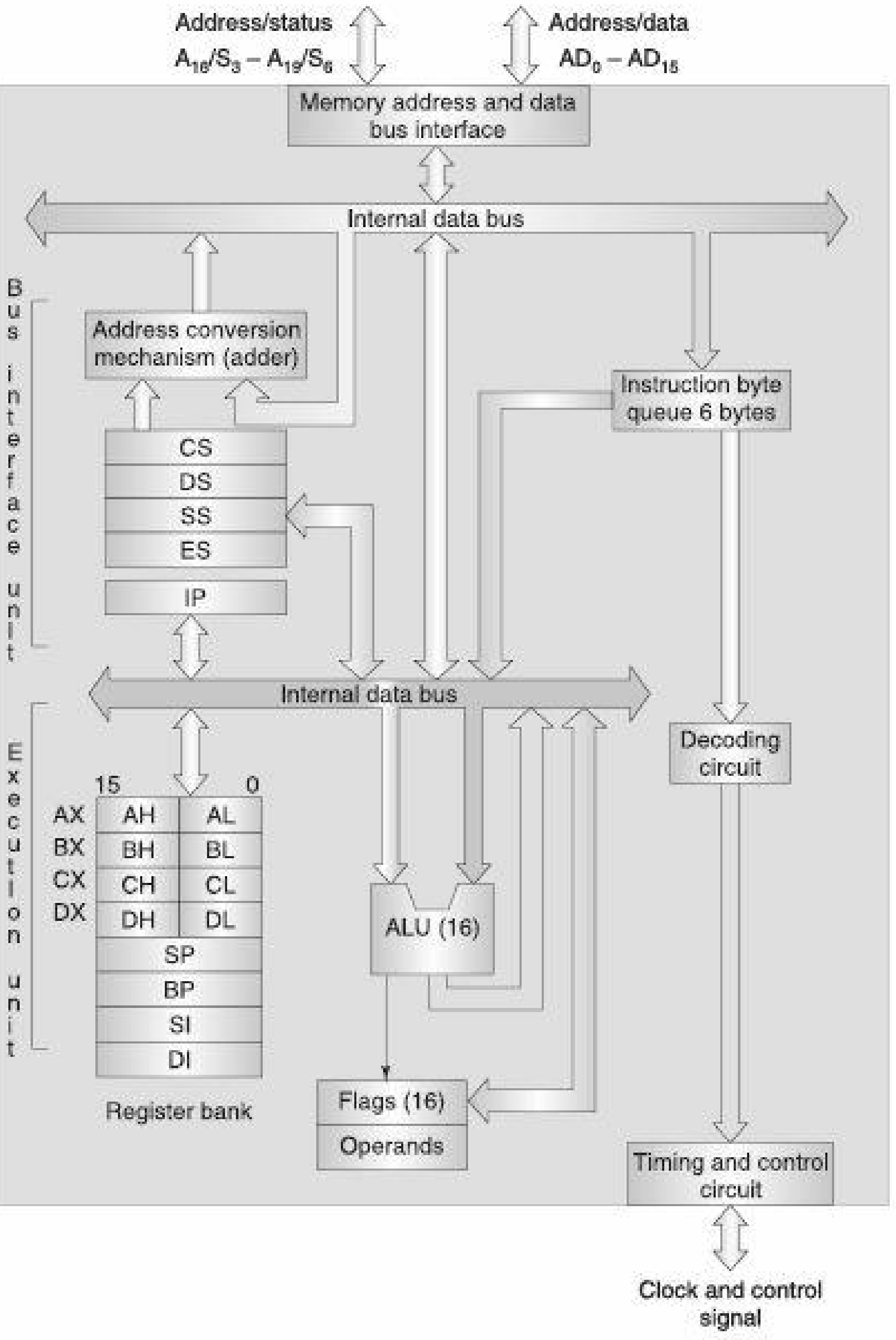


Fig. 4.1 8086 Internal Architecture.

instruction and arrange it in a queue called pre-decoded instruction byte queue. It is 6-byte long, first-in first-out structure. The instructions from the queue are taken for decoding sequentially.

Once a byte is decoded, the queue is rearranged by pushing it out and the queue status is checked for the possibility of the next opcode fetch cycle. While the opcode is fetched by the

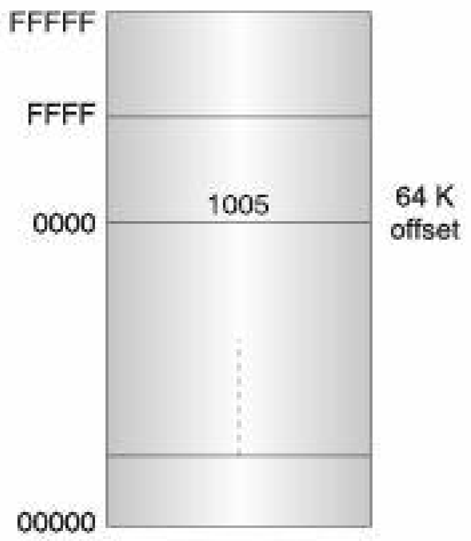


Fig. 4.2 Offset and Segment Address.

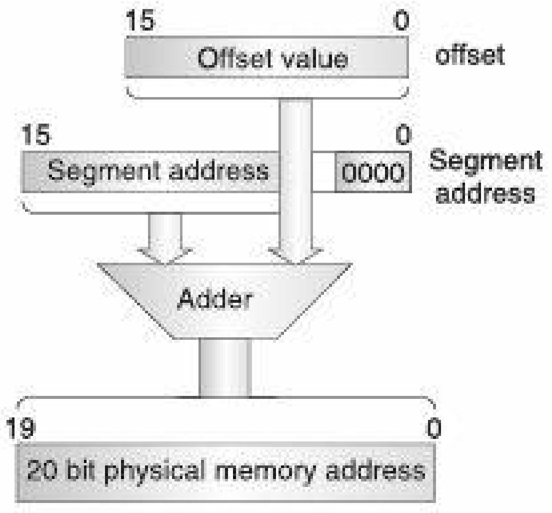


Fig. 4.3 Physical and Logical Address.

bus interface unit (BIU), the execution unit (EU) executes the previously decoded instruction concurrently. The BIU along with the execution unit (EU) thus forms a pipeline. The bus interface unit thus manages the complete interface of execution unit with memory and I/O devices, of course, under the control of the timing and control unit. The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit drives the necessary control signals to execute the instruction opcode received from the queue, depending upon the information made available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in memory.

4.3. EXTERNAL SYSTEM BUS ARCHITECTURE

8086 is 16-bit processor with 40 pins. It has 20 address pins and out of which 16 pins are used as data pins. This concept of using same pins for both address and data is called multiplexing. It has 16 control signals. It can access a memory of 1 MB (2^{20}). It has 14 registers which are 16 bits wide. There are a set of arithmetic registers, set of pointers (Base and Index registers), set of segment registers. It has program status word (PSW) or Flag register, an Instruction pointer.

Instruction queue: It can queue 6 bytes at a time.

4.3.1 Execution Unit (EU) and Bus Interface Unit (BIU)

8086 microprocessor consists of two parts called EU and BIU. These two units can work in parallel. This improves speed of execution. BIU fetches instruction and places them in instruction queue. Execution unit decodes and executes the instruction. When EU is executing an instruction, BIU can fetch the next instruction (Fig. 4.4).

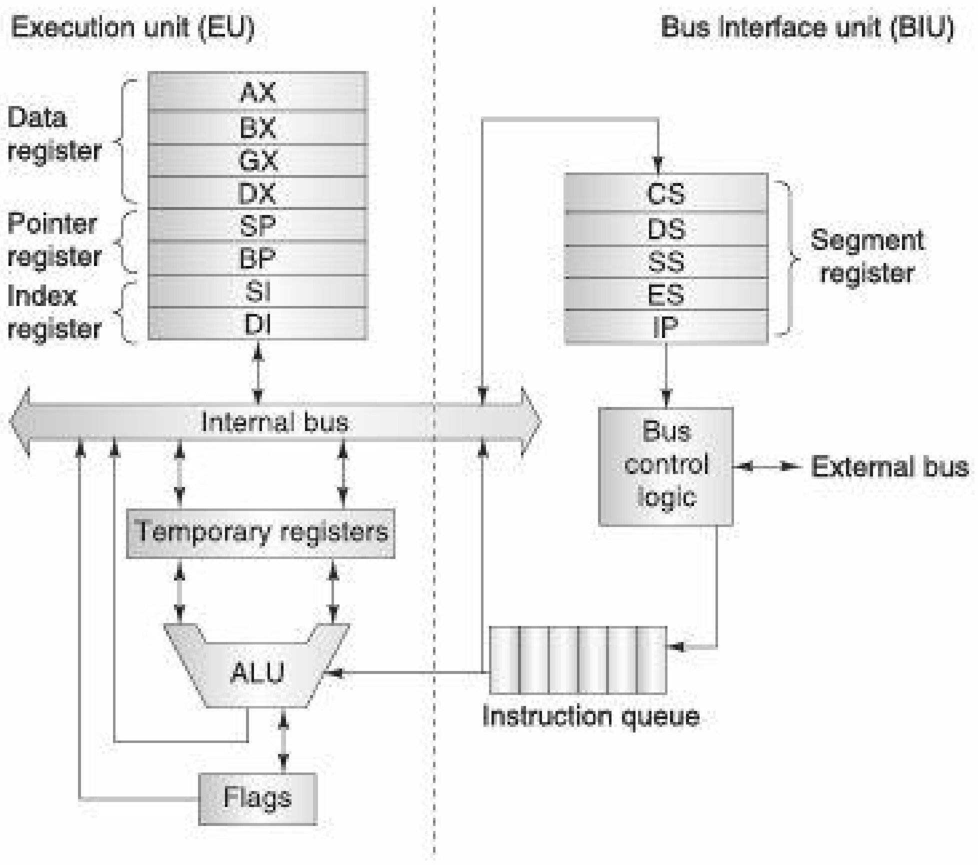


Fig. 4.4 8086's EU and BIU.

4.3.2 Register Organization of 8086 Microprocessor

8086 microprocessor has a powerful set of registers that are general-purpose and special-purpose registers. All the registers of 8086 are 16-bit registers. The general-purpose registers can be used as either 8-bit registers or 16-bit registers. The general-purpose registers are either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes, etc. The special-purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. We will categorize the register set into four groups, as follows.

4.3.2.1 General-purpose Registers

Figure 4.5 shows the register organization of 8086. The registers AX, BX, CX and DX are the general-purpose 16-bit registers. AX is used as 16-bit accumulator, with the lower 8 bits of AX designated as AL and higher 8 bits as AH. AL can be used as 8-bit accumulator for 8-bit operations. This is the most important general-purpose register having multiple functions, which will be discussed later.

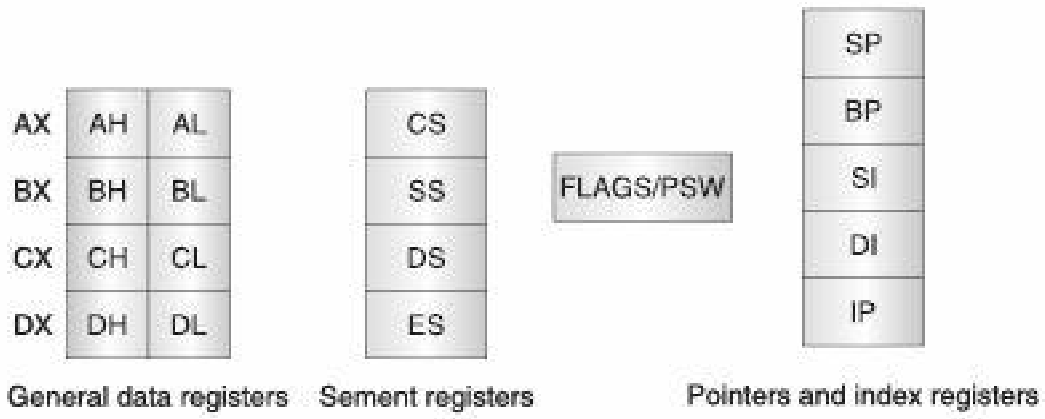


Fig. 4.5 Register Organization of 8086.

Usually, the letters L and H specify the lower and higher bytes respectively of a particular register. For example, CH means the higher 8-bit of the CX register and CL means the lower 8 bits of the CX register. The letter X is used to specify the complete 16-bit register. The register CX is also used as a default counter in case of string and loop instructions. The register BX is used as 16-bit offset storage for forming physical addresses in case of certain addressing modes. DX register is a general-purpose register which may be used as an implicit operand or destination in case of a few instructions. BX may be used as base register in address calculations.

4.3.2.2 Segment Registers

Unlike 8085, the 8086 microprocessor addresses a segmented memory. Memory segmentation allowed the 8086 microprocessor to be able to access only four 64 Kbyte segmentation within its 1Mbyte memory range at any given time. These four segments are defined by the data values stored in four segment registers in 8086 as given below.

1. Code Segment Register (CS)
2. Data Segment Register (DS)
3. Extra Segment Register (ES)
4. Stack Segment Register (SS)

The code segment register is used for addressing a memory location in the code segment of the memory, where the executable program is stored. Similarly, the data segment register points to the data segment of the memory, where the data is resided. The extra segment also refers to a segment which essentially is another data segment of the memory. Thus, the extra segment also contains data. The stack segment register is used for addressing stack segment of memory. The stack segment is that segment of memory which is used to store stack data. The CPU uses the stack for temporarily storing important data, e.g., the contents of the CPU registers which will be required at a later stage. The stack grows down, i.e., the data is pushed onto the stack in the memory locations with decreasing addresses. When this information will be required by the CPU, they will be popped off from the stack.

While addressing any location in the memory bank, the physical address is calculated from two parts, the first is segment address and the second is offset. The segment registers contain16-bit segment base addresses, related to different segments. There are specific (index) Registers. that contain the offsets for a specific segment. The advantage of this scheme is that in place of maintaining a 20-bit register for a physical address, the processor just maintains two 16-bit registers which are within the word length capacity of the machine. Thus, the CS, DS, SS and ES segment registers respectively contain the segment addresses for the code, data, stack and extra segments of memory. It may be noted that all these segments are the logical segments. They may or may not be physically separated. In other words, a single segment may require more than one memory chip or more than one segment may be accommodated in a single memory chip.

4.3.2.3 Pointers and Index Registers

The pointers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets within the code, data and stack segments respectively. The index registers are 16-bit SI (Source Index) and 16-bit DI (Destination Index) registers and 16-bit BP (Base Pointer) registers can be used for temporary storage as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. BP is used to provide indirect access to data in stack registers. The register SI is generally used to store the offset of source data in data segment while the register DI is used to store the offset of destination in data or extra segment. The index registers are particularly useful for string manipulations.

4.4 MEMORY SEGMENTATION

The memory in an 8086/8088 based system is organized as segmented memory. In this scheme, the complete physically available memory may be divided into a number of logical segments. Each segment is 64 Kbytes in size and is addressed by one of the segment registers. The 16-bit contents of the segment register actually point to the starting location of a particular segment. To address a specific memory location within a segment, we need an offset address. The offset address is also 16-bit long so that the maximum offset value can be FFFFH, and the maximum size of any segment is thus 64 K locations.

To emphasize this segmented memory concept, we will consider an example of a housing colony containing say, 100 houses. The simplest method of numbering the houses will be just to assign the numbers from 1 to 100 to each house sequentially. Suppose, now, if one wants to find out house number 67, then he will start from house number 1 and go on till he finds the house, numbered 67. Consider another case where the 100 houses are arranged in the 10 × 10 (rows × columns) pattern. In this case, to find out house number 67, one will directly go to the 6th row and then to the 7th column. In the second scheme, the efforts required for finding the same house will be less. This second scheme in our example is analogous to the segmented memory scheme, where the addresses are specified in terms of segment addresses analogous to rows and offset addresses analogous to columns.

The CPU of 8086 microprocessor is able to address 1 MB of physical memory. The complete 1 MB memory can be divided into 16 segments, each of 64 KB size. The addresses of the segments may be assigned 0000H to F000H. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.

4.4.1 Need for Segmentation

Intel designed the 8086 family devices to access memory using the segment: offset approach rather than accessing memory directly with 20-bit addresses. The segment: offset, scheme requires only a 16-bit number to represent the base address for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.

Segmentation makes it easy to keep users' programs and data separate from one another, and switch from one user's program to another user's program. For example, in a timesharing system, several users share a CPU. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy.

4.4.2 Overlapping and Non-overlapping Segments

(i) **Non-overlapping Segments:** If the complete 1 MB memory is divided into 16 segments, each of 64 Kbytes size then it is called as non-overlapping segmentation. The addresses of the segment may be assigned as 0000H to F000H respectively and the offset values are from 0000H to FFFFH.

(ii) **Overlapping Segments:** In some cases segment may overlap also. Suppose a segment starts at a particular address and its maximum size can go up to 64 Kbytes. But if another segment starts before this 64 Kbytes location of the first segment, the two segments are said to be overlapping segment.

The area of memory from the start of the second segment to the possible end of the first segment is called as overlapped segment. Non-overlapping segments and overlapping segments are shown in the Fig. 4.6.

(iii) **Visualizing the Overlapping Segments:** Segments are more like a mental construct or a way of visualizing a computer's memory, rather than being closely tied to the physical hardware itself. Figure 4.7 (a) shows how each segment of 65,536 bytes overlaps most of the preceding segment. Each segment begins only 16 bytes (or a paragraph) after the preceding one. For every 16 bytes higher in memory that we point to, the number of overlapping segments will increase by one until we arrive at the end of the first segment. At that point, each successive paragraph (16 bytes) of memory (up to 1MB) has a constant number of 4,096 overlapping segments. Figure also shows the segment: offset values for each of the four corners of the first five of segments.

Segment No.	*Absolute memory location*
0	00H to 0FFFFH (0 to 65,535)
1	10H to 1000FH (16 to 65,551)
2	20H to 0FFFFH (32 to 65,567)
3	30H to 0FFFFH (48 to 65,583)
4	40H to 0FFFFH (64 to 65,599)

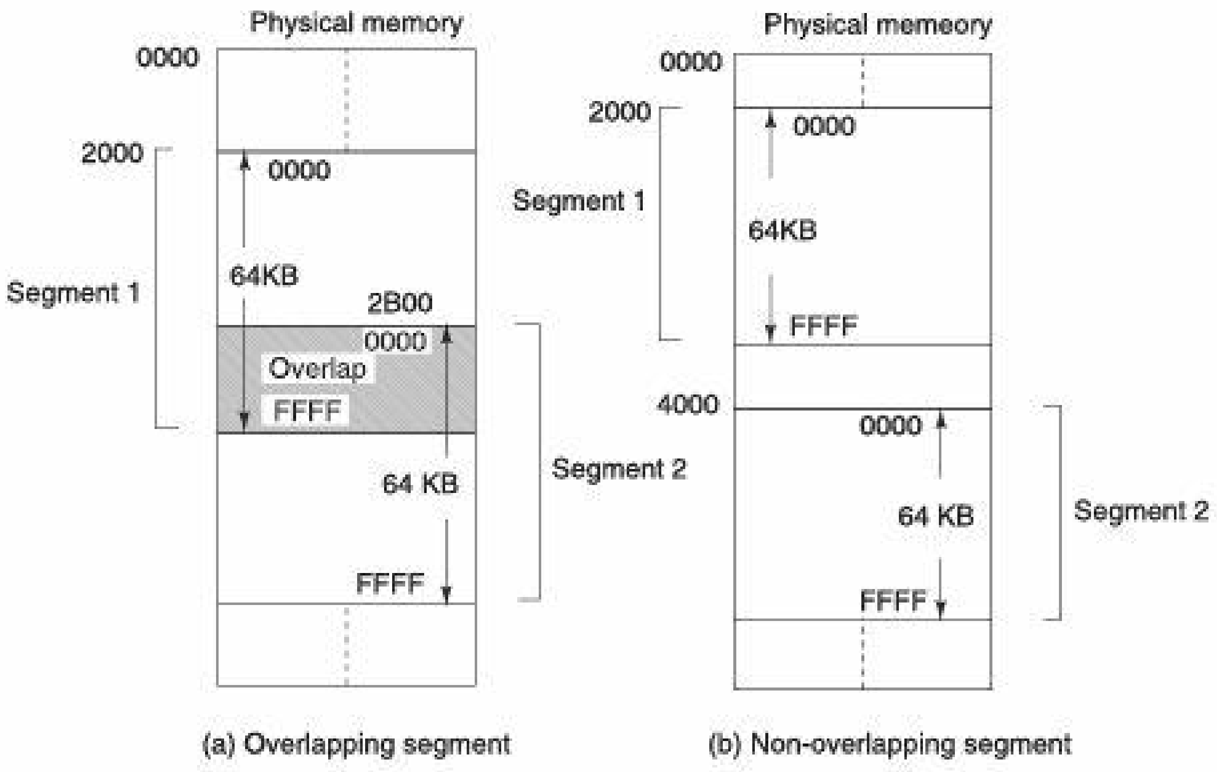


Fig. 4.6 Overlapping Segments and Non-overlapping Segments.

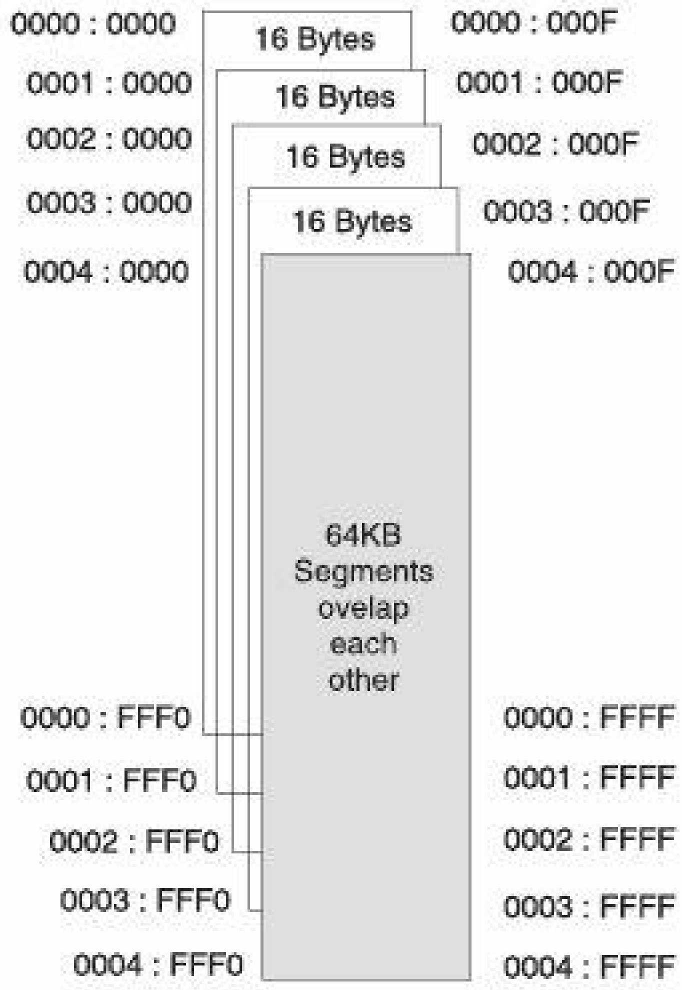


Fig. 4.7(a) Overlapping Segments.

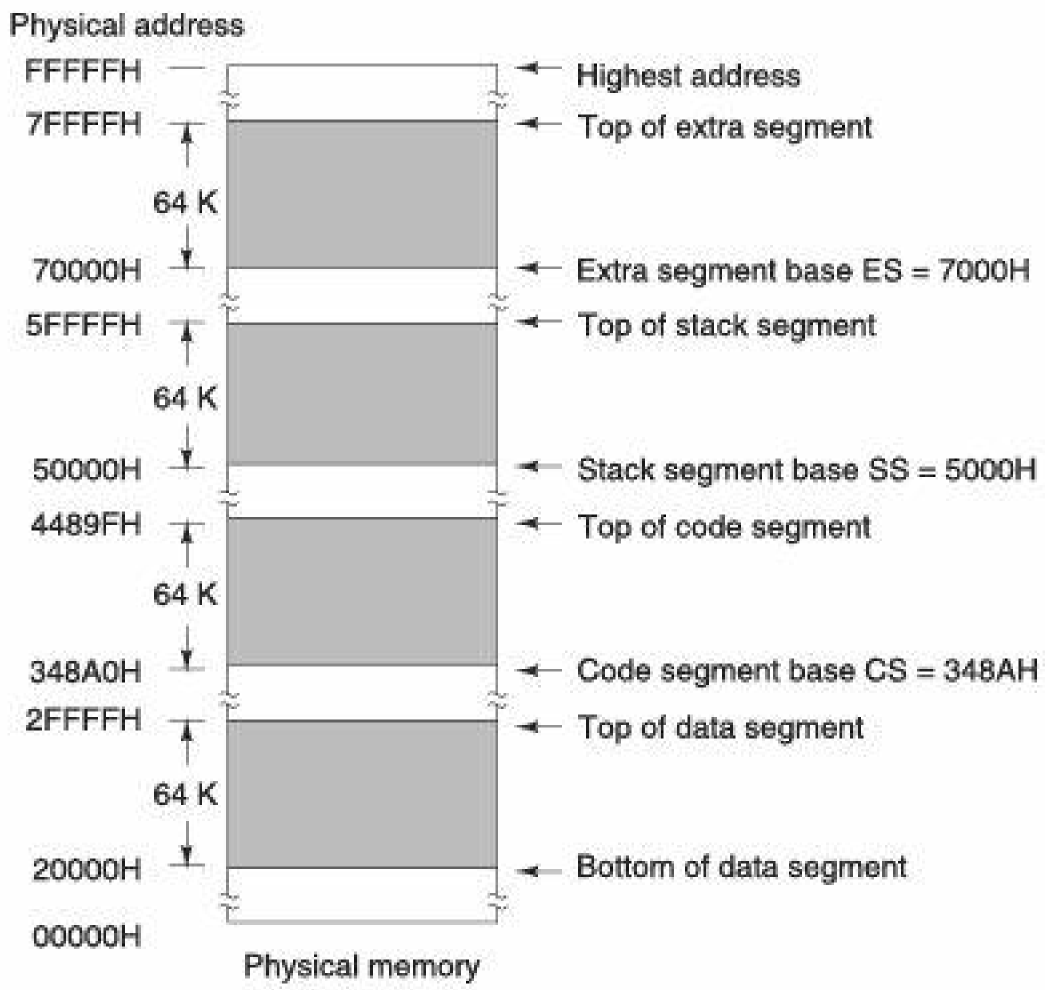


Fig. 4.7(b) Memory Segmentation of 8086.

(iv) **Advantages of memory segmentation:** The main advantages of the segmented memory scheme are as follows:

(1) Allow the memory capacity to be 1Mb even though the addresses associated with the individual instructions are only 16 bits wide.
(2) Facilitate the use of separate memory areas for the program, its data and the stack.
(3) Permit a program and/or its data to be put into different areas of memory each time the program is executed i.e. provision for relocation may be done.
(4) Multitasking becomes easy.

4.4.3 Segment Registers

Within the 1 MB of memory space the 8086/88 defines four 64 K byte memory blocks called the code segment, stack segment, data segment, and extra segment. Each of these blocks of memory is used differently by the processor. The code segment holds the program instruction codes. The data segment stores data for the program. The extra segment is an extra data segment (often used for shared data). The stack segment is used to store interrupt and subroutine return addresses.

The four segment registers (CS, DS, ES, and SS) are used to "point" at location 0 (the base address) of each segment. This is a little "tricky" because the segment registers are only 16 bits wide, but the memory address is 20 bits wide. The BIU takes care of this problem by appending four 0's to the low-order bits of the segment register. In effect, this multiplies the segment register contents by 16. Memory locations not defined to be within one of the current segments cannot be accessed by the 8086/88 without first redefining one of the segment registers to include that location. Thus, at any given instant a maximum of 256 K (64 K * 4) bytes of memory can be utilized. An immediate advantage of having separate data and code segments is that one program can work on several different sets of data. This is done by reloading register DS to point to the new data. Perhaps the greatest advantage of segmented memory is that programs that reference logical addresses only can be loaded and run anywhere in memory. This is because the logical addresses always range from 00000H to 0FFFFh, independent of the code segment base. Such programs are said to be re-locatable, meaning that they will run at any location in memory. The requirements for writing re-locatable programs are that no references be made to physical addresses, and no changes to the segment registers are allowed.

(i) **Code segment (CS):** It is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

(ii) **Stack segment (SS):** It is a 16-bit register containing address of 64 KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

(iii) **Data segment (DS):** It is a 16-bit register containing address of 64 KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

(iv) **Extra segment (ES):** It is a 16-bit register containing address of 64 KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS, DS or ES prefix.

4.4.4 Segment Offset Addressing

There are often many different Segments: offset pairs which can be used to address the same location in computer's memory. This scheme is a relative way of viewing computer memory as opposed to Linear or Absolute addressing scheme.

Segment: Offset addressing was introduced at a time when the largest register in a CPU was only 16-bit long which meant it could address only 65,536 bytes (64 KB) of memory, directly. But there was demand for higher memory capacity. Then rather than create a CPU with larger register sizes designers at Intel decided to keep the 16-bit registers for their new 8086 CPU and added a different way to access more memory. They expanded the instruction set, so programs could tell the CPU to group two 16-bit registers together whenever they needed to refer to an Absolute memory location beyond 64 KB.

If the designers had allowed the CPU to combine two registers into a high and low pair of 32-bits, it could have referenced up to 4 GB of memory in a linear fashion. They created the Segment: Offset scheme which allows a CPU to effectively address about 1 MB of memory.

The value in any segment register is multiplied by 16 (or shifted one hexadecimal byte to the left; add an extra 0 to the end of the hex number) and then the value in an Offset register is added to it. So, the Absolute address for any combination of Segment and Offset pairs is found by using the formula:

Absolute Memory Location = (Segment value * 16) + Offset value

The Absolute or Linear address for the Segment: Offset pair, F000: FFFD can be computed quite easily by simply inserting a zero at the end of the Segment value (which is the same as multiplying by 16) and then adding the Offset value:

F	0	0	0	0	
+	F	F	F	D	
F	F	F	F	D	Or 1,048,573(decimal)

Here's another example: 923F:E2FF

9	2	3	F	0	
+	E	2	F	F	
A	0	6	E	F	Or 657,135 (decimal)

Now let's compute the Absolute Memory location for the largest value that can be expressed using a Segment: Offset reference:

F	F	F	F	0	
+	F	F	F	F	
1	0	F	E	F	Or 1,114,095 (decimal)

In reality, it wasn't until quite some time after the 8086, that such a large value actually corresponded to a real Memory location. Once it became common for PCs to have over 1MB of memory, programmers developed ways to use it to their advantage and this last byte became part of what's now called the HMA (High Memory Area).

One of the downsides in using Segment: Offset pairs is the fact that a large number of these pairs refer to the same exact memory locations. For example, every Segment: Offset pair below, refers to exactly the same location in memory:

0007:7B90	0008:7B80	0049:7770	004A:7760
0047:7790	0048:7780	0079:7470	007A:7460
0077:7490	0078:7480	0201:5BF0	0202:5BE0
01FF:5C10	0200:5C00	07BD:0030	07BE:0020
07BB:0050	0009:7B70	000A:7B60	000B:7B50

Note: The Segment: Offset pairs listed above are only some of the many ways one can refer to the single Absolute Memory location of: 7C00h (or 0000:7C00).

(i) Normalized Segment: Offset Notation

Since there are so many different ways that a single byte in Memory might be referenced using Segment: Offset pairs, most programmers have agreed to use the same convention to normalize all such pairs into values that will always be unique. These unique pairs are called Normalized Address or Pointer.

By confining the Offset to just the Hex values 0H through FH (16 hex digits); or a single paragraph and setting the Segment value accordingly, we have a unique way to reference all Segment: Offset Memory pair locations. To convert an arbitrary Segment: Offset pair into a normalized address or pointer is a two-step process that's quite easy for an assembly programmer:

1. Convert the pairs into a single physical (linear) address.
2. Then simply insert the colon (:) between the last two hex digits!

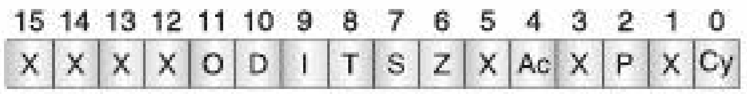
For example, in order to normalize 1000:1B0F, the steps are:

$$1000{:}1B0F \rightarrow 11B0Fh \rightarrow 11B0{:}F \text{ (or } 11B0{:}000F)$$

Since the normalized form will always have three leading zero bytes in its Offset, programmers often write it with just the digit that counts as shown here:11B0:F (when we see an address like this, it's almost a sure sign that the author is using this normalized notation).

4.5 FLAG REGISTER (PROGRAM STATUS WORD)

The 8086 has a 16-bit flag register which is divided into two parts, first one is condition code or status flags and the second one is machine control flags. The condition code flag register is the lower byte of the 16-bit flag register along with the overflow flag. The condition code flag register is identical to 8085 flag register, with an additional overflow flag, which is not present in 8085. This part of the flag register of 8086 reflects the results of the operations performed by ALU. The control flag register is the higher byte of the flag register of 8086. It contains three flags, viz., direction flag (D), interrupt flag (I) and trap flag (T). The complete bit configuration of 8086 flag register is shown in Figure 4.8.



O	–	Overflow flag
D	–	Direction flag
I	–	Interrupt flag
T	–	Trap flag
S	–	Sign flag
Z	–	Zero flag
Ac	–	Auxiliary carry flag
P	–	Parity flag
Cy	–	Carry flag
X	–	Not used

Fig. 4.8 Flag Register of 8086.

The description of each flag bit is as follows:

S-SIGN FLAG: This flag is set, if the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

Z-ZERO FLAG: This flag is set, if the result of the computation or comparison performed by the previous instruction /instructions is zero.

P-PARITY FLAG: This flag is set to 1, if the lower byte of the result contains even number of 1's.

C-CARRY FLAG: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction. For example, when two numbers are added, a carry may be generated out of the most significant bit position. The carry flag, in this case, will be set to '1'. In case, no carry is generated; it will be '0'. Some other instructions also affect or use this flag and will be discussed later in this text.

T-TRAP FLAG: If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the trap interrupt service routine.

I-INTERRUPT FLAG: If this flag is set, the maskable interrupts are recognized by the CPU, otherwise, they are ignored.

D-DIRECTION FLAG: This flag is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto- incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

AC-AUXILIARY CARRY FLAG: This flag is set, if there is a carry from the lowest nibble, i.e., bit D_3, during addition or borrow for the lowest nibble, i.e., bit D_3, during subtraction.

O-OVERFLOW FLAG: This flag is set, if an overflow occurs, i.e., if the result of a signed operation is large enough to be accommodated in a destination register. For example, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e., the result is of more than 7-bit in size in case of 8-bit signed operations and more than 15-bit in size in case of 16-bit signed operations, then the overflow flag will be set.

4.6 PIN DESCRIPTION OF 8086 MICROPROCESSOR

The 8086 microprocessor is a 16-bit CPU available in three clock rates, i.e., 5, 8 and 10 MHz, packaged in a 40-pin DIP or plastic package. The 8086 operates in single processor or multiprocessor configurations to achieve high performance. The pin configuration of 8086 microprocessor is shown in Figure 4.9. Some of the pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode (multiprocessor mode) configuration. The 8086 signals can be categorized in three groups. The first group has the signals having common functions in minimum as well as maximum mode, the second group has the signals which have special functions for minimum mode and the third group has the signals having special functions for maximum mode.

The following signal descriptions are common for both the minimum and maximum modes:

AD_{15}–AD_0: These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T_1 state, while the data is available on the data bus during T_2, T_3, T_w and T_4. Here T_1, T_2, T_3, T_4 and T_w are the clock states of a machine cycle. T_w is a wait state. These

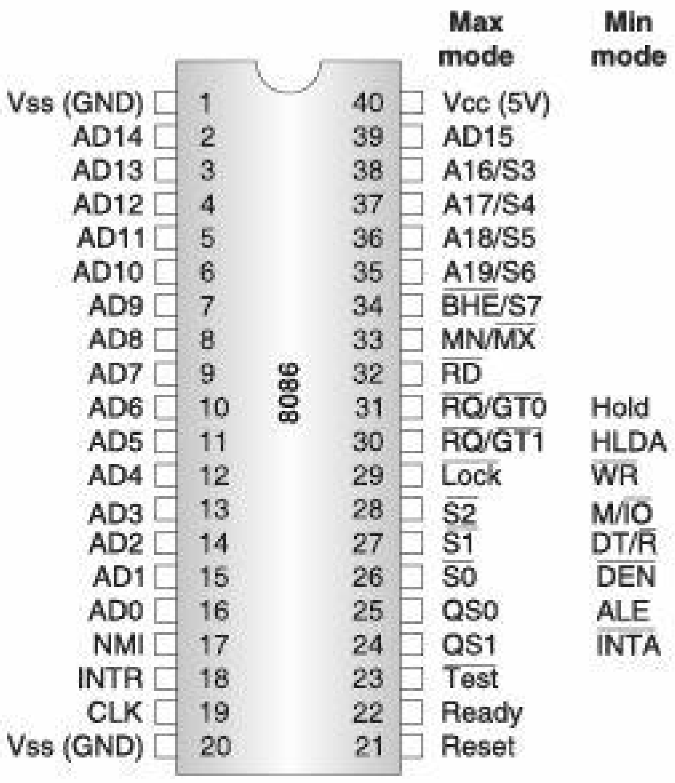


Fig. 4.9 Pin Diagram of 8086.

lines are active high and float to a tri-state during interrupt acknowledge and local bus hold acknowledge cycles.

A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3: These are the time multiplexed address and status lines. During T_1, these are the most significant address lines for memory operations. During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T_2, T_3, T_w and T_4. The status of the interrupt enable flag bit (displayed on S_5) is updated at the beginning of each clock cycle. The S_4 and S_3 combined indicate which segment register is presently being used for memory accesses as shown in Table 4.1. These lines float to tri-state off (tri-stated) during the local bus hold acknowledge.

Table 4.1 Segment Register used for Memory Access

$A_{17}/S4$	$A_{16}/S3$	*Indications*
0	0	Alternate Data (ES)
0	1	Stack (SS)
1	0	Code (CS) or none
1	1	Data (DS)

The status line S_6 is always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

$\overline{BHE}/S_7$-Bus High Enable/Status: High enable signal is used to indicate the transfer of data over the higher order (D_{15}-D_8) data bus as shown in Table 4.2. It goes low for the data transfers over D_{15}-D_8, and is used to derive chip selects of odd address memory bank or peripherals. $\overline{BHE}$ is low during T_1 for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on the higher byte of the data bus. The status information is available during T_2, T_3 and T_4. The signal is active low and is tri-stated during 'hold'. It is low during T_1 for the first pulse of the interrupt acknowledges cycle.

Table 4.2 Word/Byte Selection

$\overline{BHE}$	A_0	Indication
0	0	Whole word
0	1	Upper byte from or to odd address.
1	0	Lower byte from or to even address.
1	1	None

$\overline{\text{RD}}$-Read: Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. $\overline{\text{RD}}$ is active low and shows the state for T_2, T_3, T_w of any read cycle. The signal remains tri-stated during the 'hold acknowledge'.

READY: This signal is the acknowledgement from the slow devices or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The ready signal is active high.

INTR-Interrupt Request: It is a level triggered input. It is sampled during the last clock cycle of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. It can be internally masked by resetting the interrupt enable flag. The interrupt request signal is active high and internally synchronized.

$\overline{\text{TEST}}$: This input is examined by a 'WAIT' instruction. If the $\overline{\text{TEST}}$ input goes low, execution will continue, else, the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

NMI-Non-Maskable Interrupt: This is an edge-triggered input which causes a Type 2 interrupt. The NMI is not maskable internally by software. A transition from low to high initiates the interrupt response at the end of the current instruction. This input is internally synchronized.

RESET: This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronized.

CLK-Clock Input: The clock input provides the basic timing for processor operation and bus control activity. It is an asymmetric square wave with 33% duty cycle. The range of frequency for different 8086 versions is from 5 MHz to 10 MHz.

Vcc +5 V: Vcc is the power supply for the operation of the internal circuit.

GND: It is the ground for the internal circuit.

MN/$\overline{\text{MX}}$: The logic level at this pin decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode.

The following pin functions are for the minimum mode operation of 8086.

M/$\overline{\text{IO}}$-Memory/Input Output: This is a status line logically equivalent to S_2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous T_4 and remains active till final T_4 of the current cycle. It is tri-stated during local bus "hold acknowledge".

$\overline{\text{INTA}}$-interrupt Acknowledge: This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during T_2, T_3 and T_w of each interrupt acknowledge cycle.

ALE-Address Latch Enable: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tri-stated.

DT/$\overline{\text{R}}$-Data Transmit/Receive: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to S_1 in maximum mode. Its timing is the same as M/IO. This is tri-stated during 'hold acknowledge'.

$\overline{\text{DEN}}$-Data Enable: This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T_2 until the middle of T_4. DEN is tri-stated during 'hold acknowledge' cycle.

HOLD, HLDA-Hold/Hold Acknowledge: When the HOLD line goes high, it indicates to the processor that another device is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus (instruction) cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and it should be externally synchronized. If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T_4 provided:

1. The request occurs on or before T_2 state of the current cycle.
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address).
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode. The following pin functions are applicable for maximum mode operation of 8086.

$\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$-Status Lines: These are the status lines which reflect the type of operation, being carried out by the processor. These become active during T_4 of the previous cycle and remain active during T_1 and T_2 of the current bus cycle. The status lines return to passive state during T_3 of the current bus cycle so that they may again become active for the next bus cycle during T_4. Any change in these lines during T_3 indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in Table 4.3.

Table 4.3 Memory / I/O Read and Write Operations

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Indication
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive (No operation)

$\overline{\textbf{LOCK}}$: This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the $\overline{\text{LOCK}}$ signal is low. The $\overline{\text{LOCK}}$ signal is activated by the LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during "hold acknowledge".When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

QS_1, QS_0-Queue Status: These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table 4.4. This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus, even the largest (6-byte) instruction can be pre-fetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By pre-fetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as instruction pipelining.

Table 4.4 Status Indicated by QS_1, QS_0

QS_1	QS_0	Indication
0	0	No operation (Queue is an idle state)
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

At the starting the CS: IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS: IP address is odd or two bytes at a time, if the CS: IP address is even. The first byte is a complete opcode in case of some instructions (one byte opcode instruction) and it is a part of opcode, in case of other instructions (two-byte long opcode instructions), the remaining part of opcode may lie in the second byte. But invariably the first byte of an instruction is an opcode. These opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied. The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is single opcode byte, the next bytes are treated as data bytes depending upon the decoded instruction length, otherwise the next byte in the queue are treated as the second byte of the instruction opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions. The next byte, after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture (Figure. 4.1), there are two separate units, namely, execution unit and bus interface unit, while the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

The general operation of a computer consists of:

1. Fetching the next instruction from the address indicated by the PC.
2. Putting it in the instruction register and decoding it while the PC is incremented to point to the next instruction.
3. Executing the instruction and, if a branch is to be taken, resetting the PC to the branch address.
4. Repeating steps 1 through 3.

The operation of the 8086 follows this basic pattern, but there are differences and some of the operations may overlap.

$\overline{RQ}/\overline{GT}_0$, $\overline{RQ}/\overline{G}_1$-Request/Grant: These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with RQ/GT_0 having higher priority than RQ/GT_1.RQ/ GT pins have internal pull-up resistors and may be left unconnected. The request grant sequence is as follows:

1. A pulse of one clock wide from another bus master requests for the bus access to 8086.
2. During current or next clock cycle, a pulse, one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at next clock cycle. The CPU's bus interface unit is likely to be disconnected from the local bus of the system.

3. A one clock wide pulse from another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle.

Thus, each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus requests those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as discussed in case of HOLD and HLDA in minimum mode.

4.7 MEMORY AND INPUT/OUTPUT INTERFACE

General Bus Operation

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

Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T_1, T_2, T_3 and T_4. The address is transmitted by the processor during T_1. It is present on the bus only for one cycle. During T_2, i.e., the next cycle, the bus is tri-stated for changing the direction of bus for the following data read cycle. The data transfer takes place during T_3 and T_4. In case, an addressed device is slow and shows 'NOT READY' status, the wait states T_w are inserted between T_3 and T_4. These clock states during wait period are called idle states (T_i), wait states (T_w) or inactive states. The processor uses these cycles for internal housekeeping. The address latch enable (ALE) signal is emitted during T_1, by the processor (minimum mode) or the bus controller (maximum mode) depending upon the status of the MN/$\overline{\text{MX}}$ input. The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines $\overline{S_0}$, $\overline{S_1}$ and $\overline{S_2}$ are used to indicate the type of operation as discussed in the pin description of this unit. Status bits $\overline{\text{BHE}}/S_7$ are multiplexed with higher order address bits and the $\overline{\text{BHE}}$ signal. Address is valid during T_1 while the status bits S_3 to S_7 are valid during T_2 through T_4. Figure 4.10 shows a general bus operation cycle of 8086.

4.7.1 Minimum Mode 8086 System and Timings

In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/$\overline{\text{MX}}$ pin to logic 1. In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system. The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086. Transreceivers are the bidirectional buffers and sometimes they are called data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, namely, $\overline{\text{DEN}}$ and

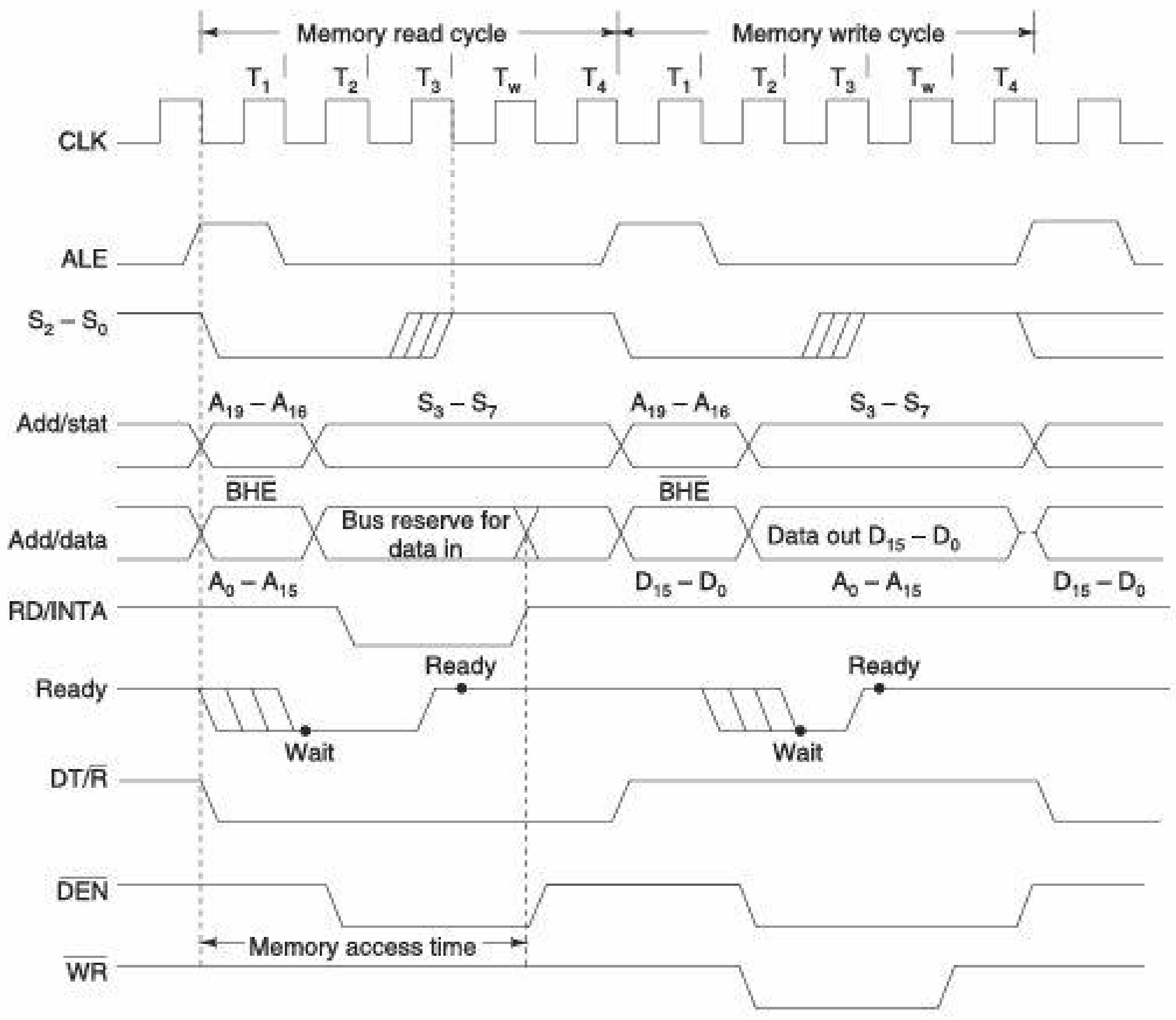


Fig. 4.10 General Bus Operation Cycle in Maximum Mode.

DT/$\overline{R}$. The $\overline{DEN}$ signal indicates that the valid data is available on the data bus, while DT/R indicates the direction of data, i.e., from or to the processor. The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users program storage.

A system may contain I/O devices for communication with the processor as well as some special-purpose I/O devices. A processor is in minimum mode when its MN/$\overline{MX}$ pin is strapped to +5 V. The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system. The clock generator also synchronizes some external signals with the system clock. The general system organization is shown in Figure 4.11.

Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation. The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations. The opcode fetch and read cycles are similar. Hence, the timing diagram can be categorized in two parts; the first is the timing diagram for

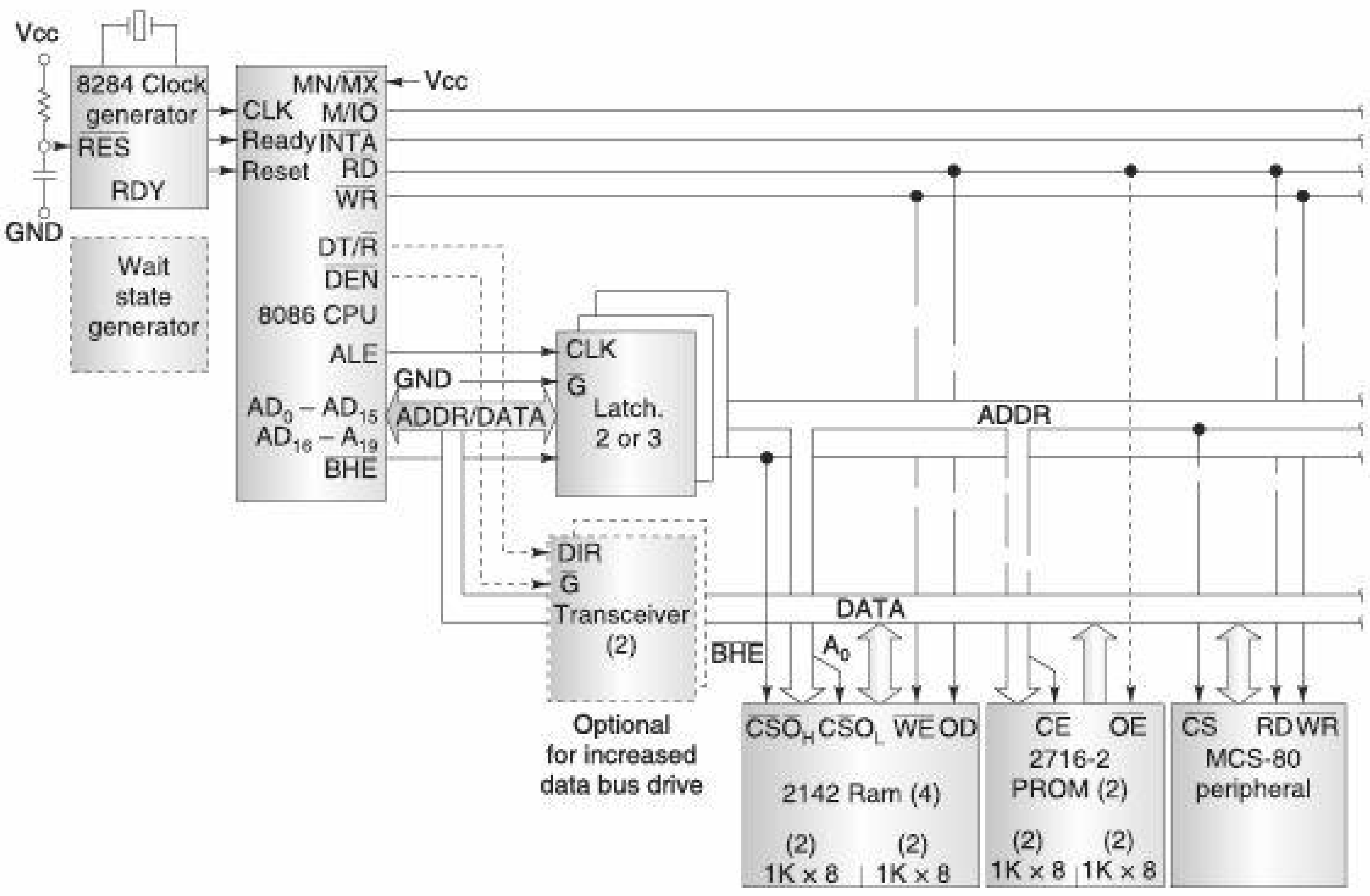


Fig. 4.11 Minimum Mode 8086 Typical Configuration.

read cycle and the second is the timing diagram for write cycle. The read cycle begins in T_1 with the assertion of the address latch enable (ALE) signal and also $M/\overline{IO}$ signal. During the negative going edge of this signal, the valid address is latched on the local bus.

The $\overline{BHE}$ and A_0 signals address low, high or both bytes. From T_1 to T_4, the $M/\overline{IO}$ signal indicates a memory or I/O operation. At T_2, the address is removed from the local bus and is sent to the output. The bus is then tri-stated. The read ($\overline{RD}$) control signal is also activated in T_2. The read ($\overline{RD}$) signal causes the addressed device to enable its data bus drivers. After $\overline{RD}$ goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tri-state its bus drivers. A write cycle also begins with the assertion of ALE and the emission of the address. The $M/\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. In T_1, after sending the address in T_2, the processor sends the data to be written to the addressed location. The data remains on the bus until middle of T_4 state. The WR becomes active at the beginning of T_2 (unlike $\overline{RD}$ is somewhat delayed in T_2 to provide time for floating). The $\overline{BHE}$ and I/O signals are used to select the proper byte or bytes of memory or I/O word to be read or written as already discussed in the signal description section of this chapter.

The $M/\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals indicate the types of data transfer as specified in Table 4.5.

Table 4.5 Memory/I/O Read, Write Operations

M/IO	RD	WR	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory read

Hold Response Sequence

The HOLD pin is checked at leading edge of each clock pulse. If it is received active by the processor before T_4 of the previous cycle or during T_1 state of the current cycle, the CPU activates HLDA in the next clock cycle and for the succeeding bus cycles, the bus will be given to another requesting master. The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock, as shown in Figure 4.12(c). The other conditions have already been discussed in the signal description section for the HOLD and HLDA signals.

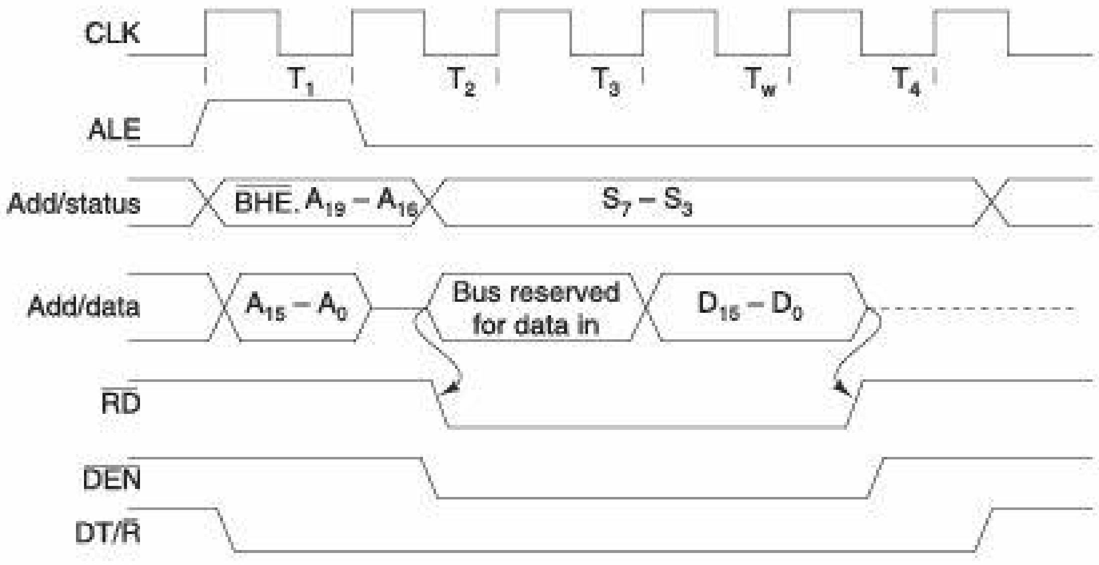


Fig. 4.12(a) Read Cycle Timing Diagram for Minimum Mode.

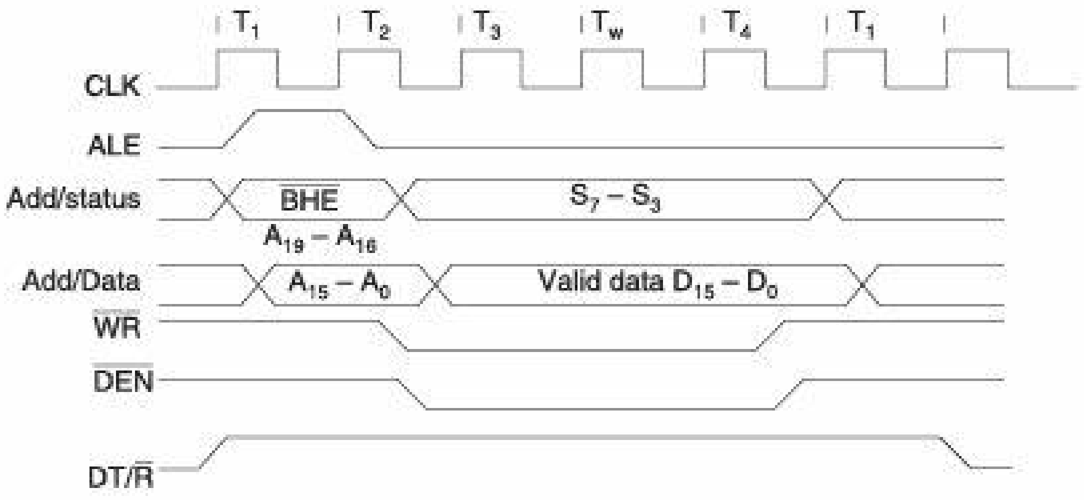


Fig. 4.12(b) Write Cycle Timing Diagram for Minimum Operation.

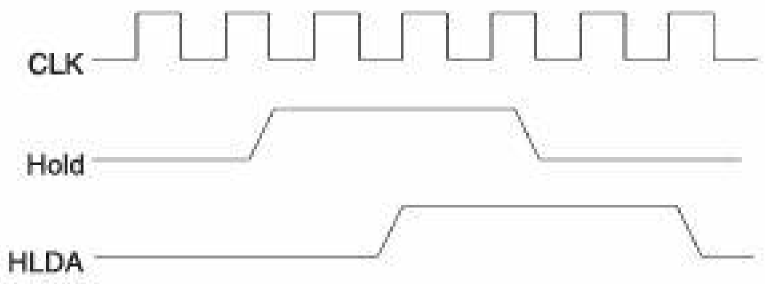


Fig. 4.12(c) Bus Request and Bus Grant Timings in Minimum Mode System.

4.7.2 Maximum Mode of 8086 System

In the maximum mode as shown in Figure 4.13, the 8086 is operated by strapping the MN/MX pin to ground. In this mode, the processor derives the status signal S_2, S_1, S_0. Another chip called bus controller derives the control signal using this status information. In the maximum mode, there may be more than one microprocessors in the system configuration. The components in the system are same as in the minimum mode system. The basic function of the bus controller chip IC8288, is to derive control signals like $\overline{RD}$ and $\overline{WR}$ (for memory and I/O devices), $\overline{DEN}$, DT/$\overline{R}$, ALE, etc. using the information by the processor on the status lines.

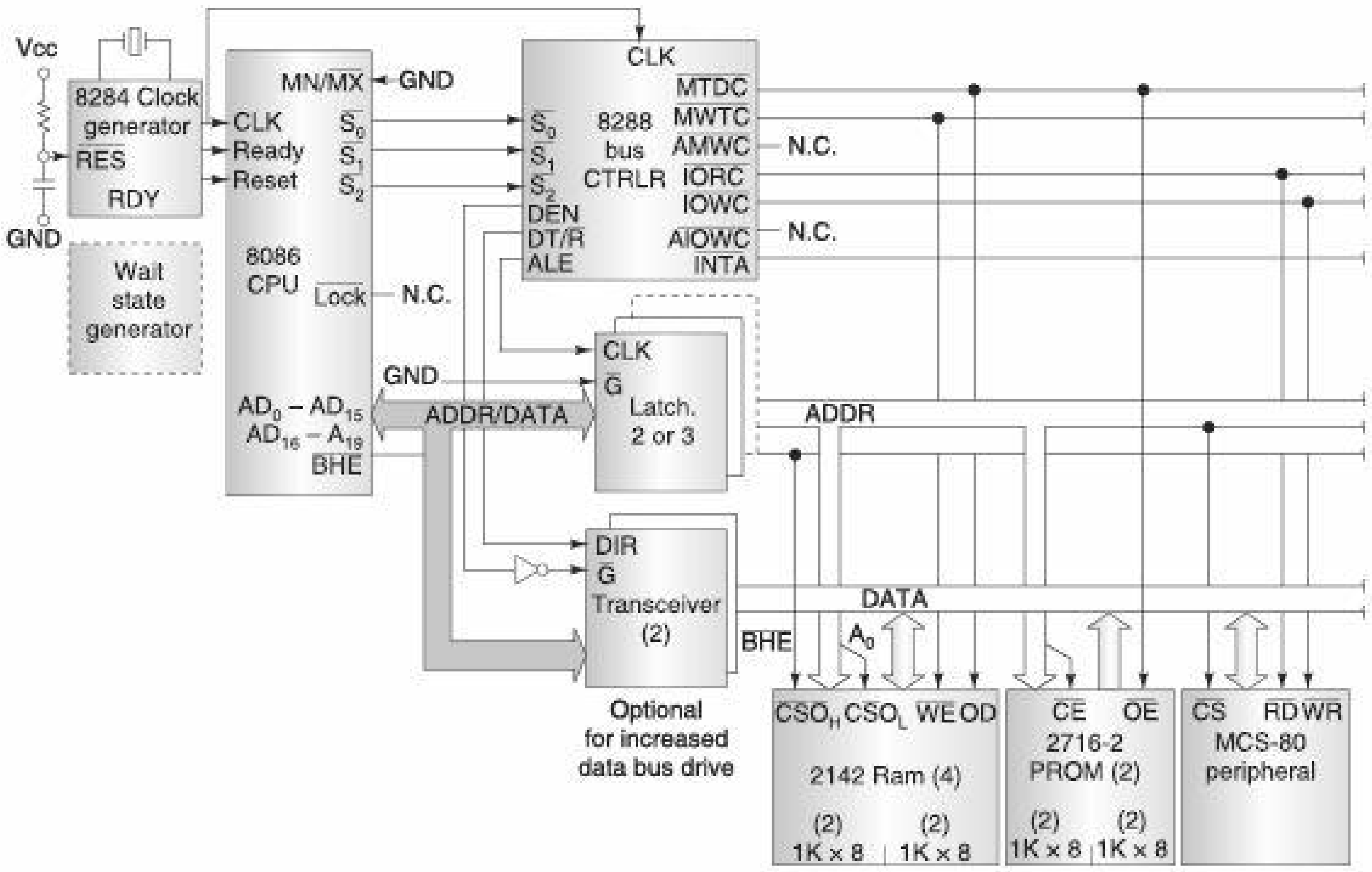


Fig. 4.13(a) Maximum Mode 8086 Microprocessor System.

The bus controller chip has input lines S_2, S_1, S_0 and CLK. These inputs to 8288 microprocessor are driven by CPU. It derives the outputs ALE, $\overline{DEN}$, DT/$\overline{R}$, $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$,

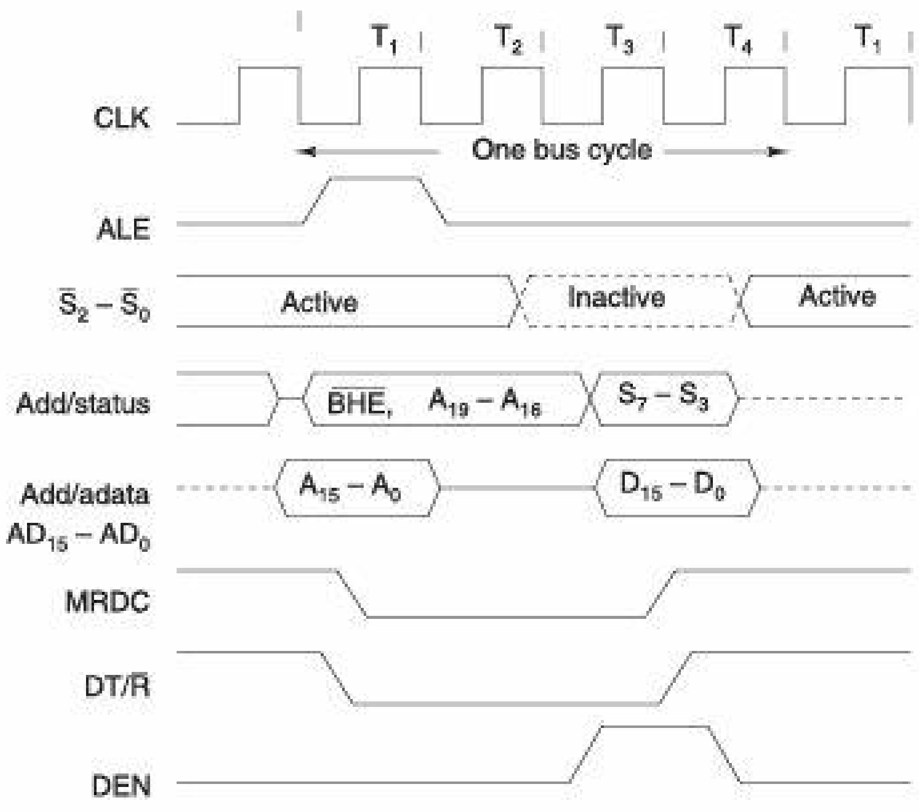


Fig. 4.13(b) Memory Read Timing in Maximum Mode.

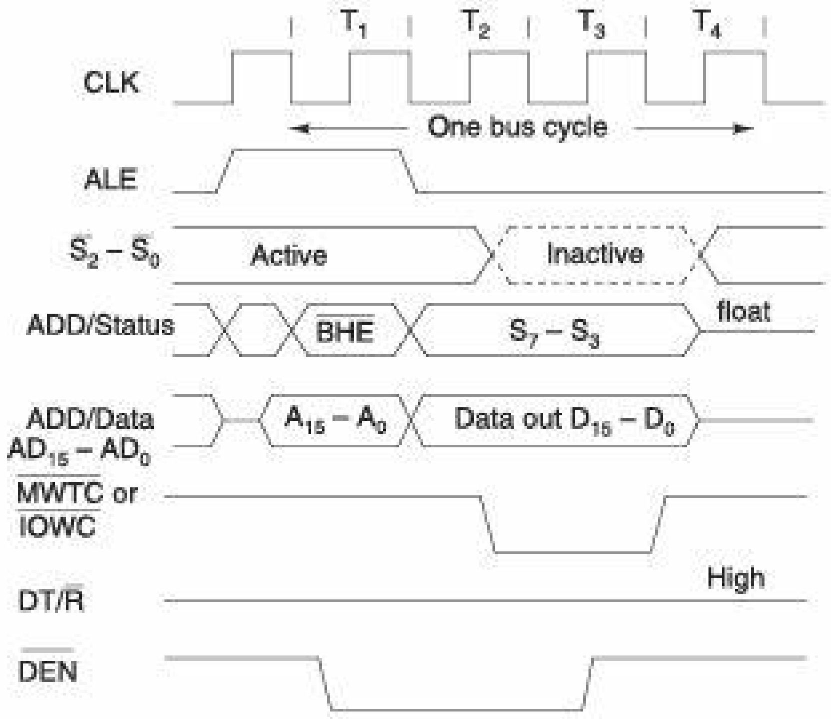


Fig. 4.13(c) Memory Write Timing in Maximum Mode.

$\overline{\text{IOWC}}$ and $\overline{\text{AIOWC}}$. The AEN, IOB and CEN pins are specially useful for multiprocessor systems. AEN and IOB are generally grounded. CEN pin is usually tied to +5 V. The significance of the MCE/PDEN output depends upon the status of the IOB pin. If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations. $\overline{\text{INTA}}$ pin used to issue two interrupt acknowledge pulses to the

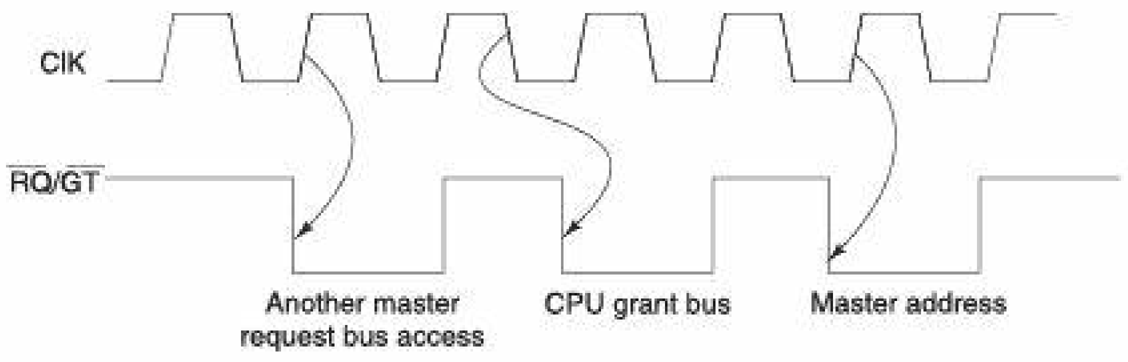


Fig. 4.13(d) RQ/GT Timings in Maximum Mode.

interrupt controller or to an interrupting device. $\overline{IORC}$, $\overline{IOWC}$ are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the address port. The $\overline{MRDC}$, $\overline{MWTC}$ are memory read command and memory write command signals respectively and may be used as memory read or write signals. All these command signals instruct the memory to accept or send data from or to the bus. For both of these write command signals, the advanced signals, namely, $\overline{AIOWC}$ and $\overline{AMWTC}$ are available. Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals. R_0, S_1, S_2 are set at the beginning of bus cycle. 8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT/$\overline{R}$ pin during T_1. In T_2, 8288 will set $\overline{DEN}$ = 1 thus enabling transreceivers, and for an input it will activate $\overline{MRDC}$ or $\overline{IORC}$.

These signals are activated until T_4. For an output, the $\overline{AMWC}$ or $\overline{AIOWC}$ is activated from T_2 to T_4 and $\overline{MWTC}$ or $\overline{IOWC}$ is activated from T_3 to T_4. The status bit S_0 to S_2 remains active until T_3 and become passive during T_3 and T_4. If reader input is not activated before T_3, wait state will be inserted between T_3 and T_4.

Timings for $\overline{RQ}/\overline{GT}$ Signals

The request/grant response sequence contains a series of three pulses. The request/grant pins are checked at each rising pulse of clock input. When a request is detected and if the conditions for HOLD request are satisfied, the processor issues a grant pulse over the $\overline{RQ}/\overline{GT}$ pin immediately during T_4 (current) or T_1 (next) state. When the requesting master receives this pulse, it accepts the control of the bus, it sends a release pulse to the processor using $\overline{RQ}/\overline{GT}$ pin.

4.7.3 Minimum Mode Interface

When the minimum mode operation is selected, the 8086 microprocessor provides all control signals needed to implement the memory and I/O interface. The minimum mode signal can be divided into the following basic groups: address/data bus, status, control, interrupt and DMA.

Address/Data Bus

Address and data bus lines serve two functions. An address bus is 20 bits long and consists of signal lines A_0 through A_{19}. A_{19} represents the MSB and A_0 represents LSB. A 20-bit address gives the 8086, 1 Mbyte memory address space. Moreover, it has an independent I/O address space which is 64 Kbytes in length. The 16 data bus lines D_0 through D_{15} are actually multiplexed with address lines A_0 through A_{15} respectively. By multiplexed, we mean that the bus works as an address bus during first machine cycle and as a data bus during next machine cycles. D_{15} is the MSB and D_0 is the LSB. When acting as a data bus, they carry read/write data for memory, input/output data for I/O devices, and interrupt type codes from an interrupt controller.

Status Signal

The four most significant address lines A_{19} through A_{16} are also multiplexed, but in this case with status signals S_6 through S_3. These status bits are output on the bus at the same time that data are transferred over the other bus lines. Bits S_4 and S_3 together form a 2-bit binary code that identifies which of the 8086 internal segment registers are used to generate the physical address that was output on the address bus during the current bus cycle. Code S_4S_3 = 00 identifies a register known as extra segment register as the source of the segment address. Status line S_5 reflects the status of another internal characteristic of the 8086. It is the logic level of the internal enable flag. The last status bit S_6 is always at the logic 0 level.

S_4	S_3	*Segment Register*
0	0	Extra
0	1	Stack
1	0	Code/none
1	1	Data

Control Signals

The control signals are provided to support the 8086 memory I/O interfaces. They control functions such as when the bus is to carry a valid address in which direction data is to be transferred over the bus, when valid write data is on the bus and when to put read data on the system bus.

ALE

ALE is a pulse to logic 1 that signals external circuitry when a valid address word is on the bus. This address must be latched in external circuitry on the 1-to-0 edge of the pulse at ALE. Another control signal that is produced during the bus cycle is $\overline{\text{BHE}}$ bus high enable. Logic 0 on this used as a memory enable signal for the most significant byte half of the data bus D_8 through D_1.

These lines also serve a second function, which is the S_7 status line. Using the M/$\overline{\text{IO}}$ and DT/$\overline{\text{R}}$ lines, the 8086 signals which type of bus cycle is in progress and in which direction data is to be transferred over the bus. The logic level of M/$\overline{\text{IO}}$ tells external circuitry whether a memory or I/O transfer is taking place over the bus. Logic 1 at this output signals a memory operation and logic 0 an I/O operation. The direction of data transfer over the bus is indicated by the logic level output at DT/$\overline{\text{R}}$. When this line is logic 1 during the data transfer part of a bus cycle, the bus is in the transmit mode. Therefore, data is either written into memory or output to an I/O device. On the other hand, logic 0 at DT/$\overline{\text{R}}$ signals that the bus is in the receive mode. This corresponds to reading data from memory or input of data from an input port. The signal read

$\overline{RD}$ and write $\overline{WR}$ indicates that a read bus cycle or a write bus cycle is in progress. The 8086 switches $\overline{WR}$ to logic 0 to signal external device that valid write or output data are on the bus. On the other hand, $\overline{RD}$ indicates that the 8086 is performing a read of data of the bus. During read operations, one another control signal is also supplied. This is $\overline{DEN}$ (Data Enable) and it signals external devices when they should put data on the bus. There is one other control signal that is involved with the memory and I/O interface. This is the READY signal.

Ready

READY signal is used to insert wait states into the bus cycle such that it is extended by a number of clock periods. This signal is provided by an external clock generator device and can be supplied by the memory or I/O subsystem to signal the 8086 when they are ready to permit the data transfer to be completed.

Interrupt Signals

The key interrupt interface signals are interrupt request (INTR) and interrupt acknowledge ($\overline{INTA}$).

INTR: Interrupt is an input to the 8086 that can be used by an external device to signal that it needs to be serviced. Logic 1 at INTR represents an active interrupt request. When an interrupt request has been recognized by the 8086, it indicates this fact to external circuit with pulse to logic 0 at the $\overline{INTA}$ output. The $\overline{TEST}$ input is also related to the external interrupt interface. Execution of a WAIT instruction causes the 8086 to check the logic level at the $\overline{TEST}$ input. If the logic 1 is found, the MPU suspends operation and goes into the idle state. The 8086 no longer executes instructions, instead it repeatedly checks the logic level of the $\overline{TEST}$ input waiting for its transition back to logic 0. As $\overline{TEST}$ switches to 0, execution resumes with the next instruction in the program. This feature can be used to synchronize the operation of the 8086 to an event in external hardware. There are two more inputs in the interrupt interface: the non-maskable interrupt NMI and the reset interrupt RESET. On the 0-to-1 transition of NMI, control is passed to a non-maskable interrupt service routine. The RESET input is used to provide a hardware reset for the 8086. Switching RESET to logic 0 initializes the internal register of the 8086 and initiates a reset service routine.

DMA interface signals: The direct memory access DMA interface of the 8086 minimum mode consists of the HOLD and HLDA signals. When an external device wants to take control of the system bus, it signals to the 8086 by switching HOLD to the logic 1 level. At the completion of the current bus cycle, the 8086 enters the hold state. In the hold state, signal lines AD_0 through AD_{15}, A_{16}/S_3 through A_{19}/S_6, $\overline{BHE}$, $M/\overline{IO}$, $DT/\overline{R}$, $\overline{RD}$, $\overline{WR}$, $\overline{DEN}$ and INTR are all in the high Z state. The 8086 signals external device that it is in this state by switching its HLDA output to logic 1 level.

4.7.4. Maximum-Mode Interface

When the 8086 is set for the maximum-mode configuration, it provides signals for implementing a multiprocessor/coprocessor system environment. By multiprocessor environment, we mean that one microprocessor exists in the system and that each is executing its own program. Usually, in this type of system environment, there are some system resources that are common to all processors. They are called global resources. There are also other resources that are assigned to specific processors. These are known as local or private resources. Coprocessor also means that there is a second processor in the system. In this, two processors do not access the bus at the same time. One passes the control of the system bus to the other and then may suspend its operation. In the maximum-mode 8086 system, facilities are provided for implementing allocation of global resources and passing bus control to other microprocessor or coprocessor.

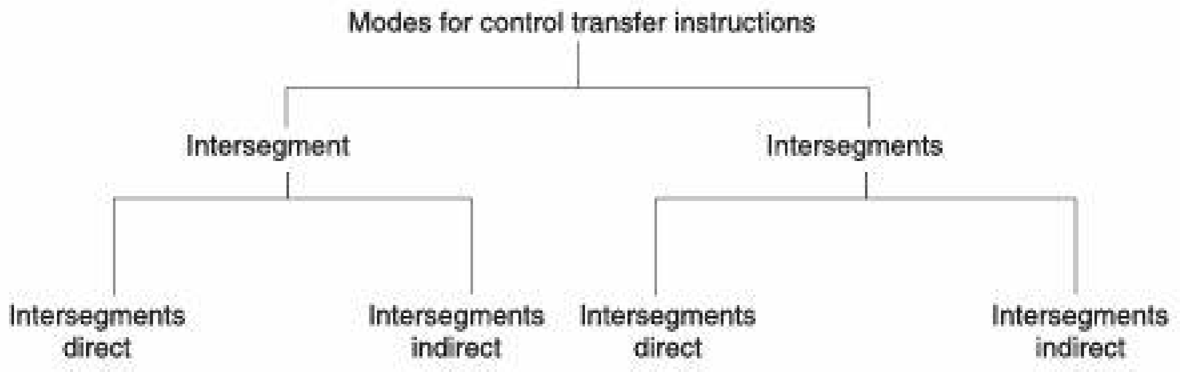
4.8 ADDRESSING MODES OF 8086

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types.

According to the flow of instruction execution, the instructions may be categorized as

(i) Sequential control flow instructions and

(ii) Control transfer instructions.



Addressing Modes for Control Transfer Instruction

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions. The control transfer instructions, on the other hand, transfer control to some predefined address somehow specified in the instruction after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are explained as follows:

1. **Immediate:** In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

   Example: MOV AX, 0005H

   In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. **Direct:** In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

   Example: MOV AX, [5000H]

   Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H*DS+5000H.

3. **Register:** In register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

   Example: MOV BX, AX.

   Table 4.6 shows internal registers, anyone can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

Table 4.6 Register Addressing Mode

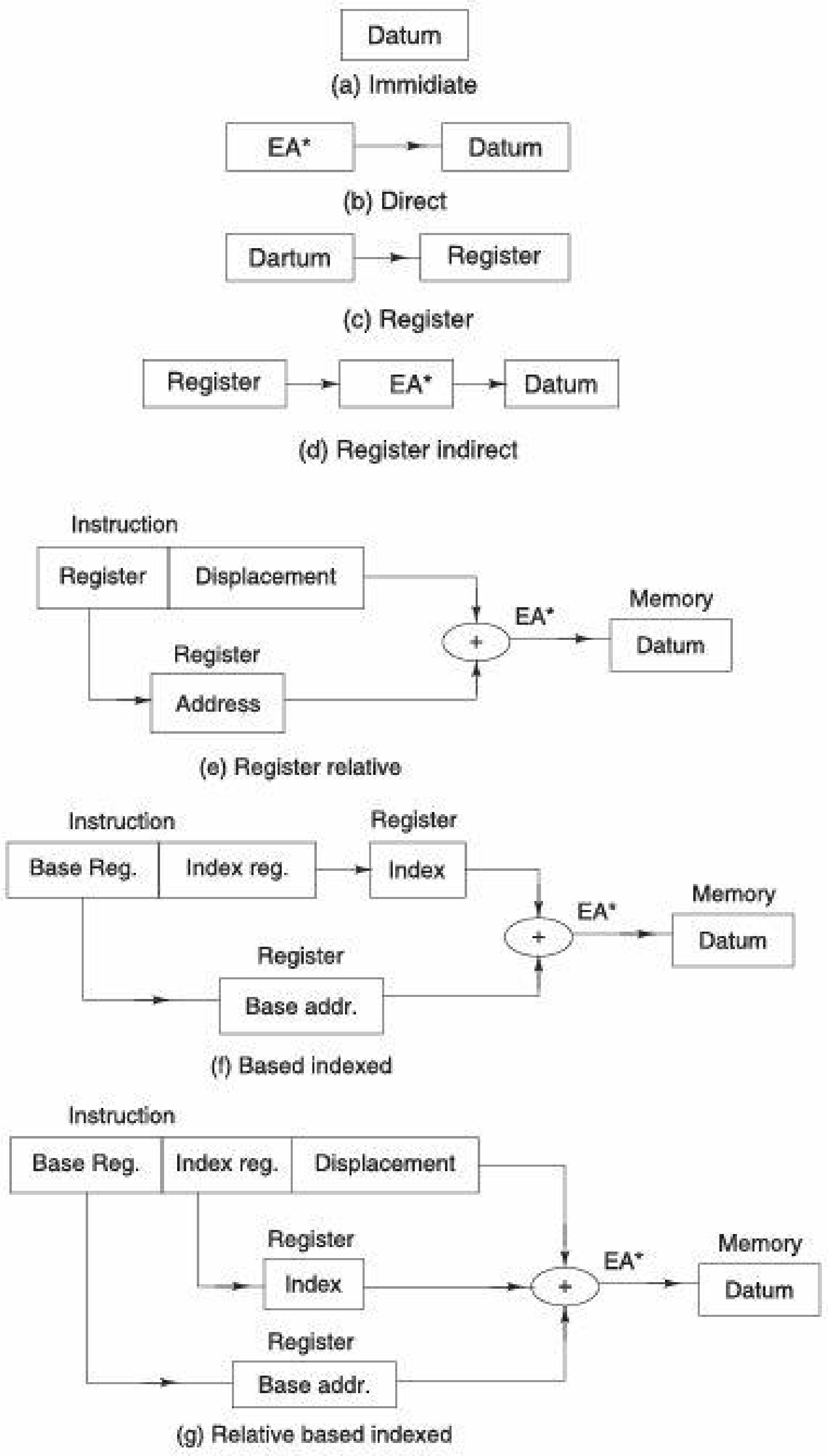
Register	*Operand sizes*	
	Byte (Reg 8)	*Word (Reg 16)*
Accumulator	AL, AH	AX
Base	BL, BH	BX
Count	CL, CH	CX
Data	DL, DH	DX
Stack pointer	–	SP
Base pointer	–	BP
Source index	–	SI
Destination index	–	DI
Code Segment	–	CS
Data Segment	–	DS
Stack Segment	–	SS
Extra Segment	–	ES

Note: MOV BX, CH is an illegal instruction. The register sizes must be the same.

4. **Register Indirect:** Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI registers. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

   Example: MOV AX, [BX]

   Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+ [BX].



EA* ia added to 1610 times contents of the appropriate segment register.

Fig. 4.14 Addressing Modes

5. **Indexed:** In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

   Example: MOV AX, [SI]

   Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H*DS+ [SI].

6. **Register Relative:** In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given explains this mode.

   Example: MOV Ax, 50H [BX]

   Here, effective address is given as 10H*DS+50H+ [BX].

7. **Based Indexed:** The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

   Example: MOV AX, [BX] [SI]

   Here, BX is the base register and SI is the index register. The effective address is computed as 10H*DS+ [BX] + [SI].

8. **Relative Based Indexed:** The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers, in a default segment.

   Example: MOV AX, 50H [BX] [SI]

   Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H*DS+ [BX] + [SI] + 50H.

   For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. inter-segment and intra-segment addressing modes.

   If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called inter-segment mode. If the destination location lies in the same segment, the mode is called intra-segment.

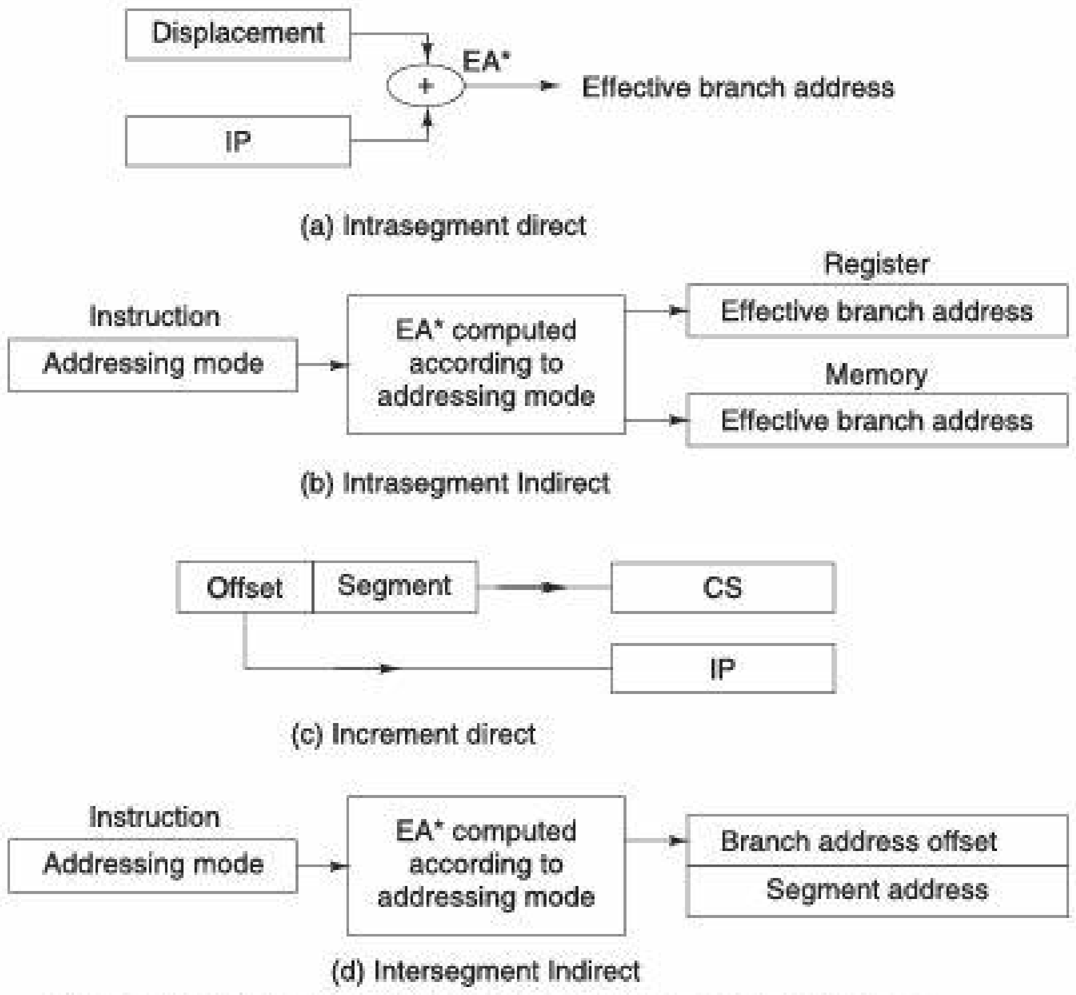
9. **Intra-segment direct mode:** In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

   The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. $-128<d<+128$), we term it as short jump and if it is of 16 bits (i.e. $-32768<+32768$), it is termed as long jump.

10. **Intrasegment Indirect Mode:** In this mode, the displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

11. **Intersegment Direct Mode:** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. **Intersegment Indirect Mode:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.



EA& ia added 16_{10} times contents of the appropriate segment register

Fig. 4.15 Intrasegment and Intersegment Status.

4.9 INSTRUCTION CODE FORMAT

There is one to one correspondence between the assembly language and machine language. As we all know that all processors only understand the binary language so we need to understand how assembly language formats are converted into machine codes. In this section, overview of the same is given. In 8086 all instruction code vary from 1 to 6-byte in length. The opcode or addressing mode is specified in the 1st or 2nd byte of the code. Some 8086 instruction formats are defined below.

OPCODE

Implied addressing mode-1bye instruction

OPCODE	REG

Register mode-1byte instruction

OPCODE	MOD	REG	REG/MEM

Register to/from memory mode-2-byte instruction

OPCODE	MOD	REG	REG/MEM	DISP. LOW	DISP.HIGH

Register to/from memory with displacement- 4-byte instruction

OPCODE	MOD	REG	REG/MEM	DATA LOW	DATA HIGH

Immediate mode-4-byte instruction

OPCODE	MOD	REG	REG/MEM	DISP. LOW	DISP.HIGH	DATA LOW	DATA HIGH

Immediate data with displacement-6-byte instruction

*REG- 3-Bit Long, MOD- 2-Bit Long, REG/MEM- 3-Bit Long

First Byte-Usually, the code for the operation to be performed and whether the operation will be performed on the byte or word is defined in the first byte of the instruction machine code or the first three bits of the second machine byte. The first byte bit has specific function which are defined below.

Opcode	D	W

First byte format

W-bit – The least significant bit of the first byte of instruction specifies whether the operation will be performed on the 8-bit data or 16-bit data

– W = 0, 8-bit data will be processed

– W = 1, 16-bit data will be processed

D-bit– In the 8086 instruction format one of the operand must be a register specified by the REG field, it will tell whether the specified register is source operand or destination operand.

- D = 0, register specified by the REG field is source operand.
- D = 1, register specified by the REG field is destination operand.

Second Byte- It has three fields

Mode field (MOD): The two most significant bits of the second byte defines the addressing mode of the instruction.

Table 4.7 2-Bit MOD Selection Code

Code	Explanation
00	Memory mode, no displacement
01	Memory mode, 8-bit displacement
10	Memory mode, 16-bit displacement
11	Register mode, no displacement

Register Field(REG): 3-bit long field used to specify one of the operands in the instruction

R/M field: 3-bit long R/M field and 2-bit MOD field together specifies the second operand.

To further examine the 8086 instruction set let us consider the ADD instruction, the 6-bit opcode for it is 0000000 for e.g.

(i) ADD CL, BH it will add the contents of the CL register with that of BH and place the result back to the CL register. The ocode for the ADD instruction is 000000. As both the operands are 8-bit wide so the W = 0, and MOD field should be 11. From Table 4.8 we have found the address for the CL to be 001 and using the Table 4.9 the value of R/M field is found to be 111

Opcode	D	W	MOD	REG	R/M	
000000	1	0	11	001	111	= machine code = 02 CFh

(ii) `MOV BL,AL` it will copy the content of AL register into the BL register. The opcode for the MOV is 100010

Opcode	D	W	MOD	REG	R/M	
100010	0	0	11	000	011	= machine code = 88 C3h

(iii) `ADD [BX+DI+2567], AX,` it is a register to memory addition. The effective address for the memory location will be calculated using the mod = 10 and REG = 000

Opcode	D	W	MOD	REG	R/M	DISP HIGH	DISP HIGH	
000000	0	1	10	000	001	67	25	= 01 81 67 25h

Table 4.8 3-Bit Register Field Code

Register Address	Registers Name	
	W = 0	W = 1
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Register address	Segment Register
00	ES
01	CS
10	SS
11	DS

Table 4.9 3-Bit R/M Field Code

Effective Address Calculation in 8086					
R/M	MOD 00	MOD 01	MOD 10	MOD 11	
				W = 0	W = 1
000	(BX) + (SI) DS	(BX) + (SI) + D8 DS	(BX) + (SI) + D16 DS	AL	AX
001	(BX) + (DI) DS	(BX) + (DI) + D8 DS	(BX) + (DI) + D16 DS	CL	CX
010	(BP) + (SI) SS	(BP) + (SI) + D8 SS	(BP) + (SI) + D16 SS	DL	DX
011	(BP) + (DI) SS	(BP) + (DI) + D8	(BP) + (DI) + D16 SS	BL	BX
100	(SI) DS	(SI) + D8 DS	(SI) + D16 DS	AH	SP
101	(DI) DS	(DI) + D8 DS	(DI) + D16 DS	CH	BP
110	D16 DS	(BP) + D8 DS	(BP) + D16 DS	DH	SI
111	(BX) DS	(BX) + D8 DS	(BX) + D16 DS	BH	DI

4.10 INSTRUCTION SET

Instruction is a command provided by the programmer which generally tells the microprocessor what operation is to be performed and on which operands. The format of instruction in 8086 is given below.

```
Label:     Mnemonic oprand1, oprand2           ; Comment
```

The 8086/8088 instructions are categorized into the following main types. This section explains the function of each of the instruction with suitable examples wherever necessary.

(i) *Data Copy/Transfer Instructions:* This type of instruction is used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) *Arithmetic and Logical Instructions:* All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) *Branch Instructions:* These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

(iv) *Loop Instructions:* If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) *Machine Control Instructions:* These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) *Flag Manipulation Instructions:* All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD, STD, CLI, STI, etc. belong to this category of instructions.

(vii) *Shift and Rotate Instructions:* These instructions involve the bit-wise shifting or rotation in either direction with or without a count in CX.

(viii) *String Instructions:* These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

4.10.1 Data Movement Instructions

These instructions include MOV, XCHG, LDS, LEA, LES, LFS, LGS, LSS, PUSH, PUSHA, PUSHAD, PUSHF, PUSHFD, POP, POPA, POPAD, POPF, POPFD, LAHF, AND SAHF.

(i) MOV: This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general- Or special-purpose registers or a memory location and another register or memory location may act as destination.

The mov instruction takes several different forms:

– Register to a register.

```
MOV  AX,BX
```

– Immediate operand to a register.

```
MOVAX,00C1H
```

– Immediate operand to a memory location.

```
MOV  [BX],  00F1
```

– Memory location to a register.

```
MOV  AX,[BX]
```

– Register to a memory location.

```
MOV  [AX],BX
```

– Register/memory location to a segment register (except CS).

MOV DS,BX

– Segment register to a register/memory location.

```
MOV CX,DS
```

There are two very important details to note about the mov instruction.

- There is no memory to memory move operation. The mod-reg-r/m addressing mode byte allows two register operands or a single register and a single memory operand.
- Second, you cannot move immediate data into a segment register.

`MOV DS, 0301H;` Not permitted (invalid)

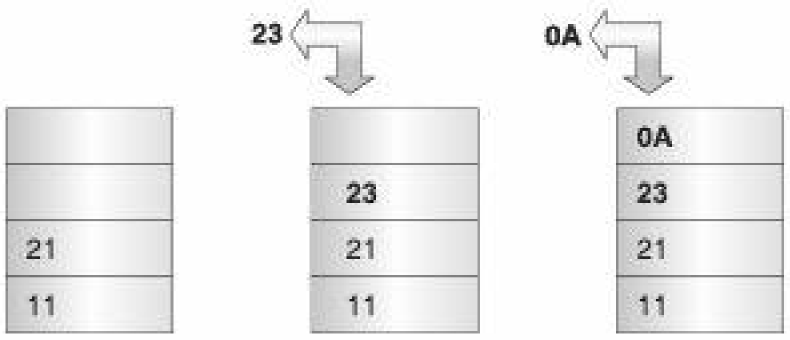
Thus, to transfer an immediate data into the segment register, the correct procedure is given below.

```
MOV AX, 0301H
MOV DS, AX
```

(ii) PUSH: This is a 16-bit data transfer instruction where two bytes are transferred onto the stack. The various types of PUSH instructions are:

```
PUSH Register
PUSH Memory
PUSH Segment Register
PUSH Flag
```

The transfer of data is done in the following way. Whenever a data is pushed onto the stack the first data byte is moved onto the stack segment memory location addressed by SS: SP-1 and the second data byte on to the memory location addressed by SS: SP-2. After the storage of data the SP register is decremented by 2.



The examples of these instructions are as follow

```
-PUSH AX
```

Consider the segment address (SS) is 0200 H and segment pointer (SP) is 0722 H at the start of the first instruction PUSH AX. On encountering the first PUSH AX instruction the AH (High) is transferred to SS: SP-1 and AL (Lower) register is transferred to SS: SP-2. Thus, 01 H is transferred to 02721 H and 02 H to 02720 H. After this operation SP is decremented by 2 and thus stores 0720 H.

(iii) POP: This instruction performs the exact opposite of the PUSH instruction. Here the data is transferred back from the stack onto the specified target register (segment register or memory location or to flag register). When the POP instruction executes, it transfers the SS: SP memory location to the lower byte and SS: SP + 1 memory location to the higher byte of target

For example

```
POP BX
```

Consider the initial stack memory pointer to SS = 0100 H and SP = 071BH, i.e. 0171BH. Now when POP BX instruction is executed BL (lower byte) is loaded with SS: SP memory location data and BH is loaded by SS: SP + 1 (0171CH) memory location contents. After this, SP is incremented by 2 and, therefore, stores 071DH.

(iv) XCHG (Exchange): This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them, may be a memory location. However, exchange of data contents of two memory locations is not permitted. Also the segment registers cannot be used by XCHG instruction. The examples are as shown:

There are four specific forms of this instruction on the 80x86:

```
XCHG REG, MEM
XCHG REG, REG
XCHG AX, REG16
```

For example,

```
XCHG AL, CL        ; Exchanges contents of AL with CL
XCHG BP, SI        ; Exchanges data of BP with SI.
```

(v) IN: This instruction transfers data from a port to the accumulator. In case of 8-bit transfer, the data is placed in AL register. In case of 16-bit data transfer; data placed in AX register is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. DX is the only register (implicit) which is allowed to carry the port address. The examples are given as shown:

For example,

```
MOV DX, port address    ; This instruction puts the port address in DX register.
IN AX, DX               ; for 16-bit data transfer
```

The advantage of the variable port is that the port address can be computed or dynamically determined during the execution of the program.

(vi) OUT: This instruction is used for writing to an output port. This instruction transfers data from the accumulator to an I/O port identified by the second byte of the instruction. The address of the output port may be specified in the instruction directly or implicitly in DX. The data to an odd addressed port is transferred on D_8-D_{15} while that to an even addressed port is transferred on D_0-D_7. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

```
MOV DX, address of an 8- or 16-bit port   ; address is specified in DX.
OUT DX, AX                                ; 8 or 16-bit data transfer
```

(vii) LEA: Load Effective Address

The load effective address instruction loads the offset of an operand in the specified register. It can also determine the offset address of a variable memory location specified in the instruction to load it into the specified register. The format for the LEA is given by

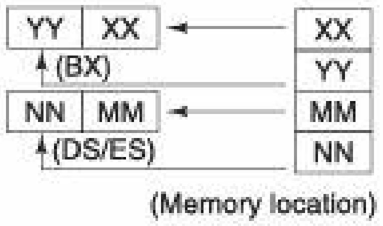
```
lea reg16, mem
```

For example

```
LEA CX, [BX + SI + 0532H]
```

(viii) LDS/LES: The LDS and LES instructions are the only instructions that directly process values larger than 32 bits. The instruction loads a word from the specified register and also loads a word from the next two memory locations into DS/ES register.

Let us consider `LDS BX,5000H`



(ix) XLAT: The translate instruction is used for code conversion using look-up table technique. When an XLAT instruction executes, it first adds AL to BX register to form a memory offset in the data segment. Then the contents of the memory location are transferred to AL register. We will explain this instruction with the aid of the following example.

For example,

```
00000                    00                       ; GRAY 0
00001                    01                       ; GRAY 1
00002                    03                       ; GRAY 2
00003                    02                       ; GRAY 3
00004                    06                       ; GRAY 4
00005                    07                       ; GRAY 5
00006                    04                       ; GRAY 6
00007                    05                       ; GRAY 7
00008                    0C                       ; GRAY 8
00009                    0D                       ; GRAY 9
00018        B0          05          00           ;MOV AX,0005 H
0001A        BB          00          00           ;MOV BX,0000 H
00010                                D7           ;XLATE
```

As we have loaded the gray equivalent from 0 to 9. Now we move 05 H to AL and 0000 H to BX. Then XLAT is executed, AL will be added to BX to produce the offset address which will be

```
offset = 05 + 0000 = 0005 H
Effective Address = DS * 10 H + 0005 H = 00005 H
```

Thus, XLAT will copy contents of memory location DS: [BX] + [AL] to AL register which will be 07 H. Now 07 H is gray equivalent of 05 H.

(x) The LAHF and SAHF Instructions: The LAHF (load ah from flags) and SAHF (store ah into flags) instructions are archaic instructions included in the 80x86's instruction set to help improve compatibility with Intel's older 8080 chip. As such, these instructions have very little use in modern day 80x86 programs. The LAHF instruction does not affect any of the flag bits. The SAHF instruction, by its very nature, modifies the S, Z, A, P, and C bits in the processor status register. These instructions do not require any operands and you use them in the following manner

```
SAHF
LAHF
```

SAHF has one major use. When using a floating point processor (8087, 80287, 80387, 80486, Pentium, etc.) you can use the SAHF instruction to copy the floating point status register flags into the 80x86's flag register.

(xi) PUSHF: The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

(xii) POPF: The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

4.10.2 Arithmetic Instructions

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The 8086/8088 instructions falling under this category are discussed below in significant detail. The arithmetic instructions affect all the condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode. The instructions that handle these operations are add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg, and some miscellaneous conversion instructions: aaa, aad, aam, aas, daa, and das. In Figure 4.16 the summary of all arithmetic instruction is given.

(i) ADD and ADC: This instruction adds an immediate 8-bit data specified in the instruction or a register (source) to the accumulator or to the contents of another register (destination). The result is in the destination operand. However, both the source and destination operands cannot be memory operands. The format of the following instructions is given below.

```
ADD  dest,  src          ;dest  ←dest  +  src
ADC  dest,  src          ;dest  ←dest  +  src  +  CF
```

Memory to memory addition is not possible in 8086 and the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected, depending upon the result.

Example of ADD

```
ADD  BP,  0100H      ;Immediate  mode
ADD  AX,  DX         ;Register  mode,  AX  =  AX+DX
```

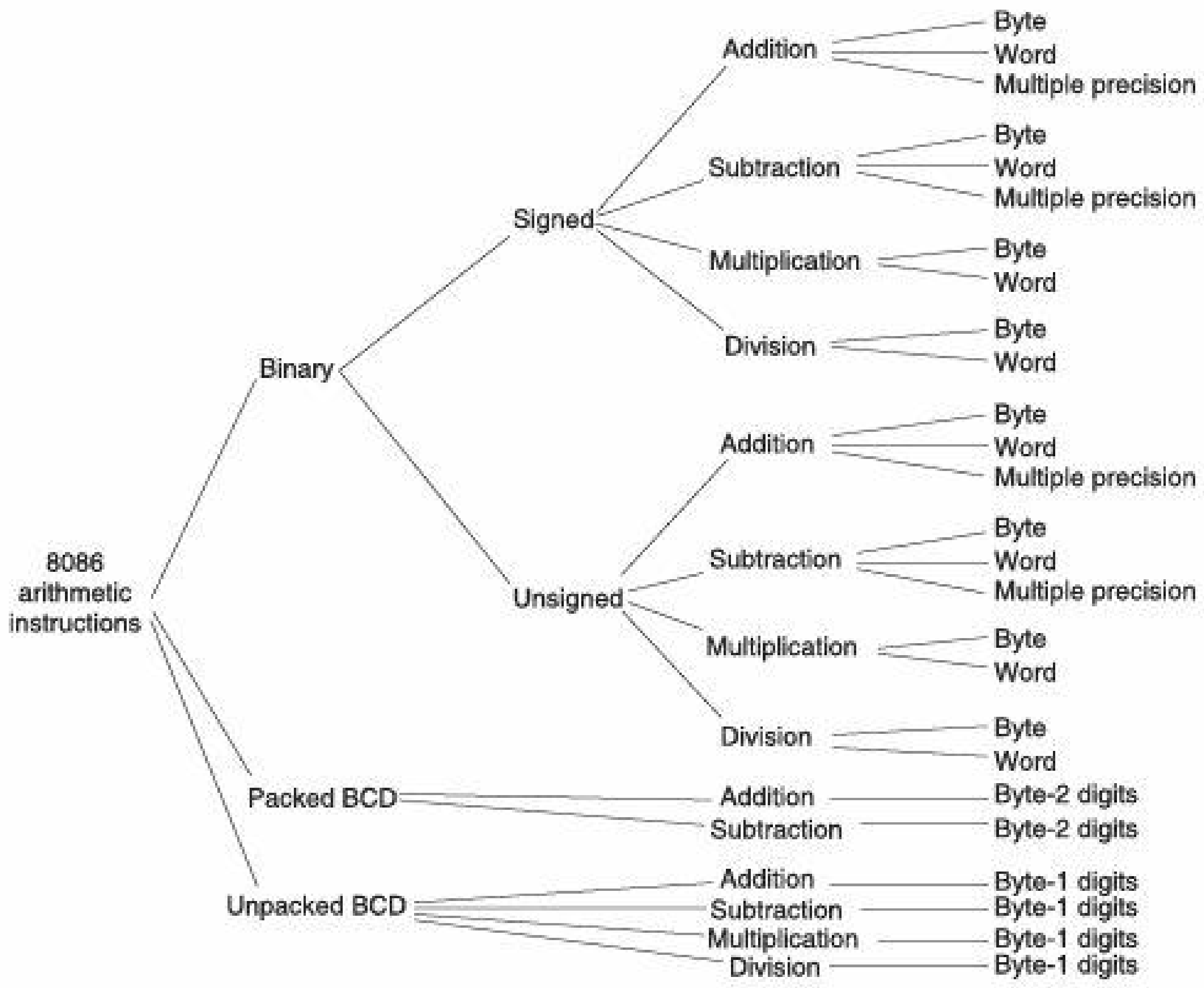


Fig. 4.16 Summary of the Arithmetic Operations that are Directly Implemented by 8086 Instructions.

```
ADD BX, [BP]  ;Register indirect, BX = BX+ Contents of SS:[SP]
ADD [DI],CX   ;Register indirect Contents of DS: [DI]+BP
```

(ii) SUB and SBB: The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set. The result is stored in the destination operand. All the flags are affected (condition code) by this instruction. The formats of this instruction are as follows:

```
SUB dest ,src          ;dest ←dest - src
SBB dest ,src          ;dest ←dest - src - CF
```

(iii) MULL (Unsigned Multiplication Byte or Word): This instruction multiplies two unsigned numbers. If the content of a specified register or memory location is 8-bit, it is multiplied by the content of AL register and the result is kept in the AX register while if content is of 16-bit, then it is multiplied by AX and the most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The formats for unsigned multiplication are given below.

```
MUL  Reg8-bit               ;AX ←AL*Reg8-bit
MUL  Reg16-bit              ;DX.AX ← AX*Reg16-bit
MUL  Mem
```

Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is 0, CF and OF both will be set.

(iv) IMUL(Signed Multiplication): This instruction multiplies two signed numbers. The result is a signed number. For 8-bit multiplication, the result is placed in AX register. For 16-bit multiplication, the low order 16 bits of the result are placed in AX register and the high order 16 bits of the result are placed in DX register. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16- and 32-bit result respectively, CF and OF both will be set. Sign bit and CF fills the unused higher bits of the result, AF are cleared.

```
IMUL  Reg8-bit              ;AX ← AL*Reg8-bit
IMUL  Reg16-bit             ;DX.AX ← AX*Reg16-bit
IMUL  Mem
```

(v) DIV and IDIV: These instructions are used to divide a 16-bit number by an 8-bit number or a 32-bit unsigned number by a 16-bit unsigned number. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. This instruction does not affect any flag.

```
DIV  Reg                        ;For unsigned division
DIV  Mem

IDIV  Reg                       ;For signed division
IDIV  Mem
```

Some examples for these instruction are given below:

```
DIV  BL                         ;AL ←AX / BL
                                ;AH ← AX % BL
DIV  CX                         ;AX ← DX.AX / CX
                                ;DX ← DX.AX % CX
IDIV  BL                        ;AL ← AX / BL
                                ;AH ← AX % BL
IDIV  CX                        ;AX ← DX.AX / CX
                                ;DX ← DX.AX % CX
```

On 80x86 you cannot divide one eight-bit value by another. If the denominator is an eight-bit value, the numerator must be a sixteen-bit value. If you need to divide one unsigned eight-bit value by another, you must zero extend the numerator to sixteen-bits. You can accomplish this by loading the numerator into the AL register and then moving zero into the AH register. Then you can divide AX by the denominator operand to produce the correct result. Failing to zero extend AL before executing div may cause the 80x86 to produce incorrect results!

(vi) AAA (ASCII Adjust after Addition): AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand. Two operands of the addition must have its lower 4 bits contain a number in the range from 0-9.The AAA instruction then adjusts AL so that it contains a correct BCD digit. If the addition produce carry (AF=1), the AH register is incremented and the carry CF and auxiliary carry AF flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases the higher 4 bits of AL are cleared to 0. AAA instruction will adjust the result of the two ASCII characters that were in the range from 30h("0") to 39h("9"). This is because the lower 4 bits of those character fall in the range of 0-9.The result of addition is not a ASCII character but it is a BCD digit.

For example,

(i) if AL = 38H (Before AAA Execution)
AL = 08H (After AAA Execution)

(ii) if AL = 5AH	and	AL = 00
AH = 00H		AH = 01
(Before Execution)		(After AAA Execution)

Flags Modifies: AF CF (OF, PF, SF, ZF undefined)

(vii) AAS (ASCII Adjust for Subtraction): AAS instruction is used to get the result in unpacked BCD after subtracting two unpacked ASCII operands. This instruction works only AL register. The procedure is similar to the AAA instruction. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

(viii) AAM (ASCII Adjust for Multiplication): AAM after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of the result (unpacked) remains in AL and the higher byte of the result remains in AH. Suppose, a product is available in AL, say AL = 5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add 6 (0110) to it D + 6 = 13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5 + 1 = 6 will be the upper unpacked byte of the result. Thus, after the execution, AH = 06 and AL = 03. Flags affected are: AF and CF. Others flags remain undefined.

```
AH = AL / 10
AL = AL mod 10
```

(ix) AAD (ASCII Adjust for Division): AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD.

```
AL = 10*AH+AL
AH = 0
```

(x) DAA (Decimal Adjust after BCD Addition): When two BCD numbers are added, the DAA instruction is used after ADD or ADC instruction to get correct answer in BCD. This instruction is for 8-bit operation and works only when the result is in AL register. The instruction DAA affects AF, CF, PF, and ZF flags. The OF is undefined. The examples given below explain the instruction.

For example,

```
i) AL = 53,CL = 29
ADD AL,CL                 ; AL = 7C = 53+29
DAA                       ; AL = 82= 7C+06(As C>9)

ii) AL = 73,CL= 29
ADD AL,CL                 ; AL = 9C = 73+29
DAA                       ; AL = 02 and CF = 1
```

(xi) DAS (Decimal Adjust after Subtraction): This instruction is used to get correct result in BCD, when a BCD number is subtracted from another BCD number. It is used after SUB or SBB instruction. This instruction is for 8-bit operation and works only when the result is in AL register. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

```
(i) AL = 75,CL = 46
    SUB AL,CL                 ; AL = 2F = 75-46 And AF = 1
    DAS                       ; AL = 29 = F-6(As F>9)
(ii) AL = 38,CL = 61
    SUB AL, CL                ; AL = D7  And CF = 1(Borrow)
    DAS                       ; AL = 77= D-6(As D>9)
                              CF = 1(Borrow)
```

(xii) NEG: The NEG (negate) instruction takes the two's complement of a byte or word. It takes a single (destination) operation and negates it. The syntax for this instruction is

```
NEG dest
```

It computes the following:

```
dest = - dest
```

This effectively reverses the sign of the destination operand. If the operand is zero, its sign does not change, although this clears the carry flag. Negating any other value sets the carry flag. Negating a byte containing –128, a word containing 32,768 or a double word containing 2,147,483,648 does not change the operand, but will set the overflow flag. NEG always updates the A, S, P, and Z flags as though you were using the sub instruction.

(xiii) CBW (Convert Signed Byte to Word): This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted into all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

(xiv) CWD (Convert Signed Word to Double Word): This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before the signed division. It does not affect any flag.

(xv) INC/DEC (Increment or Decrement): These will Add or sub one to destination unsigned binary operand. Flag modified are AF OF PF SF ZF, e.g.,

```
    INC  SP                  ;SP = SP + 1
    INC  WORD [BP]           ;word contents of SS: [BP] +1
    DEC  AX                  ;AX=AX-1
    DEC  BYTE [DI]           ;byte content of DS: [DI]-1
```

(xvi) CMP (Compare): The cmp (compare) instruction is identical to the sub instruction with one crucial difference. It does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is

```
CMP dest, src
```

The specific forms are

```
    CMP  reg,  reg
    CMP  reg,  mem
    CMP  mem,  reg
    CMP  reg,  immediate data
    CMP  mem,  immediate data
```

Usually, you'll want to execute a conditional jump instruction after a cmp instruction. This two-step process, comparing two values and setting the flag bits then testing the flag bits with the conditional jump instructions, is a very efficient mechanism for making decisions in a program. For example, CMP AX,BX ;AX-BX and Flag are set or reset according to subtraction result.

4.10.3 Logical Instruction

(i) AND, OR, XOR, and NOT: The 8086 logical instructions operate on a bit-by-bit basis. Both eight- and sixteen-bit versions of each instruction exist. These instructions perform bit-by-bit logical AND, XOR, OR and NOT operation of two operands and place the result in destination operand. Both the operands cannot be memory locations.

```
AND dest, source             ;dest:= dest and source
OR dest, source              ;dest = dest or source
XOR dest, source             ;dest = dest xor source
NOT dest                     ;dest = not dest
```

Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.
- The not instruction does not affect any flags.

For example,

```
AND AL,BL                  ;Suppose AL = 23H and BL = 50H
AL =  0  0  1  0  0  0  1  1
BL =  0  1  0  1  0  0  0  0
     ------------------------
AL =  0  0  0  0  0  0  0  0        (After AND execution)
```

(ii) TEST (Logical Compare Instructions): These instructions perform a bit-by-bit logical AND operation on the two operands. 8-bit or 16-bit operation is allowed. The result of this ANDing operation is discarded but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

(iii) SHR/SAR (Shift logical/Arithmetic Right): This instruction shifts each bit of operand which is contained in an 8-bit or 16-bit resister or memory location(s), right by specified number of bits the operand word or byte bit by bit to the right and inserts zeros in the newly introduced most significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 4.17 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

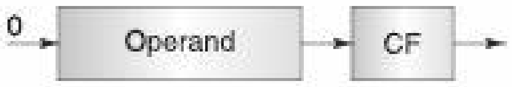


Fig. 4.17 Logical Shift Right.

Operand	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1	CF
Count = 1	0	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1
Count = 2	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0

Execution of SHR Instruction

Operand	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1	CF
Count = 1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1
Count = 2	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0

Execution of SHR Instruction

(iv) SHR/SAR (Shift logical/Arithmetic Right): The working of these instructions is same except data shift flow will be in the left direction and least significant bit is flooded with zero in case of logical Right.

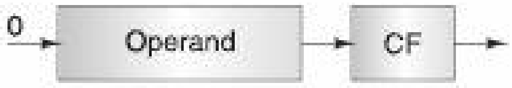


Fig. 4.18 Shift Arithmetic Right.

(v) ROR (Rotate Right without Carry): This instruction rotates all the bits of the operand contained in the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand.

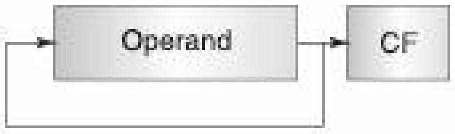


Fig. 4.19 Rotate Right without Carry.

Operand	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1	CF
Count = 1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1
Count = 2	0	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0

(vi) ROL (Rotate Left without Carry): This instruction rotates all the bits of operand contained in an 8-bit or 16-bit resistor or memory location(s) left by specified number of bits. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged by this rotate operation. Figure 4.20 explains the operation.

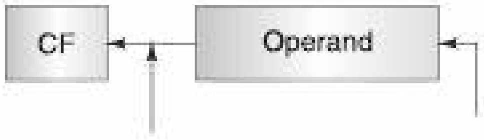


Fig. 4.20 Rotate Left without Carry.

(vii) RCR (Rotate Right through Carry): This instruction rotates all the bits of operand contained in an 8-bit or 16-bit resistor or memory location(s) right by specified number of bits. For each operation, carry flag is pushed into the MSB of the operand and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 4.21 explains the operation.

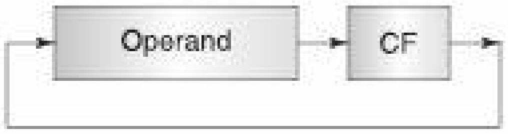


Fig. 4.21 Rotate Right through Carry.

Operand	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1	CF
Count = 1	X	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	1
Count = 2	1	X	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0

*X = carry flag status not known.

(viii) RCL (Rotate Left through Carry): This instruction rotates all the bits of the operand contained in an 8-bit or 16-bit resistor or memory location(s) left by specified number of bits through carry flag. For each operation, the carry flag is pushed into LSB, and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left operations. The operand may be a register or a memory location. Figure 4.22 explains the operation.

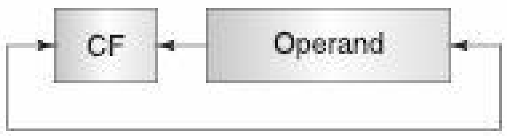


Fig. 4.22 Rotate Left through Carry.

(ix) REP (Repeat Instruction Prefix): This instruction is used as a prefix to other instructions before string instructions MOVS or STOS. It decrements CX register and repeats the string instruction until the CX register becomes zero. When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ, i.e., repeat operation while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero.

4.10.4 String Instructions

These instructions can operate on strings of bytes, words, or double words. The instructions which come under this group are MOVS, STOS, SCAS, CMPS, INS.

(i) MOVSB/MOVSW (Move String Byte or String Word): The starting byte of the source string is located in the memory location whose address may be computed using SI (source index) and DS (data segment) contents. The MOVSB/MOVSW instruction thus, moves a string of bytes words pointed to by DS: SI pair (source) to the memory location pointed to by ES: Dl pair (destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e., SI and Dl depend upon the direction flag DF. If DF is 0, the index registers are incremented; otherwise, they are decremented, if DF is 1, in case of all the string manipulation instructions. The following string of instructions explains the execution of the MOVS instruction.

(ii) CMPS (Compare String Byte or String Word): The CMPS instruction can be used to compare two strings addressed by DI register with the strings addressed by SI register. The length of the string must be stored in the register CX. For comparison the content of memory location addressed by DI register is subtracted from the content of the memory location addressed by SI register. The result is not stored. Both the operands remain unaffected. REP instruction prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explains the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further. If both strings are completely equal, i.e., CX becomes zero, the ZF is set, and otherwise, ZF is reset.

```
MOV SI, OFFSET FIRST_STRING       ; Point SI at source string
MOV DI, OFFSET SECOND_STRING      ; Point DI at destination string
CLD                               ; DF Cleared so SI and DI is
                                    auto increment after CX = 0
MOVCX, 100                        ; Put number of string element
                                    in CX
REPE CMPSB                        ; Repeat the comparison of
                                    string byte until end of
                                    string or until compared bytes
                                    are not equal
```

(iii) SCAS (Scan String Byte or String Word): It compares the content of the accumulator with the content of memory location addressed by DI register in extra segment ES. This string to be scanned must be in the extra segment. The string is pointed to by ES: DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in the string must be stored in the register CX. For comparison the content of memory location addressed by DI register is subtracted from the content of the memory location addressed by SI register. The result is not stored. Both the operands remain unaffected. REP instruction prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explains the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When the mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further. If both strings are completely equal, i.e., CX becomes zero, the ZF is set, and otherwise, ZF is reset.

```
MOV SI, OFFSET FIRST_STRING       ; Point SI at source string
MOV DI, OFFSET SECOND_STRING      ; Point DI at destination string
CLD                               ; DF Cleared so SI and DI is
                                    auto increment after CX = 0
MOVCX, 100                        ; Put number of string element
                                    in CX
REPE CMPSB                        ; Repeat the comparison of
                                    string byte until end of
                                    string or until compared bytes
                                    are not equal
```

(iv) LODS (Load String Byte or String Word): The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is automatically incremented or decremented depending upon DF. If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

```
LODSB -                       AL ← DS:[SI]
                              If DF = 0, SI ← SI+1
                              If DF = 1, SI ← SI-1.
LODSW -                       AL ← DS:[SI]: AH ← DS:[SI+1]
                              If DF = 0, SI ← SI+2;
                              If DF = 1, SI ← SI-2
```

(v) STOS (Store String Byte or String Word): The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES: DI register pair. The DI is automatically incremented or decremented if DF flag is 0 or 1. No flags are affected by this instruction.

```
STOSB -                       ES:[DI] ← AL
                              If DF = 0, DI ← DI+1
                              If DF = 1, DI ← DI-1
STOSW -                       ES:[DI] ← AL; ES:[DI+1] ← AH
                              If DF = 0, DI ← DI+2
                              If DF = 1, DI ← DI-2
```

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF = 1, then the execution follows auto decrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in auto decrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses.

4.10.5 Control Transfer or Branching Instructions

The instructions discussed thus far execute sequentially, that is, the CPU executes each instruction in the sequence it appears in the program. To write real programs requires several control structures, not just the sequence. The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes, the CS may or may not be modified. 80x86 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions. These instructions can be categorized as given below.

- Unconditional branch instruction
- Conditional branch instruction

4.10.5.1 Unconditional Branch Instructions

(i) Unconditional Jumps: The jmp (jump) instruction unconditionally transfers control to another point in the program. The various forms of this instruction are as: an intersegment/direct jump, two intrasegment/direct jumps, an intersegment/indirect jump. Intrasegment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment. These instructions generally use the same syntax, it is JMP target.

```
JMP  disp8          ;direct  intrasegment,   8-bit  displacement.
JMP  disp16         ;direct  intrasegment,   16-bit  displacement.
```

Near Jump: The first form consists of an opcode and a single byte displacement. The CPU sign extends this displacement to 16 bits and adds it to the IP register. This instruction can branch to a location –128 +127–from the beginning of the next instruction.

Far Jump: The second form of the intrasegment jump or far jump is three bytes long with a two-byte displacement. This instruction allows an effective range of –32,768 +32,767 bytes and can transfer control to anywhere in the current code segment. The CPU simply adds the two-byte displacement to the ip register.

(ii) CALL (Unconditional Call): This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. There are two basic types of calls, near and far. A near call is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near call instruction it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL onto the stack. This offset saved on the stack is referred to as return address, because this is the address that execution will return to after the procedure executes. A RET instruction at the end of the procedure will return execution to the instruction after the call by copying the offset saved on the stack back to IP. A far call is a call to a procedure which is in a different segment from the one that contains the CALL instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of CS register to the stack. A RET instruction at the end of the procedure will return execution to the next instruction after the call by restoring the saved values of CS and IP from the stack. Example is given below.

```
CALL  1024               ;SP ← SP-2;  [SP] ←IP,  IP ← 1024
CALL  [SI]               ;SP ← SP-2;  [SP] ←IP,  IP ← [SI]
CALL  1122:3344          ;SP ← SP-2;  [SP] ←CS,  CS ← 1122
                         ;SP ← SP-2;  [SP] ←IP,  IP ← 3344
CALL  [SI]               ;SP ← SP-2;  [SP] ←CS,  CS ← [SI+2]
                         ;SP ← SP-2;  [SP] ←IP’  IP ← [SI]
```

(iii) RET (Return from the Procedure): At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure. In case of a FAR procedure, the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

(iv) INT N-Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from OOH to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N×4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IF must be enabled.

Thus, the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT 20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000: 0080 H

(v) IRET-Return from ISR: When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored onto the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

(vi) LOOP-Loop Unconditionally: This instruction is used to repeat a series of instructions a number of times specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMF IF NOT ZERO structure.

4.10.5.2 Conditional Branch Instructions

Conditional jump: When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions here mean the status of condition code flags. These types of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within –80H to 7FH (or –128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The following Table gives conditional jumps.

Instruction	*Description*	*Condition*
JC	Jump if carry	Carry = 1
JNC	Jump if no carry	Carry = 0
JZ	Jump if zero	Zero = 1
JNZ	Jump if not zero	Zero = 0
JS	Jump if sign	Sign = 1
JNS	Jump if no sign	Sign = 0
JO	Jump if overflow	Overflow = 1
JNO	Jump if no overflow	Overflow = 0
JP	Jump if parity	Parity = 1
JPE	Jump if parity even	Parity = 1
JNP	Jump if no parity	Parity = 0
JPO	Jump if parity odd	Parity = 0
JA	Jump if above (>)	Carry = 0, Zero = 0
JNBE	Jump if not below or equal (not <=)	Carry = 0, Zero = 0
JAE	Jump if above or equal (>=)	Carry = 0
JNB	Jump if not below (not <)	Carry = 0
JB	Jump if below (<)	Carry = 1
JNAE	Jump if not above or equal (not >=)	Carry = 1
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1
JE	Jump if equal (=)	Zero = 1
JNE	Jump if not equal	Zero = 0
JG	Jump if greater (>)	Sign = Overflow or Zero = 0
JNLE	Jump if not less than or equal (not <=)	Sign = Overflow or Zero = 0
JGE	Jump if greater than or equal (>=)	Sign = Overflow
JNL	Jump if not less than (not <)	Sign = Overflow
JL	Jump if less than (<)	Sign □ Overflow
JNGE	Jump if not greater or equal (not >=)	Sign □ Overflow
JLE	Jump if less than or equal (<=)	Sign □ Overflow or Zero = 1
JNG	Jump if not greater than (not >)	Sign □ Overflow or Zero = 1
JE	Jump if equal (=)	Zero = 1
JNE	Jump if not equal (□)	Zero = 0

4.10.6 Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types: (a) flag manipulation instructions and (b) machine control instructions. The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

```
CLI - Clear interrupt flag
STI - Set interrupt flag
```

These instructions modify the carry (CF), direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto increment or auto decrement modes. Thus, the respective instructions may also be called machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed data transfer instructions, in this text. No direct instructions are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

```
WAIT - Wait for Test input pin to go low
HLT - Halt the processor
NOP No- operation
ESC - Escape to external device like NDP (numeric coprocessor)
LOCK - Bus lock instruction prefix.
```

After executing the HLT instruction, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further.

DATA TRANSFER INSTRUCTIONS

MOV	Move byte or word to register or memory.
XCHG	Exchange byte or word.
LDS	Load pointer using data segment.
LEA	Load effective address.
LES	Load pointer using extra segment.

POP	pop word off stack.
POPF	pop flags off stack.
IN	Input byte or word from port.
OUT	output word to port.
XLAT	Translate byte using look-up table.

LOGICAL INSTRUCTIONS

NOT	Logical NOT of byte or word.
AND	Logical AND of byte or word.
OR	Logical OR of byte or word.
XOR	Logical exclusive-OR of byte or word.
TEST	Test byte or word (AND without storing).

SHIFT AND ROTATE INSTRUCTIONS

SHL	Logical shift left byte or word by 1 or CL.
SHR	Logical shift right byte or word by 1 or CL.
SAL	Arithmetic shift left byte or word by 1 or CL.
SAR	Arithmetic shift right byte or word by 1 or CL.
ROL	Rotate left, t byte or word by 1 or CL.
ROR	Rotate right byte or word by 1 or CL.
RCL	Rotate left through carry byte or word by 1 or CL.
RCR	Rotate right through carry byte or word by 1 or CL.

ARITHMETIC INSTRUCTIONS

ADD	Add byte or word.
SUB	Subtract byte or word.
ADC	Add byte or word and carry .
SBB	Subtract byte or word and borrow.
INC	Increment, decrement byte or word.
NEG	Negate byte or word (two's complement).
CMP	Compare byte or word (subtract without storing).
MUL	Multiply byte or word (unsigned).
DIV	Divide byte or word (unsigned).
IMUL	Integer multiply, divide byte or word (signed).
IDIV	Divide byte or word (signed).
CBW	Convert byte to word (useful before multiply/divide).
CWD	Convert word to double word (useful before multiply/divide).
AAA	ASCII adjust for addition.
AAS	ASCII adjust for addition, subtraction.

AAM	ASCII adjust for multiplication.
AAD	ASCII adjust for multiplication.
DAA	Decimal adjust for addition (binary coded decimal numbers).
DAS	Decimal adjust for subtraction (binary coded decimal numbers).

TRANSFER INSTRUCTIONS

JMP	Unconditional jump (*short* 127/128, *near* 32 K, *far* between segments)
LOOP	Loop unconditional, count in CX, short jump to target address.
LOOPE	Loop if equal (zero), count in CX, short jump to target address.
LOOPNE	Loop if not equal (not zero), count in CX, short jump to target address.
CALL	Call inside or outside current segment.
RET	Return from procedure.
INT	Software interrupt.
INTO	Software interrupt, interrupt if overflow.
IRET	Return from interrupt.

STRING INSTRUCTIONS

MOVS	Move byte or word string.
MOVSB	Move byte string.
MOVSW	Move word string.
CMPS	Compare byte or word string.
SCAS	Scan byte or word string (comparing to AL or AX).
LTOS	Load byte or word string to AL or AX.
STOS	Store byte or word string to AL or AX.
REP	Repeat.
REPE	Repeat while equal, zero.
REPZ	Repeat while equal.
REPNE	Repeat while not equal.
REPNEZ	Repeat while not equal (zero).

PROCESSOR CONTROL INSTRUCTIONS

STC	Set carry flag.
CLC	Clear carry flag.
CMC	Complement carry flag.
STD	Set direction flag.
CLD	Clear direction flag.
STI	Set interrupt enable flag.
CLI	Clear interrupt enable flag.
LAHF	Load ah from flags.

SAHF	Store ah into flags.
PUSHF	Push flags onto stack.
POPHF	Pop flags off stack.
ESC	Escape to external processor interface.
LOCK	Lock bus during next instruction.
NOP	No operation.
WAIT	Wait for test pin activity.
HLT	Halt processor.

4.11 ASSEMBLER DIRECTIVES

The Assembler directives are directions to the assembler, not instructions for the 8086. A program that translates programs from assembly language to machine language is called an assembler. An assembler directive is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process. The assembler directives described here are those for the Intel 8086.

(i) ASSUME: The ASSUME directive is used to inform the assembler the name of the logical segment it should use for a specified segment. The 8086 works with 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

```
ASSUME CS:CODE
```

Inform the assembler that the instruction for a program is in logical segment named code.

```
ASSUME DS:DATA
```

Inform the assembler that for any program instruction which refers to the data segment.

If, for example, the assembler reads

The statement MOV AX, [BX] after it reads this ASSUME, it will know that the memory location referred to by [BX] is in the logical segment DATA.

(ii) DB (Define Byte): The directive is used to tell the assembler that a variable is declared which would hold byte data variable or to set aside one or more storage location.

```
Score DB 01H
```

This above directive will declare score as a variable and data value of 01H will stored in it.

```
Ranks DB 01H, 02H, 03H, 04H
```

This statement directs the assembler to reserve four memory location for a list named ranks and initialize them with the above specified data.

```
NAME_EMPLOY DB 'VISHAL'
```

Declare array of 6 bytes and initialize with ASCII codes for letters in YOHANNAS.

(iii) DW(Define Word): The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory.

```
INTEREST DW 437A H
```

This declares a variable of type word and named it as INTEREST. This variable is initialized with value 437Ah when it is loaded into memory to run.

```
NAME DW 100 DUP(0)
```

Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named NAME.

(iv) DD (Define Double Word): This directive is used to define a variable of type double word or to reserve storage location of type double word in memory. For example,

```
NAME DD 12341234H
```

(v) DT – Define Ten Bytes: This directive is used to define a variable which is 10 bytes in length or to reserve 10 bytes of storage in the memory. For example,

```
NAME DT 11223344556677889900
```

(vi) END: This directive is always placed after the last statement of a program to inform the assembler that this is the end of the program. The assembler will ignore any statement after an END directive.

(vii) ENDP: This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler.

```
FACTORIAL ENDP
```

The above statement will inform the assembler to end the procedure named as FACTORIAL.

(viii) ENDS: This ENDS directive is used with name of the segment to indicate the end of that logical segment. ENDS is used with the segment directive to bracket a logical segment containing instruction or data

```
CODE ENDS
```

It will direct the Assembler to End of segment named as CODE.

(ix) GROUP: This directive is used to tell the assembler to group the logical segment named after directive into one logical group segment. The assembler passes an information to the linker/ loader to form the code such that the group declared segments or operands must lie within a 64 Kbyte memory segment.

(x) LABEL: This directive is used to assign a name to the current content of the location counter. This directive must be followed by a term which specifies the type you want associated with that name. The type of the label must be specified, i.e., whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

(xi) LENGTH: This directive is not available in MASM. This is used to refer to the length of a data array or a string. It is an operator which tells the assembler to determine the number of elements in some named data item, such as a string or an array.

```
MOV AX, LENGTHSTRING1
```

The above instruction will find the length of the string 1 and place the result into the AX register.

(xii) NAME: The NAME directive is used to assign a name to an assembly language program module. When program consisting of several modules are written.

(xiii) OFFSET: When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. For example,

```
MOV  DI,OFFSET  NAME
```

The above instruction will move the displacement address of the variable NAME defined before.

(xiv) ORG: The ORG directive directs the assembler to set the location counter to a desired value at any point in the program. If the ORG statement is not written in the program, the location counter is initialized to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the location counter will get initialized to the address 0200H instead of 0000H.

(xv) PROC: The PROG directive is used to identify the start procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The PROC directive precedes an ENDP directive. For example,

```
FACTORIAL PROC NEAR
```

The above instruction will find the start of the procedure named FACTORIAL and also tells the assembler that procedure is near.

(xvi) PTR-Pointer: The pointer operator is used to declare the type of a label, variable or memory operand. It is necessary to do this in any instruction where the type of the operand is not clear.

The examples of the PTR operator are as follows:

(xvii) PUBLIC: This directive is used to tell the assembler that a specified name or label will be accessed from other modules. The PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive.

(xviii) SEG: The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of "SEG label".

(xix) SHORT: Operator indicates to the assembler that only one byte is required to code the displacement for a jump. This method of specifying the jump address saves the memory. Otherwise, the assembler may reserve two bytes for the displacement.

(xx) EVEN: This assembler directive tells the assembler to increment the memory location to next possible even address if it is not at the even address. The processor can read a word in one clock cycle if the memory address is even and take two clock cycles if the memory address is odd.

Assembler directive	*Description*
ALIGN	Aligns next variable or instruction to byte which is multiple of operand.
COMMENT	Indicates a comment.
DW	Allocates and optionally initializes words of storage.
DQ	Allocates and optionally initializes quad words of storage.
END	Terminates assembly; optionally indicates program entry point.
ENDP	Marks end of procedure definition.
EQU	Assigns expression to name.
EXITM	Terminates macro expansion.
LABEL	Creates a new label with specified type and current location counter.
MACRO	Starts macro definition.
MODEL	Specifies mode for assembling the program.
LOCAL	Declares local variables in macro definition.
EXTRN	Indicates externally defined symbols.
EVEN	Aligns next variable or instruction to even byte.
ENDS	Marks end of segment or structure.
ENDM	Terminates a macro definition.
DT	Allocates and optionally initializes 10-byte-long storage units.
DD	Allocates and optionally initializes double words of storage.
DB	Allocates and optionally initializes bytes of storage.
ASSUME	Selects segment register(s) to be the default for all symbol in segment.

4.12 ASSEMBLY LANGUAGE PROGRAMMING

Program 1: Block movement of data (word transfer) without overlap.

```
DATA
     SOURCE_BLOCK  DW  89ABH,0ABCDH,0020H,0F0H,0100H
     DESTN_BLOCK  DW  5  DUP(?)
CODE
     MOV  AX,@DATA
     MOV  DS,AX
     LEA  SI,  SOURCE_BLOCK
     LEA  DI,  DESTN_BLOCK
     MOV  CX,05H
LOC1:  MOV  AX,[SI]
     MOV  [DI],AX
```

```
      INC  SI
      INC  SI
      INC  DI
      INC  DI
      DEC  CX
      JNZ  LOC1
      MOV  AH,4CH
      INT  21H
END
```

Program 2: To copy a string from one set of locations to another.

```
DATA SEGMENT
      TESTMSG DB 'THAPAR INSTITUTE OF TECHNOLOGY'
      LEN  DW  30
DATA  ENDS

DST   SEGMENT
      DEST  DB  30  DUP(0)
DST   ENDS

CODE SEGMENT
   ASSUME  CS:CODE,DS:DATA,ES:DST
START:MOV  AX,DATA
      MOV  DS,AX
      MOV  AX,DST
      MOV  ES,AX
      LEA  SI,TESTMSG
      LEA  DI,DEST
      MOV  CX,LEN
      CLD
REP:  MOVSB
      MOV  AH,4CH
        INT  21H
CODE  ENDS
   END  START
```

Program 3: Write a program to store 256 bytes of data into memory starting from address 18950H. The source of data is input port of address 1500H.

```
      MOV  AX,  1895H
      MOV  ES,  AX
      MOV  DI,  0H
      MOV  DX,  1500H
      MOV  CX,  100H
      CLD
   L1  : IN  AL,  DX
     STOSB
     LOOP   L1
     HLT
```

Program 4: Write an ALP to transfer 100 bytes from the starting source memory address 15009 H to output port 9137H.

```
      MOV  AX,  1500H
      MOV  DS,  AX
      MOV  SI,  09  H
      MOV  CX,  64H
      MOV  DX,  9137H
      CLD
   L1:  LODSB
      OUT  DX,  AL
      LOOP  L1
      HLT/INT3
```

Program 5: To add two 16-bit numbers (without carry).

```
DATA  SEGMENT
      OPERAND  1  DW  1200H
      OPERAND  2  DW  3283H
      RESULT  DW  ?
DATA  ENDS

CODE  SEGMENT
  ASSUME  CS:  CODE,  DS:DATA
  START:MOV  AX,DATA
MOV  DS,  AX
      MOV  AX,  OPERAND  1
      ADD  AX,  OPERAND  2
      MOV  RESULT,  AX
   INT  3
CODE  ENDS
END  START
```

Program 6: Add the contents of memory location 2000H : 0500H to contents of 3000H : 0600H and store the result in 5000H : 0700H.

```
   MOV  CX,  2000H
   MOV  DS,  CX
   MOV  AX,  [500H]
   MOV  CX,  3000H
   MOV  DS,  CX
   MOV  BX,  [600H]
   ADD  AX,  BX
   MOV  CX,  5000H
   MOV  DS,  CX
   MOV  [0700H],  AX
   HLT
```

Program 7: Addition of 232-bit numbers.

```
DATA   SEGMENT
       NUM1  DW  0FFFFH,0FFFFH
       NUM2  DW  1111H,1111H  SUM
       DW  4  DUP(0)
DATA  ENDS

CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:MOV  AX,DATA
       MOV  DS,AX
       MOV  AX,NUM1
       ADD  AX,NUM2
       MOV  SUM,AX
       MOV  AX,NUM1+2
       ADC  AX,NUM2+2
       JNC  DOWN
       MOV  SUM+4,01H
OWN:   MOV  SUM+2,AX
       MOV  AH,4CH
       INT  21H
CODE  ENDS
END  START
```

Program 8: Sum of a series of 16-bit numbers; sum :32- bits

```
       MOV  AX,  0000H
       MOV  BX,  0000H
       MOV  SI,  0201H
       MOV  CX,  [SI]
BACK   INC  SI
       INC  SI
       ADD  AX,  [SI]
       JAE   GO
       INC  BX
GO     LOOP  BACK
       MOV  [0401],  AX
       MOV  [0403],  BX
       INT  3
```

Program 9: Subtraction of 16-bit numbers.

```
DATA   SEGMENT
       NUM  DW  4567H,2345H
       DIF  DW  1  DUP(0)
       DATA  E    NDS  CODE
CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:  MOV  AX,DATA
       MOV  DS,AX
       CLC
       LEA  SI,NUM
       MOV  AX,[SI]
       SBB  AX,[SI+2]
       MOV  DIF,AX
       MOV  AH,4CH
       INT  21H
CODE  ENDS
END  START
```

Program 10: Perform the sub (X-Y) where X and Y are 16-bit BCD numbers present in registers AX and CX.Store 16-bit BCD result in register SI.

```
   SUB  AL,  CL
   DAS
   MOV  BL,  AL
   MOV  AL,  AH
   SBB  AL,  CH
   DAS
   MOV  BH,  AL
   MOV  SI,  BX
   HLT
```

Program 11: Write a program for the multiplication of two 16-bit numbers in the memory locations. The result is 16 or 32 bits and is to be stored in another memory location.

```
DATA-SEG  SEGMENT
      MULTIPLICAND  DW  1111H
      MULTIPLIER  DW  1111H
      RESULT  DW  ?
DATA-SEG  ENDS

CODE-SEG   SEGMENT
ASSUME CS:  CODE-SEG,  DS:  DATA-SEG
START:MOV  DS,  AX
      MOV  AX,  DATA-SEG
      MOV  AX,  MULTIPLICAND
      MOV  DX,  MULTIPLIER
      MUL  AX,  DX
      MOV  RESULT,  AX
      MOV  RESULT  +  2,  DX
CODE-SEG  ENDS
   END      START
```

Program 12: Multiplication of two unsigned 32-bit numbers.

```
DATA
      MPD  DW  0C432H,765BH
      MPR  DW  3679H,0B397H
      RES  DW  4  DUP(0)
CODE
      MOV  AX,@DATA
      MOV  DS,AX
      MOV  BX,OFFSET  MPD
      MOV  AX,[BX]
      MUL  MPR
      MOV  RES,AX
      MOV  RES+2,DX
      MOV  AX,[BX]
      MUL  MPR+2
      ADD  RES+2,AX
      ADC  RES+4,DX
      JNC  L1
      INC  RES+6
L1:   MOV  AX,[BX+2]
      MUL  MPR
      ADD  RES+2,AX
      ADC  RES+4,DX
      JNC  L2
      INC  RES+6
L2:   MOV  AX,[BX+2]
      MUL  MPR+2
      ADD  RES+4,AX
      ADC  RES+6,DX
      MOV  AH,4CH
      INT  21H
END
```

Program 13: Write an ALP to perform multiplication of +4H and –5H. Store the result in register CX.

```
   MOV  AL,  04H
   MOV  BL,  05H
   NEG  BL
   IMUL  BL
   MOV  CX,  AX
   INT  3/HLT
```

Program 14: 16-bit division for unsigned numbers.

```
DATA   SEGMENT
       NUM1  DW  4567H,2345H
       NUM2  DW  4111H
       QUO  DW  2  DUP(0)
       REM  DW  1  DUP(0)
DATA  ENDS

CODE  SEGMENT
```

```
ASSUME  CS:CODE,DS:DATA
START:MOV  AX,DATA
       MOV  DS,AX
       MOV  AX,NUM1
       MOV  DX,NUM1+2
       DIV  NUM2
       MOV  QUO,AX
       MOV  REM,DX
       MOV  AH,4CH
       INT  21H
CODE  ENDS
END  START
```

Program 15: 16-bit division for signed numbers.

```
DATA   SEGMENT
       NUM1  DW  4567H,2345H
       NUM2  DW  4111H
       QUO  DW  2  DUP(0)
       REM  DW  1  DUP(0)
DATA  ENDS

CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:  MOV  AX,DATA
       MOV  DS,AX
       MOV  AX,NUM1
       MOV  DX,NUM1+2
       CWD
       IDIV  NUM2
       MOV  QUO,AX
       MOV  REM,DX
       MOV  AH,4CH
       INT  21H
CODE  ENDS
END  START
```

Program 16: ASCII addition of two numbers 1 H and 7 H.

```
MOV  AL,  31  H
MOV  BL,  37  H
```

```
AAA
HLT
```

Program 17: Write an ALP to find the average of two numbers.

```
MOV  AL,  72  H
ADD  AL,  78  H
ADC  AH,  00  H
SAR  AX,  1
MOV  [3000  H],  AL
HLT
```

Program 18: Write an 8086 assembly program to clear 10010 consecutive bytes. Assume CS and DS are already initialized.

```
      LEA  BX,  ADDR
      MOV  CX,  100
START MOV  [BX],  00H
      INC  BX
     LOOP  START
     HLT
```

Program 19: Write an ALP that reverses the contents of the bytes TABLE.

```
MOV  CL,  N
MOV  CH,  00  H
MOV  SI,  TABLE
2007  MOV  DI,  SI
2009  SUB  DI,  01
200B  ADD  DI,  CX
200D  SHR  CX,  01
200F  MOV  AL,  [SI]
2011  XCHG  AL,  [DI]
2013  MOV  [SI],  AL
2014  INC  SI
2015  DEC  DI
LOOP  200F  H
2019  HLT
```

Program 20: Write ALP that saves the contents of 8086's flags in memory location having an offset 101A H and then to reload the flags from the contents of the memory location having an offset 2238 H.

```
LAHF
MOV  [101A],  AH
MOV  AH,  [2238]
SAHF
HLT.
```

Program 21: Two blocks of data are present in memory from location 95000H and 96000H, compare the corresponding bytes of the two blocks and when the comparison is over or mismatching is found, then stop the program. Each block has 1K bytes.All the segment register used should contain the same base address.

```
      MOV  AX,  9500H
      MOV  DS,  AX
      MOV  SI,  0H
      MOV  AX,  9500H
      MOV  ES,  AX
      MOV  DI,  1000H
      MOV  CX,  3FFH
      CLD
REPZ: CMPSB
      HLT/INT3
```

Program 22: To find the smallest number in a data array. There are five 16-bit numbers in the given data array, the count is 0005.Two consecutive memory locations store a 16-bit number i.e. two bytes of a 16-bit number. The result is to be stored in memory locations 0251H and 0252H.

```
      MOV  AX,  FFFFH
      MOV  SI,  0200H
      MOV  CX,  [SI]
BACK  INC  SI
      INC  SI
      CMP  AX,[SI]
      JB    GO
      MOV  AX,  [SI]
GO    LOOP  BACK
      MOV  [0251],  AX
      INT  3
```

Program 23: To find the largest of 5 bytes.

```
DATA  SEGMENT.
      INPUT  DB  09,02,99,02,05
      COUNT  DW  05H
      LARGEST  DB  (?)
DATA  ENDS
```

```
CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:MOV  AX,DATA
      MOV  DS,AX
      MOV  SI,OFFSET[INPUT]
      MOV  CX,COUNT
      DEC  CX
      MOV  AL,[SI]
LAB:  INC  SI
      CMP  AL,[SI]
      JNC  LAB1
      MOV  AL,[SI]
LAB1: LOOP  LAB
      MOV  LARGEST,AL
      INT  3
CODE  ENDS
END  START
```

Program 24: To sort an array in ascending order (using the bubble sorting method).

```
DATA  SEGMENT
      ARR  DB  5,  4,  3,  1,  2
      COUNT  DW  05
DATA  ENDS

CODE  SEGMENT
      ASSUME  CS:CODE,  DS:DATA
START:MOV  AX,DATA
      MOV  DS,AX
      MOV  CX,COUNT
      DEC  CX
UP2:  MOV  DX,CX
      MOV  BX,0000H
UP1:  MOV  AL,ARR[BX]
      CMP  AL,ARR[BX+1]
      JBE  DOWN
      XCHG  AL,ARR[BX+1]
      MOV  ARR[BX],AL
      DOWN:INC  BX
```

```
        LOOP  UP1
        MOV  CX,DX
        LOOP  UP2
       INT  3H
CODE  ENDS
END  START
```

Program 25: ALP to find whether the given number is even or odd.

```
DATA   SEGMENT  X
        DW  27H
        MSG1  DB  19,13,'NUMBER  IS  EVEN$'
        MSG2  DB  10,13,'NUMBER  IS  ODD$'
DATA  ENDS

CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:  MOV  AX,DATA
        MOV  DS,AX
        MOV  AX,BX
        TEST  AX,01H
        JNZ  EXIT
        MOV  BL,2
        DIV  BL
        CMP  AH,0H
        JNZ  EXIT
        LEA  DX,MSG1
        MOV  AH,09H
        INT  21H
        JMP  LAST
EXIT:  LEA  DX,MSG2
        MOV  AH,09H
        INT  21H
LAST:  MOV  AH,4CH
        INT  21H
CODE  ENDS
END  START
```

Program 26: To find the number of logical ones and zeros in a given data.

```
DATA   SEGMENT
        X  DB  0AAH
```

```
      ONE  DB  (?)
      ZERO  DB  (?)
DATA  ENDS

CODE  SEGMENT
ASSUME  CS:  CODE,DS:DATA
START:  MOV  AX,DATA
      MOV  DS,AX
      MOV  AH,X
      MOV  BL,8
      MOV  CL,1
UP:   ROR  AH,CL
      JNC  DOWN
      INC  ONE
      JMP  DOWN1
DOWN:  INC  ZERO
DOWN1:  DEC  BL
      JNZ  UP
      MOV  AH,4CH
      INT  21H
CODE  ENDS
END  START
```

Program 27: To find out whether the given year is a leap year or not.

```
ASSUME  DS:DATA1,CS:CODE1
DATA1  SEGMENT
       MSG  DB  0AH,0DH,'ENTER  THE  YEAR$'
       NUMBER  DB  6,0,6  DUP('$')
       YS  DB  0AH,0DH,'YES,IT  IS  A  LEAP  YEAR$'
       N  DB  0AH,0DH,'NO,IT  IS  NOT  A  LEAP  YEAR$'
DATA1  ENDS

CODE1  SEGMENT
START:  MOV  AX,DATA1
       MOV  DS,AX
       LEA  DX,MSG
       MOV  AH,09H
       INT  21H
       LEA  DX,NUMBER
```

```
        MOV  AH,0AH
I       NT  21H
        LEA  BX,NUMBER+4
        MOV  AH,[BX]
        MOV  AL,[BX+1]
        AAD
        MOV  BL,04H
        DIV  BL
        AND  AH,0FFH
        JZ  YES
        LEA  DX,N
        MOV  AH,09H
        INT  21H
        JMP  DOWN
YES:    LEA  DX,YS
        MOV  AH,09H
        INT  21H
DOWN:   MOV  AH,4CH
        INT  21H
CODE1  ENDS
END  START
```

Program 28: To use software and hardware interrupts for receiving an input from keyboard and display it on screen.

```
DATA  SEGMENT
      INKEY  DB  ?
      BUF  DB  20  DUP(0)
      MES  DB  10,13,  BAPATLA  ENGINEERING  COLLEGE  $'  DATA  ENDS

CODE  SEGMENT
ASSUME  CS:CODE  ,  DS:DATA
START:  MOV  AX,DATA
      MOV  DS,AX
      MOV  AH,01H
      INT  21H
      MOV  INKEY,AL
      MOV  BUF,10
      MOV  AH,0AH
      LEA  DX,BUF
      INT  21H
      MOV  AH,06H
      MOV  DL,'A'
      INT  21H
      MOV  AH,09H
      LEA  DX,MES
      INT  21H
      MOV  AH,4CH
      INT  21H
CODE  ENDS
END  START
```

Program 29: To find factorial of a number using procedure.

```
HEX_ASC  PROC
      NUM  EQU  3
      MSG  DB  'FACTORIAL  OF  ',NUM+'0','  IS:'
      ASCRES  DB  4  DUP(?),'H',0DH,0AH,'$'
      RES  DW  ?
      HEXCODE  DB  '0123456789ABCDEF'
      MOV  DL,10H
      MOV  AH,0
      MOV  BX,0
      DIV  DL
      MOV  BL,AL
      MOV  DH,HEXCODE[BX]
      MOV  BL,AH
      MOV  DL,HEXCODE[BX]
RET
HEX_ASC  ENDP

FACT  PROC
      CMP  AX,01
      JE  EXIT
      PUSH  AX
      DEC  AX
      CALL  FACT
      POP  AX
      MUL  RES
      MOV  RES,A  X
EXIT:MOV  RES,01
RET

FACT  ENDP
MAIN:MOV  AX,@DATA
      MOV  DS,AX
      MOV  AX,NUM
      CALL  FACT
      MOV  AL,BYTE  PTR  RES+1
      MOV  ASCRES,DH
      MOV  ASCRES+1,DL
      MOV  AL,BYTE  PTR  RES
      MOV  ASCRES+2,DH
      MOV  ASCRES+3,DL
      MOV  AH,09H
      MOV  DX,OFFSET  MSG
      INT  21H
      MOV  AH,4CH
      INT  21H
      ALIGN  16
END  MAIN
```

Program 30: ALP for Fibonacci series.

```
DATA  SEGMENT
DATA  ENDS
CODE  SEGMENT
ASSUME  DS:DATA,CS:CODE
MAIN  PROC
       MOV  AH,02H
       MOV  DL,'1'
       INT  21H
       MOV  DL,','
       INT  21H
       MOV  DL,'1'
       INT  21H
       MOV  BL,01H
       MOV  CH,01H
       MOV  DL,'1'
       INT  21H
START1:MOV  CL,BL
       ADD  BL,CH
       MOV  CH,CL
       MOV  AL,BL
       MOV  AH,00
       MOV  BH,10
       DIV  BH
       MOV  CL,AH
       MOV  DL,AL
       MOV  AH,02H
       ADD  DL,30H
       INT  21H
       MOV  DL,CL
       ADD  DL,30H
       INT  21H
       MOV  DL,CL
       ADD  DL,30H
       INT  21H
       MOV  DL,'1'
       INT  21H
       CMP  BL,50H
       JL  START1
       MOV  AX,4C00H
       INT  21H
CODE  ENDS
END  MAIN
```

Program 31: Stepper motor interface.

```
DATA  SEGMENT
      PORTA  EQU  120H
      PORTB  EQU  121H
      PORTC  EQU  122H
      CWRD  EQU  123H
DATA  ENDS
```

```
CODE  SEGMENT
ASSUME  CS:CODE,DS:DATA
START:  MOV  AX,DATA
      MOV  DS,AX
      MOV  AL,80H
      MOV  DX,CWRD
      OUT  DX,AL
      MOV  DX,PORTA
      MOV  AL,88H
      OUT  DX,AL
      UP: CALL  DELAY
      ROL  AL,01H
      OUT  DX,AL
      JMP  UP
      DELAY:  MOV  CX,0FFFFH
      UP2:  MOV  BX,0FFH
      UP1:  DEC  BX
      JNZ  UP1
      DEC  CX
      JNZ  UP2
      RET
      MOV  AH,4CH
      INT  21H
CODE  ENDS
END  START
```

4.13 8086 INTERRUPTS AND INTERRUPT RESPONSES

An 8086 microprocessor interrupt can be caused from any one of the three sources. One source is an external signal applied to the non-maskable interrupt (NMI) input pin or to the interrupt (INTR) input pin. This is referred to as hardware interrupt. The second source of an interrupt is execution of interrupt instruction, INT. This is referred to as software interrupt. The third source of an interrupt is some error condition produced in the 8086 by the execution of an instruction. An example of this is the divide-by-zero interrupt. If you attempt to divide an operand by zero, the 8086 will automatically interrupt the currently executing program. At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions:

1. It decrements the stack pointer by 2 and pushes the flag register on the stack.
2. It disables the 8086 INTR interrupt input by clearing the interrupt flag (IF) in the flag register.

3. It resets the trap flag (TF) in the flag register.
4. It decrements the stack pointer by 2 and pushes the current code segment register contents on the stack.
5. It decrements the stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect far jump to the interrupt service procedure.

Figure 4.23 summarizes these steps in diagram form. As you can see, the 8086 pushes the flag register on the stack, disables the INTR input and the single step function and does essentially an indirect far call to the interrupt service procedure. An IRET instruction at the end of the interrupt service procedure returns execution to the main program. Now let's see how the 8086 actually gets to the interrupt procedure.

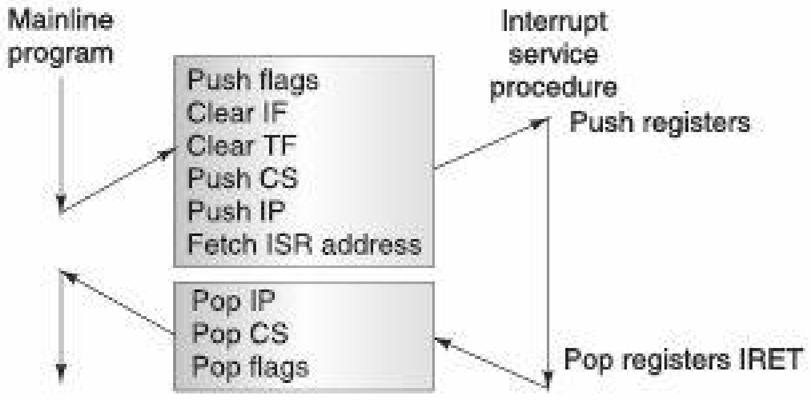


Fig. 4.23 8086 Interrupt Response.

When the 8086 does a far call to a procedure, it puts a new value in the code segment register and a new value in the instruction pointer. And thus, the 8086 gets the new values for CS and IP from four memory addresses. Likewise, when the 8086 responds to an interrupt, it goes to four memory locations to get the CS and IP values for the start of the interrupt-service procedure. Since 4 bytes are required to store the CS and IP values for each interrupt-service procedure, the table can hold the starting addresses for up to 256 interrupt procedures. The starting address of an interrupt-service procedure is often called the interrupt-vector or the interrupt-pointer, so the table is referred to as the interrupt-vector table or the interrupt-pointer table. Figure 4.24 shows how the 256 interrupt vectors are arranged in the table in memory. Note that the instruction pointer value is put in as the low word of the vector and the code segment register is put in as the high word of the vector. Each double word interrupt-vector is identified by a number from 0 to 255. Intel calls this number the type of the interrupt.

The lowest five types are dedicated to specific interrupts, such as the divide-by-zero interrupt, the single-step interrupt, and the non-maskable interrupt. Later in this chapter, we explain the operation of these interrupts in detail. Interrupt types 5 to 31 are reserved by Intel for use in more complex microprocessors, such as the 80286, 80386 and 80486. In a later chapter, we discuss some of these interrupt types. The upper 224 interrupt types, from 32 to 255, are available to use for hardware or software interrupts. As you can see in Figure 4.22, the vector for each interrupt type requires four memory locations. Therefore, when the 8086 responds to a particular type interrupt. It automatically multiplies the type by 4 to produce the desired address in the vector table. It then goes to that address in the table to get the starting address of the interrupt service procedure. We will show you later how you use instructions at the start of your program to load the starting address of a procedure into the vector table. Now that you have an overview of how the 8086 responds to interrupts, we will discuss one type of interrupt in detail and show you how to write a procedure to service that interrupt.

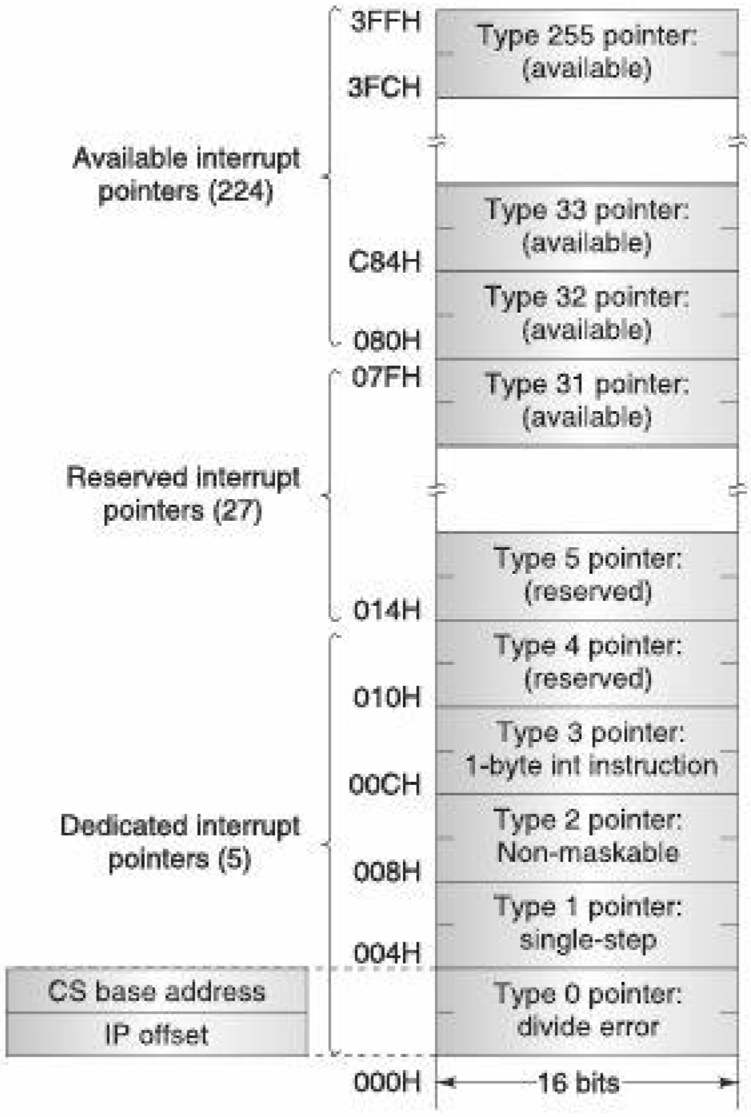

3FFH
Type 255 pointer: (available)
3FCH
Available interrupt pointers (224)
Type 33 pointer: (available)
C84H
Type 32 pointer: (available)
080H
07FH
Type 31 pointer: (available)
Reserved interrupt pointers (27)
Type 5 pointer: (reserved)
014H
Type 4 pointer: (reserved)
010H
Type 3 pointer: 1-byte int instruction
00CH
Dedicated interrupt pointers (5)
Type 2 pointer: Non-maskable
008H
Type 1 pointer: single-step
004H
CS base address
IP offset
Type 0 pointer: divide error
000H
16 bits

Fig. 4.24 8086 Interrupt Pointer Table.

An 8086 Interrupt Response Example—Type 0

The easiest 8086 interrupt to understand is the divide-by-zero interrupt, identified as type 0 in Figure 4.24. Before we get into the details of the type 0 interrupt response, let's refresh your memory about how the 8086 DIV and IDIV instructions work. The 8086 DIV instruction allows you to divide a 16-bit unsigned binary number in AX register by an 8-bit unsigned number from a specified register or memory location. The 8-bit result (quotient) from this division will be left in the AL register. The 8-bit remainder will be left in the AH register. The DIV instruction also allows you to divide a 32-bit unsigned binary number in DX and AX by a 16-bit number in a specified register or memory location. The 16-bit quotient from this division is left in the AX register and the 16-bit remainder is left in the DX register. In the same manner, the 8086 IDIV instruction allows you to divide a 16-bit signed number in AX by an 8-bit signed number in a specified register or a 32-bit signed number in DX and AX by a 16-bit signed number from a specified register or memory location. If the quotient from dividing a 16-bit number is too large to fit in AL or the quotient from bit dividing a 32 number is too large to fit in AX, the result of the division will be meaningless. A special case of this is where an attempt is made to divide a 32-bit number or a 16-bit number by zero. The result of dividing by zero is infinity (actually undefined), which is somewhat too large to fit in AX or AL. Whenever the quotient from a DIV or IDIV operation is too large to fit in the result register, the 8086 will automatically do a type 0 interrupt. In response to this interrupt the 8086 proceeds as follows. As you can see in Figure 4.24, it gets the new value for CS from addresses 0002H and 0003H and the new value for IP from addresses 0000H and 000IH. After the starting address of the procedure is loaded into CS and IP, the 8086 then fetches and executes the first instruction of the procedure.

At the end of the interrupt-service procedure, an IRET instruction is used to return execution to the interrupted program. The IRET instruction pops the stored value of IP off the stack and increments the stack pointer by 2. It then pops the stored value of CS off the stack and increments the stack pointer again by 2. Finally, it restores the flags, by popping off the stack the values stored during the interrupt response and increments the stack pointer by 2. Remember from the previous paragraph that during its interrupt response, the 8086 disables the INTR and single-step interrupts by clearing IF and TF. If the INTR input and/or the trap interrupt were enabled before the interrupt, they will be enabled upon return to the interrupted program. The reason for this is that flags from the interrupted program were pushed on the stack before IF and TF were cleared by the 8086 in its interrupt response.

To summarize, then, IRET returns execution to the interrupted program and restores IF and TF to the state they were in before the interrupt. Now that we have described the type 0 response, we can show you how to write a program to handle this interrupt.

4.14 8086 MICROPROCESSOR INTERRUPT TYPES

In this section, we discuss in detail the different ways an 8086 can be interrupted and how the 8086 responds to each of these interrupts. We discuss these in order, starting with type 0, so that you can easily find a particular discussion when you need to refer back to it. Read through all the types to get an overview and we then focus on the details of the hardware-caused NMI interrupt. The software interrupts produced by the INT instruction and the hardware interrupt produced by applying a signal to the INTR input pin.

Divide-by-Zero Interrupt-Type 0

As we described in the preceding section, the 8086 will automatically do a type 0 interrupt, if the result of a DIV operation or an IDIV operation is too large to fit in the destination register. For a type 0 interrupt, the 8086 pushes the flag register on the stack. Resets IF and TF, and pushes the return address (CS and IP) on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 00002H in the interrupt-pointer table and the IP value for the start of the procedure from address 00000H in the interrupt-pointer table. Since the 8086 type 0 response is automatic and cannot be disabled in any way, you have to account for it in any program where you use the DIV or IDIV instruction. One way is to make sure the result will never be too large for the result register. We first make sure the divisor is not zero and then we do the division in several steps so that the result of the division will never be too large.

Another way to account for the 8086 type 0 response is to simply write an interrupt-service procedure, which takes the desired action when an invalid division occurs. The advantage of this approach is that you don't have the overhead of a more complex division routine in your mainline program. The 8086 automatically does the checking and does the interrupt procedure only if there is a problem.

Single-Step Interrupt—Type 1

When you tell a system to single-step, it will execute one instruction and stop. You can then examine the contents of registers and memory locations. If they are correct, you can tell the system to go on and execute the next instruction. In other words, when in single-step mode, a system will stop after it executes each instruction and wait for further direction from you. The 8086 trap flag and type 1 interrupt response make it quite easy to implement a single-step feature in an 8086 based system.

If the 8086 trap flag is set, the 8086 will automatically do a type 1 interrupt after each instruction executes. When the 8086 does a type 1 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the type interrupt-service procedure from address 0006H and it gets the IP value for the start of the procedure from address 0004H. The tasks involved in implementing single stepping are: Set the trap flag, write an interrupt-service procedure which saves all registers on the stack, where they can later be examined or perhaps displayed on the CRT, and load the starting address of the type 1 interrupt-service procedure into addresses 0004H and 0006 H.

The actual single-step procedure will depend very much on the system on which it is to be implemented. We do not have space here to show you the different ways to do this. We will, however, show you how the trap flag is set or reset because this is somewhat unusual. The 8086 has no instructions to directly set or reset the trap flag. These operations are done by pushing the flag register on the stack, changing the trap flag bit to what you want it to be and then popping the flag register back off the stack. Here is the instruction sequence to set the trap flag.

```
PUSHF                          ; Push flags on stack
MOV BP, SP                     ; Copy SP to BP for use as index
OR WORD PTR (BP+0), 0100H      ; Set TF bit
POPF                           ; Restore flag register
```

To reset the trap flag, simply replace the OR instruction in the preceding sequence with the instruction AND WORD PTR [BP+0], 0FEFFH. The trap flag is reset when the 8086 does a type 1 interrupt. So the single-step mode will be disabled during the interrupt-service procedure.

Maskable Interrupt—Type 2

The 8086 will automatically do a type 2 interrupt response when it receives a low-to-high transition n its NMI input pin. When it does a type 2 interrupt, the 8086 will push the flags on the stack.

Reset TF and IF, and push the CS value and the IP value for the next instruction on the stack. It will then get the CS value for the start of the type 2 interrupt-service procedure from address 0000AH and the IP value for the start of the procedure from address 0008H.

The name non-maskable given to this input pin on the 8086 means that the type 2 interrupt response cannot be disabled (masked) by any program instructions. Because this input cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of. We could, for example, have a pressure-sensor on a large steam boiler connected to the NMI input. If the pressure goes above some preset limit, the sensor will send an interrupt signal to the 8086. The type 2 interrupt-service procedure for this case might turn off the fuel to the boiler, open a pressure-relief valve and sound an alarm.

Another common use of the type 2 interrupt is to save program data in case of a system power failure. Some external circuitry detects when the ac power to the system fails and sends an interrupt-signal to the NMI input. Because of the large filter capacitors in most power supplies, the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for a type 2 interrupt-service procedure to copy program data to some RAM which has a battery backup power supply. When the ac power returns program data can be restored from the battery-backed RAM and the program can resume execution where it left off.

Breakpoint Interrupt—Type 3

The type 3 interrupt is produced by execution of the INT 3 instruction. The main use of the type 3 interrupt is to implement a breakpoint function in a system. When you insert a break-point, the system executes the instructions up to the breakpoint and then goes to the break-point procedure. Unlike the single-step feature, which stops execution after each instruction, the breakpoint feature executes all the instructions up to the inserted breakpoint and then stops execution.

When you tell most 8086 systems to insert a breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CCH. The 8086 code for the INT 3 instruction. When the 8086 executes this INT 3 instruction, it pushes the flag register on the stack, resets TF and IF and pushes the CS and IP values for the next mainline instruction on the stack. The 8086 then gets the CS value of the start of the type 3 interrupt-service procedure from address 000EH and the IP value for the procedure from address 000CH. A breakpoint interrupt-service procedure usually saves all the register contents on the stack. Depending on the system, it may then send the register contents to the CRT display and wait for the next command from the user or in a simple system. It may just return control to the user. In this case an examine register command can be used to check if the register contents are correct at that point in the program.

Overflow Interrupt—Type 4

If the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location, the 8086 overflow flag (OF) will be set. For example, if you add the 8-bit signed number 01101100 (108 decimal) and the 8-bit signed number 01010001 (81 decimal), the result will be 10111101 (189 decimal). This would be the correct result if we were adding unsigned binary numbers, but it is not the correct signed result. For signed operations, the 1 in the most significant bit of the result indicates that the result is negative and in 2's complement form, the result, 10111101, then actually represents –67 decimal, which is obviously not the correct result for adding +108 and +89. An overflow error in program can be detected and responded by two major ways. One way is to put the jump if overflow instruction, JO, immediately after the arithmetic instruction. If the overflow flag is set as a result of the arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds to the overflow in the way you want.

An overflow instruction, INTO can be used immediately after the arithmetic instruction in the program and this is the second way of detecting and responding to an overflow error. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will, simply function as an NOP. However, if the overflow flag is set, indicating an overflow error, the 8086 will do a type 4 interrupt after it executes the INTO instruction.

When the 8086 does a type 4 interrupt, it pushes the flag register on the stack, resets TF and IF and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 0012H and the IP value for the procedure from address 0010H instructions in the interrupt-service procedure then perform the desired response to the error condition. The procedure might, for example, set a "flag" in a memory location as we did on the BAD-DIV procedure. The advantage of using the INTO and type 4 interrupt approach is that the error routine is easily accessible from any program.

4.15 SOFTWARE INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. Any interrupt which is desired is specified as part of the instruction. For example, the instruction INT 32 will cause the 8086 to do a type 32 interrupt response. The 8086 will push the flag register on the stack, reset TF and IF and push the CS and IP values of the next instruction on the stack. It will then get the CS and IP values for the start of the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher. For a type 32 interrupt, then, the IP value will be put at 4 × 32 or 128 decimal (80H) and the CS value will be put at address 82H in the interrupt-vector table. Software interrupts produced by the INT instruction have many uses. In a previous section, we discussed the use of the INT 3 instruction to insert breakpoints in programs for debugging, another use of software interrupts is to test various interrupt-service procedures. For example, an INT 0 instruction is used to send execution to a divide-by-zero interrupt-service procedure without having to run the actual division program. As another example, an INT 2 instruction is used to send execution to an NMI interrupt-service procedure. This allows you to test the NMI procedure without needing to apply an external signal to the NMI input of the 8086. In a later section of the chapter, we show an example of another important application of software interrupts.

4.16 INTR INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INTR input allows some external signal to interrupt execution of a program. Unlike the NMI input, however, INTR can be masked (disabled) so that it cannot cause an interrupt. If the Interrupt Flag (IF) is cleared, then the INTR input is disabled. IF can be cleared at any time with the Clear Interrupt Instruction, CLI. If the interrupt flag is set, the INTR input will be enabled. IF can be set at any time with the Set Interrupt Instruction (SII). When the 8086 is reset, the interrupt flag is automatically cleared. Before the 8086 can respond to an interrupt signal on its INTR input, you have to set IF with an STI instruction. The 8086 was designed this way so that ports, timers, registers, etc. can be initialized before the INTR input is enabled. In other words, this allows you to get the 8086 ready to handle interrupts before letting an interrupt in, just as you might want to get yourself ready in the morning with a cup of coffee before turning on the telephone and having to cope with the interruptions it produces. Remember that the interrupt flag (IF) is also automatically cleared as part of the response of an 8086 to an interrupt. This is done for two reasons. First, it prevents signal on the INTR input from interrupting a higher priority interrupt-service procedure in progress. However, if you want another INTR input signal to be able to interrupt an interrupt procedure in progress, you can re-enable the INTR input with an STI instruction at any time. The second reason for automatically disabling the INTR input at the start of an INTR interrupt-service procedure is to make sure that a signal on the INTR input does not cause the 8086 to interrupt itself continuously. The INTR input is activated by a high level. In other words, whenever the INTR input is high and INTR is enabled, the 8086 will be interrupted. If INTR were not disabled during the first response, the 8086 would be continuously interrupted and would never get to the actual interrupt-service procedure.

The IRET instruction at the end of an interrupt-service procedure restores the flags to the condition they were in before the procedure by popping the flag register off the stack. This will re-enable the INTR input. If a high-level signal is still present on the INTR input, it will cause the 8086 to be interrupted again. If you do not want the 8086 to be interrupted again by the same input signal, you have to use external hardware to make sure that the signal is made low again before you re-enable INTR with the STI instruction or before the IRET from the INTR service procedure. When the 8086 responds to an INTR interrupt signal, its response is somewhat different from its response to other interrupts. The main difference is that for an INTR interrupt, the interrupt type is sent to the 8086 from an external hardware device such as the 8259A priority interrupt controller, as shown in Figure 4.25.

When an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the INTR input of the 8086. If the INTR input of the 8086 has been enabled with an STI instruction, the 8086 will respond. The 8086 first responds to the two interrupt acknowledge machine ready signal, it cannot easily be doing other tasks. In systems where the microcomputer must be doing many tasks, polling is a waste of time, so interrupt input and output is used. In this case, the data ready or strobe signal is connected to an interrupt input on the microcomputer. The microcomputer is busy in doing its other tasks until it is interrupted by a data ready signal from the external device. An interrupt-service procedure can read in or send out the desired data in a few micro seconds and return execution to the interrupted program. The time taken by input or output operation will be very less compared to the microprocessor time.

4.17 8288 BUS CONTROLLER–BUS COMMAND AND CONTROL SIGNALS

8086 does not directly provide all the signals that are required to control the memory, I/O and interrupt interfaces. Specially the WR, M/IO, DT/R, DEN, ALE and INTA, signals are no longer produced by the 8086. Instead it outputs three status signals S_0, S_1, S_2 prior to the initiation of each bus cycle. This 3-bit bus status code identifies which type of bus cycle is to follow. S_2, S_1, S_0 are input to the external bus controller device, the bus controller generates the appropriately timed command and control signals.

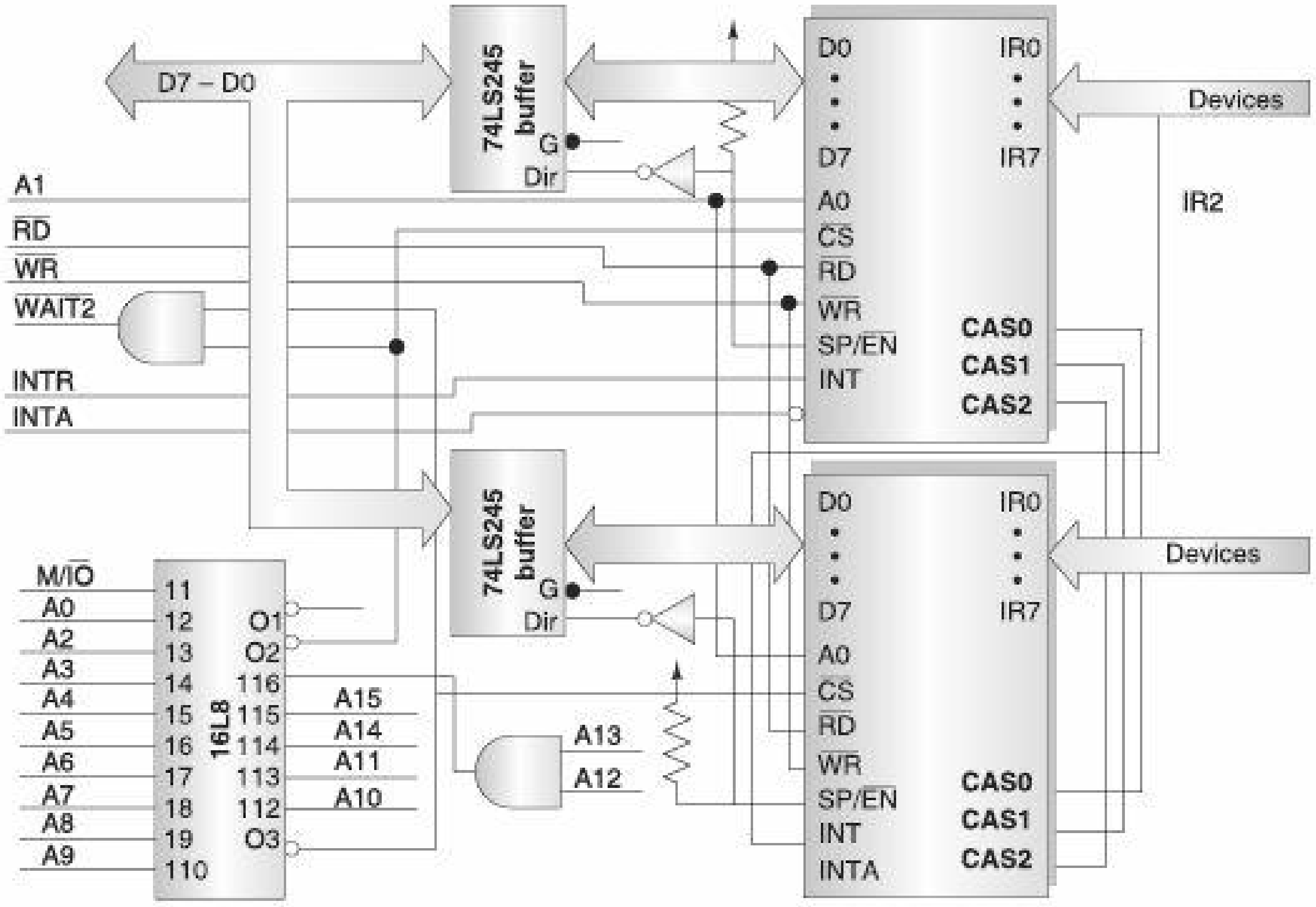


Fig. 4.25 Block Diagram Showing an 8259 Connected to an 8086.

The 8288 produces one or two of these eight command signals for each bus cycle. For instance, when the 8086 outputs the code $S_2S_1S_0$ equals 001, it indicates that an I/O read cycle is to be performed. If the code 111 is output by the 8086, it is signaling that no bus activity is to take place. The control outputs produced by the 8288 are DEN, DT/$\overline{R}$ and ALE. These 3 signals provide the same functions as those described for the minimum system mode. This set of bus commands and control signals is compatible with the multibus and industry standard for interfacing microprocessor systems.

Table 4.10 CPU Cycle and Commands of 8288

S_2 S_1 S_0	S_0CPU Cycles	8288 Command
0 0 0	Interrupt Acknowledge	INTA
0 0 1	Read I/O Port	IORC
0 1 0	Write I/O Port	IOWC, IOWC
0 1 1	Halt	None
1 0 0	Instruction Fetch	MRDC
1 0 1	Read Memory	MRDC
1 1 0	Write Memory	MWTC, AMWC
1 1 1	Passive	None

4.17.1 The Output of 8289 are Bus Arbitration Signals

Bus busy (BUSY), common bus request (CBRQ), bus priority out (BPRO), bus priority in (BPRN), bus request (BREQ) and bus clock (BCLK) correspond to the bus exchange signals of the multibus and are used to lock other processor off the system bus during the execution of an instruction by the 8086. In this way, the processor can be assured of uninterrupted access to common system resources such as **global memory.**

4.17.2 Queue Status Signals

Two new signals that are produced by the 8086 in the maximum mode system are queue status outputs QS_0 and QS_1. Together they form a 2-bit queue status code, QS_1, QS_0. Table 4.11 shows the four different queue status.

Table 4.11

QS_1	QS_0	*Queue Status*
0	0	No Operation. During the last clock cycle, nothing was taken from the queue.
0	1	First Byte. The byte taken from the queue was the first byte of the instruction.
1	0	Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction.
1	1	Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction.

4.17.3 Local Bus Control Signal–Request/Grant Signals

In a maximum mode configuration, the minimum mode HOLD, HLDA interface is also changed. These two are replaced by request/grant lines RQ/GT0 and RQ/GT1, respectively. They provide a prioritized bus access mechanism for accessing the local bus 8086 maximum mode.

4.18 PROCEDURE AND MACROS OF 8086

4.18.1 Procedure

Named group of statement which performs particular task is called procedure. Procedure or subroutine may require input data or constants for their execution. Their data or constants may be passed to the subroutine by main program or some subroutine may access readily available data of constants available in memory.

Generally, the following techniques are used to pass input/parameter to procedures in ALP.

(a) Using global declared variable

(b) Using registers of CPU architecture

(c) Using memory location

(d) Using stack

(e) Using PUBLIC and EXTERN

4.18.1.1 Using Global Declared Variable

A variable or a parameter label may be declared global in the main program and the same variable or parameter label can be used by all the procedures of the application. Examples of passing parameters

```
ASSUME CS: CODE,DS: DATA
DATA SEGMENT
NUMBER EQY 77H GLOBAL
DATA ENDS
CODE1 SEGMENT
START: MOV          AX,DATA
MOV DS,AX
–
–
CODE1 ENDS
ASSUME CS: CODE2
CODE2 SEGMENT
START: MOV          AX,DATA
                 DS,AX MOV
–
–
CODE2 ENDS
END START
```

4.18.1.2 Using Registers of CPU Architecture

CPU general-purpose registers may be used to pass parameters to the procedures. The main program may store the parameters to be passed to the procedure in the variable CPU registers and the procedure may use the same register content for execution. The original content of the used CPU register may change during execution of the procedure. This may be avoided by pushing all the register content to be used to the stack sequentially at the start of the procedure and popping all the register contents at the end of the procedure in the opposite sequence.

Example:

```
   ASSUME CS: CODE,DS: DATA
   CODE SEGMENT
   START: MOV AX 5555H
   MOV BX 5456H
   –
   –
```

```
PROCEDURE  P1  NEAR
—
—
ADD  AX,  BX
—
—
RET
P1  ENDP
CODE  ENDS
END  START
```

4.18.1.3 Using Memory Location

Memory location may also be used to pass parameter to a procedure in the same way as the registers. A main program may store the parameter to be passed to a procedure at known memory address location and the procedure may use the same location for accessing the parameter.

Example:

```
ASSUME  CS:  CODE,DS:  DATA
DATA  SEGMENT
NUM  DB  (55H)
COUNT  EQY  77H
DATA  ENDS
CODE  SEGMENT
START:  MOV  AX,DATA
DS,AX  MOV
—
—
CALL  ROUTINE
—
—
PROCEDURE  ROUTINE  NEAR
MOV  BX,  NUM
MOV  CX,  COUNT
—
ROUTINE  ENDP
CODE  ENDS
END  START
```

4.18.1.4 Using Stack

Stack memory can also be used to pass parameters to procedure. A main program may store the parameters to be passed to a procedure in its CPU registers. The registers will further be pushed on to the stack. The procedure during its execution pops back the appropriate parameters as and when required. This procedure of popping back the parameters must be implemented carefully because besides the parameters to be passed to the procedure the stack contains other important information like contents of other pushed registers, return addresses from the current procedure and other procedure or interrupt service routines.

Example:

```
ASSUME CS: CODE,SS: STACK
STACK SEGMENT
STACKDATA DB 200H DUP (?)
STACK ENDS
CODE SEGMENT
START: MOV AX, STACK
MOV SS,AX
MOV BX, 55H
MOV CX, 10H
- -
PUSH AX
PUSH CX
CALL ROUTINE
- -
- -
PROCEDURE ROUTINE NEAR
- -
MOV DX, SP
ADD SP, 02H
POP CX
POP BX
MOV SP, DX
- -
ROUTINE ENDP
CODE ENDS
END START
```

4.18.1.5 Using PUBLIC and EXTERN

For passing the parameters to procedures using the PUBLIC and EXTERN directives, must be declared PUBLIC (for all routine) in the main routine and the same should be declared EXTERN in the procedure. Thus, the main program can pass the PUBLIC parameter to a procedure in which it is declared EXTERN (external).

Example:

```
ASSUME CS: CODE,DS: DATA
DATA SEGMENT
PUBLIC NUMBER EQY 77H
DATA ENDS
CODE SEGMENT
START: MOV      AX,DATA
MOV DS,AX
- -
- -
CALL ROUTINE
- -
PROCEDURE ROUTINE NEAR
EXTERN NUMBER
MOV AX, NUMBER
- -
ROUTINE ENDP
CODE ENDS
END START
```

4.18.2 Macros

The macro is also similar to subroutine. Suppose, a number of instructions are repeating through in the main program, the listing becomes lengthy. So a macro definition, i.e., a label, is assigned within the repeatedly appearing string of instructions. The process of assigning a label or macro name to the string is called macro. A macro within the macro is called nested macro. The macro name or macro definition is then used throughout the main program to refer to the string of instructions. The difference between a macro and a subroutine is that in the macro the complete code of the instructions string is inserted at each place where the macro name appears. Hence, the EXE file becomes lengthy. Macro does not utilize the service of stack. There is no question of transfer of control as the program using the macro inserts the complete code of the macro at every reference of the macro name. On the other hand, subroutine is called whenever necessary, i.e., the control of execution is transferred to the subroutine, every time it is called. The executable code in case of the subroutines becomes smaller as the subroutine appears only once in the complete code. Thus, the EXE file is smaller as compared to the program using macro. The control is transferred to the subroutine whenever it is called, and this utilizes the stack service. The program using subroutine requires less memory space for execution than the using the macro. Macro requires less time for execution, as it does not contain CALL and RET instructions as the subroutine do.

4.18.2.1 Defining a MACRO

A MACRO can be defined anywhere in the program using the directives MACRO and ENDM. The label prior to MACRO is the macro name which should be used in the actual program. The ENDM directive marks the end of the instructions. The following macro DISP displays the message MSG on the CRT. The syntax is as given:

```
DISP  MACRO
MOV  AX,  SEG  MSG
MOV  DS,  AX
MOV  DX,  OFFSET  MSG
MOVE  AH,  09H
INT  21H
ENDM
```

The above definition of macro assign the name DISP to the instruction sequence between the directives MACRO and ENDM. While assembling, the above sequence of the instructions will replace the label 'DISP', whenever it appears in the program. A macro may be called by quoting its name, along with any value to be passed to the macro. Calling a macro means inserting the statements and instructions represented by the macro directly at the place of the macro name in the program.

4.18.2.2 Passing Parameters to a MACRO

Using parameters in a definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called. For example, the DISP macro written above can be made to display two different messages—MSG1 and MSG2 as shown.

```
DISP  MACRO
MOV  AX,  SEG  MSG
MOV  DS,  AX
MOV  DX,  OFFSET  MSG
MOVE  AH,  09H
INT  21H
ENDM
```

This parameter MSG can be replaced by MSG1 and MSG2 while calling the macro as shown below:

```
ASSUME CS: CODE,DS: DATA
CODE SEGMENT
START: MOV AX,DATA
- -
DISP MSG1
- -
DISP MSG2
- -
CODE ENDS
END START
MSG1 DB OAH,ODH,"PROGRAM TERMINATED NORMALLY"
MSG1 DB OAH,ODH,"Retry, Abort, Fail"
```

4.19 COPROCESSOR 8087

4.19.1 Introduction

- Each processor in the 80 × 86 family has a corresponding coprocessor with which it is compatible.
- Numeric processor extension (NPX),
- Numeric data processor (NDP),
- Floating point unit (FPU).

4.19.2 Compatible Processor and Coprocessor

Processors

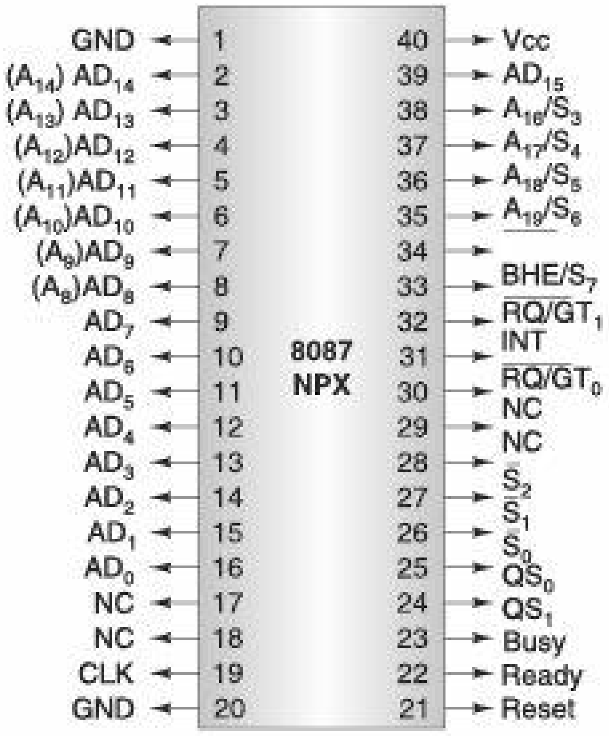
1. 8086 & 8088
2. 80286
3. 80386DX
4. 80386SX
5. 80486DX
6. 80486SX

Coprocessors

1. 8087
2. 80287, 80287XL
3. 80287, 80387DX
4. 80387SX
5. It is inbuilt
6. 80487SX

4.19.3 Architecture of 8087

1. Control Unit
2. Execution Unit



4.19.3.1 Control Unit

Control unit: To synchronize the operation of the coprocessor and the processor. This unit has a Control word and Status word and Data Buffer. If instruction is an ESC (coprocessor) instruction, the coprocessor executes it, if not the microprocessor. Status register reflects the overall operation of the coprocessor.

4.19.3.2 Status Register

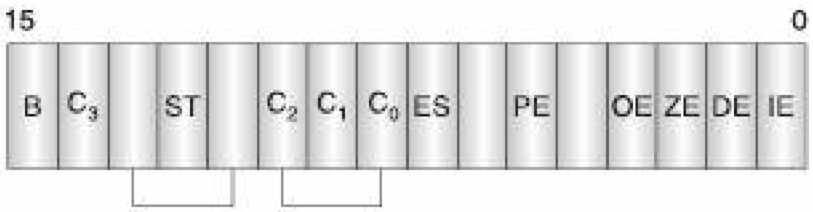


Fig. 4.26 Status Register.

C_3-C_0 Condition code bits

TOP Top-of-stack (ST)

ES Error summary

PE Precision error

UE Underflow error

OE Overflow error

ZE Zero error

DE Denormalized error

IE Invalid error

B Busy bit

B–Busy bit indicates that coprocessor is busy executing a task. Busy can be tested by examining the status or by using the FWAIT instruction. Newer coprocessor automatically synchronizes with the microprocessor, so busy flag need not be tested before performing additional coprocessor tasks.

TOP–Top of the stack (ST) bit indicates the current register address as the top of the stack.

ES–Error summary bit is set if any unmasked error bit (PE, UE, OE, ZE, DE, or IE) is set. In the 8087, the error summary also causes a coprocessor interrupt.

PE–Precision error indicates that the result or operand executes selected precision.

UE–Underflow error indicates the result is too large to be represents with the current precision selected by the control word.

OE–Overflow error indicates a result that is too large to be represented. If this error is masked, the coprocessor generates infinity for an overflow error.

ZE–A zero error indicates the divisor was zero while the dividend is a non-infinity or non-zero number.

DE–Denormalized error indicates at least one of the operand is denormalized.

IE–Invalid error indicates a stack overflow or underflow, indeterminate from (0/0, 0, –0, etc.) or the use of a NAN as an operand. This flag indicates error such as those produced by taking the square root of a negative number.

4.19.3.3 Control Register

Control register selects precision, rounding control and infinity control. It also masks and unmasks the exception bits that correspond to the rightmost six bits of status register. Instruction FLDCW is used to load the value into the control register.

IC Infinity control	**ZM** Division by zero mask
RC Rounding control	**DM** Denormalized operand mask
PC Precision control	**IM** Invalid operand mask
PM Precision Mask	
UM Underflow mask	
OM Overfl ow mask	

IC–Infinity control selects either affine or projective infinity. Affine allows positive and negative infinity, while projective assumed infinity is unsigned.

INFINITY CONTROL

0 = Projective

1 = Affine

RC–Rounding control determines the type of rounding.

ROUNDING CONTROL

00 = Round to nearest or even

01 = Round down towards minus infinity

10 = Round up towards plus infinity

11 = Chop or truncate towards zero

PC- Precision control sets the precision of the result as defined in the table.

PRECISION CONTROL

00 = Single precision (short)

01 = Reserved

10 = Double precision (long)

11 = Extended precision (temporary)

4.19.3.4 Exception Masks

It determines whether the error indicated by the exception affects the error bit in the status register. If a logic is placed in one of the exception control bits, corresponding status register bit is masked off.

4.19.3.5 Numeric Execution Unit

This unit performs all operations that access and manipulate the numeric data in the coprocessor's registers. Numeric registers in NUE are 80 bits wide. NUE is able to perform arithmetic, logical and transcendental operations as well as supply a small number of mathematical constants from its on-chip ROM. Numeric data is routed into two parts—a 64-bit mantissa bus and a 16-bit sign /exponent bus.

Multiplexed address-data bus lines are connected directly from the 8086 to 8087. The status lines and the queue status lines connected directly from 8086 to 8087. The Request/Grant signal RQ/GT0 of 8087 is connected to RQ/GT1 of 8086. BUSY signal 8087 is connected to TEST pin of 8086. Interrupt output INT of the 8087 to NMI input of 8086. This intimates an error condition. The main purpose of the circuitry between the INT output of 8087 and the NMI input is to make sure that an NMI signal is not present upon reset, to make it possible to mask NMI input and to make it possible for other devices to cause an NMI interrupt. BHE pin is connected to the system BHE line to multiplexed address-data bus lines are connected directly from the 8086 to 8087. The status lines and the queue status lines connect directly from 8086 to 8087.

The Request/Grant signal RQ/GT0 of 8087 is connected to RQ/GT1 of 8086. BUSY signal 8087 is connected to TEST pin of 8086. Interrupt output INT of the 8087 to NMI input of 8086. This intimates an error condition. The main purpose of the circuitry between the INT output of 8087 and the NMI input is to make sure that an NMI signal is not present upon reset, to make it possible to mask NMI input and to make it possible for other devices to cause an NMI interrupt. BHE pin is connected to the system BHE line to enable the upper bank of memory. The RQ/ GT1 input is available so that another coprocessor such as 8089 I/O processor can be connected

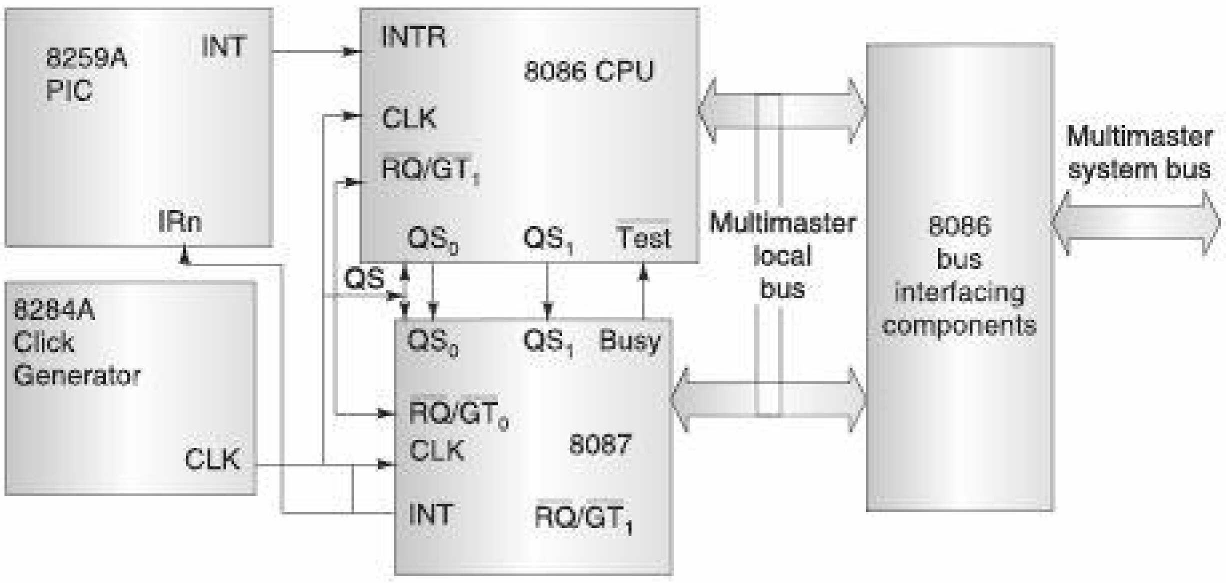


Fig. 4.27

and function in parallel with the 8087. One type of cooperation between the two processors that you need to know about it is how the 8087 transfers data between memory and its internal registers. When 8086 reads an 8087 instruction that needs data from memory or wants to send data to memory, the 8086 sends out the memory address code in the instruction and sends out the appropriate memory read or memory write signal to transfer a word of data. In the case of memory read, the addressed word will be kept on the data bus by the memory. The 8087 then simply reads the word of data bus. The 8086 ignores this word. If the 8087 only needs this one word of data, it can then go on and executes its instruction. Some 8087 instructions need to read in or write out up to 80-bit word. For these cases 8086 outputs the address of the first data word on the address bus and outputs the appropriate control signal. The 8087 reads the data word on the data bus by memory or writes a data word to memory on the data bus. The 8087 grabs the 20-bit physical address that was output by the 8086. To transfer additional words it needs to/from memory, the 8087 then takes over the buses from 8086. To take over the bus, the 8087 sends out a low-going pulse on RQ/GT0 pin. The 8086 responds to this by sending another low- going pulse back to the RQ/GT0 pin of 8087 and by floating its buses. The 8087 then increments the address it grabbed during the first transfer and outputs the incremented address on the address bus. When the 8087 outputs a memory read or memory write signal, another data word will be transferred to or from the 8087. The 8087 continues the process until it has transferred all the data words required by the instruction to/from memory. When the 8087 is using the buses for its data transfer, it sends another low-going pulse out on its RQ/GT0 pin to 8086 to know it can have the buses back again. The next type of the synchronization between the host processor and the coprocessor is that is required to make sure the 8086 does not attempt to execute the next instruction before the 8087 has completed an instruction. Taking one situation, in the case where the 8086 needs the data produced by the execution of an 8087 instruction to carry out its next instruction. In the instruction sequence, for example, the 8087 must complete the FSTSW STATUS instruction before the 8086 will have the data it needs to execute the MOV AX, STATUS instruction. Without some mechanism to make the 8086 wait until the 8087 completes the FSTSW instruction, the 8086 will go on and execute the MOV

AX, STATUS with erroneous data. We solve this problem by connecting the 8087 BUSY output to the TEST pin of the 8086 and putting on the WAIT instruction in the program. While 8087 is executing an instruction it asserts its BUSY pin high. When it is finished with an instruction, the 8087 will drop its BUSY pin low. Since the BUSY pin from 8087 is connected to the TEST pin 8086 the processor can check its pin of 8087 whether it finished its instruction or not. You place the 8086 WAIT instruction in your program after the 8087 FSTSW instruction. When 8086 executes the WAIT instruction it enters an internal loop where it repeatedly checks the logic level on the TEST input. The 8086 will stay in this loop until it finds the TEST input asserted low, indicating the 8087 has completed its instruction. The 8086 will then exit the internal loop, fetch and execute the next instruction.

Example:

```
FSTSW  STATUS             ;copy 8087 status word to memory
MOV  AX,  STATUS          ;copy status word to AX to check; bits
```

(a) In this set of instructions, we are not using WAIT instruction. Due to this, the flow of execution of command will take place continuously even though the previous instruction has not finished its completion of its work, so we may lose data.

```
FSTSW  STATUS          ;copy 8087 status word to memory
FWAIT                  ;wait for 8087 to finish before-
                       ;doing next 8086 instruction
MOV  AX,STATUS         ;copy status word to AX to check; bits
```

(b) In this code, we are adding up FWAIT instruction so that it will stop the execution of the command until the above instruction has finished its work so that you are not losing data and after that you will allow to continue the execution of instructions. Another case where you need synchronization of the processor and the coprocessor is the case where a program has several 8087 instructions in sequence. The 8087 executes only one instruction at a time so you have to make sure that 8087 has completed one instruction before you allow the 8086 to fetch the next 8087 instruction from memory. Here again you use the BUSY-TEST connection and the FWAIT instruction to solve the problem. If you are hand coding, you can just put the 8086 WAIT(FWAIT) instruction after each instruction to make sure that the instruction is completed before going on to the next. If you are using the assembler which accepts 8087 mnemonics, the assembler will automatically insert the 8-bit code for the WAIT instruction, 10011011 binary (9BH), as the first byte of the code for 8087 instruction.

4.19.4 Interfacing

Multiplexed address-data bus lines are connected directly from the 8086 to 8087. The status lines and the queue status lines are connected directly from 8086 to 8087. The Request /Grant signal RQ/GT0 of 8087 is connected to RQ/GT1 of 8086. BUSY signal 8087 is connected to TEST pin of 8086. Interrupt output INT of the 8087 to NMI input of 8086. This intimates an error condition. A WAIT instruction is passed to keep looking at its TEST pin, until it finds pin low to indicate that the 8087 has completed the computation. SYNCHRONIZATION must be established between the processor and coprocessor in two situations.

(a) The execution of an ESC instruction that requires the participation of the NUE must not be initiated if the NUE has not completed the execution of the previous instruction.

(b) When a processor instruction accesses a memory location that is an operand of a previous coprocessor instruction. In this case, CPU must synchronize with NPX to ensure that it has completed its instruction. Processor WAIT instructions is provided.

4.19.5 Exception Handling

The 8087 detects six different types of exception conditions that occur during instruction execution. These will cause an interrupt if unmasked and interrupts are enabled. The six different types of exception conditions are:

1. Invalid Operation
2. Overflow
3. Zero Divisor
4. Underflow
5. Denormalized Operand
6. Inexact Result

4.19.6 Data Types

Internally, all data operands are converted into the 80-bit temporary real format. We have 3 types. These are:

- Integer data type
- Packed BCD data type
- Real data type

Coprocessor Data Types

- Integer data type
- Packed BCD
- Real data type

Example: Converting a decimal number into a floating-point number.

1. Converting the decimal number into binary form.
2. Normalize the binary number.
3. Calculate the biased exponent.
4. Store the number in the floating-point format.

Example: Step Result

1. 100.25
2. 1100100.01 = 1.10010001 × 26
3. 110 + 01111111 = 10000101

4. Sign = 0

   Exponent = 10000101

   Significant = 10010001000000000000000

   - In step 3, the biased exponent is the exponent a 26 or 110, plus a bias of 01111111(7FH), single precision no use 7F and double precision no. use 3FFFH.
   - IN step 4, the information found in prior step is combined to form the floating point no.

4.19.7 Instruction Set

The 8087 instruction mnemonics begins with the letter F which stands for floating point and is distinguished from 8086. These are grouped into four functional groups. The 8087 detects an error condition usually called an exception when it executes an instruction and it will set the bit in its status register.

The types of instructions are:

I. Data Transfer Instructions.

II. Arithmetic Instructions.

III. Coprocessor Control Operations.

IV. Transcendental Instructions (Trigonometric and Exponential)

V. Constant Instructions

4.19.7.1 Data Transfer Instructions

4.19.7.1.1 Real transfer

FLD Load real

FST Store real

FSTP Store real and pop

FXCH Exchange registers

4.17.7.1.2 Integer Transfer

FILD Load integer

FIST Store integer

FISTP Store integer and pop

PACKED DECIMALTRANSFER(BCD)

FBLD Load BCD

FBSTP Store BCD and pop

Example:

FLD Source decrements the stack pointer by one and copies a real number from a stack element or memory location to the new ST.

```
FLD  ST(3)                              Copies  ST(3)  to  ST.
FLD  LONG_REAL[BX]        ;Number  from  memory;copied  to  ST.
```

FLD Destination: Copies ST to a specified stack position or to a specified memory location.

```
FST  ST(2)                ;Copies  ST  to  ST(2),and;increment  stack
                          pointer.
FST  SHORT_REAL[BX]      ;Copy ST to a memory at a;SHORT_REAL[BX].
```

FXCH Destination: Exchange the contents of ST with the contents of a specified stack element.

```
   FXCH  ST(5)       ;Swap  ST  and  ST(5)
```

FILD Source: Integer load. Convert integer number from memory to temporary—real format and push on 8087 stack.

```
FILD  DWORD  PTR[BX]     ;Short  integer  from  memory  at  [BX].
```

FIST Destination: Integer store. Convert number from ST to integer and copy to memory.

```
FIST  LONG_INT        ;ST  to  memory  locations  named  LONG_INT.
```

FISTP Destination: Integer store and pop. Identical to FIST except that stack pointer is incremented after copy.

FBLD Source: Convert BCD number from memory to temporary—real format and push on top of 8087 stack.

4.19.7.2 Arithmetic Instructions

Four basic arithmetic functions: Addition, Subtraction, Multiplication, and Division.

4.19.7.2.1 Addition

FADD Add real

FADDP Add real and pop

FIADD Add integer

4.19.7.2.2 Subtraction

FSUB Subtract real

FSUBP Subtract real and pop

FISUB Subtract integer

FSUBR Subtract real reversed

FSUBRP Subtract real and pop

FISUBR Subtract integer reversed

4.19.7.2.3 Multiplication

FMUL Multiply real

FMULP Multiply real and pop

FIMUL Multiply integer

4.19.7.2.4 Advanced

FABS Absolute value

FCHS Change sign

FPREM Partial remainder

FPRNDINT Round to integer

FSCALE Scale

FSQRT Square root

FXTRACT Extract exponent and mantissa.

Example:

FADD–Add real from specified source to specified destination. Source can be a stack or memory location. Destination must be a stack element. If no source or destination is specified, then ST is added to ST(1) and stack pointer is incremented so that the result of addition is at ST.

```
FADD ST(3), ST       ;Add ST to ST(3), result in ST(3)
FADD ST,ST(4)        ;Add ST(4) to ST, result in ST.
FADD;ST + ST(1       ; pop stack result at ST
FADDP ST(1)          ;Add ST(1) to ST. Increment stack;pointer
                      so ST(1) become ST.
FIADD Car_Sold       ;Integer number from memory + ST
```

FSUB: Subtract the real number at the specified source from the real number at the specified destination and put the result in the specified destination.

```
FSUB ST(2), ST        ;ST(2) = ST(2)-ST.
FSUB Rate             ;ST = ST-real no from memory.
FSUB                  ;ST = (ST(1)-ST)
```

FSUBP: Subtract ST from specified stack element and put result in specified stack element. Then increment the pointer by one.

```
FSUBP ST(1)         ;ST(1)-ST. ST(1) becomes new ST
```

FISUB: Integer from memory subtracted from ST, result in ST.

```
FISUB Cars_Sold          ;ST becomes ST-integer from memory
```

4.19.7.3 Compare Instructions

FCOM Compare real

FCOMP Compare real and pop

FCOMPP Compare real and pop twice

FICOM Compare integer

FICOMP Compare integer and pop

FTST Test ST against +0.0

FXAM Examine ST

4.19.7.4 Transcendental Instruction

FPTAN Partial tangent

FPATAN Partial arc tangent

F2XM1 2x – 1

FYL2X Y log2X

FYL2XP1 Y log2(X + 1)

Example:

FPTAN: Compute the values for a ratio of Y/X for an angle in ST. The angle must be in radians, and the angle must be in the range of $0 < \text{angle} < \pi/4$.

F2XM1: Compute Y = 2x–1 for an X value in ST. The result Y replaces X in ST. X must be in the range $0 \leq X \leq 0.5$.

FYL2X: Calculate Y(LOG2X). X must be in the range of $0 < X < \infty$ any Y must be in the range $-\infty < Y < +\infty$.

FYL2XP1 Compute the function Y(LOG2(X+1)). This instruction is almost identical to FYL2X except that it gives more accurate results when we compute log of a number very close to one.

4.19.7.5 Constant Instructions

Load Constant Instruction

FLDZ Load +0.0

FLDI Load +1.0

FLDPI Load π

FLDL2T Load log210

FLDL2E Load log2e

FLDLG2 Load log102

FLDLN2 Load loge2

ALGORITHM

To calculate x to the power of y

Load base, power.

Compute (y) × (log2 x)

Separate integer (i), fraction(f) of a real number

Divide fraction (f) by 2

Compute (2 f/2) × (2f/2)

xy = (2x) × (2y)

Program:

Program to calculate x to the power of y

```
MODEL  SMALL
.DATA
x            Dq          4.567              ;Base
y            Dq          2.759              ;Power
temp  DD
temp1  DD
temp2   DD                                  ;final  real result
tempint   DD
tempint1 DD                                 ;final integer result
two DW
diff DD
trunc_cw DW 0fffh
.STACK 100h
.CODE
start: mov ax,@DATA                         ;init data segment
mov ds,ax
load: fld y                                 ;load the power
fld x                                       ;load the base
comput: fyl2x                               ;compute (y * log2(x))
fst temp                                    ;save the temp result
trunc: fldcw  trunc_cw                      ;set truncation command
Frndint
fld  temp                                   ;load real number of
fyl2x
fist  tempint                               ;save integer after
;truncation
fld  temp                                   ;load the real number
getfrac: fisub  tempint                     ;subtract the integer
fst   diff                                  ;store the fraction
fracby2: fidiv two                          ;divide the fraction by 2
twopwrx: f2xm1                              ;calculate the 2 to the
;power fraction
fst temp1                                   ;minus 1 and save the
Result
fld1                                        ;load1
fadd                                        ;add 1 to the previous
Result
fst temp1                                   ;save the result
sqfrac: fmul st(0),st(0)                    ;square the result as fraction
fst temp1                                   ;was halved and save the result
fild tempint                                ;save the integer portion
fxch                                        ;interchange the integer and power of fraction.
scale: fscale                               ;scale the result in real  and integer
fst temp2                                   ;in st(1) and store
fist tempint1                               ;save the final result in real and integer
over: mov ax,4c00h                          ;exit to dos
int 21h
end start
```

Things to Remember

- ◊ **BIU:** Sends out addresses, fetches instructions from memory, reads data from ports and memory, and write data to ports and memory.
- ◊ **EU:** Tell the BIU where to fetch instructions or data from, decodes instructions and execute instructions.
- ◊ **Queue:** Set of six registers each of 1 byte, prefetch the instruction bytes for EU, speed up processing
- ◊ **Effective Address:** Offset from the base of segment, it is 16-bit long.
- ◊ **Physical Address:** It is 20-bit memory address, generated by combining segment register value/or segment, base address and offset value.
- ◊ **Addressing Mode:** Addressing modes are ways of specifying operands.
- ◊ **Assembler Directives:** These are defined in the form of words and these are directions to the assembler and not instructions for the 8086.
- ◊ **Opcode:** An opcode or mnemonic is a short alphabetic code consisting of two to four alphabets. Opcode assists the memory in remembering a CPU instruction. The opcode can be an instruction or directive.
- ◊ **Procedure:** A sequence of instructions written separately to carry out a specific subtask in a program is called subprogram, subroutine or procedure. If the procedure is in another program then it is a FAR procedure.
- ◊ **Recursive Procedure:** When a procedure calls itself, it is called a recursive procedure.
- ◊ **Minimum Mode:** When MN/MX pin of 8086 are connected to +5 V supply, processor is said to be operated in minimum mode.
- ◊ **Maximum Mode:** When MN/MX pin of 8086 is tied to ground, the processor is said to be operated in maximum mode.

Questions and Answers

1. What are the different features of 8086 Microprocessor?

Answer:

- It is a 16-bit μp.
- The 8086 has a 20-bit address bus and can access up to 220 memory locations (1 MB)
- It can support up to 64 K I/O ports.
- It provides 14, 16-bit registers.
- It has multiplexed address and data buses $AD_0 - AD_{15}$ and $A_{16} - A_{19}$.
- It requires single phase clock with 33% duty cycle to provide internal timing.
- 8086 is designed to operate in two modes, viz., minimum and maximum.
- It can pre-fetch up to 6 instructions bytes from memory and queue them in order to speed up instruction execution.
- It requires +5V power supply.
- A 40-pin dual in line package.

2. Write a short note on internal architecture of 8086.

Answer: 8086 has two blocks, viz. BIU and EU. The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue. EU executes instructions from the instruction system byte queue. Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called pipelining. This results in efficient use of the system bus and system performance. BIU contains Instruction queue, Segment registers, Instruction pointer, Address adder. EU contains Control circuitry, Instruction decoder, ALU, Pointer, Index register and Flag register.

Bus Interface Unit

It provides a full 16-bit bidirectional data bus and 20-bit address bus. The bus interface unit is responsible for performing all external bus operations like Instruction fetching, Instruction queuing, Operand fetch and storage, Address relocation and Bus control. The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture. This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction. These prefetching instructions are held in its FIFO queue. With its 16-bit data bus, the BIU fetches two instruction bytes in a single memory cycle. After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output. If the BIU is already in the process of fetching an instruction when the EU requests it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read/write cycle. The BIU also contains a dedicated adder which is used to generate the 20-bit physical address that is output on the address bus. This address is formed by adding an appended 16-bit segment address and a 16-bit offset address. For example, the physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register. The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Execution Unit

The execution unit is responsible for decoding and executing all instructions. The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bys cycles to memory or I/O and perform the operation specified by the instruction on the operands. During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction. If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to the top of the queue. When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions. Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

3. What are the different internal registers in 8086?

Answer: The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointers and index register, four-segment registers The 8086 has a total of fourteen 16-bit registers including a 16-bit register called the status register, with 9 bits implemented for status and control flags. Most of the registers contain data/ instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data. To specify wherein 1 MB of processor memory these 4 segments are located, the processor uses four-segment registers:

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack segment (SS) is a 16-bit register containing address of 64 KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

Data segment (DS) is a 16-bit register containing address of 64 KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low-order byte of the word and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low-order byte of the word, and CH contains the high-order byte. Count register can be used in loop, shift/rotate instructions and as a counter in string manipulation.

Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction, the DX register contains high-order word of the initial or resulting number.

4. Which registers are used as general and index registers in 8086?

Answer: **Stack Pointer** (SP) is a 16-bit register pointing to program stack.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation instructions.

5. What are the different flags in 8086?

Answer: Flags are 16-bit registers containing 9 one-bit flags.

Overflow Flag (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.

Direction Flag (DF) - if set then string manipulation instructions will auto-decrement index registers. If cleared, then the index registers will be auto-incremented.

Interrupt-enable Flag (IF) - setting this bit enables maskable interrupts.

Single-step Flag (TF) - if set then single-step interrupt will occur after the next instruction.

Sign Flag (SF) - set if the most significant bit of the result is set.

Zero Flag (ZF) - set if the result is zero.

Auxiliary carry Flag (AF) - set if there was a carry from or borrow to bits 0-3 in the AL register.

Parity Flag (PF) - set if parity (the number of "1" bits) in the low-order byte of the result is even.

Carry Flag (CF) - set if there was a carry from or borrow to the most significant bit during last result calculation.

6. Write a short note on memory segmentation in 8086.

Answer: The memory in an 8086/8088 based system is organized as segmented memory. In this scheme, the complete physically available memory may be divided into a number of logical segments. Each segment is 64 KB in size and is addressed by one of the segment registers. The 16-bit contents of the segment register actually point to the starting location of a particular segment. To address a specific memory location within a segment, we need an offset address. The offset address is also 16-bit long so that the maximum offset value can be FFFFH, and the maximum size of any segment is thus 64 K locations. To emphasize this segmented memory concept, we will consider an example of a housing colony containing say, 100 houses. The simplest method of numbering the houses will be just to assign the numbers from 1 to 100 to each house sequentially. Suppose, now, if one wants to find out house number 67, and then he will start from house number 1 and go on till he finds the house, numbered 67. Consider another case where the 100 houses are arranged in the 10 × 10 (rows × columns) pattern. In this case, to find out house number 67, one will directly go to the 6th row and then to the 7th column. In the second scheme, the efforts required for finding the same house will be too less. This second scheme in our example is analogous to the segmented memory scheme, where the addresses are specified in terms of segment addresses analogous to rows and offset addresses analogous to columns.

7. Briefly explain the minimum mode of 8086.

Answer: The logic level at this mode decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode.

The following pin functions are for the minimum mode operation of 8086.

M/I/O-Memory/IO: This is a status line logically equivalent to S_2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous T_4 and remains active till final T_4 of the current cycle. It is tri-stated during local bus "hold acknowledge".

INTA-interrupt Acknowledge: This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during T_2, T_3 and T_w of each interrupt acknowledge cycle.

ALE-Address Latch Enable: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tri-stated.

DT/$\overline{R}$-DataTransmit/Receive: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to S_1 in maximum mode. Its timing is the same as M/IO. This is tristated during 'hold acknowledge'.

DEN-Data Enable: This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T_2 until the middle of T_4. DEN is tri-stated during 'hold acknowledge' cycle.

HOLD, HLDA-Hold/Hold Acknowledge: When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus (instruction) cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and it should be externally synchronized.

If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T_4 provided:

1. The request occurs on or before T_2 state of the current cycle.
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address).
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode.

The following pin functions are applicable for maximum mode operation of 8086.

S_2, S_1, S_0-Status Lines: These are the status lines which reflect the type of operation being carried out by the processor. These become active during T_4 of the previous cycle and remain active during T_1 and T_2 of the current bus cycle. The status lines return to passive state during T_3 of the current bus cycle so that they may again become active for the next bus cycle during T_4. Any change in these lines during T_3 indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in the table given below.

S_2	S_1	S_0	*Indication*
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

LOCK: This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during "hold acknowledge". When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

QS_1, QS_0-Queue Status: These lines give information about the status of the code prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in given table.

QS1	*QS0*	*Indication*
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus, even the largest (6-bytes) instruction can be prefetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (Opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as instruction pipelining.

At the starting the CS: IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS: IP address is odd or two bytes at a time, if the CS: IP address is even. The first byte is a complete Opcode in case of some instructions (one byte Opcode instruction) and it is a part of Opcode, in case of other instructions (two-byte long Opcode instructions), the remaining part of Opcode may lie in the second byte. But invariably the first byte of an instruction is an Opcode. These Opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied. The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single Opcode byte or double Opcode byte. If it is single Opcode byte, the next bytes are treated as data bytes depending upon the decoded instruction length, otherwise the next byte in the queue is treated as the second byte of the instruction Opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.

RQ/CT$_0$, RQ/G$_1$-Request/Grant: These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with RQ/GT$_0$ having higher priority than RQ/GT$_1$. RQ/GT pins have internal pull-up resistors and may be left unconnected. The request grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.
2. During current or next clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at next clock cycle. The CPU's bus interface unit is likely to be disconnected from the local bus of the system.
3. A one clock wide pulse from another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle.

Thus, each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus requests those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as discussed in case of HOLD and HLDA in minimum mode.

8. Describe different addressing modes in 8086?

Answer: Addressing mode indicates a way of locating data or operands. The required operand may be placed in the accumulator, in a general-purpose register or in a memory location. The way by which an operand is specified for an instruction is called addressing mode. Thus, the addressing modes describe the types of operands and the way they are accessed for executing an instruction.

Immediate Addressing

In this type of addressing, the operand is specified in the instruction itself, e.g. : **MOV**

AX, 21325H

Direct Data Addressing

Most instructions of the 8086/8088 can be used as direct data addressing because it specifies the location of the data which is to be operated on by the instruction. There are two types of direct data addressing mode.

1. Direct Addressing Mode
2. Displacement Addressing Mode

1. **Direct Addressing Mode:** An instruction executing in the direct addressing mode operates between the memory location in the data segment and the AX (16-bit) or AL (8-bit). Consider the following move instruction which uses this type of addressing mode. These instructions can be two or three bytes long, e.g.: **MOVAL, [B894 H]**
2. **Displacement Addressing Mode:** Displacement addressing mode can use any of the 8086 register (excluding segment register) unlike the direct addressing mode where only the AX or AL register can be used. In this type of addressing the instruction formed is 4-byte long, e.g.: **MOV CL, [4568]**

Register Addressing Mode: In register addressing mode, a register is a source of an operand. The destination location is specified in the instruction before comma and source is specified after comma. Hence, in register addressing mode both source and destination are two different registers, e.g.: **MOV BX, AX.**

Register Indirect Addressing Mode: Sometimes, the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES, e.g.: **MOV AX, [BX]**

Register Relative Addressing Mode: In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and Dl in the default (either DS or ES) segment. The example given below explains this mode, e.g.: **MOVAX, [BX+1001H]**

Offset = [SI or DI or BX or BP + 8-bit or 16-bit displacement]

Based Indexed Addressing Mode: The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS, e.g. : **MOVAX, [BX + SI]**

Relative Based Indexed Addressing Mode: The effective address is formed by adding an 8-or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers (SI or DI) in a default segment, e.g.: **MOVAX, [BX + SI + d 8 or d 16]**

9. What do you mean by INC, DIV, IMUL, DIV in 8086?

Answer: **IMUL: Signed Multiplication:** This instruction multiplies two signed numbers. The result is a signed number. For 8-bit multiplication, the result is placed in AX register. For 16-bit multiplication, the low order 16 bits of result are placed in AX register and the high order 16 bits of the result are placed in DX register. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16- and 32-bit result respectively, CF and OF both will be set. Sign bit and CF fills the unused higher bits of the result, AF are cleared.

DIV: Unsigned Division: This instruction is used to divide a 16-bit unsigned number by an 8-bit unsigned number or a 32-bit unsigned number by a 16-bit unsigned number. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. This instruction does not affect any flag.

IDIV: Signed Division: This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. In case of 16-bit operation, if the quotient is greater than 7FFF, the divide-by-zero (Type 0) interrupt is generated. All the flags are undefined after IDIV instruction.

INC: This instructions increment the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to either a memory or register contents. Immediate data cannot be operand of this instruction.

10. Explain CALL, LOOP and RET instructions in 8086?

Answer: CALL: Unconditional Call: This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. There are two basic types of calls— near and far. A near call is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near call instruction it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL onto the stack. This offset saved on the stack is referred to as return address, because this is the address that execution will return to after the procedure executes. A RET instruction at the end of the procedure will return execution to the instruction after the call by copying the offset saved on the stack back to IP. A far call is a call to a procedure which is in a different segment from the one that contains the CALL instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of CS register to the stack. A RET instruction at the end of the procedure will return execution to the next instruction after the call by restoring the saved values of CS and IP from the stack.

RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure. In case of a FAR procedure, the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP.

Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

LOOP: Loop Unconditionally: This instruction is used to repeat a series of instructions a number of times specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMF IF NOT ZERO structure.

11. What are the various types of interrupts in 8086? Briefly explain Single-Step Interrupt, Non-Maskable Interrupt, Breakpoint Interrupt, Overflow Interrupt and Software Interrupts.

Answer: Single-Step Interrupt—Type 1

When you tell a system to single-step, it will execute one instruction and stop. You can then examine the contents of registers and memory locations. If they are correct, you can tell the system to go on and execute the next instruction. In other words, when in single-step mode, a system will stop after it executes each instruction and wait for further direction from you. The 8086 trap flag and type 1 interrupt response make it quite easy to implement a single-step feature in an 8086-based system.

If the 8086 trap flag is set, the 8086 will automatically do a type 1 Interrupt after each instruction executes. When the 8086 does a type 1 Interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the type interrupt-service procedure from address 00006H and it gets the IP value for the start of the procedure from address 00004H.

The tasks involved in implementing single stepping are: Set the trap flag, write an interrupt-service procedure which saves all registers on the stack, where they can later be examined or perhaps displayed on the CRT, and load the starting address of the type 1 interrupt-service procedure into addresses 00004H and 00006H. The actual single-step procedure will depend very much on the system on which it is to be implemented. We do not have space here to show you the different ways to do this. We will, however, show you how the trap flag is set or reset because this is somewhat unusual. The 8086 has no instructions to directly set or reset the trap flag. These operations are done by pushing the flag register on the stack, changing the trap flag bit to what you want it to be and then popping the flag register back off the stack. Here is the instruction sequence to set the trap flag.

```
PUSHF ; Push flags on stack
MOV BP, SP        ; Copy SP to BP for use as index
OR WORD PTR (BP+0), 0100H; Set TF bit
POPF                          ; Restore flag register
```

To reset the trap flag, simply replace the OR instruction in the preceding sequence with the instruction AND WORD PTR [BP+0], 0FEFFH. The trap flag is reset when the 8086 does a type 1 interrupt. so the single-step mode will be disabled during the interrupt-service procedure.

Non-Maskable Interrupt—Type 2

The 8086 will automatically do a type 2 interrupt response when it receives a low-to-high transition on its NMI input pin. When it does a type 2 interrupt, the 8086 will push the flags on the stack. Reset TF and IF, and push the CS value and the IP value for the next instruction on the stack. It will then get the CS value for the start of the type 2 interrupt-service procedure from address 0000AH and the IP value for the start of the procedure from address 00008H.

The name non-maskable given to this input pin on the 8086 means that the type 2 interrupt response cannot be disabled (masked) by any program instructions. Because this input cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of. We could, for example, have a pressure-sensor on a large steam boiler connected to the NMI input. If the pressure goes above some preset limit, the sensor will send an interrupt signal to the 8086.

The type 2 interrupt-service procedure for this case might turn off the fuel to the boiler, open a pressure-relief valve and sound an alarm. Another common use of the type 2 interrupt is to save program data in case of a system power failure. Some external circuitry detects when the ac power to the system fails and sends an interrupt-signal to the NMI input. Because of the large filter capacitors in most power supplies, the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for a type 2 interrupt-service procedure to copy program data to some RAM which has a battery backup power supply. When the ac power returns program data can be restored from the battery-backed RAM and the program can resume execution where it left off.

Breakpoint Interrupt—Type 3

The type 3 interrupt is produced by execution of the INT 3 instruction. The main use of the type 3 interrupt is to implement a breakpoint function in a system. We hope that you have been using them in debugging your programs. When you insert a breakpoint, the system executes the instructions up to the breakpoint and then goes to the breakpoint procedure. Unlike the single-step feature, which stops execution after each instruction, The breakpoint feature executes all the instructions up to the inserted breakpoint and then stops execution.When you tell most 8086 systems to insert a breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CCH. The 8086 code for the INT 3 instruction. When the 8086 executes this INT 3 instruction, it pushes the flag register on the stack, resets TF and IF and pushes the CS and IP values for the next mainline instruction on the stack. The 8086 then gets the CS value of the start of the type 3 interrupt: service procedure from address 0000EH and the IP value for the procedure from address 000CH. A breakpoint interrupt-service procedure usually saves all the register contents on the stack. Depending on the system, it may then send the register contents to the CRT display and wait for the next command from the user or in a simple system. It may just return control to the user. In this case an examine register command can be used to check if the register contents are correct at that point in the program.

Overflow Interrupt—Type 4

If the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location, the 8086 overflow flag (OF) will be set. For example, if you add the 8-bit signed number 01101100 (108 decimal) and the 8-bit signed number 01010001 (81 decimal), the result will be 10111101 (189 decimal). This would be the correct result if we were adding unsigned binary numbers, but it is not the correct signed result. For signed operations, the 1 in the most significant bit of the result indicates that the result is negative and in 2's complement form, the result, 10111101, then actually represents -67 decimal, which is obviously not the correct result for adding +108 and +89. An overflow error in program can be detected and responded by two major ways. One way is to put the jump if overflow instruction, JO, immediately after the arithmetic instruction. If the overflow flag is set as a result of the arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds to the overflow in the way you want. An overflow instruction, INTO can be used immediately after the arithmetic instruction in the program and this is the second way of detecting and responding to an overflow error. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will, simply function as an NOP. However, if the overflow flag is set, indicating an overflow error, the 8086 will do a type 4 interrupt after it executes the INTO instruction.

When the 8086 does a type 4 interrupt, it pushes the flag register on the stack, resets TF and IF and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 0012H and the IP value for the procedure from address 00010H instructions in the interrupt-service procedure then performs the desired response to the error condition. The procedure might, for example, set a "flag" in a memory location as we did on the BAD-DIV procedure. The advantage of using the INTO and type 4 interrupt approach is that the error routine is easily accessible from any program.

Software Interrupts—Types 0 Through 255

The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. Any interrupt which is desired is specified as part of the instruction. For example, the instruction INT 32, will cause the 8086 to do a type 32 interrupt response. The 8086 will push the flag register on the stack, reset TF and IF and push the CS and IP values of the next instruction on the stack. It will then get the CS and IP values for the start of the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher. For a type 32 interrupt, then, the IP value will be put at 4×32 or 128 decimal (80H) and the CS value will be put at address 82H in the interrupt-vector table. Software interrupts produced by the INT instruction have many uses. In the previous section, we discussed the use of the INT 3 instruction to insert breakpoints in programs for debugging, another use of software interrupts is to test various interrupt-service procedures. For example, an INT 0 instruction is used to send execution to a divide-by-zero interrupt-service procedure without having to run the actual division program. As another example, an INT 2 instruction is used to send execution to an NMI interrupt-service procedure. This allows you to test the NMI procedure without needing to apply an external signal to the NMI input of the 8086.

Exercise

1. (a) Draw the internal diagram of 8086.
   (b) Briefly explain functioning of BIU and EU.

2. Describe instruction stream byte queue and explain how does it speed up processing of 8086.
3. Describe memory segmentation and explain the role of segment registers. How is the physical address generated in 8086?
4. Describe the bus cycle of 8086. What is the utility of wait states into a bus cycle?
5. What is the advantage of using general-purpose data register for temporary storage over using a memory location? What are the advantages of memory segmentation?
6. What do you mean by addressing mode? Explain indexed addressing modes.
7. What do you understand by assembler directive? Explain (a) SEGMENT (b) ENDS (c) ENDP (d) PROC (e) EQU (f) ASSUME (g) DW (h) DH.
8. How the 8086 instructions are encoded? Construct the machine code for (a) MOV AL, [BX] (b) MOV SP, DX (c) MOV SS, AX instructions.
9. Write an 8086 program to subtract two numbers and store the result in a location named "SUBTRACTION".
10. Briefly explain instructions, MOV, PUSHF, LEA, INC, CAMP, XOR, ROR, SCAS, JIM, CALL, CLD and LOCK.
11. How CALL and RET instructions are used in the execution of a procedure in a main line program?
12. How assembler macro differ from procedure? What are the advantages of macro over procedure? What is recursive procedure?
13. What do you understand by STACK and how it can be defined in a program?
14. What are the signals used in the 8086 microprocessor? Explain demultiplexing of address/ data bus in 8086 microprocessor. How does it differ from 8088?
15. Explain interrupt acknowledge cycle of 8086 microprocessor. What are the sources of interrupts?
16. What is the major difference between minimum mode and maximum mode architecture of 8086?
17. What are the important signals of Intel 8086?
18. How many operating modes does 8086 have?
19. How many functional units does 8086 contain?
20. What is the function of a segment register in 8086?
21. What are conditional and control flags in 8086?
22. How many interrupt lines does 8086 have?
23. What physical address is represented by:

    (i) 4370 : 561E H (ii) 7A32 : 0028 H
24. Describe the difference between the instructions:

    (i) MOV AL, 0DB H (ii) MOV AL, DB H
25. Briefly explain the maximum mode configuration of 8086.
26. What is the difference between minimum and maximum modes of 8086?

27. How many interrupts are available in 8086? List the predefined software interrupts available in 8086.
28. Briefly explain the maximum mode configuration of 8086.
29. What is the purpose of MN/MX pin? Explain.
30. Explain the concept of segmented memory? What are its advantages?
31. Explain the concept of pipelining in 8086. Discuss its advantages and disadvantages.
32. Discuss the interrupt system of Intel 8086. What is interrupt pointer? What is 'type' of an interrupt?
33. Discuss the various addressing modes of 8086. What are displacement, base and index? What is an effective address or offset?
34. What is the difference between minimum and maximum modes of 8086? How are these modes selected?
35. Draw and explain the architecture of 8086.
36. Write an 8086 program to add two 16-bit numbers in CX and DX and store the result in location 0500H addressed by DI.

Chapter 5

Interfacing Devices

- Introduction
- Data Transfer Schemes
- Interfacing Devices and I/O Devices
- Programmable Peripheral Interface (PPI)
- Programmable Keyboard/ Display Interface (Intel 8279)
- Centronix Parallel Communication
- Serial Communication
- RS-232C
- Universal Asynchronous Receiver/Transmitter
- 8251 Programmable/Communication Interface
- Special-Purpose Interfacing Devices
- Programmable Interval Timer
- 8259A—Priority Interrupt Controller
- Direct Memory Access Controller (Intel 8237A)
- Programmable DMA Controller–Intel 8257
- Interfacing Analog to Digital Data Converters
- Interfacing Digital to Analog Converters

5.1 INTRODUCTION

A microprocessor when combined with memory and input/output devices forms a microcomputer. The microprocessor is the heart of a microcomputer. Memories and input/output devices are interfaced to microprocessor to form a microcomputer. The memories and input/output devices are interfaced to CPU by the manufacturer in case of large and minicomputers. In a microprocessor-based system, the designer has to select suitable memories and input/output devices for his task and interface them to the microprocessor. The selected memories and input/output devices should be compatible with the microprocessor. An additional electronic circuit has to be designed through which the device may be interfaced to the CPU if a particular device is not compatible. The incompatibility between two devices may arise due to one or more of the following reasons:

1. Timing according to which data is transferred
2. Data format
3. Electrical characteristics

Timing differences can arise when a slow memory is to be interfaced with a fast processor. It can be avoided by using a processor which will be capable of waiting for the interfaced device for as long as desired. The second option is an external buffer between the processor and the memory.

Different data formats can also introduce incompatibility, e.g., a modem may send data on the telephone lines serially, but the processor receives data in parallel. So, there is a requirement of a serial-to-parallel conversion device at the processor end.

Different electrical characteristics can also be a source of incompatibility. Electrical characteristics differ due to different voltage levels and current requirements. With the help of level translators and current drivers, these incompatibilities can be reduced.

The I/O devices such as keyboards and displays establish communication of computer with outside world. Such devices can be interfaced in two ways:

1. I/O mapped I/O
2. Memory-mapped I/O

In I/O mapped I/O, device is identified with a unique device number and data is transferred through IN/OUT instruction. In memory-mapped I/O, each device is identified with 16-bit address. The I/O devices are considered to be a part of memory and memory related instruction is used for data transfer.

An I/O interface must be able to perform the following tasks:

1. Determine whether or not it is being interfaced.
2. Determine whether it has to send data to CPU or receive data from CPU.
3. Send ready signal informing CPU that transfer is over.
4. Send interrupt requests to CPU and receive interrupt acknowledgement and send an Interrupt type.

An interface can be divided into two parts. First, a part that interfaces to the I/O device and second that interfaces to the system bus. There must be drivers and receivers to maintain signal quality, logic for translating the interface control signals to proper handshaking signals, logic for decoding address that appears on the bus.

Handshaking signals are used to determine the direction in which transfer has to take place whether from CPU or to CPU. It also determines whether it is a READ or WRITE operation. Also, the interrupt signals are handled here.

Address decoder determines whether it is I/O mapped I/O or memory-mapped I/O from one of the bits. If the decoder finds that an interface is referred, it sends signal to the appropriate device. Interfaces can be categorized according to the way I/O devices transfer data either in serial or parallel form.

5.2 DATA TRANSFER SCHEMES

A variety of I/O devices having wide range of speed and other different characteristics are available which use different manufacturing technologies such as electronic, electrical, mechanical, optical, etc. Due to these reasons, designers face difficulties in interfacing I/O devices with microprocessor. Special interfacing circuitries have to be designed for the purpose. In a microprocessor-based system or in a computer, the data transfer takes place between two devices such as microprocessor and memory, microprocessor and I/O devices, and memory and I/O device. Usually, semiconductor memories are compatible with microprocessor as the same technology is employed in the manufacturing of both semiconductor memories and microprocessors. Hence, the problem associated with the interfacing of memory is less. A microprocessor-based system or a computer may have several I/O devices of different speed. A slow I/O device cannot transfer data when microprocessor issues an instruction for the same because it takes some time to get ready. To solve the problem, a number of data transfer techniques have been developed. The data transfer schemes are classified into the following two broad categories.

1. Programmed Data Transfer Schemes
2. DMA Data Transfer Schemes

5.2.1 Programmed Data Transfer Schemes

Programmed data transfer schemes are those which are controlled by the CPU. The transfer of data from an I/O device to the CPU (or to the memory through the CPU) or vice versa takes place under the control of programs which reside in the memory. These programs are executed by the CPU when an I/O device is ready to transfer data. The microprocessor executes the program to transfer data. The programmed data transfer schemes are employed to transfer a small amount of data, e.g., A/D, D/A converters and peripherals mass storage devices like a hard disk or a high-speed line printer. The programmed data transfer schemes are classified into the following three categories:

(i) Synchronous data transfer scheme
(ii) Asynchronous (or handshaking) data transfer scheme
(iii) Interrupt driven data transfer scheme

All the schemes require both software and hardware for their implementation. Within a microcontroller, more than one scheme can be used for interfacing different I/O devices.

(i) **Synchronous Data Transfer Scheme:** Synchronous means "at the same time". There is synchronization between the device which sends data and the device which receives the data with the same clock input. This technique of the data transfer is employed when the CPU and I/O devices match in speed, the data transfer with I/O devices is performed executing IN or OUT instructions for I/O mapped I/O devices or using memory read/write instructions for memory mapped I/O devices. The IN instruction is used to read data from an input device or input port and the OUT instruction to send data from the CPU to an output device or output port due to the speed match between the CPU and the I/O device, the I/O device is ready to transfer data when IN or OUT instruction is issued by the CPU. The status of the I/O device, i.e., whether it is ready or not, is not examined before data is transferred.

(ii) **Asynchronous Data Transfer Scheme:** Asynchronous means "at irregular intervals". In this method, data transfer is not based on predetermined timing pattern. The technique of data transfer is used when the speed of an I/O device does not match with the speed of the microprocessor, and the timing characteristics of I/O device is not predictable. In this technique, before the data is transferred the status of the I/O device, i.e., whether the device is ready or not, is checked by the microprocessor. The microprocessor initiates the I/O device to get ready and then continuously checks the status of the I/O device till the I/O device becomes ready to transfer data. The microprocessor sends instructions to transfer data when I/O device becomes ready. This mode of data transfer is also called handshaking mode of data transfer because some signals are exchanged between the I/O device and the microprocessor before the actual transfer of data takes place. An initiating signal is issued by the microprocessor to the I/O device to get ready (or to start) and when I/O device is ready, it sends signals to the processor to indicate that it is ready. Such signals are called handshake signals.

(iii) **Interrupt Driven Data Transfer Scheme:** In this the microprocessor initiates an I/O device to get ready, and then executes its main program instead of remaining in a program loop to check the status of the I/O device. When the I/O device is ready to transfer data, it sends a high signal to the microprocessor through a special input line called an interrupt line. In other words, it interrupts the normal processing sequence of the microprocessor. On receiving an interrupt, the microprocessor completes the current instruction at hand, and then attends the I/O device. The contents of the program counter are placed on stack first, and then the microprocessor takes up a subroutine called ISS (Interrupt Service Subroutine). It executes ISS to transfer data from or to the I/O device. Different ISSs are to be provided for different I/O devices. After completing the data transfer, the microprocessor returns back to the main program which it was executing before the interrupt occurred. Interrupt driven data transfer is used for slow I/O devices as it is an efficient technique as compared to asynchronous data transfer scheme because precious time of the microprocessor is not wasted in waiting while an I/O device is getting ready.

5.2.2 DMA Data Transfer Scheme

In this scheme CPU does not participate and data is directly transferred from an I/O device to the memory or vice versa. The data transfer is controlled by the I/O device or a DMA controller. This scheme is employed for transferring a large amount of data. If bulk data is transferred through the CPU, it takes appreciable time and the process becomes slow. An I/O device which wants to send data using DMA technique sends the HOLD signal to the CPU. On receiving a HOLD signal from an I/O device the CPU gives up the control of buses as soon as the current machine cycle is completed. The CPU sends a hold acknowledge signal, i.e., HLDA to the I/O device to indicate that it has received the HOLD request and it has released the buses. The I/O device takes over the control of buses and directly transfers data to the memory or reads data from the memory. When a HOLD request is received, the 8085 may be in one of the following states:

- In a T_i state of any machine cycle $(1 \leq i \leq 6)$,
- In a T_{HALT} state (i.e., it had executed a HLT instruction prior to the HOLD request).

DMA data transfer scheme is a faster scheme as compared to programmed data transfer scheme. It is used to transfer data from mass storage devices such as hard disks, floppy disks, etc. It is also used for high-speed printers. When data transfer is over, the CPU regains the control over the buses. DMA data transfer schemes are of the following two types:

(i) Burst mode of DMA data transfer
(ii) Cycle stealing techniques of DMA data transfer

(i) **Burst Mode of DMA Data Transfer:** A scheme of DMA data transfer is one in which the I/O device withdraws the DMA request only after all the data bytes have been transferred. This is called burst mode of data transfer. By this technique a block of data is transferred. This technique is employed by magnetic disks drives. In case of magnetic disks, data transfer cannot be stopped or slowed down without loss of data. Hence, block transfer is a must.

(ii) **Cycle Stealing Technique of DMA Data Transfer:** By this technique a long block of data is transferred by a sequence of DMA cycles. In this method, the I/O device withdraws DMA request after transferring one byte or several bytes. This method reduces interference in CPU's activities. The interference can be eliminated completely by designing an interfacing circuitry which can steal bus cycle for DMA data transfer only when the CPU is not using the system bus. In DMA data transfer schemes, I/O devices control data transfer and hence the VO devices must have registers to store memory addresses and byte count. It must also have electronic circuitry to generate necessary control signals required for DMA operations. Usually, I/O devices do not have these facilities. To solve this problem, DMA controllers have been designed and developed. Examples of DMA controller chips are: Intel 8237A, 8257, etc.

5.3 INTERFACING DEVICES AND I/O DEVICES

To communicate with the outside world, microcomputers use peripherals (I/O devices) which are: A/D converter, D/A converter, CRT, printers, hard disks, floppy disks, magnetic tapes, etc.

Peripherals are connected to the microcomputer through electronic circuits known as interfacing circuits. Generally, each I/O device requires a separate interfacing circuit. The interfacing circuit converts the data available from an input device into compatible format for the computer. The interface associated with the output device converts the output of the microcomputers into the desired peripheral format. To simplify the work of the designer microprocessor, manufacturers have developed a number of general-purpose and special-purpose single chip-interfacing devices. Some of the general-purpose devices are:

(i) I/O Port
(ii) Programmable Peripheral Interface (PPI) or Peripheral Interface Adapter (PIA)-8255
(iii) DMA Controller-8257
(iv) Programmable Interrupt Controller (PIC)-8259
(v) Communication Interface-8251
(vi) Programmable Counter/Interval Timer-8353

Special-purpose interfacing devices are designed to interface a particular type of I/O device to the microprocessor. Examples of such devices are:

(i) CRT Controller
(ii) Floppy Disk Controller
(iii) Keyboard and Display Interface (8279)
(iv) Hard Disk Controller

A large number of interfacing devices have been developed by various manufacturers. The detailed description of these devices is beyond the scope of the book. Some important interfacing devices developed by INTEL Corporation are described in the subsequent subsections.

5.3.1 Programming I/O Ports

An input device is connected to the microprocessor through an input port. An input port is a place for unloading data which is done through an input device which unloads data into the port. The microprocessor reads data from the input port. Thus, data is transferred from the input device to the accumulator via input port. Similarly, an output device is connected to the microprocessor through an output port. The microprocessor unloads data into an output port. As the output port is connected to the output device, data is transferred to the output device. Figure 5.1 shows a schematic connection of the CPU, I/O port and I/O devices. An I/O port may be programmable or non-programmable. A non-programmable port behaves as an input port if it has been designed and connected in input mode. Similarly, a port connected in output mode acts as an output port. But a programmable I/O port can be programmed to act either as an input port or output port; the electrical connections remain the same.

5.3.2 Address Space Partitioning

The Intel 8085 uses a 16-bit wide address bus for addressing memories and I/O devices. Using 16-bit wide address bus it can access 2^{16} = 64 KB of memory and I/O devices. The 64 K addresses are to be assigned to memories and I/O devices for their addressing. There are two schemes for the allocation of addresses to memories and input/output devices:

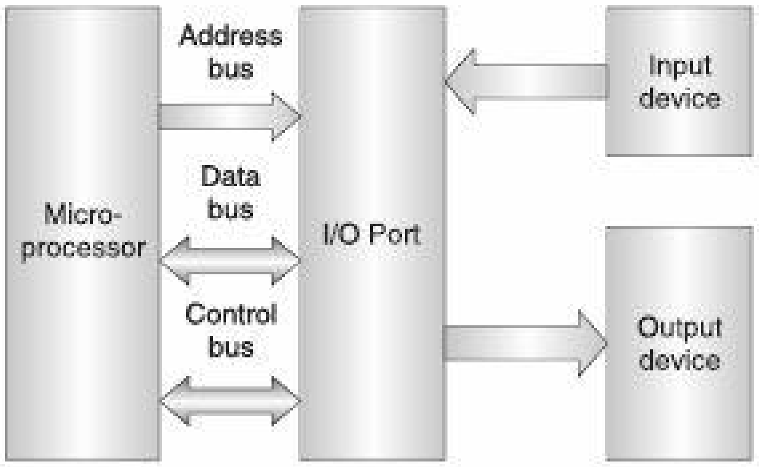


Fig. 5.1 Interfacing of I/O Device through I/O Port.

(i) Memory-mapped I/O scheme

(ii) I/O mapped I/O scheme

(i) **Memory-Mapped I/O Scheme:** In memory-mapped I/O scheme, there is only one address space which is defined as the set of all possible addresses that a microprocessor can generate. Some addresses are assigned to memories and some addresses to I/O devices. An I/O device is also treated as a memory location and one address is assigned to it. Suppose that memory locations are assigned the addresses 2000 to 24FF. One address is assigned to each memory location. Any one of these addresses cannot be assigned to an I/O device. The addresses for I/O devices are different from the addresses which have been assigned to memories. The addresses which have not been assigned to memories can be assigned to I/O devices. For example, 2500, 2501, 2502, etc. may be assigned to I/O devices. One address is assigned to each I/O device.

In this scheme, all the data transfer instructions of the microprocessor can be used for both memory as well as I/O devices. For example, MOV A, M will be valid for data transfer from the memory location or I/O device whose address is in H-L pair. If the H-L pair contains the address of a memory location, data will be transferred from the memory location to the accumulator. If the H-L pair contains the address of an I/O device, data will be moved from the I/O device to the accumulator. The memory mapped I/O scheme is suitable for a small system.

(ii) **I/O Mapped I/O Scheme:** In this scheme, the addresses assigned to memory locations can also be assigned to I/O devices. Since the same address may be assigned to a memory location or an I/O device, the microprocessor must issue a signal to distinguish whether the address on the address bus is for a memory location or an I/O device. The Intel 8085 issues an $IO/\overline{M}$ signal for this purpose. When this signal is high, the address on the address bus is for an I/O device. When this signal is low, the address on the address bus is for a memory location. Two extra instructions IN and OUT are used to address I/O devices. The IN instruction is used to read data data of an input device. The OUT instruction is used to send data to an output device. This scheme is suitable for a large system.

5.4 PROGRAMMABLE PERIPHERAL INTERFACE (PPI)

A programmable peripheral interface is a multiport device. The ports may be programmed in a variety of ways as required by the programmer. The device is very useful for interfacing peripheral devices. The term PIA, Peripheral Interface Adapter is also used by some manufacturers.

INTEL 8255: The Intel 8255 is a programmable peripheral interface (PPI). It has two versions, namely, the Intel 8255A and the Intel 8255A-5. General descriptions for both are the same. There are some differences in their electrical characteristics. Hereafter, they will be referred to as 8255. Its main functions are to interface peripheral devices to the microcomputer. It has three 8-bit ports, namely, Port A, Port B and Port C. The Port C has been further divided into two 4-bit ports, Port C upper and Port C lower. Thus, a total of 4 ports are available, two 8-bit ports and two 4-bit ports. Each port can be programmed either as an input port or an output port.

5.4.1 Architecture of Intel 8255A

Figure 5.2 shows the schematic diagram of Intel 8255 programmable peripheral interface. It is a 40-pin IC package. It provides a good example of a parallel interface. As shown in Figure 5.2, the interface contains a control register and three separately addressable ports, denoted by A, B, and C. Whether or not an 8255 is being accessed, it is determined by the signal on the $\overline{CS}$ pin and the direction of the access is according to the RD and WR signals. Which of the four registers is being addressed is determined by the signals applied to the pins A_1 and A_0. Therefore, the lowest port address assigned to an 8255A must be divisible by 4. Data is transferred between microprocessor/CPU and 8255 or the $D_0 - D_7$ bus. The $\overline{RD}$ and $\overline{WR}$ signals are used for indicating the 8255 whether a read or write operation is being preformed. Before the microprocessor/CPU can read or write any data, the 8255 chip must be selected using $\overline{CS}$ input. When this pin is high, the $D_0 - D_7$ lines are tri-stated.

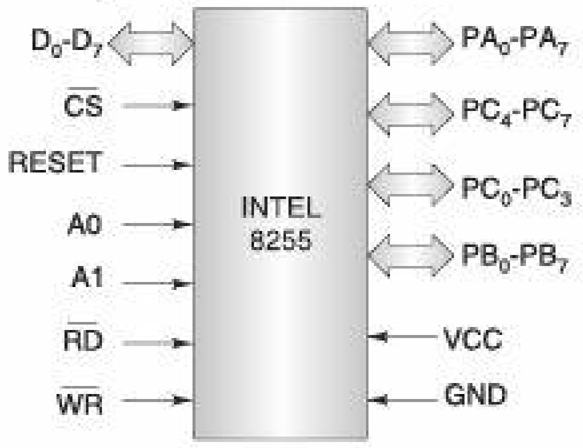


Fig. 5.2 Schematic Diagram of Intel 8255.

5.4.2 Control Signals

The function performed by control signals are explained as follows:

1. **$\overline{CS}$ (Chip Select):** It is a chip select signal. The LOW status of this signal enables communication between the CPU and 8255.
2. **$\overline{RD}$ (Read):** When $\overline{RD}$ goes LOW the 8255 ends out data or status information to the CPU on the data bus. In other words, it allows the CPU to read data from the input port of 8255.

3. **$\overline{\text{WR}}$ (Write):** When $\overline{\text{WR}}$ goes LOW, the CPU writes data or control word into 8255. The CPU writes data into the output port of 8255 and the control word into the control word register.
4. **A_0 and A_1:** The selection of input port and control word register is done using A_0 and A_1 in conjunction with $\overline{\text{RD}}$ and $\overline{\text{WR}}$. A_0 and A_1 are normally connected to the least significant bits of the address bus.

A summary of the 8255 A's addressing is:

$\overline{CS}$	A_1	A_0	*Selected*
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
0	x	x	8255 is not selected

If we write the instruction IN 00, it means that it is for the Port A of 8255. When this instruction is executed, data is transferred from the Port A to the accumulator. The instruction OUT 03 will transfer the content of the accumulator to the control word register of 8255. The instruction IN 09 transfers the data from Port B of 8255 to the accumulator. OUT OA transfers the content of the accumulator to the Port C of 8255. The instruction OUT OB transfers the content of the accumulator to the control word register of 8255. The IN instruction is used for the port which has been defined as input port. After IN, the address of the port is specified. Similarly, OUT instruction is used for the ports which have been defined as output ports. After OUT instruction, the address of the port is specified. For control word register only OUT instruction is used. Its contents cannot be read.

5.4.3 Control Word

According to the requirement, a port can be programmed to act either as an input port or an output port. For programming the ports of 8255, a control word is formed as shown in Figure 5.3. Control word is written into the control word register which is within 8255. No read operation of the control word register is allowed. The control word bit corresponding to particular ports set to either 1 or 0 depending upon the definition of the port, whether it is to be made an input port or output port. If a particular port is to be made an input port, the bit corresponding to that port is set to 1. For making a port an output port, the corresponding bit for the port is set to 0. The detailed description of the bits of the control word is as follows:

Bit 0: It is for Port C_{lower}. To make Port C_{lower} an input port, the bit is set to 1. To make Port C_{lower} an output port, the bit is set to 0.

Bit 1: It is for Port B. To make Port B an input port, the bit is set to 1. To make Port B an output port, the bit is set to 0.

Bit 2: It is for the selection of the mode for the Port B. If the Port B has to operate in Mode 0, the bit is set to 0. For Mode 1 operation of the Port B, it is set to 1.

Bit 3: It is for the Port C_{upper}. To make Port C_{upper} an input port, the bit is set to 1. To make Port C_{upper} an output port, the bit is set to 0.

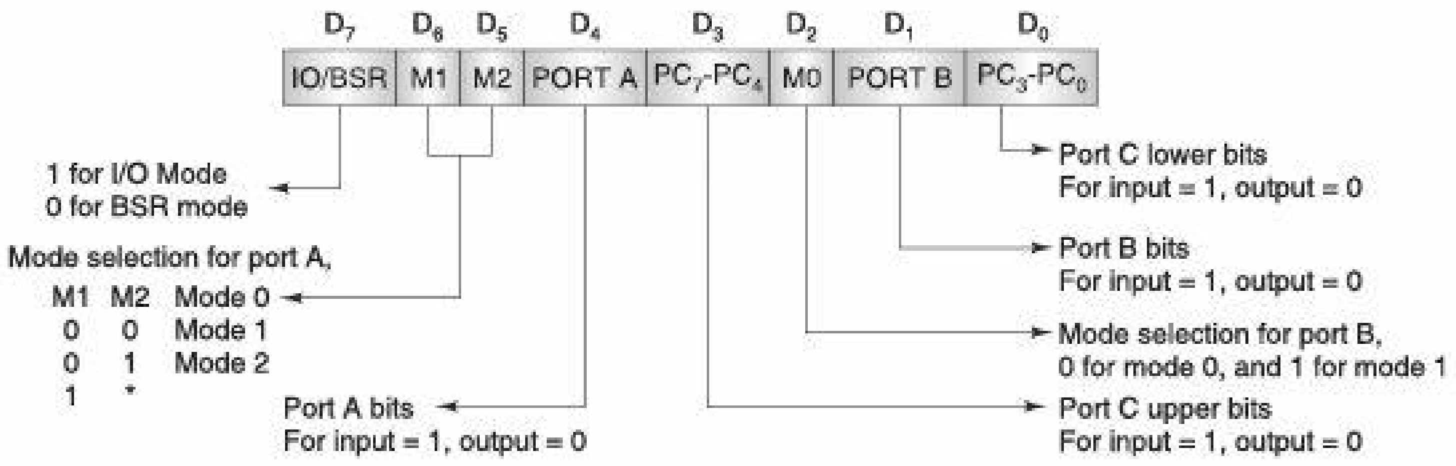

D7 D6 D5 D4 D3 D2 D1 D0
IO/BSR
M1
M2
PORT A
PC7-PC4
M0
PORT B
PC3-PC0
1 for I/O Mode
0 for BSR mode
Mode selection for port A,
M1 M2 Mode 0
0 0 Mode 1
0 1 Mode 2
1 *
Port A bits
For input = 1, output = 0
Port C lower bits
For input = 1, output = 0
Port B bits
For input = 1, output = 0
Mode selection for port B,
0 for mode 0, and 1 for mode 1
Port C upper bits
For input = 1, output = 0

Fig. 5.3 Control Word Format for Intel 8255.

Bit 4: It is for Port A. To make Port A an input port, the bit is set to 1. To make Port A an output port, the bit is set to 0.

Bit 5, 6: These bits are used to define the operating mode of the Port A. For the various modes of Port A, these bits are defined as follows:

Table 5.1(a) Mode selection for Port A

Mode of Port A	*Bit No. 6*	*Bit No. 5*
Mode 0	0	0
Mode 1	0	1
Mode 2	1	0 or 1

For Mode 2, Bit No. 5 is set to either 0 or 1; it is immaterial.

Bit 7: It is set to 1 if Port A, B and C are defined as input/output port. It is set to 0 if the individual pins of the Port C are to be set or reset (in BSR mode).

Table 5.1(b) Shows control words for various configurations of the ports of 8255 for Mode 0 operation below:

Control word	*Port A*	*Port C (upper)*	*Port B*	*Port C (lower)*
80H	Output	Output	Output	Output
81H	Output	Output	Output	Input
82H	Output	Output	Input	Output
83H	Output	Output	Input	Input

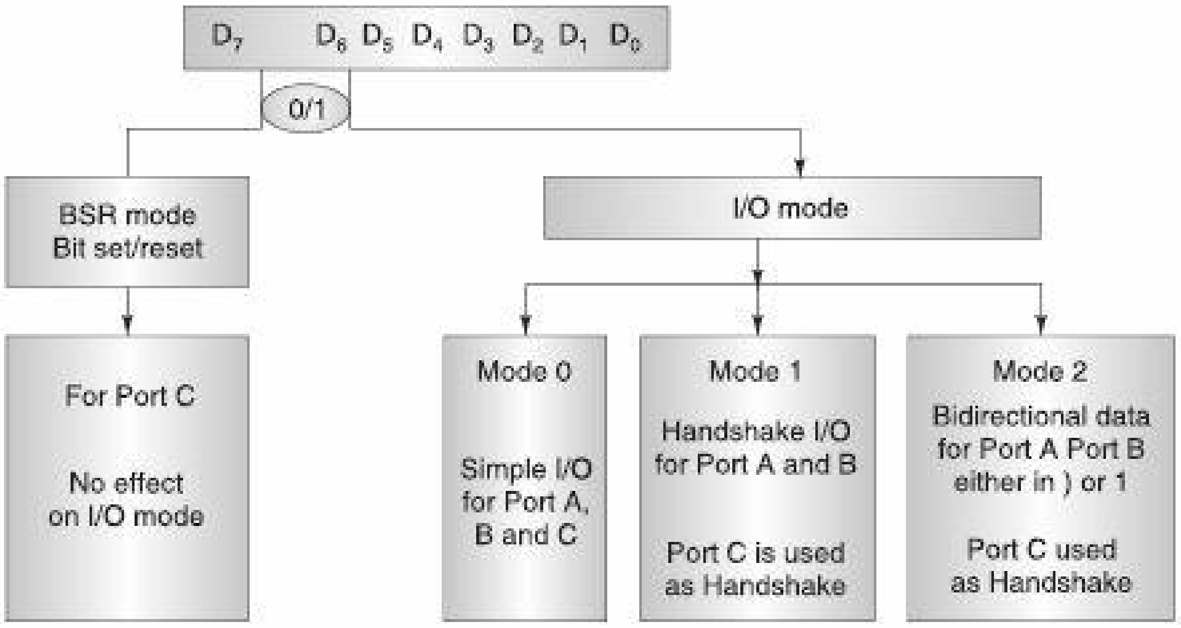
5.4.4 Operating Modes of 8255

Intel 8255 works in two modes: (1) Input/output mode (2) Bit set/reset mode (BSR). In I/O mode, 8255 has the following three modes of operation which are selected by software:

1. Mode 0—Simple or Basic Input/Output
2. Mode 1—Strobed Input/Output

3. Mode 2—Bidirectional Port/Data Bus

These are shown in Figure 5.4 and their explanation is also described as:



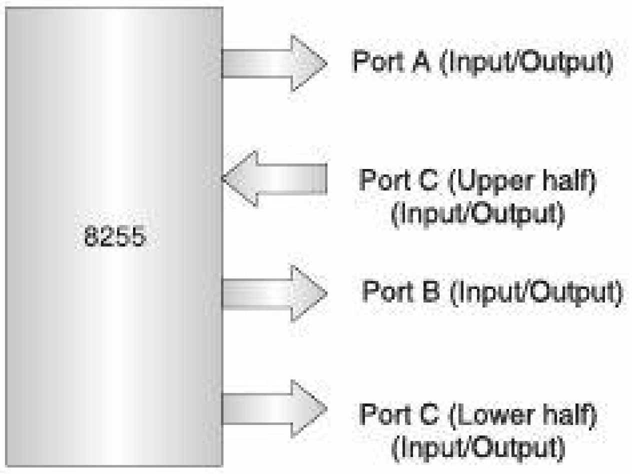
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
0/1
BSR mode Bit set/reset
I/O mode
For Port C
No effect on I/O mode
Mode 0
Simple I/O for Port A, B and C
Mode 1
Handshake I/O for Port A and B
Port C is used as Handshake
Mode 2
Bidirectional data for Port A Port B either in) or 1
Port C used as Handshake

Fig. 5.4 Modes of 8255.

Mode 0: The 8255 has two 8-bit ports (Port A and Port B) and two 4-bit ports (Port Cupper and Port Clower) in Mode 0 operation; a port can be operated as a simple input or output port. Each of the port of 8255 can be programmed to be either an input or output port.

If port C is programmed to be in Mode 0 (Input), then a peripheral device can write data into this port whenever required. Further, the microprocessor can read data from this port. Any output device connected to this port will immediately receive this data.

When a port acts as an output port in Mode 0, any data written into the port is latched. However, if the port acts as an input, then the data written into it by the peripheral device is only buffered but not latched. Figure 5.5 shows different ports in Mode 0.



8255
Port A (Input/Output)
Port C (Upper half) (Input/Output)
Port B (Input/Output)
Port C (Lower half) (Input/Output)

Fig. 5.5 8255A PPI Configuration in Mode 0.

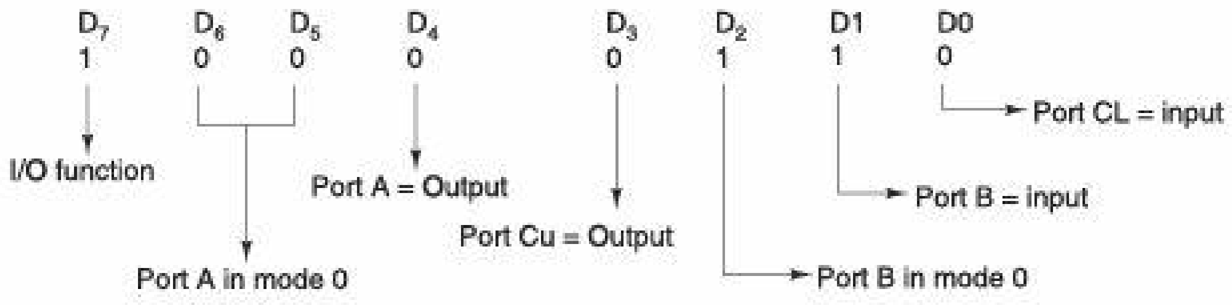
Example 5.1: Write a program to read the DIP switches and display the reading from port B to port A and from port C_L to C_U. (In Mode 0).

1. Port Address: This is a memory-mapped I/O, when the address line A_{15} is high; the chip select line is enabled. Assuming all don't care lines are at logic 0, the port addresses are as follows:

Port A = 8000H (A_1 = 0, A_0 = 0)
Port B = 8001H (A_1 = 0, A_0 = 1)
Port C = 8002H (A_1 = 1, A_0 = 0)
Control Register = 8003H (A_1 = 1, A_0 = 1)

Control Word:

After evaluating the data that we need to load in control word and port addresses of Port A, Port B, Port C, we can make the program as:



```
MVI A, 83H      ; Load accumulator with the control word
STA 8003H       ; Write word in control register to initialize the ports
LDA 8001H       ; Read switches at port B
STA 8000H       ; Display reading at port A
LDA 8002H       ; Read switches at port C
ANI 0FH         ; Mask the upper four bits of port C; these bits are not input data bits
RLC             ; Rotate and place data in the upper half of the accumulator
RLC
RLC
RLC
STA 8002H       ; Display data at port CU
HLT             ; End of the program
```

Program Description: The instructions are written as for memory mapped I/O, because all the 8255A ports are memory locations in this program. The ports are initialized by placing the control word 83H in the control register. The instructions STA and LDA are equivalent to the instructions OUT and IN, respectively.

In this example, the low four bits of port C are configured as input and high four bits are configured as output; even though port C has one address for both halves. Control signals like $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ are used to differentiate read and write operations. When microprocessor read port C, it receives eight bits in the accumulator. High-Order bits of port C is ignored because the input data bits are in PC_0-PC_3. To display these bits at upper half of port C, bits (PC_0-PC_3) must be shifted to PC_4-PC_7.

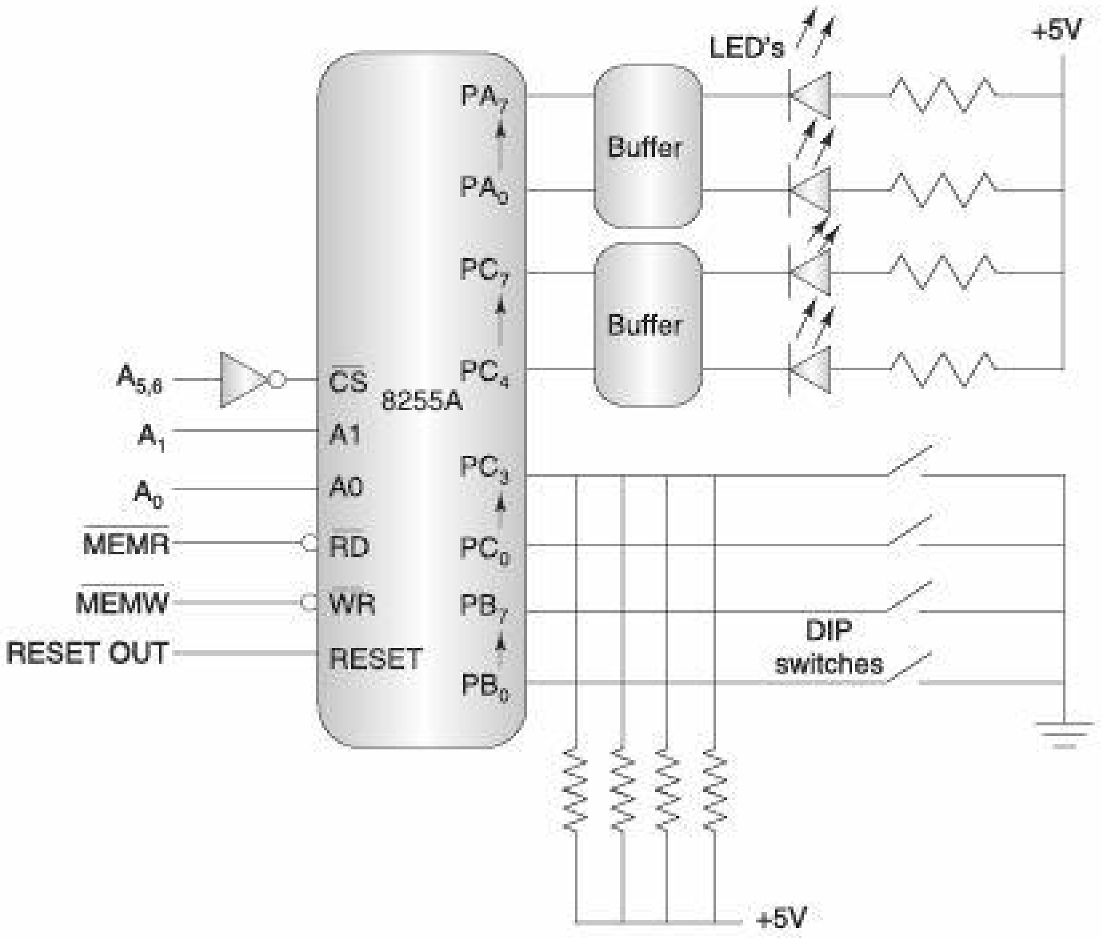


Fig. 5.6 Interfacing 8255A I/O Ports in Mode 0.

Mode 1 (Input/output configuration): In this mode of configuration, ports A and ports B are used as I/O ports. They can be used either as input port or output port. The port C is used for providing six handshake signals and remaining two lines are used as simple I/O lines. If port A is operated as an input port, pins PC_3, PC_4, PC_5 are used for its control. The remaining pins of port C, i.e., PC_6-PC_7 can be used either as input or output. And if port A is used as output port, pins PC_3, PC_6, PC_7 are used for its control. The pins PC_4 and PC_5 can be used either as input or output. The input and output configuration of mode 1 are shown in Figures 5.7(a) and 5.7(b) respectively. In this mode, data is transferred from output peripheral device into port (in input configuration) using the strobe signal (STB). Similarly, in strobed output mode, the device is informed with the aid of the $\overline{\text{OBF}}$ (output buffer full) signal that the microprocessor has written data into port.

Input control signals in Mode 1: The associated control signal used for handshaking, when port A and port B are configured as input ports as shown in Figure 5.7(a). The functions of these signals are as follows:

$\overline{\text{STB}}$ (Strobe Input): This signal is generated by a peripheral device to indicate that it has transmitted a byte of data. The 8255A, in response to $\overline{\text{STB}}$, generate IBF and INTR signal as shown in Figure 5.10.

IBF (Input Buffer Full): This signal is an acknowledgement by the 8255A to indicate that input latch has received the data byte. This is reset, when the MPU reads the data.

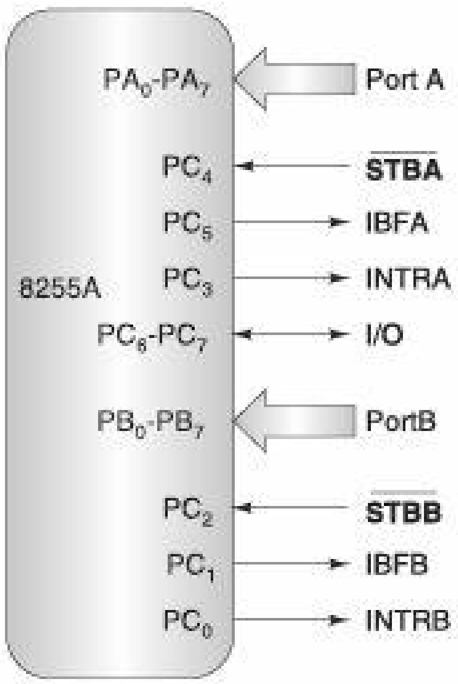


Fig. 5.7(a) Mode 1 Input Configuration.

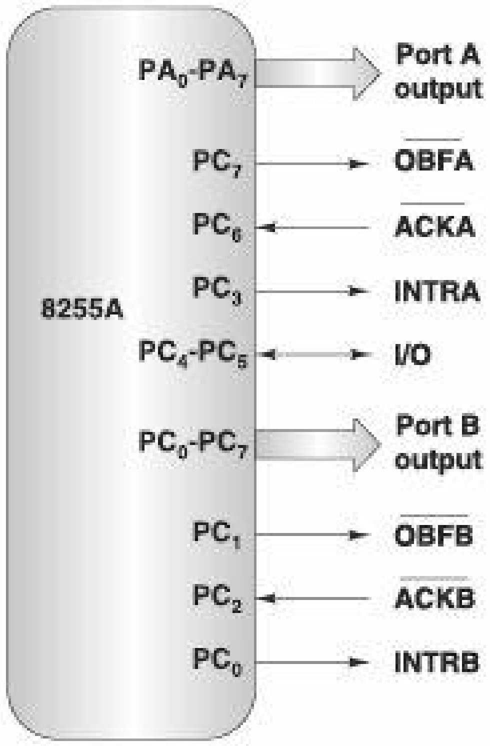


Fig. 5.7(b) Mode 1 Output Configuration.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
I/O	I/O	IBFA	INTEA	INTRA	INTEB	IBFB	INTRB

Fig. 5.8 Status Word in Mode 1 Input Configuration.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
$\overline{OBFA}$	INTEA	I/O	I/O	INTRA	INTEB	$\overline{OBFB}$	INTRB

Fig. 5.9 Status Word in Mode 1 Output Configuration.

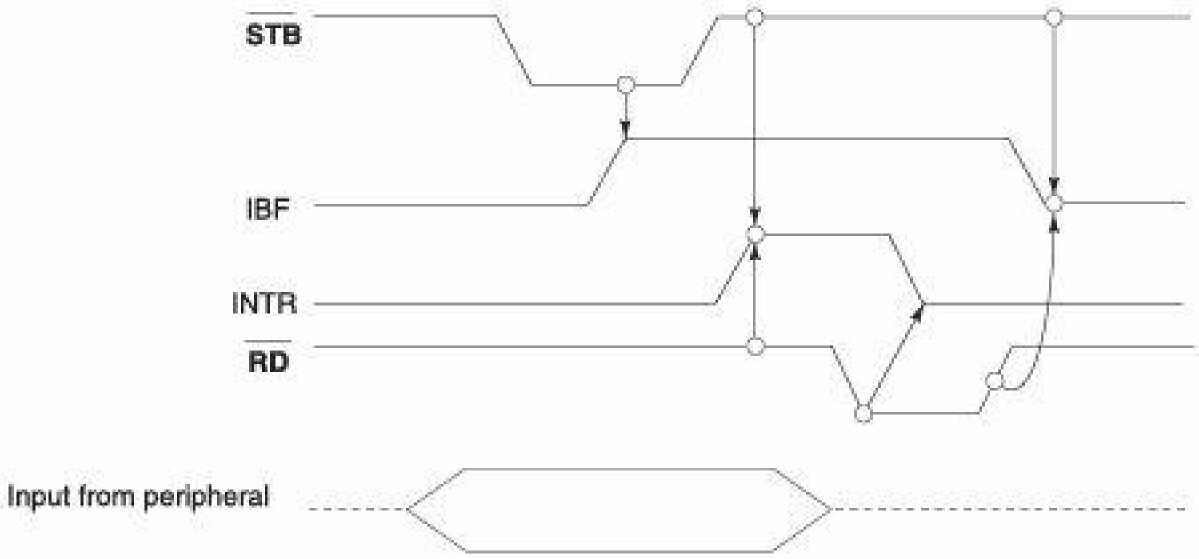


Fig. 5.10 Timing Waveform for Strobed Input with Handshake in Mode 1.

INTR (Interrupt Request): This is an output signal that may be used to interrupt the MPU. This signal is generated if $\overline{STB}$, IBF and INTE (interrupt flip-flop) are all at logic 1. This is reset by the falling edge of the $\overline{RD}$ signal.

INTE (Interrupt Enable): This signal is an internal flip-flop used to enable or disable the generation of the INTR signal. The two flip-flops INTEA and INTEB are set/reset using BSR mode. The $INTE_A$ is enabled or disabled through PC_4 and INTEB is enabled or disabled through PC_2.

Output control signals in Mode 1: Figure 5.7(b) show the Mode 1 Output Configuration, with control signals, that are defined as follows:

OBF (Output Buffer Full): This is an output signal that goes low when the MPU writes data into output latch of 8255A. This signal indicates to an output peripheral that new data is ready to be read as shown in Figure 5.11. It goes high again after the 8255A receive an acknowledgement from peripheral.

$\overline{ACK}$ (Acknowledgement): This is an input signal from a peripheral that must output a low, when the peripheral receives the data from 8255A ports (Figure 5.11).

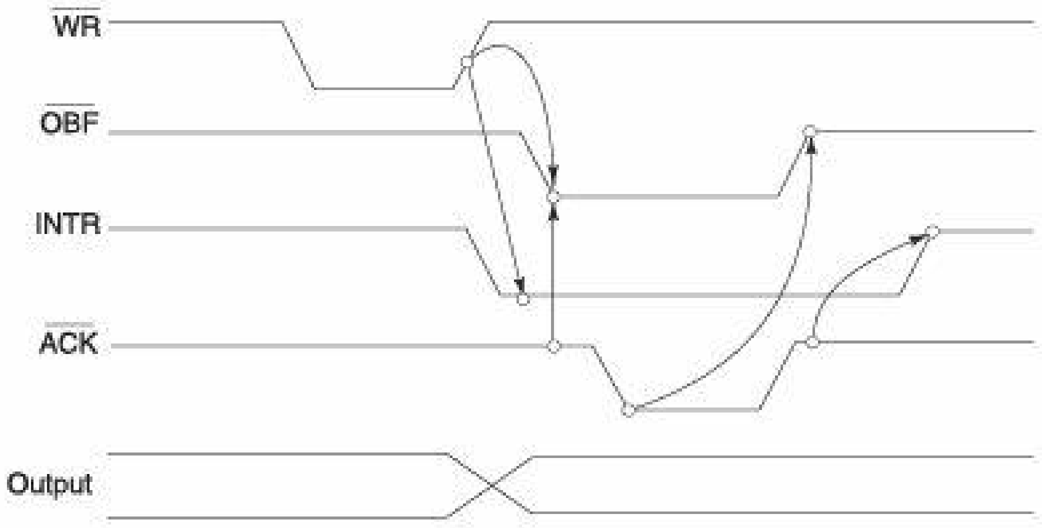


Fig. 5.11 Timing Waveform for Strobed Output with Handshake in Mode 1.

INTR (Interrupt Request): This is an output signal and it is set by rising edge of $\overline{ACK}$ signal. This signal can be used to interrupt the MPU to request the next data byte for output. The INTR is set when $\overline{OBF}$, $\overline{ACK}$ and $\overline{INTE}$ are all one and reset by falling edge of $\overline{WR}$.

INTE (Interrupt Enable): This is an internal flip-flop to a port and needs to be set to generate the INTR signal. The two flip-flops $INTE_A$ and $INTE_B$ are controlled by bits PC_6 and PC_2 respectively, through the BSR mode.

$PC_{4, 5}$: These two lines can be set up either as input or output.

Example 5.2: An interface circuit using the 8255A in Mode 1 is shown in Figure 5.12. Port B is designed as the input port for a keyword with interrupt I/O, and port A is designed as the output port for a printer with status check I/O.

Find port addresses by analyzing the decode logic.

1. Determine the control word to set up port B as input port and port A as output port in Mode 1.
2. Determine the BSR word to enable $INTE_B$ (port B).
3. Determine the masking byte to verify the $\overline{OBF_A}$ line in status check I/O (port A).
4. Write initialization instruction and a printer subroutine to output characters that are stored in memory.

1. Port Addresses: The 8255A is connected as peripheral I/O. The port addresses are determined by analyzing the decode logic as shown in Figure 5.12. According to this, the port addresses are given in Table 5.2.

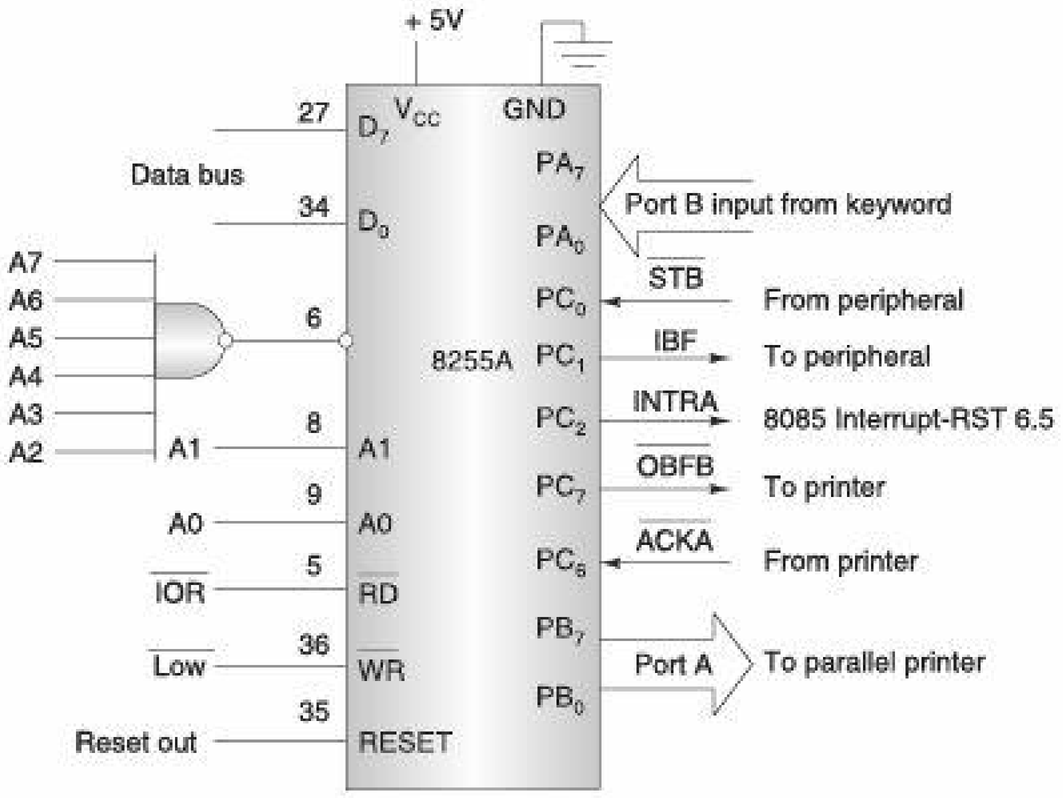


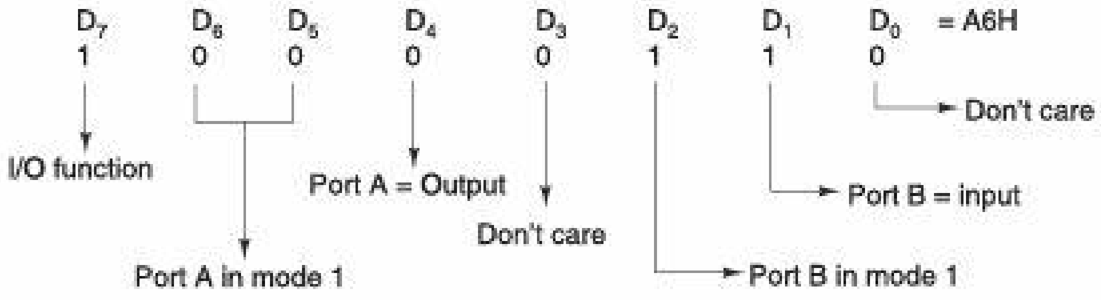
Fig. 5.12 Interfacing the 8255 in Mode 1.

Table 5.2 Port Address

$\overline{CS}$						Port Selection Bits		Hex Address	Port Selected
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		
1	1	1	1	1	1	0	0	FCH	A
1	1	1	1	1	1	0	1	FDH	B
1	1	1	1	1	1	1	0	FEH	C
1	1	1	1	1	1	1	1	FFH	Control register

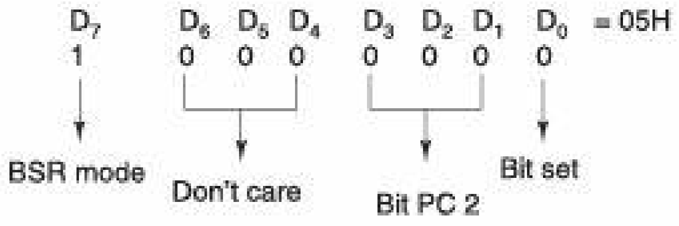
2. Control Word to set up Port B as input port and port A as output port in Mode 1: In the above control word, all the bits have values (0 or 1) according to our program requirement.

Bits D_3 and D_0 are in don't care logic state. To generate interrupt $INTR_B$ signal, $INTE_B$ flip-flop is set to 1, which is done by setting PC_2 bit in BSR mode.



The output of Port A is status controlled. Therefore, the status of line $\overline{OBF_A}$ can be checked by reading D_7 bit of port C_U.

1. BSR Word to set $INTE_B$: To set INTEB, bit PC_2 should be 1and it is set in BSR mode as given below:



2. Status word to check $\overline{OBF_A}$: It is checked by masking address with status word as shown in Figure 5.11(b). The OBFA bit is set to 1and it is given as:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	x	x	x	x	x	x	x

Masking Byte = 80H

3. Initialization Program:

```
                 MVI A, A6H     ;Word to initialize port B as input and port A as
                                output port in Mode 1
                 OUT FFH
                 MVI A, 05H     ;Set INTEB  (PC2)
                 OUT FFH        ;Using BSR Mode
                 EI             ;Enable Interrupt
                 CALL PRINT     ;Continue other tasks
  Print Subroutine
  PRINT:         LXI H, MEM     ;Point index to location of stored characters
  MVI B, COUNT                  ;Number of character to be printed
  NEXT:          MOV A, M       ;Get character from the memory
```

```
                MOV C, A      ;Save character
STATUS:         IN FEH        ;Read port C for status of OBF
                ANI 80H       ;Mask all bits except D1
                JZ STATUS     ;If it is low, the printer is not ready; wait in the loop
                MOV A, C
                OUT FCH       ;Send a character to port B
                INX H         ;Point to next character
                DCR B
                JNZ NEXT
                RET
```

Program Description: In this program, 8255A in Mode 1 allow two operations, first is output to printer and second is input, the data from keyboard. For printer, we use status check interfacing and interrupt for keyboard interfacing.

In the print subroutine, the character is placed in the accumulator first and then the status is read by the instruction "IN FEH". Initially, the port A is empty, bit PC_1 ($\overline{OBF_A}$) is high. The instruction "OUT FCH "send the first character to port A. The presence of data byte in port A is indicated by $\overline{OBF}$ signal, when it goes low in the rising edge of $\overline{WR}$ signal as shown in Figure 5.11(c). After receiving a character, the printer sends back an acknowledgement signal, which in turn sets $\overline{OBF_A}$ high, and indicate that the port A is ready for next characters, and print subroutine continues.

Suppose a key is pressed during the PRINT, then a data byte is transmitted to port B and the $\overline{STB_B}$ goes low, which set IBF_B high. When $\overline{STB_B}$ goes high, all the conditions to generate $INTR_A$ are met. This signal interrupts the microprocessor through RST 6.5 interrupt. Then program control is transferred to the service routine and it read the content of port B, enables the interrupts and returns to the PRINT routine.

Mode 2: It is strobed bidirectional mode of operation. In this mode of operation, Port A can be programmed to operate as a bidirectional port. The Mode 2 operation is only for Port A. When Port A is programmed in Mode 2, the Port B can be used either in Mode 1 or Mode 0. For Mode 2 operation PC_3 to PC_7 are used for the control of Port A.

The device can strobe the data into the port, using the STB input. The 8255A can strobe data to the device using the OBF output. Figure 5.13 shows different control signals associated with Mode 2 operation.

Individual bits of Port C can be controlled directly by microprocessor. Using this bit set/reset feature, the microprocessor can send a command to the 8255A for setting/resetting one of the 8 bits of Port C. It may be noted that bit set corresponds to setting a bit to 0 (low) and reset to setting a bit to 1(high).

Programming of Control Register

Example 5.3 Make control word when the ports of Intel 8255 are defined as follows:

Port A as an input port.

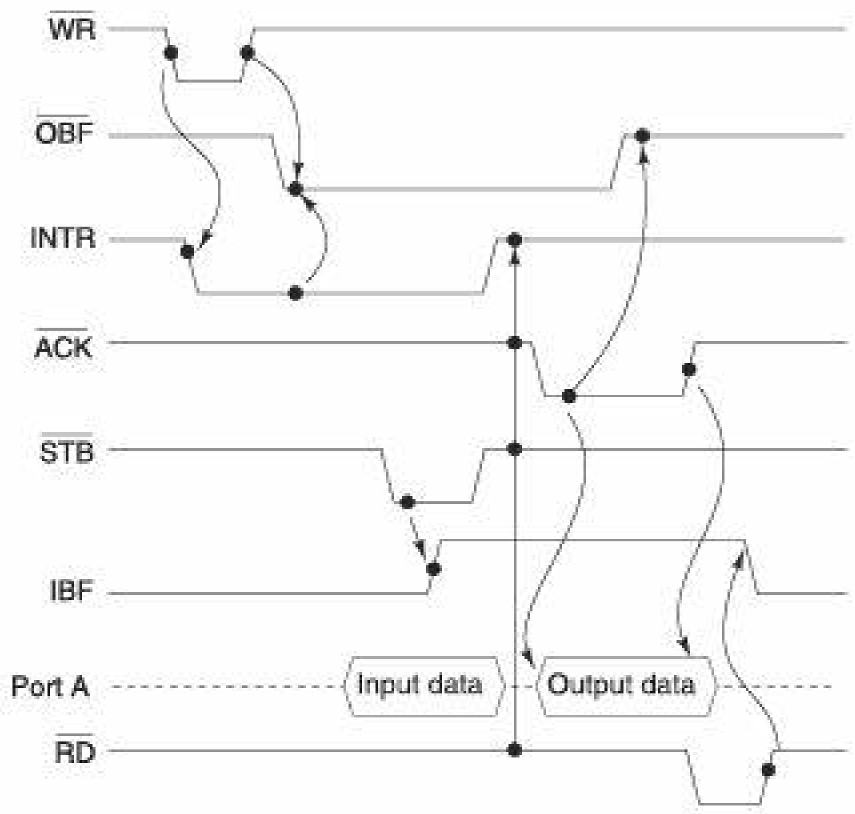


Fig. 5.13 Mode 2 Data Transfer Timing.

Mode of the Port A—Mode 0

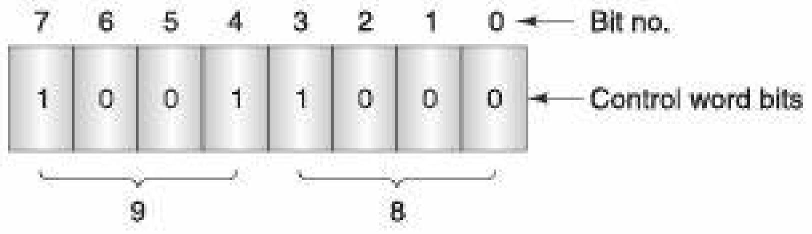
Port B as an output port.

Mode of the Port B—Mode 0

Port Cupper as an input port

Port Clower as an output port

It has already been explained with the help of Figure 5.6 how to set the bits of the control word for various ports depending on their definition to act either as input or output port. The control word bits for the above definition of the ports are as shown below:



The control word = 98H

Bit No. 0 is set to 0, as the Port Clower is an output port.

Bit No. 1 is set to 0, as the Port B is an output port.

Bit No. 2 is set to 0, as the Port B has to operate in Mode 0.

Bit No. 3 is set to 1, as the Port Cupper is an input port.

Bit No. 4 is set to 1, as the Port A is an input port.

Bit No. 5 and 6 are set to 00 as the Port A has to operate in Mode 0.

Bit No. 7 is set to 1, as the Ports A, B and C are used as simple input/output port.

Thus, the control word = 98H.

Example 5.4: Form control word for the following configuration of the ports of Intel 8255 for Mode 0 operation:

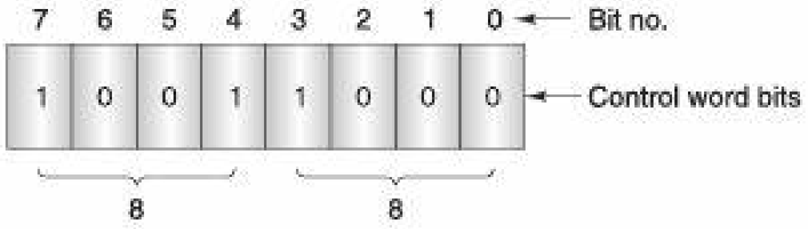
Port A—output

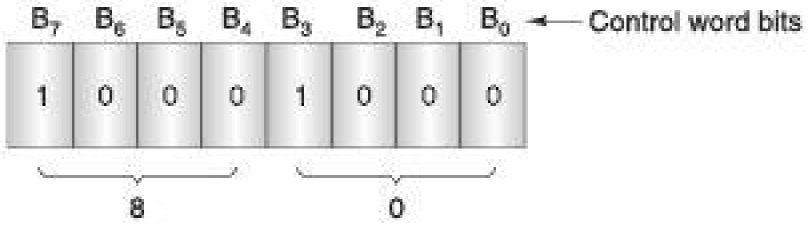
Port B—output

Port Clower—output

Port Cupper—input

The control word bits for the above configuration of the ports are as shown in figure. The control word for the above definition of the ports of Intel 8255 is 88H.





Example 5.5: Make control word for the following arrangement of the ports of Intel 8255 for mode 0 operation:

Port A—output

Port B—output

Port Cupper—output

Port Clower—output

The control word bits for the above configuration is 80H.

Example 5.6: Make control word for the following configuration of the ports of Intel 8255 for Mode 1 operation:

Port A—input

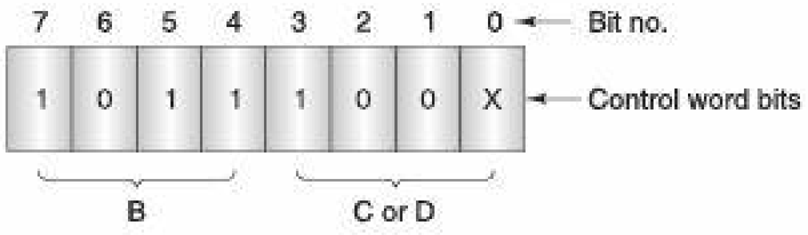
Mode of Port A—Mode 1

Port B—output

Mode of Port B—Mode 1

Remaining pins of the Port Cupper, i.e., PC_6 and PC_7—input.

When Port A and Port B are operated in Mode 1, six pins of the Port C are used for their control. PC_0, PC_1, PC_2 are used for the control of the Port B which can be programmed to be either an input or output port. If the Port A is operated as an input port, PC_3, PC_4 and PC_5 are used for its control. The pins PC_6 and PC_7 can be used as either input or output. When Port A is programmed as an output port, pins PC_3, PC_6 and PC_7 are used for its control. Pins PC_4 and PC_5 can be used as either input or output. When Port A and Port B are operated in Mode 1, six pins of the Port C are used for their control. PC_0, PC_1, PC_2 are used for the control of the Port B which can be programmed to be either an input or output port. If the Port A is operated as an input port, PC_3, PC_4 and PC_5 are used for its control. The pins PC_6 and PC_7 can be used as either input or output. When Port A is programmed as an output port pins PC_3, PC_6 and PC_7 are used for its control. Pins PC_4 and PC_5 can be used either as input or output.



As the Port C_{lower} is used for the control, the bit no. 0 which is for the Port C_{lower} remains undefined. It is immaterial whether the bit no. 0 is 1 or 0. The computer ignores the value of the bit no. 0. If the bit no. 0 is considered 0, the control word is BC. If its value is considered 1, the control word is BD.

Note. For the examples on Mode 1 and Mode 2 operation, see details in Intel's Microprocessor and Peripheral Handbook.

Example 5.7: Frame control word for the following configuration of the ports of Intel 8255 for Mode 1 operation:

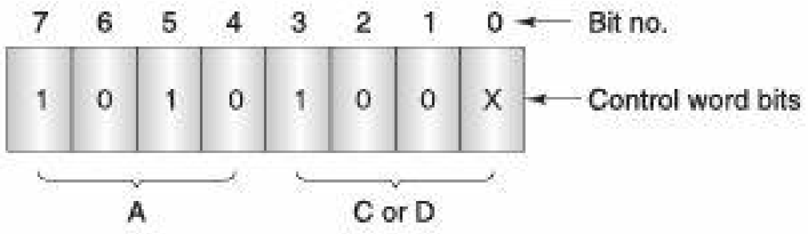
Port A—output

Mode of Port A—Mode 1

Port—output

Mode of Port—Mode 1

Remaining pins of Port C, i.e., PC_4 and PC_5—input.



For Mode 1 operation PC_0, PC_1 and PC_2 are used for the control of the Port B. When the Port A is used as an output port, PC_3, PC_6 and PC_7 are used for its control. PC_4 and PC_5 are available to be used either as input or output. The bit number 0 is undefined as the Port C_{lower} is used for control signals. If it is assumed 0, the control word is AC. If it is assumed 1, the control word is AD.

Example 5.8: Determine the control word for the following configuration of the ports of 8255.

Port A—Output

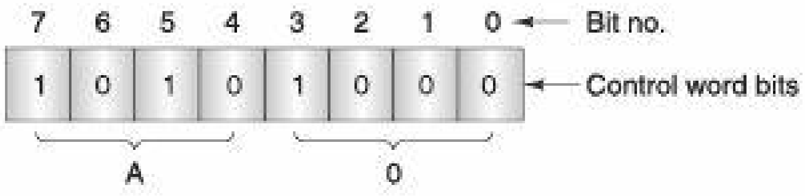
Mode of Port A—Mode 1

Port B—Output

Mode of Port B—Mode 0

Port Clower (Pin PC_0-PC_2)—Output

Remaining pins of Port Cupper, i.e., PC_4 and PC_5—Output



The example illustrates the combination of Mode 1 and Mode 0. As the Port A operates in Mode 1 as an output port, pins PC_3, PC_6, and PC_7 are used for control purposes. Remaining pins of Port C_{upper}, i.e., PC_0 and PC_5 can be used as I/O port. As Port B operates in Mode 0 pins PC_0-PC_2 are free to be used as I/O port in Mode 0. The control word is formed as follows:

Example 5.9: Determine the control word for the following configuration of the ports of Intel 8255.

Port A—Input

Mode of Port A—Mode 1

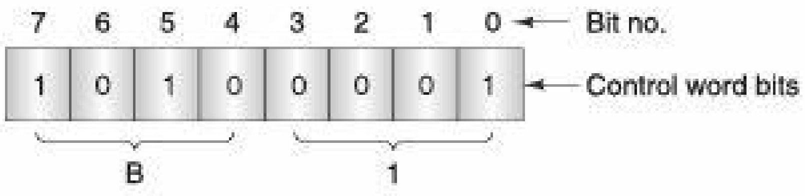
Port B—output

Mode of Port B—Mode 0

Port C_{lower} (PC_0-PC_2)—input

Remaining pins of Port C_{upper}, i.e., PC_6 and PC_7—output.

The control word is formed as follows:



The control word = B1H.

5.4.5 BSR (Bit Set/Reset) Mode

The BSR mode is concerned only with the eight bits of Port C, which can be set or reset by writing an appropriate control word in the control register. A control word with bit $D_7 = 0$ is recognized as BSR control word whereas control word with $D_7 = 1$ is input/output (I/O) mode as discussed earlier. Hence, it does not alter any previously transmitted control word. BSR Control Word has the following format:

Table 5.3 BSR Control Word

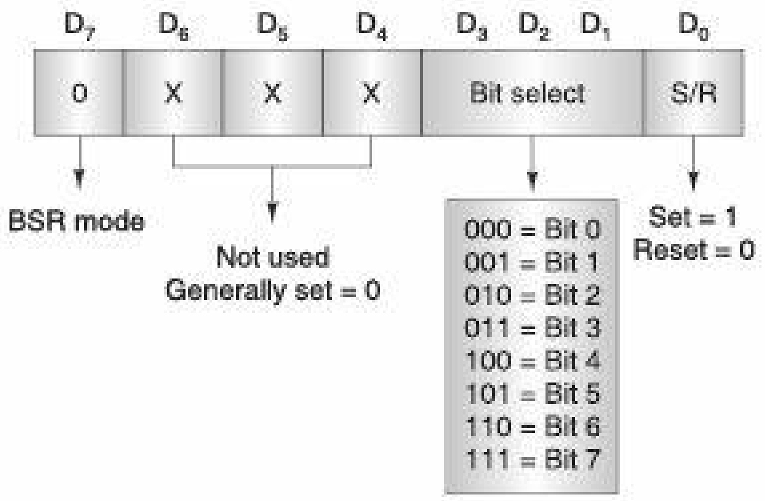
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	X	X	X	Bit Select			
BSR Mode	Not used	Not used	Not used	000 = Bit 0 001 = Bit 1 010 = Bit 2 011 = Bit 3 100 = Bit 4 101 = Bit 5 110 = Bit 6 111 = Bit 7			$S/\overline{R}$ Set = 1, Reset = 0

BSR control word:

To set bit PC_7	=	0	0	0	0	1	1	1	1	= 0FH
To reset bit PC_7	=	0	0	0	0	1	1	1	0	= 0EH
To set bit PC_3	=	0	0	0	0	0	1	1	1	= 07H
To reset bit PC_3	=	0	0	0	0	0	1	1	0	= 06H

Example 5.10: Write a BSR control word subroutine to set bits PC_7 and PC_4 and reset them after 25 ms. Assume that a delay subroutine is available.

Port Address: Control register address = 83H; refer to Table 5.1.



Subroutine

```
BSR:    MVI A, 0FH      ;Load byte in accumulator to set PC7
        OUT 83H         ;Set PC7  = 1
        MVI A, 09H      ;Load byte in accumulator to set PC4
        OUT 83H         ;Set PC4  = 1
        CALL DELAY      ;This is a 25 ms delay
        MVI A, 08H      ;Load accumulator with the byte to reset PC4
        OUT 83H         ;Reset PC4
```

```
MVI A, 0EH      ;Load accumulator with the byte to reset PC7
OUT 83H         ;Reset PC7
RET
```

Description: To set/reset bits in port C, a control word is loaded in the control register and not in port C. We set PC_7 and than PC_4 separately by setting bits of control word in BSR mode. This control word, when written in the control register, set or reset one bit at a time. The delay of 25 ms is assumed to be specified by call delay subroutine which is available.

Note.

1. A BSR control word affects only one bit of port C and not other bits.
2. It does not affect the I/O Mode.

5.5 PROGRAMMABLE KEYBOARD/DISPLAY INTERFACE (INTEL 8279)

The Intel 8279 is a programmable keyboard interfacing device. Data input and display are the integral part of microprocessor kits and microprocessor-based systems. The designer requires a suitable interface for the same. The 8279 has been designed for the purpose of 8-bit Intel microprocessors. The 8279 has two sections, namely, keyboard section and display section. The function of the keyboard section is to interface the keyboard which is used as input device for the microprocessor. It can also interface toggle or thumb switches. The purpose of the display section is to drive alphanumeric displays or indicator lights. It is directly connected to the microprocessor bus.

The microprocessor is relieved from the burden of scanning the keyboard or refreshing the display. Some important features are: simultaneous keyboard display operations, scanned sensor mode, scanned keyboard mode, 8-character keyboard FIFO, strobed input entry mode, 2-key lock out or N-key roll over with contact debounce, single 16-character display, dual 8-or 16-numerical display, interrupt output on key entry, programmable scan timing and mode programmable from CPU. 8279 provides three input modes. These modes are as follows:

Scanned Keyboard Mode: This mode follows a key matrix to be interfaced using either encoded or decoded scans. In encoded scan, an 8 × 8 keyboard or in decoded scan, a 4 × 8 keyboard can be interfaced. The code of key pressed with SHIFT and CONTROL status is stored into the FIFO RAM.

Scanned Sensor Matrix: In this mode, a sensor array can be interfaced with 8279 using either encoded or decoded scan 8 × 8 sensor matrix can be interfaced. The sensor codes are stored in the CPU addressable sensor RAM.

Strobed input: In this mode, if the control lines go low, the data on return lines is stored in the FIFO byte by byte.

5.5.1 8279 Keyboard/Display Controller

Figure 5.14(a) shows the structure of 8279 and its interface to the bus. Its four main sections are: Keyword section, scan section, display section and MPU interface section.

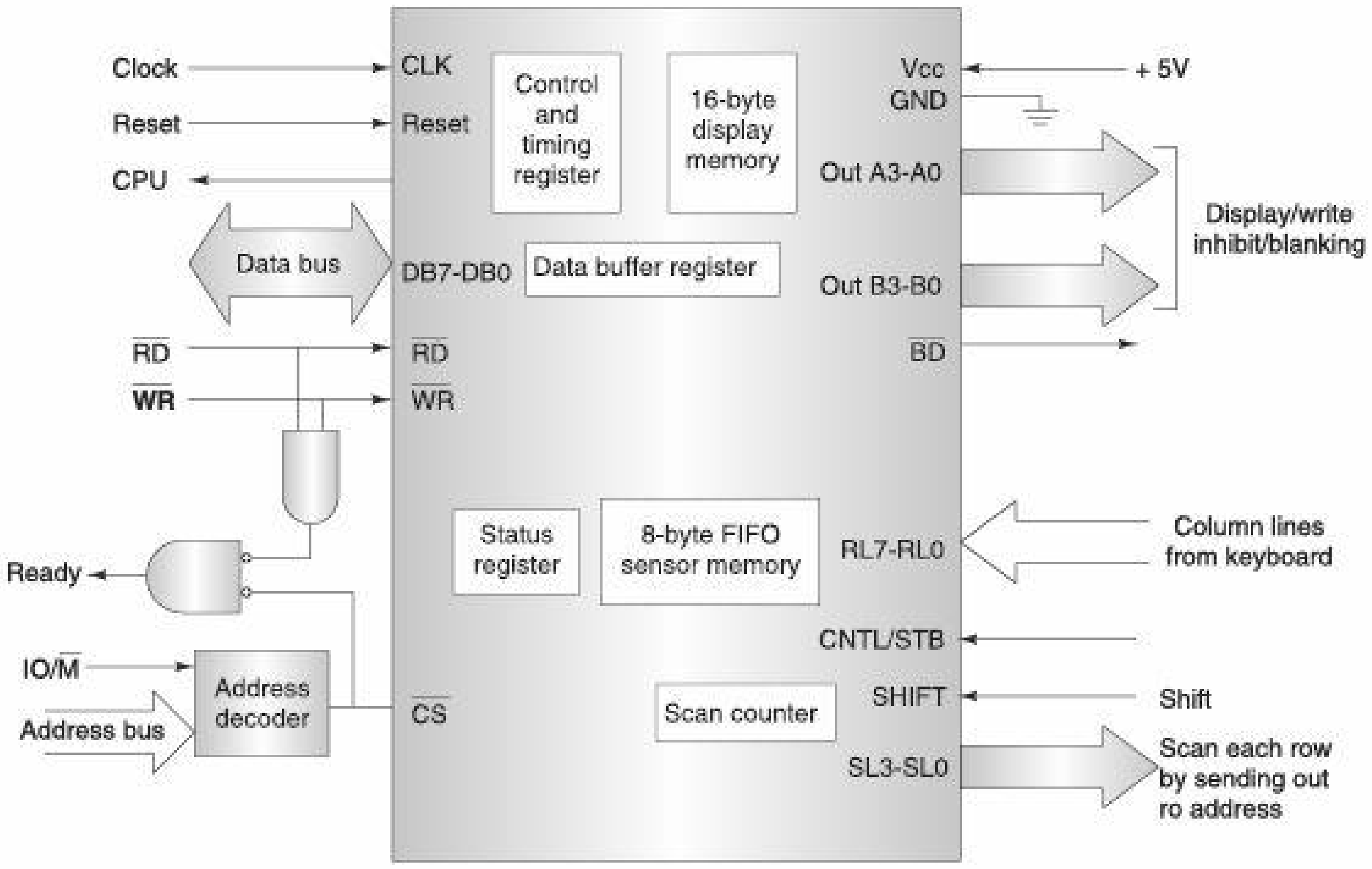


Fig. 5.14(a) Structure of the 8279.

Table 5.4 Transfer Description of Data

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_0	Transfer Description
0	1	0	0	Data bus to data bus buffer
0	0	1	x	Data bus to control register
0	1	0	x	Data buffer register to data bus
0	0	1	1	Status register to data bus

Keyboard Section: It contains RL_0-RL_7 lines that are connected to eight columns of the keyboard. Two other lines used are CNTL/STB (Control/Strobe) and Shift. The status of both lines can be stored with a key closure. Keyboard section includes 8 × 8 FIFO (First In First Out) RAM. It consists of eight registers to store eight keyboard entries. The status logic keeps the record of number of keyboard entries and provides an IRQ (Interrupt Request) signal to indicate that FIFO is full.

Scan Section: 8279 scans each row of the keyboard by sending out row addresses on SL_3-SL_0 and inputting signals on the return lines RL_7-RL_0, which are column addresses. When a depressed key is detected the key is automatically debounced by weighing 10 ms to check if the same key remains depressed. If a depressed key is detected and 8-bit code word corresponding to key position is assembled by combining column, row position and shift and control status as follows.

CNTL SHIFT R R R C C C

R-row (scan line address)

C-column (return line address)

The SHFT and CNTL pins are used to support shift and control keys. The key position is then entered into 8 × 8 FIFO sensor memory and IRQ (Interrupt Request) line is activated if sensor memory was previously empty.

The control and timing registers are collection flags and registers that are accessed by commands. The 3 MSBs of command determine its type and meaning of the remaining 5 bits. Out of 8 types of commands, 3 commands are important. Keyboard display mode set – it specifies the input and display methods and is used to initialize the 8279. Its format is:

Control bits to set up input mode

0 0 0 D D K K K

DD Control bits to set display modes

00-left entry, 8 × 8 bit display

01-left entry, 16 × 8 bit display

10-right entry, 8 × 8 bit display

11-right entry, 16 × 8 bit display

KKK

000 encoded keyboard scan mode with 2-key lockout

001 decoded keyboard scan mode with 2-key lockout

010 encoded keyboard scan mode with N-key rollover

011 decoded keyboard scan mode with N-key rollover

100 encoded sensor matrix scan mode

101 decoder sensor matrix scan mode

110 strobed input with encoded display scan

111 strobed input with decoded display scan

Display Section: The eight output lines are divided into two groups $A_0 - A_3$ and $B_0 - B_3$. These are used as a group of eight lines or two groups of four lines. This section use 16 × 8 display RAM. The display can be blanked by $\overline{\text{BD}}$ line. 8279 provides two output modes for selecting the display options.

These are discussed briefly as:

1. **Display Scan:** In this mode 8279 provides 8 or 16 character multiplexed displays which can be organized as dual 4-bit or single 8-bit display units.
2. **Display Entry:** (right entry or left entry mode) 8279 allows options for data entry on the displays. The display data is entered for display either from the right side or from the left side.

MPU Interface Section: In this section eight bidirectional data lines ($DB_0 - DB_7$), one IRQ line and six lines for interfacing (including A_0) are used. IRQ signal used to interrupt the MPU to indicate the availability of data.

5.5.2 Pin Description

The 8279 is a 40 pin IC package. Figure 5.14(b) shows the pin diagram of 8279 keyboard/ display controller. The explanation of pins is given below:

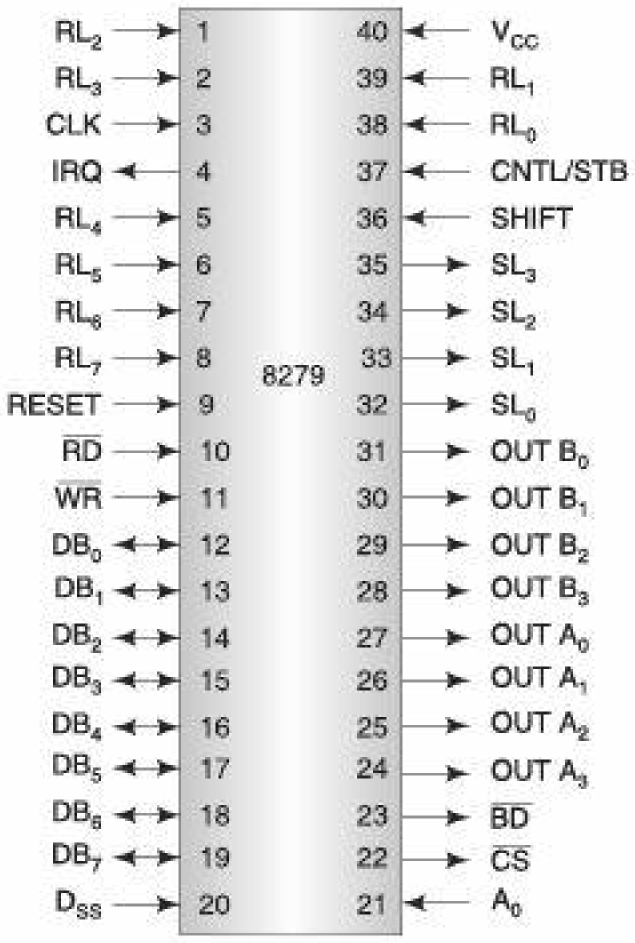


Fig. 5.14(b) Pin Diagram of 8279.

DB_0-DB_7 (Bidirectional data bus): The data, commands and status information between the CPU and the 8279 are transmitted on these data lines.

$\overline{RD}$ (Read): It is an active low input signal. When $\overline{RD}$ signal is low, CPU reads the contents of selected register (display RAM, status register or FIFO RAM) from 8279; depending on the status of A0 signal and command type.

$\overline{WR}$ (Write): It is an active low input signal. When $\overline{WR}$ signal is low, CPU writes data into selected register.

A_0 (Address line): It is an input signal. When it is at high logic, signals are interpreted as a command or status. When it goes low, signals are interpreted as data.

$\overline{CS}$ (Chip select): It is an active low input signal. When low, enable the 8279 IC.

RESET: A high signal on this pin reset the 8279 registers, commands and memory.

CLK (Clock): Clock input control the internal operation of 8279 and usually driven by system clock. It is used to generate internal timings.

IRQ (Interrupt Request): This signal is an output signal. It is used to implement interrupt driven input system. In scanned keyboard mode, the interrupt line goes low when there is data present in FIFO/sensor RAM. In sensor matrix mode, the interrupt line goes high whenever a change in a sensor is detected. The IRQ line is cleared by first data read operation if Auto-increment flag is zero and by End of Interrupt command if the Auto-increment flag is set to one.

RL0-RL7 (Return lines): These input lines are used to interface matrix keyboard with 8279. These lines have internal pull-ups, which keeps their status high. When any key is pressed, corresponding return line goes low.

SHIFT: It is an input line that remains high until a switch closure pulls it low. Its status is stored along with key position on the key closure in scanned keyboard modes.

CNTL/STB (Control/Strobe): In scanned keyboard mode, this line is used as a control input and similarly to shift key, its status is stored along with key position on key closure. In strobed input mode, this line used as a strobed input. A high logic on it, loads the status of keyboard into FIFO RAM.

OUT A_3-A_0 and OUT B_3-B_0: These are two ports, having four bits each. These are used for sending data to display drivers and connected to the segment inputs of 7-segment display. These lines are synchronized to scan lines for multiplexed digit display. Both ports can be blanked independently.

BD (Blank Display): This is an active low output signal that is used to blank the display during digit switching or by a display blanking command.

5.5.3 Keyboard Modes

1. **Scanned Keyboard Mode with 2 Key Lockout:** In this mode of operation, when a key is pressed, debounce logic comes into operation. During the next two scans, other keys are checked for closure and if no other key is pressed the first pressed key is identified. The key code of the identified key is entered into the FIFO with SHIFT and CNTL status, provided the FIFO is not full, i.e., it has at least one byte free. If the FIFO does not have any free byte, naturally the key data will not be entered and the error flag is set. If FIFO has at least one byte free, the above code is entered into it and the 8279 generates an interrupt on IRQ line to the CPU to inform about the previous key closures. If another key is found closed during the first key, the key code is entered in FIFO. If the first pressed key is released before the others, the first will be ignored. A key code is entered to FIFO only once for each valid depression, independent of other keys pressed along with it, or released before it. If two keys are pressed within a debounce cycle (simultaneously), no key is recognized till one of them remains closed and the other is released. The last key that remains depressed is considered single valid key depression.
2. **Scanned Keyboard with N-Key Rollover:** In this mode, each key depression is treated independently. When a key is pressed, the debounce circuit waits for 2 keyboard scans and then checks whether the key is still depressed. If it is still depressed, the code is entered in FIFO RAM. Any number of keys can be pressed simultaneously and recognized

in the order, the keyboard scan recorded them. All the codes of such keys are entered into FIFO. In this mode, the first pressed key need not be released before the second is pressed. All the keys are sensed in the order of their depression, rather in the order the keyboard scan senses them, and are independent of the order of their release.

3. **Scanned Keyboard Special Error Mode:** This mode is valid only under the N Key roll-over mode. This mode is programmed using end interrupt/error mode set command. If during debounce period (two keyboard scans) two keys are found pressed, this is considered a simultaneous depression and an error flag is set. This flag, if set, prevents further writing in FIFO but allows the generation of further interrupts to the CPU for FIFO read. The error flag can be read by reading the FIFO status word. The error flag is set by sending normal clear command with CF = 1.
4. **Sensor Matrix Mode:** In the sensor matrix mode, the debounce logic is inhibited. The 8-byte FIFO RAM now acts as 8 × 8-bit memory matrix. The status of the sensor switch matrix is fed directly to sensor RAM matrix. Thus, the sensor RAM bits contain the row-wise and column-wise status of the sensors in the sensor matrix. The IRQ line goes high, if any change in sensor valve is detected at the end of a sensor matrix scan or the sensor RAM has a previous entry to be read by the CPU. The IRQ line is reset by the first data read operation, if $A_1 = 0$, otherwise, by issuing the end interrupt command. AI is a bit in read sensor RAM word.

5.5.4 Display Modes

There are various options of data display. For example, the command number of characters can be 8 or 16, with each character organized as single 8-bit or dual 4-bit codes. Similarly, there are two display formats. The first one is known as left entry mode or typewriter mode, since in a typewriter the character typed appears at the leftmost position, while the subsequent characters appear successively to the right of the first one. The other display format is known as right entry mode, or calculator mode, since in a calculator the first character entered appears at the rightmost position and this character is shifted to one position left when the next character is entered. Thus, all the previously entered characters are shifted to left by one position when a new character is entered.

1. **Left Entry Mode:** In the left entry mode, the data is entered from left side of the display unit. Address 0 of the display RAM contains the leftmost display characters and address 15 of the RAM contains the rightmost display characters. It is just like writing in our address is automatically updated with successive reads or writes. The first entry is displayed on the leftmost display and the sixteenth entry on the rightmost display. The seventeenth entry is again displayed at the leftmost display position.
2. **Right Entry Mode:** In this right entry mode, the first entry to be displayed is entered on the rightmost display. The next entry is also placed in the rightmost display but after the previous display is shifted to left by one display position. The leftmost character is shifted out of that display at the seventeenth entry and is lost, i.e., it is pushed out of the display RAM.

Command Words

All the words or status words are written or read with $A_0 = 1$ and $\overline{CS} = 0$ to or from the 8279. This section describes the various commands available in 8279.

(a) **Keyboard Display Mode Set:** The format of the command word to select different modes of operation of 8279 is given below with its bit definitions:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	0	D	D	D	K	K	K	I

Fig. 5.15 Command Word.

Table 5.5 Function of K-bits in Display Mode

K	*K*	*K*	*Display Mode*
0	0	0	Encoded Scan, 2 key lockout (Default after reset)
0	0	1	Decoded Scan, 2 key lockout
0	1	0	Encoded Scan, N-key Rollover
0	1	1	Decoded Scan, N-key Rollover
1	0	0	Encode Scan, N-key Rollover
1	0	1	Decoded Scan, N-Rollover
1	1	0	Strobed Input Encoded Scan
1	1	1	Strobed Input Decoded Scan

Table 5.6 Function of $D_7 - D_6$ Bits in Display Mode

D_7	D_6	*Display Mode*
0	0	Eight-bit character left entry
0	1	Sixteen 8-bit character left entry
1	0	Eight 8-bit character right entry
1	1	Sixteen 8-bit character right entry

(b) **Programmable Clock:** The clock for operation of 8279 is obtained by dividing the external clock input signal by a programmable constant called prescaler. PPPPP is a 5-bit binary constant. The input frequency is divided by a decimal constant ranging from 2 to 31, decided by the bits of an internal prescaler, PPPP.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	0	1	P	P	P	P	P	I

Fig. 5.16 Programmable Clock.

(c) **Read FIFO/Sensor RAM:** The format of this command is given below. This word is written to set up 8279 for reading FIFO/sensor RAM. In scanned keyboard mode, AI and

AAA bits are of no use. The 8279 will automatically drive data bus for each subsequent read, in the same sequence, in which the data was entered. In sensor matrix mode, the bits AAA select one of the 8 rows of RAM. If AI flag is set, each successive read will be from the subsequent RAM location.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	1	0	A1	X	A	A	A	I

Fig. 5.17 Read FIFO/Sensor RAM.

X-Don't care

AI-Auto Increment Flag

AAA-Address pointer to 8-bit FIFO RAM

(d) **Read Display RAM:** This command enables a programmer to read the display RAM data. The CPU writes this command word to 8279 to prepare it for display RAM read operation. AI is auto increment flag AAAA, the 4-bit address points to the 16-byte display RAM that is to be read. If AI = 1, the address will be automatically incremented after each read or write to the display RAM. The same address counter is used for reading and writing.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	1	1	A1	A	A	A	A	I

Fig. 5.18 Read Display RAM.

(e) **Write Display RAM:**

AI–Auto increment flag

AAAA–4-bit addresses for 16-bit display RAM to be written.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	0	0	A1	A	A	A	A	I

Fig. 5.19 Write Display RAM.

(f) **Display Write Inhibit/Blanking:** The IW (inhibit write flag) bits are used to mask the individual nibble as shown in the below command word. The output lines are divided into two nibbles ($OUTA_0$–$OUTA_3$) and ($OUTB_0$–$OUTB_3$), those can be masked by setting the corresponding IW bit to 1. Once a nibble is masked by setting the corresponding IW bit to 1, the entry to display bit flags (BL) are used for blanking A and B nibbles. Here D_0, D_2 correspond to $OUTB_0$–$OUTB_3$ while D_1 and D_3 correspond to $OUTA_0$–$OUTA_3$ for blanking and masking.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	0	1	X	IW	IW	BL	BL	I

Fig. 5.20 Display Write Inhibit/Blanking.

If the user wants to clear the display, blank (BL) bits are available for each nibble as shown in format. Both BL bits will have to be cleared for blanking both the nibbles.

(g) **Clear Display RAM:** The CD_2, CD_1, CD_0 are used for selecting the blanking code to clear all the rows of the display RAM as given below. CD_2 must be 1 for enabling the clear display command. If $CD_2 = 0$, the clear display command is invoked by setting CA = 1 and maintaining CD_1, CD_0 bits exactly same as above. If CF = 1, FIFO status is cleared and IRQ line is pulled down. Also the sensor RAM pointer is set to row 0. If CA = 1, this combines the effect of CD and CF bits. Here, CA represents clear all and CF as clear FIFO RAM.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	1	0	CD2	CD1	CD0	CF	CA	I

Fig. 5.21 Clear Display RAM.

Table 5.7 Blanking Code

CD_1	CD_2	*Blanking Code*
0	x	All zeros for common cathode display
1	0	20H for alphanumeric display
1	1	All ones for common anode display

CD (CD1, CD0) bits are used to set Blanking code. Bit CD2 when set to one, enables clear display.

(h) **End Interrupt/Error Mode Set:** For the sensor matrix mode, this command lowers the IRQ line and enables further writing into the RAM. Otherwise, if a change in sensor value is detected, IRQ goes high that inhibits writing in the sensor RAM. For N-key rollover mode, if the E bit is programmed to be '1', the 8279 operates in special error mode. Details of this mode are described in scanned keyboard special error mode. X- Don't care.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	1	1	E	X	X	X	X	I

Fig. 5.22 End Interrupt/Error Mode Set.

5.5.5 Read FIFO Sensor Memory

Indicates a read operation that is memory. In the sensor mode, it specifies which row has to be read. This command is needed before inputting data from the FIFO memory. Its format is as given below:

0 1 0 1 X A A A

Row address to be read in a sensor mode X-don't.

Bit 4 is auto increment bit. If 1, next input is from the next byte in the FIFO.

Write to display memory

Indicates that write to data buffer register will put data in display memory.

Its format is:

1 0 0 1 A A A A

Address of the location in the display memory where the data for the next write will be stored. 8279 provides two options for handling the situation in which more than 1 key is depressed at about the same time. With the two-key lockout option, if another key is depressed while the first key is being debounced, the key which is released last will be entered into the FIFO. If the second key is depressed within two scan cycles after the first is debounced and the first key remains depressed after the second one is released, then the first key is recognized. If more than one is depressed, after they are depressed they are all entered in the order they were sensed. 8279 has a sensor matrix mode in which signals in the return lines are stored into the FIFO at the row corresponding to the scan address.

5.5.6 Display Control

8279 provides a 16-byte display memory and refresh logic. Each address in the display memory corresponds to a display unit with address 0 representing the leftmost display unit. Output is accomplished by 8279 repeatedly sending out characters over the lines OUT $A_3 - A_0$ and OUT $B_3 - B_0$ and unit select address is over $SL_3 - SL_0$. For the auto increment left entry, after each write to the display the addresses is incremented by one, so that the next character appears in the display unit to the right. Auto increment right entry allows character to be displayed in electronic calculators form. It causes the display to be shifted to left by one character and stores the next character from the right.

5.5.7 Interfacing a Microprocessor to Keyboard

The interfacing of 8086 microprocessor with 8 × 8 matrix keyboard is done with the help of 8279 (Programmable Keyboard/Display Interface) device as shown in Figure 5.23.

5.5.7.1 Keyboard Circuit Connections and Interfacing

In most keyboards, the key switches are connected in a matrix of rows and columns, as shown in Figure 5.24. We will use simple mechanical switches for our examples, but the principle is same for other type of switches. Getting meaningful data from a keyboard, requires the following three major tasks:

1. Detect a key press.
2. Debounce the key press.
3. Encode the key press.

Three tasks can be done with hardware, software, or a combination of two, depending on the application.

5.5.7.1.1 Software keyboard interfacing

Circuit Connection and Algorithm: Figure 5.25 shows how a hexadecimal keypad can be connected to a couple of microcomputer ports so that the three interfacing tasks can be done as part of a

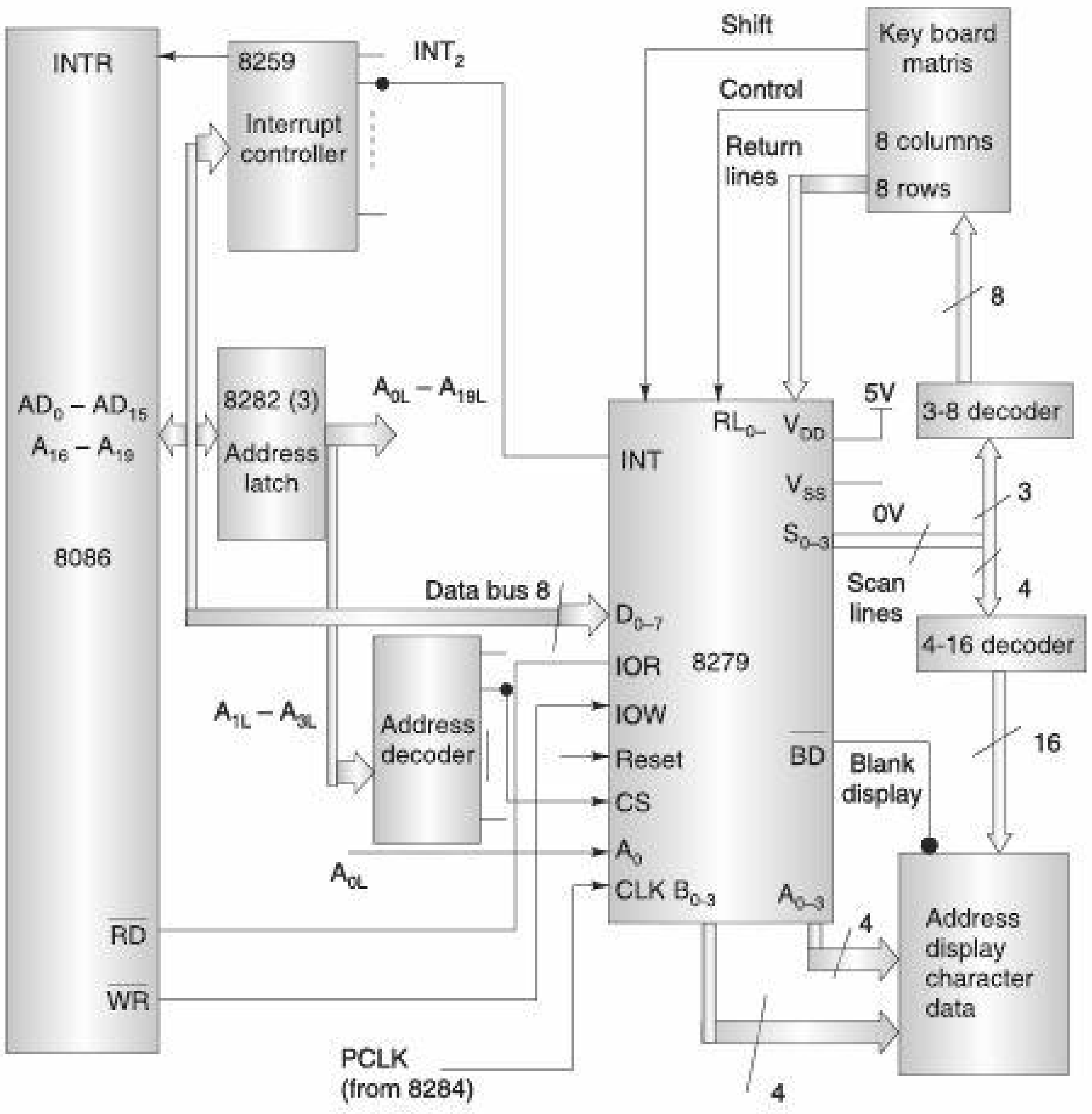


Fig. 5.23 Interfacing 8086 with 8279.

program. The rows of the matrix are connected to four output port lines. The column lines of matrix are connected to four input lines. When no keys are pressed, the column lines are held high by the pull-up resistor connected to +5 V.

Pressing a key connects a row to a column. If the output is low on a row and a key in that row is pressed, then the low will appear on the column which contains that key and can be detected on the input port. If you know the row and column of the pressed key, you then know which key was pressed, and you can convert this information into any code you want to represent that key.

The flow chart for a procedure to detect, debounce and produce the hex code for a pressed key is shown in Figure 5.24. An easy way to detect if any key in the matrix is pressed is to output 0's to all rows and then check the column to see if a pressed key has connected a low to a column. In the algorithm, we first output low to all the rows and check the columns over and over until the columns are all high. This is done before the previous key has been released before looking for the next one. In the standard keyboard terminology, this is called two-key lockout.

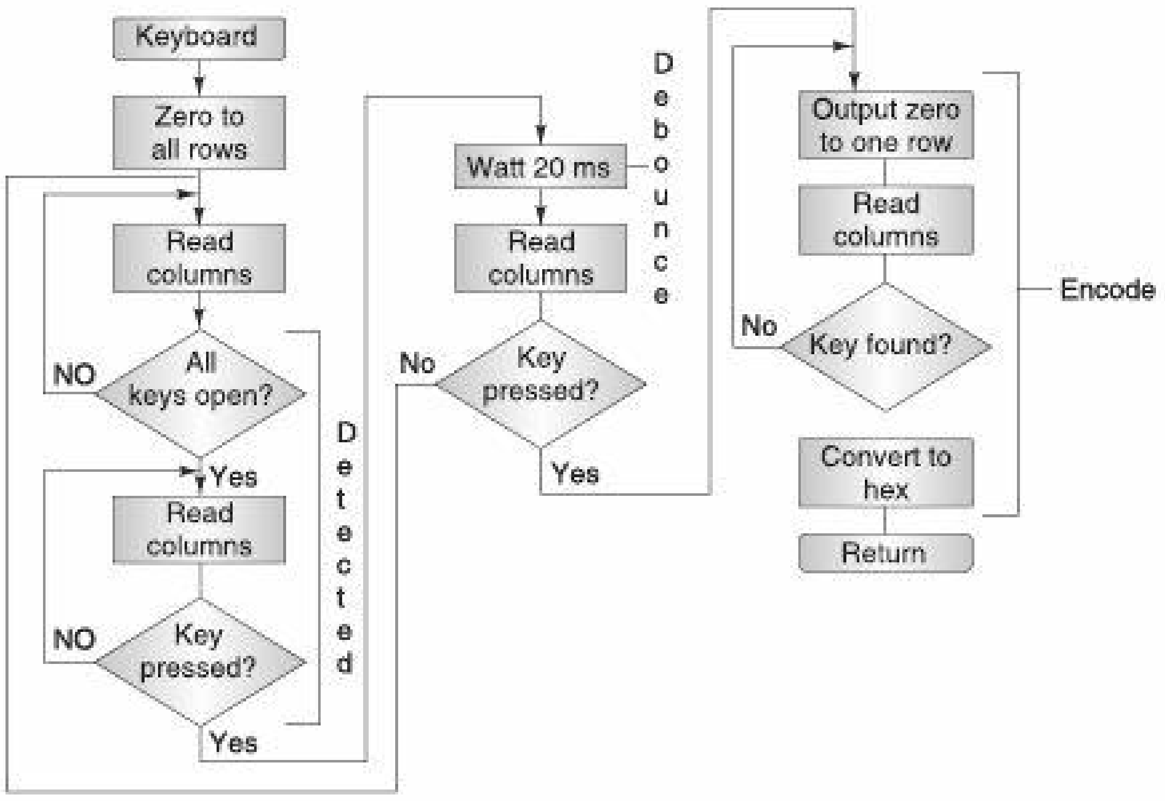


Fig. 5.24 Flow Chart for a Procedure to Detect, Debounce and Produce.

Once the columns are found to be all high, the program enters another loop, which waits until a low appears on one of the columns, indicating that a key has been pressed. This second loop does the detect task for us. A simple 20 ms delay procedure then does the debounce task. After the debounce time, another check is made to see if the key is still pressed. If the columns are now all high, then no key is pressed and the initial detection was caused by a noise pulse or a light brushing past a key. If any of the columns are still low, then the assumption is made that it was a valid key press. The final task is to determine the row and column of the pressed key. To get the row and column information to the hex code for the pressed key. To get the row and column information, a low is output to one row and the column is read. If none of the columns is low, the pressed key is not in that row. So the low is rotated to the next row and the columns are checked again. The process is repeated until a low on a row produces a low on one of the column. The pressed key then is in the row which is low at that time.

The connection in Figure 5.25 shows the byte read from the input port will contain a 4-bit code which represents the row of the pressed key and 4-bit code which represents the column of the pressed key.

Error trapping: The concept of detecting some error condition such as "no match found" is called error trapping. Error trapping is a very important part of real programs. Even in simple programs, think what might happen with no error trap if two keys in the same row were pressed at exactly the same time and a column code with lows in it was produced. This code would not match any of the row-column codes in the table, so after all the values in the table were checked, assigned register in program would be decremented from 0000H to FFFFH. The compare decrement cycle would continue through 65,636 memory locations until, by change the value in a memory location matched the row-column code. The contents of the lower byte register

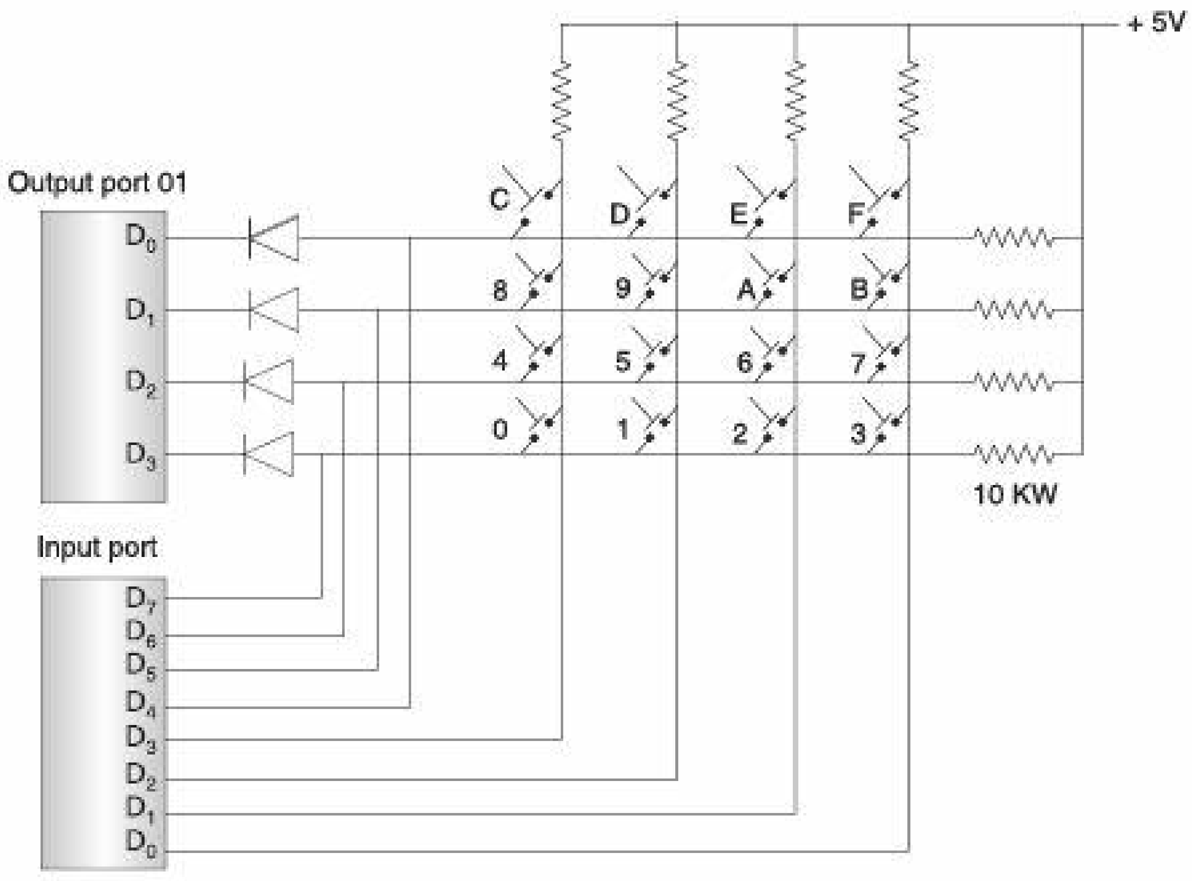


Fig. 5.25 Port Connections.

at that point would be passed back to the calling routine. The changes are 1 in 256 that would be the correct value for one of the pressed keys. You should keep an error trap in a program whenever there is a chance for it.

5.5.7.1.2 Keyboard interfacing with hardware

For the system where the CPU is too busy to be bothered doing this task in software, an external device is used to do them. One MOS device which can do this is the General Instruments AY5-2376 which can be connected to the rows and columns of a keyboard switch matrix. The AY5-2376 independently detects a key press by cycling a low down through the rows and checking the columns. When it finds a key pressed, it waits a debounce time.

If the key is still pressed after the debounce time, the AY5-2367 produces the 8-bit code for the pressed key and sends it out to microcomputer port on 8 parallel lines. The microcomputer knows that a valid ASCII code is on the data lines, the AY5-2376 outputs a strobe pulse. The microcomputer can detect this strobe pulse and read in ASCII code on a polled basis or it can detect the strobes pulse on an interrupt basis. With the interrupt method the microcomputer doesn't have to pay any attention to the keyboard until it receives an interrupt signal. So this method uses very little of the microcomputer time. The AY5-2376 has a feature called two-key rollover. This means that if two keys are pressed at nearly the same time, each key will be detected, debounced and converted into ASCII. The ASCII code for the first key and a strobe for it will be sent out then the ASCII code for the second key and a strobe signal for it will be sent out and compare this with two-key lockout.

5.6 CENTRONIX PARALLEL COMMUNICATION

A parallel port is that type of socket which is found on personal computers for interfacing with various peripherals. It is also known as a printer port or Centronics port. The IEEE 1284 standard defines the bidirectional version of the port.

For the most part, the USB interface has replaced the Centronics-style parallel port as of 2006, most modern printers are connected through a USB connection, and often don't even have a parallel port connection. On many modern computers, the parallel port is omitted for cost savings, and is considered to be a legacy port. In laptops, access to a parallel port is still commonly available through docking stations.

In Windows 98 and previous operating systems, programs could access the parallel port with simple outportb () and inportb () subroutine commands. In Windows NT, 2000 and XP, the microprocessor is operated in a different security ring, and accesses to the parallel port are inhibited, unless using the required driver. This improves security and arbitration of device contention. However, this also means that many legacy products will not work with later operating systems such as Windows XP.

Centronics parallel is generally compliant with IEEE 1284 compatibility mode. The original Centronics implementation called for the busy lead to toggle with each received line of data (busy by line), whereas IEEE 1284 calls for busy to toggle with each received character (busy by character). Some host systems or print servers may use a strobe signal with a relatively low voltage output or a fast toggle. Any of these issues might cause no or intermittent printing, missing or repeated characters or garbage printing. Some printer models may have a switch or setting to set busy by character; others may require a handshake adapter.

5.7 SERIAL COMMUNICATION

All IBM PCs and compatible computers are typically equipped with two serial ports and one parallel port. Although these two types of ports are used for communicating with external devices, they work in different ways.

A parallel port sends and receives data eight bits at a time over 8 separate wires. This allows data to be transferred very quickly; however, the cable required is more bulky because of the number of individual wires it contains. Parallel ports are typically used to connect a PC to a printer and are rarely used for much else. A serial port sends and receives data one bit at a time over one wire. While it takes eight times as long to transfer each byte of data this way, only a few wires are required. In fact, two-way (full duplex) communication is possible with only three separate wires—one to send, one to receive, and a common signal ground wire.

5.7.1 Synchronous and Asynchronous Communications

There are two basic types of serial communications, synchronous and asynchronous. With synchronous communications, the two devices initially synchronize themselves to each other, and then continually send characters to stay in synchronous; a block of character is transmitted along with the synchronization information as shown in Figure 5.26. Even when data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each character that is sent is either actual data or an idle character.

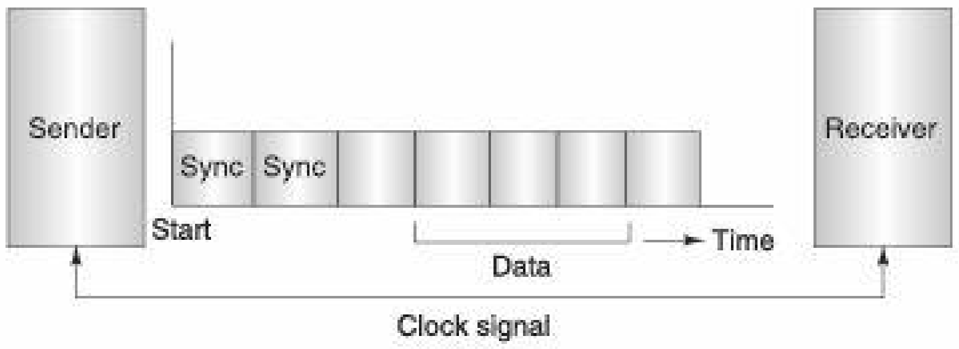


Fig. 5.26 Transmission Format for Synchronous Communication.

Synchronous communication allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

The serial ports on IBM-style PCs are asynchronous devices and, therefore, only support asynchronous serial communications. Asynchronous means "no synchronization", and thus does not require sending and receiving idle characters. However, the beginning and end of each byte of data must be identified by start and stop bits as shown in Figure 5.27. The start bit indicates when the data byte is about to begin and the stop bit signals when it ends. Transmission begins with one start bit (low) or two stop bits (high), known as "Framing". The requirement to send these additional two bits causes asynchronous communication to be slightly slower than synchronous; however, it has the advantage that the processor does not have to deal with the additional idle characters.

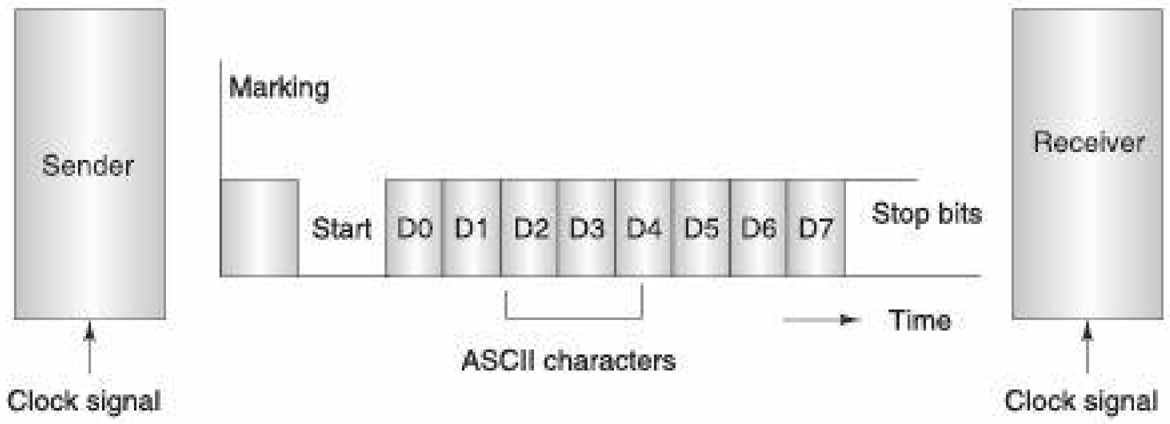


Fig. 5.27 Transmission Format for Asynchronous Communication.

An asynchronous line that is idle is identified with a value of 1 (also called a mark state). By using this value to indicate that no data is currently being sent, the devices are able to distinguish between an idle state and a disconnected line. When a character is about to be transmitted, a start bit is sent. A start bit has a value of 0 (also called a space state). Thus, when the line switches from a value of 1 to a value of 0, the receiver is alerted that a data character is about to be sent.

5.8 RS-232C

RS-232 stands for Recommend Standard number 232 and C is the latest revision of the standard. The serial ports on most computers use a subset of the RS-232C standard. The full RS-232C standard specifies 25-pin D connectors of which 22 pins are used. The pin description of RS-232C is shown in Figure 5.28(a). Most of these pins are not needed for normal PC communications, and indeed, most new PCs are equipped with male D type connectors having only 9 pins.

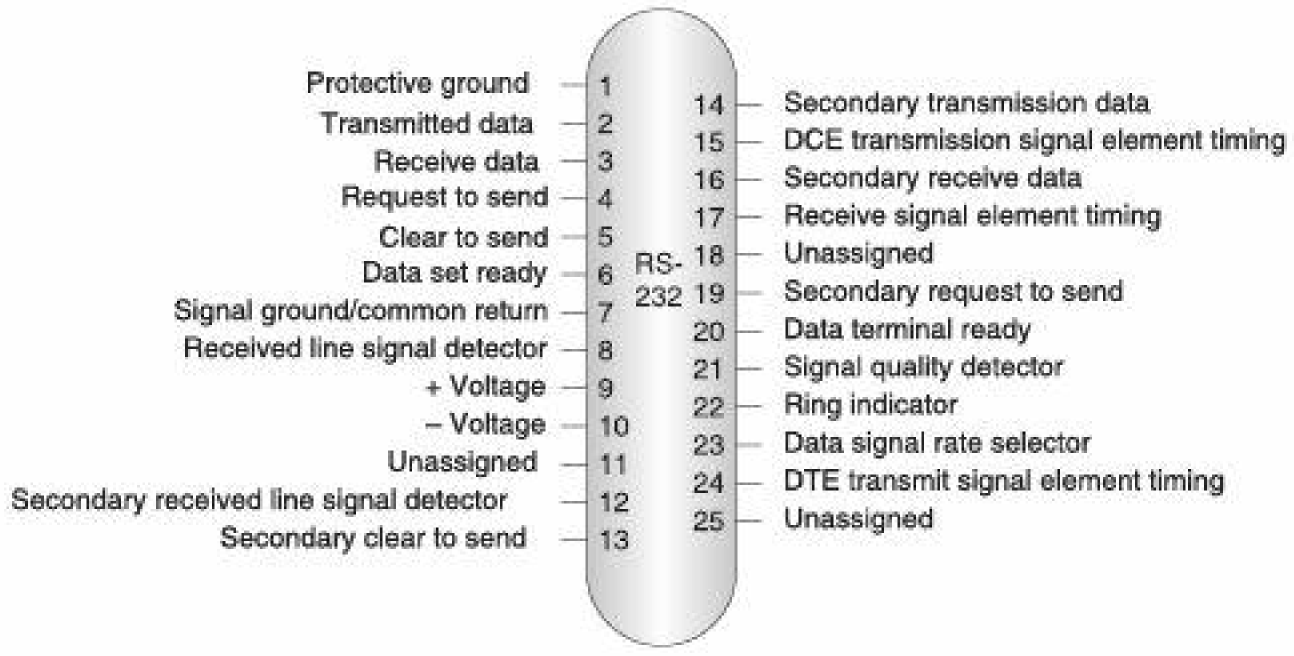

Protective ground — 1
Transmitted data — 2
Receive data — 3
Request to send — 4
Clear to send — 5
Data set ready — 6
Signal ground/common return — 7
Received line signal detector — 8
+ Voltage — 9
– Voltage — 10
Unassigned — 11
Secondary received line signal detector — 12
Secondary clear to send — 13
RS-232
14 — Secondary transmission data
15 — DCE transmission signal element timing
16 — Secondary receive data
17 — Receive signal element timing
18 — Unassigned
19 — Secondary request to send
20 — Data terminal ready
21 — Signal quality detector
22 — Ring indicator
23 — Data signal rate selector
24 — DTE transmit signal element timing
25 — Unassigned

Fig. 5.28(a) RS-232C 25-Pin Connector with Signal Definitions.

As shown in Figure of 25 pin connector, the signals are divided into four groups: data signals, timing signals, control signals and ground. For data lines, the voltage level +3V to +15V is defined as logic 0 and –3V to –15V is defined for logic 1. So, this is a negative true logic. The other signals are compatible with TTL logic. To make data lines compatible with TTL logic, we use a voltage translator known as line drivers and line receiver. MC 1488 is a line driver, which converts the logic 1 into –9V and logic 0 into +9V as shown in Figure 5.28(b). It is again converted by line receiver, MC1489, into TTL compatible logic, before received by the DCE.

Now the question arises that why we need to convert transmitted signal into higher level? The first reason is that this standard was defined before the TTL levels came into existence and that times most equipment were designed to handle higher voltages. Second reason is to provide a higher level of noise margin –from –3V to +3V.

Figure 5.28(b) also shows the minimum interface between a computer and a peripheral with pins 2, 3 and 7. Also the transmit and receive signals are defined in figure. Now, the dilemma is: How does a manufacturer define the role of the equipment. Suppose a user connects its microcomputer to the serial printer as a DTE (Data Terminal Equipment). To remain compatible with the RS-232C signal, it is configured as shown in Figure 5.29(b). In Figure 5.29(a) the microcomputer is defined as DTE and the modem defined as the DCE (Data Communication Equipment). It shows the microcomputer is connected to the modem without any modification in the RS-232C cable. However, when it is connected to the printer, the transmit and receive lines are crossed as shown in Figure 5.29(b). This type of connection is known as Null-Modem Connection. Typically, data transmission requires eight-handshake signals that are explained in next topic.

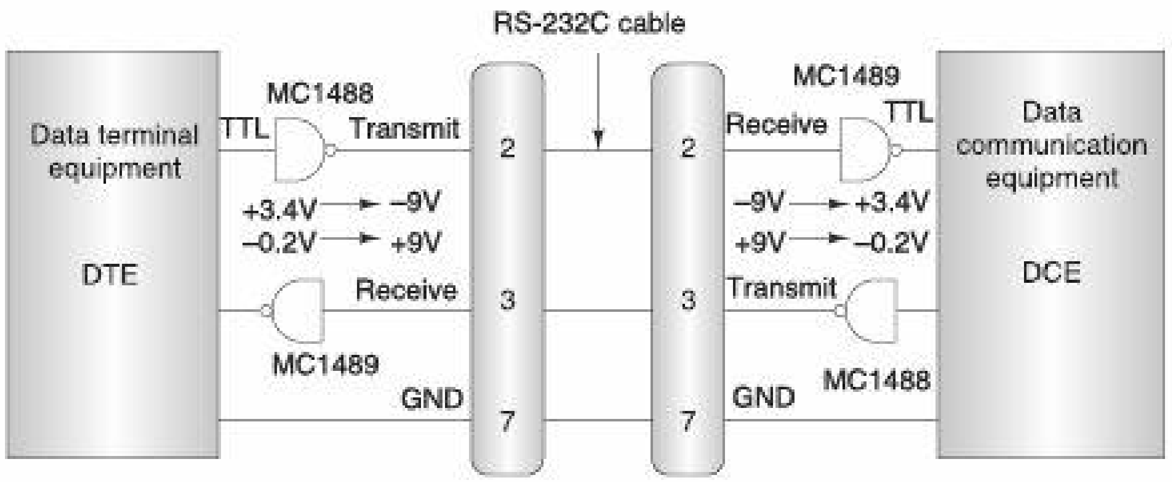


Fig. 5.28(b) Interface between DTE and DCE through RS-232C and Voltage Levels.

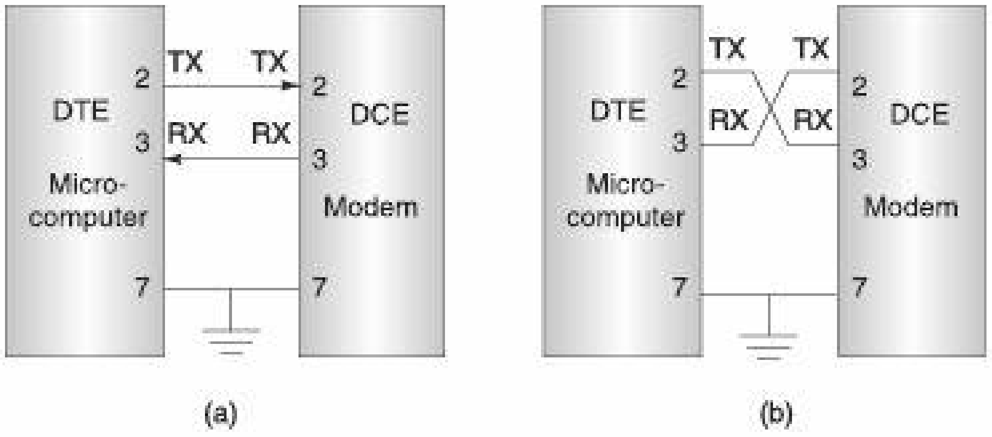


Fig. 5.29(a) DTE to DCE Connections. **Fig. 5.29(b)** DTE to DTE Connections.

5.8.1 DCE and DTE Devices

Two terms you should be familiar with are DTE and DCE. DTE stands for Data Terminal Equipment, and DCE stands for Data Communications Equipment. These terms are used to indicate the pin-out for the connectors on a device and the direction of the signals on the pins. Your computer is a DTE device, while most other devices are usually DCE devices.

If you have trouble keeping the two straight, then replace the term "DTE device" with "your PC" and the term "DCE device" with "remote device" in the following discussion.

The RS-232 standard states that DTE devices use a 25-pin male connector, and DCE devices use a 25-pin female connector. You can, therefore, connect a DTE device to a DCE using a straight pin-for-pin connection. However, to connect two like devices, you must instead use a null modem cable.

The TD (transmit data) wire is the one through which data from a DTE device is transmitted to a DCE device. This name can be deceiving, because this wire is used by a DCE device to receive its data. The TD line is kept in a mark condition by the DTE device when it is idle. The RD (receive data) wire is the one on which data is received by a DTE device, and the DCE device keeps this line in a mark condition when idle.

RTS stands for Request to Send. This line and the CTS line are used when "hardware flow control" is enabled in both the DTE and DCE devices. The DTE device puts this line in a mark condition to tell the remote device that it is ready and able to receive data. If the DTE device is not able to receive data (typically because its receive buffer is almost full), it will put this line in the space condition as a signal to the DCE to stop sending data. When the DTE device is ready to receive more data (i.e., after the data has been removed from its receive buffer), it will place this line back in the mark condition. The complement of the RTS wire is CTS, which stands for Clear To Send. The DCE device puts this line in a mark condition to tell the DTE device that it is ready to receive the data. If the DCE device is unable to receive data, it will place this line in the space condition. Together, these two lines make up what is called RTS/CTS or "hardware" flow control. The Software Wedge supports this type of flow control, as well as X_{on}/X_{off} or "software" flow control. Software flow control uses special control characters transmitted from one device to another to inform other device to stop or start sending data. With software flow control the RTS and CTS lines are not used.

DTR stands for Data Terminal Ready. Its intended function is very similar to the RTS line. DSR (Data Set Ready) is the companion to DTR in the same way that CTS is to RTS. Some serial devices use DTR and DSR as signals to simply confirm that a device is connected and is turned on. The Software Wedge sets DTR to the mark state when the serial port is opened and leaves it in that state until the port is closed. The DTR and DSR lines were originally designed to provide an alternate method of hardware handshaking. It would be pointless to use both RTS/CTS and DTR/DSR are not used at the same time for flow control signals. Because of this, DTR and DSR are rarely used for flow control.

CD stands for Carrier Detect. Carrier Detect is used by a modem to signal that it has made a connection with another modem, or has detected a carrier tone. The last remaining line is RI or Ring Indicator. A modem toggles the state of this line when an incoming call rings your phone.

The Carrier Detect (CD) and the Ring Indicator (RI) lines are only available in connections to a modem. Because most modems transmit status information to a PC, when either a carrier signal is detected (i.e., when a connection is made to another modem) or when the line is ringing, but they are least in use now.

5.8.2 Cable Length

The RS-232C standard has a cable length limit of 50 feet. This standard can be ignored since a cable can be as long as 10000 feet at baud rates up to 19200 if you use a high quality, well shielded cable. The external environment has a large effect on lengths for unshielded cables. In electrically noisy environments, even very short cables can pick up stray signals. You can greatly extend the cable length by using additional devices like optical isolators and signal boosters. Optical isolators use LEDs and photo diodes to isolate each line in a serial cable including the signal ground. Any electrical noise affects all lines in the optically isolated cable equally – including the signal ground line. This causes the voltages on the signal lines relative to the signal ground line to reflect the true voltage of the signal and thus cancelling out the effect of any noise signals.

Table 5.8 Cable Lengths and Baud Rates

Baud Rate	Shielded Cable Length	Unshielded Cable Length
110	5000	1000
300	4000	1000
1200	3000	500
2400	2000	500
4800	500	250
9600	250	100

5.9 UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER

A universal asynchronous receiver/transmitter (usually abbreviated UART) is a type of "Asynchronous Receiver/Transmitter", a piece of computer hardware that translates data between parallel and serial interfaces. Used for serial data telecommunication, an UART converts bytes of data into and from asynchronous start-stop bit streams represented as binary electrical impulses. UARTs are commonly used in conjunction with other communication standards such as EIA RS-232.

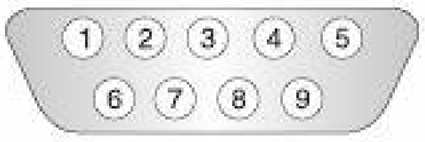
An UART is usually an individual (or part of) an integrated circuit used for serial communications over a computer or peripheral device serial port. UARTs are now commonly included in microcontrollers (for example, PIC16F628). A dual UART or DUART combines two UARTs into a single chip. Many modern ICs now come with a UART that can also communicate synchronously; these devices incorporate the word synchronous into the acronym to become USARTs. Transmitting and receiving serial data bits have to be moved from one place to another using wires or some other medium. Over many miles, the expense of the wires becomes large.

To reduce the expense of long distance communication links carrying several bits in parallel, data bits are sent sequentially, one after another, using an UART to convert the transmitted bits between sequential and parallel form at each end of the link. Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms.

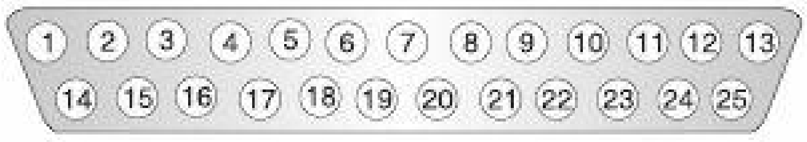
By convention, teletype-style UARTs send a "start" bit, five to eight data bits, least-significant bit first, an optional "parity" bit, and then one, one and a half, or two "stop" bits. The start bit is the opposite polarity of the data-line's idle state. The stop bit is the data-line's idle state, and provides a delay before the next character can start. (This is called asynchronous start-stop transmission.) In mechanical teletypes, the "stop" bit was often stretched to two bit times to give the mechanism more time to finish printing a character. A stretched "stop" bit also helps resynchronization. The parity bit can either makes the number of "one" bits between any start/ stop pair odd, or even, or it can be omitted. Odd parity is more reliable because it assures that there will always be at least one data transition, and this permits many UARTs to resynchronize. The UART usually does not directly generate or receive the external signaling levels (such as voltages on wires) that are used between different equipment. Typically, an interface is used to convert the logic level signals of the UART to the external signaling levels. "Signaling levels" is a very broad term encompassing all the various possible schemes to convey a level from one place to another. Voltage is by far the most common kind of signaling used. Examples of standards for voltage signaling are RS-232, RS-422 and RS-485 from the EIA. Historically, the presence or absence of current (in current loops) was the dominant kind of signaling used.

Depending on the limits of the communication channel to which the UART is ultimately connected, communication may be "full duplex" (both send and receive at the same time) or "half duplex" (devices take turns transmitting and receiving). Optical fibers are example of signalling scheme which, infrared, and (wireless) Bluetooth in its Serial Port Profile (SPP). Some signaling schemes use modulation (with or without wires). Examples are modulation of audio signals with phone line modems, RF modulation with data radios, and the DC-LIN for power line communication.

As of 2006, UARTs are commonly used with RS-232 for embedded systems communications. It is useful to communicate between microcontrollers and also with PCs. Many chips provide UART functionality in silicon, and low-cost chips exist to convert logic level signals (such as TTL voltages) to RS-232 level signals (for example, Maxim MAX232).



Male Rs 232 DB9



Male Rs 232 DB25

Table 5.9 Pin Description of 9-Pin Connector on a DTE Device (PC Connection)

Pin Number	*Function*
1	Carrier Detect (CD) (from DCE) Incoming signal from a modem
2	Received Data (RD) Incoming Data from a DCE
3	Transmitted Data (TD) Outgoing Data to a DCE
4	Data Terminal Ready (DTR) Outgoing handshaking signal.
5	Signal Ground Common reference voltage
6	Data Set Ready (DSR) Incoming handshaking signal
7	Request To Send (RTS) Outgoing flow control signal
8	Clear To Send (CTS) Incoming flow control signal
9	Ring Indicator (RI) (from DCE) Incoming signal from a modem

Table 5.10 Pin Description of 25-Pin Connector on a DTE Device (PC Connection)

Pin Number	*Function*
1	Protective Shield
2	Transmitted Data (TD) Outgoing Data (from a DTE to a DCE)
3	Received Data (RD) Incoming Data (from a DCE to a DTE)
4	Request To Send (RTS) Outgoing flow control signal controlled by DTE
5	Clear To Send (CTS) Incoming flow control signal controlled by DCE
6	Data Set Ready (DSR) Incoming handshaking signal controlled by DCE
7	Signal Ground Common reference voltage
8	Carrier Detect (CD) Incoming signal from a modem

Contd...

Contd...

9	Reserved for data set testing
10	Reserved for data set testing
11	Unassigned
12	Secondary Carrier Detect
13	Secondary Clear to send.
14	Secondary Transmit Data
15	Transmit Clock
16	Secondary Receive Data
17	Receive Clock
18	Unassigned
19	Secondary Request to Send
20	Data Terminal Ready (DTR)
21	Signal Quality Detector/Remote loopback
22	Ring Indicator (RI) Incoming signal from a modem.
23	Signal Rate Selector
24	DTE Transmit Signal Element Timing
25	Unassigned

5.9.1 Asynchronous Receives and Transmits

The word "asynchronous" indicates that UARTs recover character timing information from the data stream, using "start" and "stop" bits to indicate the framing of each character. In synchronous transmission, the clock data is recovered separately from the data stream and no start/stop bits are used. This improves the efficiency of transmission on suitable channels; more of the bits sent are data. An asynchronous transmission sends nothing over the interconnection when the transmitting device has nothing to send; but a synchronous interface must send "pad" characters to maintain synchronism between the receiver and transmitter. The usual filler is the ASCII "SYN" character. This may be done automatically by the transmitting device.

USART chips have both synchronous and asynchronous modes.

5.9.2 Universal Data Rates

The UART is a subset of "asynchronous receiver/transmitters" in that UARTs add the ability to receive and transmit serial data using different serial bit rates. (Receive and transmit rates are usually the same in most applications.) For example, better teletypes and computers with early MODEMs might use baud rates (nearly the same as bit rates) of roughly 110 or 300 bits per second for data telecommunication, while computers might use rates of 9600 to 38400 bits per second internally or locally; one UART would fit these applications universally.

Speeds for UARTs are in bits per second (bit/s or bps), although often incorrectly called the baud rate. Standard mechanical teletype rates are 45.5, 110, and 150 bit/s. Computers have used from 110 to 230,400 bit/s. Standard speeds are 110, 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 460800, 921600, 1382400, 1843200 and 2764800 bit/s.

5.9.3 Structure

A Universal Asynchronous Receiver/Transmitter includes:

1. A clock generator, usually a multiple of the bit rate to improve sampling in the middle of a bit period
2. Input and output shift registers
3. Transmit/receive control
4. Read/write control logic
5. Optional transmit/receive buffers
6. Optional parallel data bus buffer
7. FIFO (optional)

5.10 8251 PROGRAMMABLE/COMMUNICATION INTERFACE

The 8251 is an USART (Universal Synchronous Asynchronous Receiver Transmitter) for serial data communication. The block diagram is shown in Figure 5.30. As a peripheral device of a microcomputer system, the 8251 receives parallel data from the CPU and transmits serial data after conversion. This device also receives serial data from the outside and transmits parallel data to the CPU after conversion.

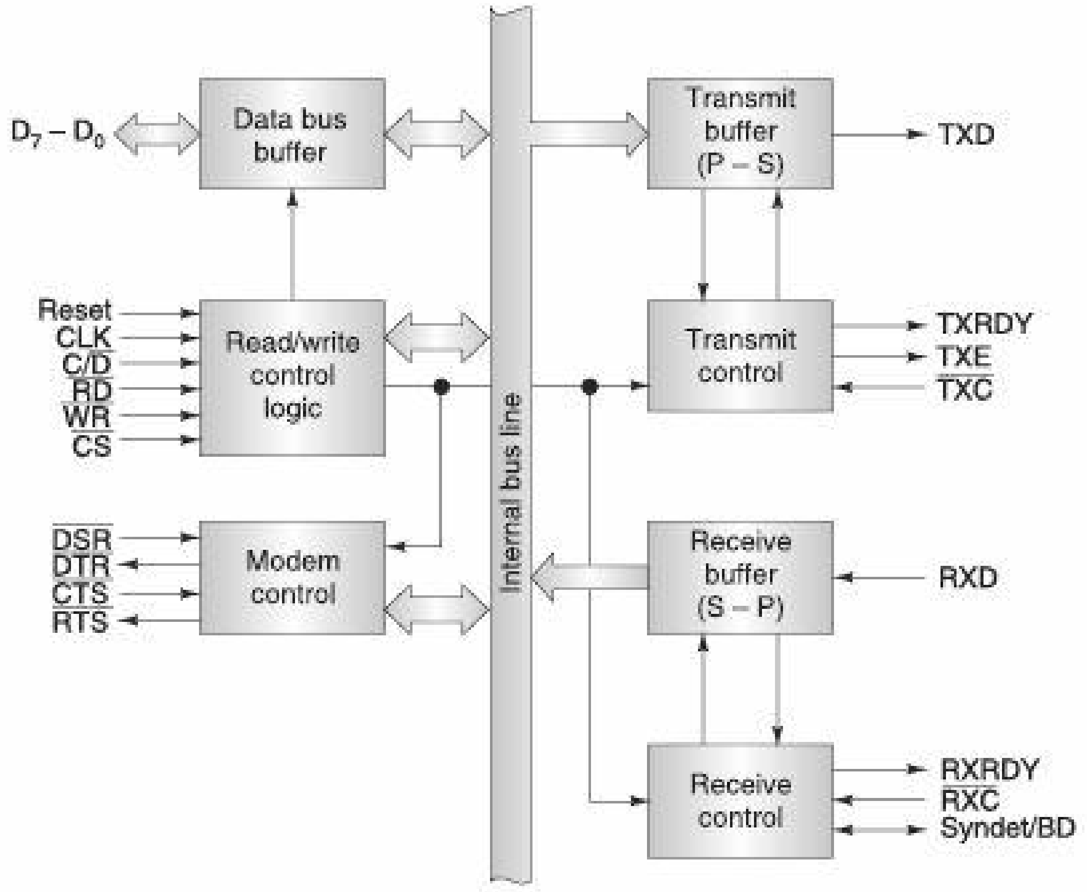


Fig. 5.30 Block Diagram of the 8251 USART.

The 8251 functional configuration is programmed by software. Operation between the 8251 and a CPU is executed by program control. Table 5.11 shows the operation between a CPU and the device.

Table 5.11 Operation between a CPU and 8251

$\overline{CS}$	$C/\overline{D}$	$\overline{RD}$	$\overline{WR}$	Operation
1	×	×	×	Data bus Tri-State
0	×	1	1	Data bus Tri-State
0	1	0	1	Status → CPU
0	1	1	0	Control Word ← CPU
0	0	0	1	Data → CPU
0	0	1	0	Data → CPU

5.10.1 Control Words

There are two types of control words:

(a) Mode instruction (setting of function)

(b) Command (setting of operation)

(a) Mode Instruction: Mode instruction is used for setting the function of 8251 USART. Mode instruction will be in "wait for write" at either internal reset or external reset. That is, the writing of a control word after resetting will be recognized as a "mode instruction". Items set by mode instruction are as follows:

- Synchronous/asynchronous mode
- Stop bit length (asynchronous mode)
- Character length
- Parity bit
- Baud rate factor (asynchronous mode)
- Internal/external synchronization (synchronous mode)
- Number of synchronous characters (synchronous mode)

The bit configuration of mode instruction is shown in Figures 5.31 and 5.32. In the case of synchronous mode, it is necessary to write one or two byte sync characters.

If sync characters were written, a function will be set because the writing of sync characters constitutes part of mode instruction.

(b) Command: Command is used for setting the operation of the 8251. It is possible to write a command whenever necessary after writing a mode instruction and sync characters. The command format is shown in Figure 5.33. The items to be set by command are as follows:

- Transmit Enable/Disable
- Receive Enable/Disable
- DTR, RTS Output of data
- Resetting of error flag
- Sending to break characters
- Internal resetting
- Hunt mode (synchronous mode)

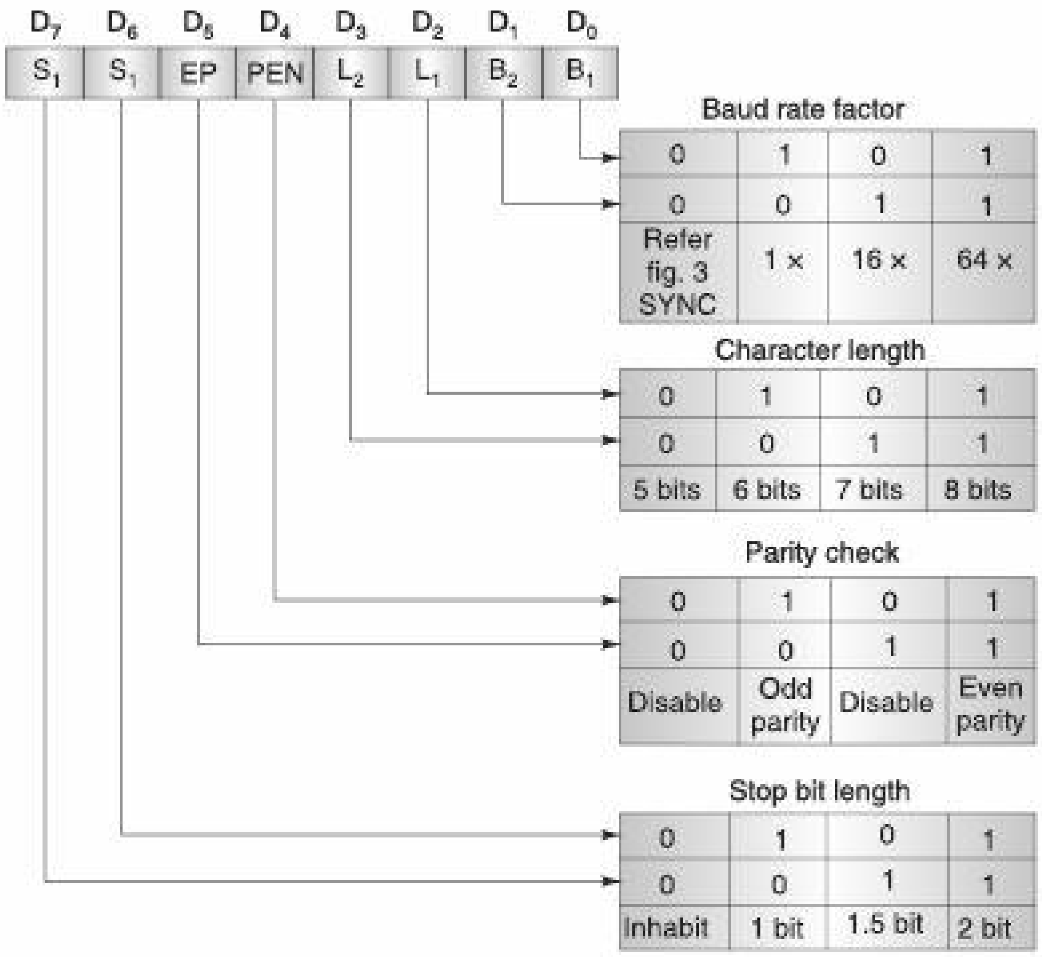


Fig. 5.31 Bit Configuration of Mode Instruction (Asynchronous).

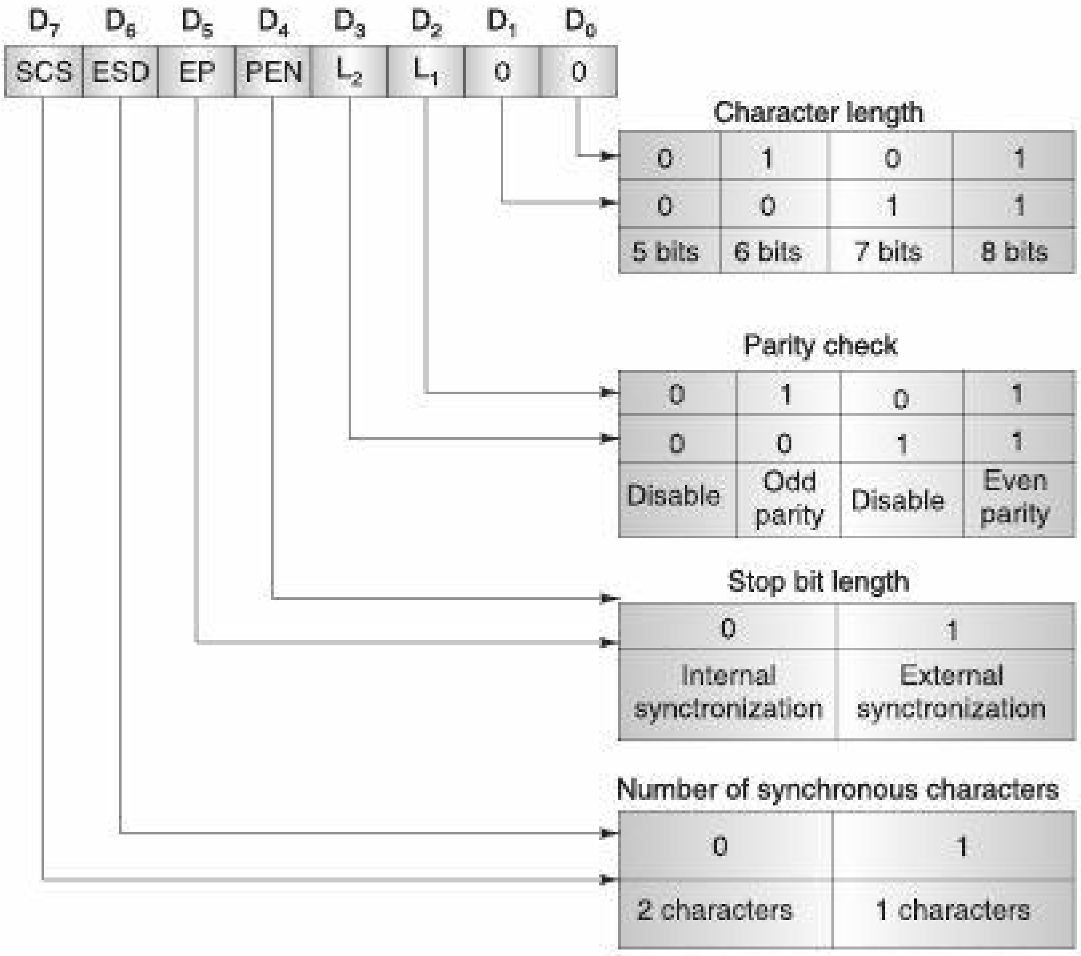


Fig. 5.32 Bit Configuration of Mode Instruction (Synchronous).

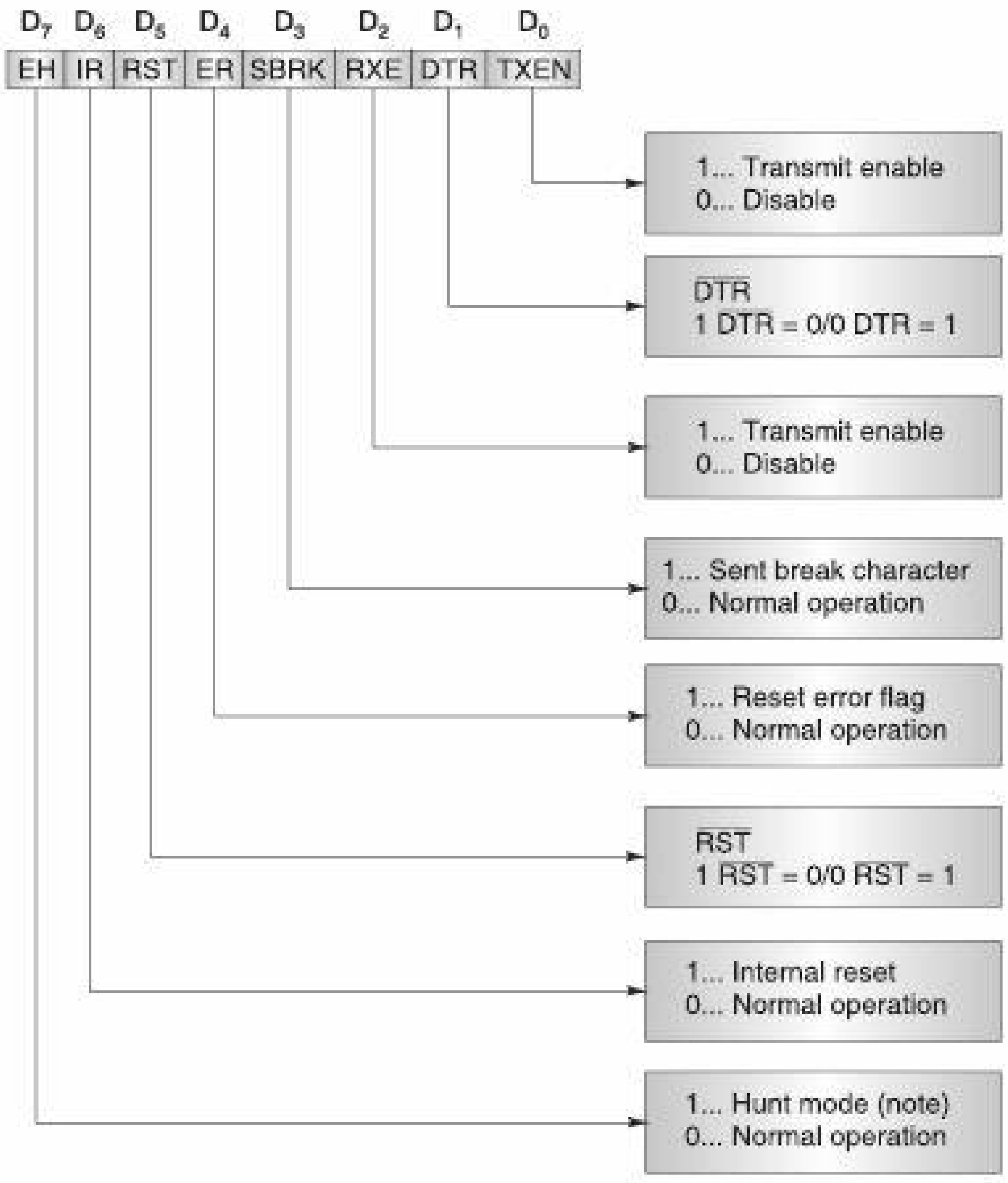


Fig. 5.33 Bit Configuration of Command.

5.10.2 Status Word

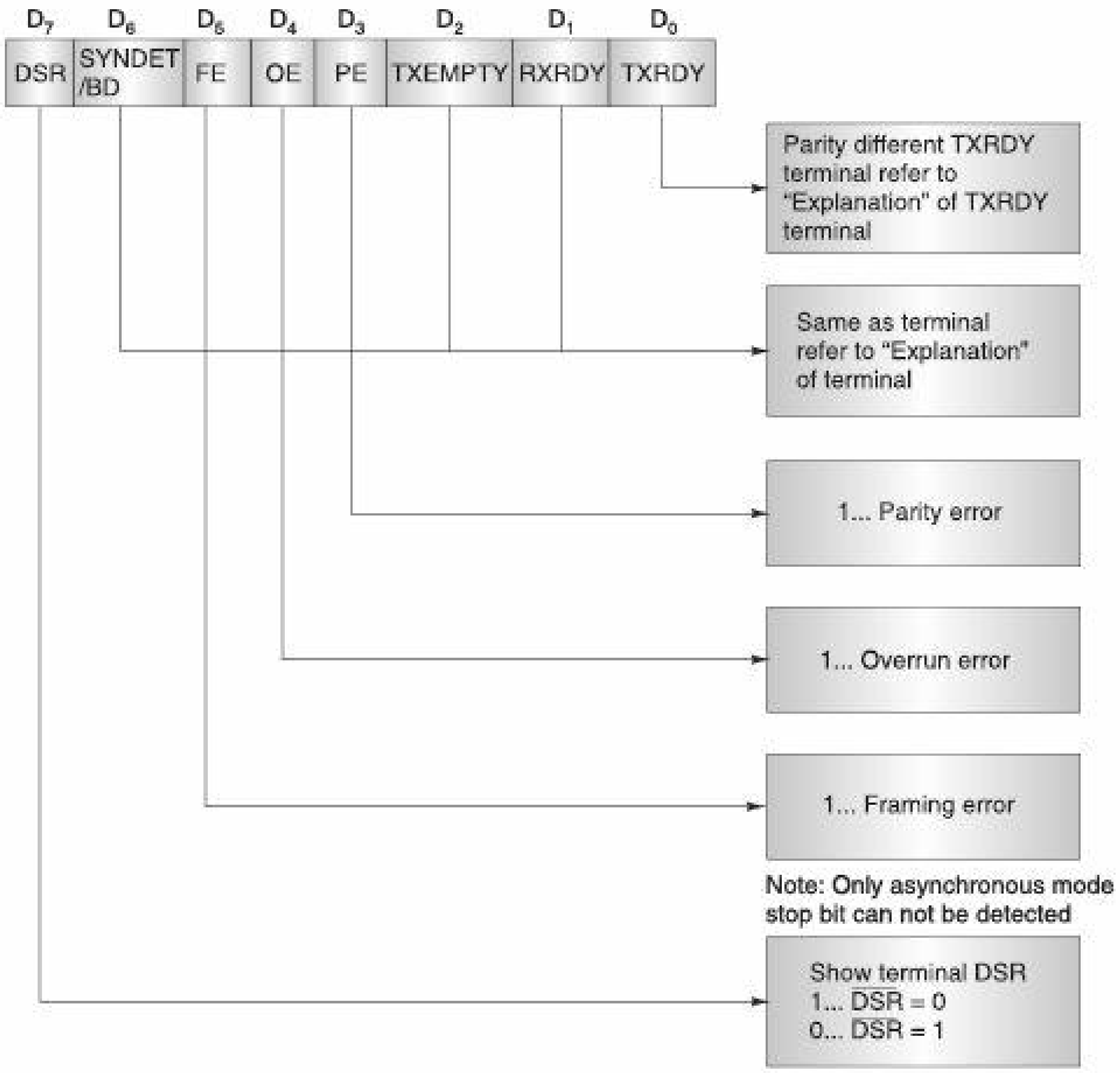
It is possible to see the internal status of the 8251 by reading a status word. The bit configuration of status word is shown in Figure 5.34.

5.10.3 Pin Description

D_0 to D_7 (I/O terminal): This is a bidirectional data bus which receives control words and transmits data from the CPU and sends status words and received data to CPU.

RESET (Input terminal): A “High” on this input forces the 8251 into “reset status.” The device waits for the writing of “mode instruction”. The min. reset width is six clock inputs during the operating status of CLK.

CLK (Input terminal): CLK signal is used to generate internal device timing. CLK signal is independent of RXC or TXC. However, the frequency of CLK must be greater than 30 times the RXC and TXC at Synchronous mode and Asynchronous “x1” mode, and must be greater than 5 times at Asynchronous “x16” and “x64” mode.



D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
DSR
SYNDET /BD
FE
OE
PE
TXEMPTY
RXRDY
TXRDY
Parity different TXRDY terminal refer to "Explanation" of TXRDY terminal
Same as terminal refer to "Explanation" of terminal
1... Parity error
1... Overrun error
1... Framing error
Note: Only asynchronous mode stop bit can not be detected
Show terminal DSR
1... $\overline{DSR}$ = 0
0... $\overline{DSR}$ = 1

Fig. 5.34 Bit Configuration of Status Word.

$\overline{\text{WR}}$ (Input terminal): This is the "active low" input terminal which receives a signal for writing transmit data and control words from the CPU into the 8251.

$\overline{\text{RD}}$ (Input terminal): This is the "active low" input terminal which receives a signal for reading receive data and status words from the 8251.

C/$\overline{\text{D}}$ (Input terminal): This is an input terminal which receives a signal for selecting data or command words and status words when the 8251 is accessed by the CPU. If C/$\overline{\text{D}}$ = low, data will be accessed. If C/$\overline{\text{D}}$ = high, command word or status word will be accessed.

CS (Input terminal): This is the "active low" input terminal which selects the 8251 at low level when the CPU accesses. Note: The device won't be in "standby status"; only setting CS = $\overline{\text{H}}$igh.

TXD (Output terminal): This is an output terminal for transmitting data from which serial-converted data is sent out. The device is in "mark status" (high level) after resetting or during a status when transmit is disabled. It is also possible to set the device in "break status" (low level) by a command.

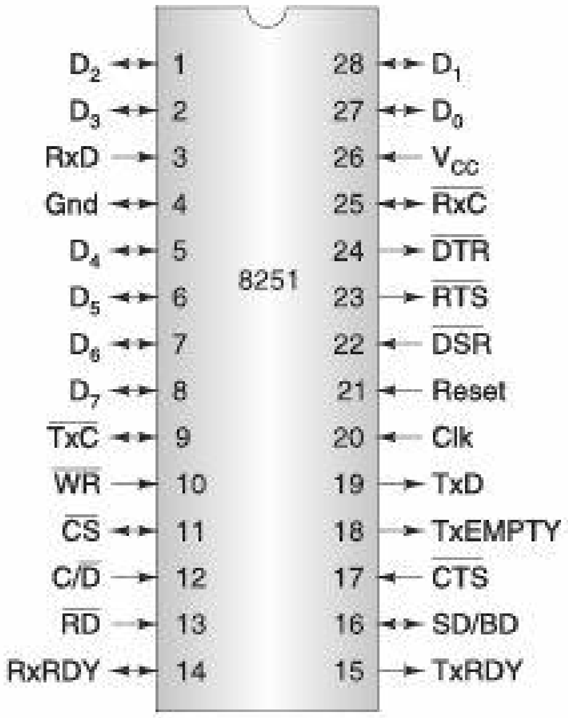


Fig. 5.35 Pin Diagram of 8251.

TXRDY (Output terminal): This is an output terminal which indicates that the 8251 is ready to accept a transmitted data character. But the terminal is always at low level if CTS = high or the device was set in "TX disable status" by a command. Note: TXRDY status word indicates that transmit data character is receivable, regardless of CTS or command. If the CPU writes a data character, TXRDY will be reset by the leading edge or WR signal.

TXEMPTY (Output terminal): This is an output terminal which indicates that the 8251 has transmitted all the characters and had no data character. In "synchronous mode," the terminal is at high level, if transmit data characters are no longer remaining and sync characters are automatically transmitted. If the CPU writes a data character, TXEMPTY will be reset by the leading edge of WR signal.

Note. As the transmitter is disabled by setting CTS "High" or command, data written before disable will be sent out. Then TXD and TXEMPTY will be "High". Even if data is written after disable, that data is not sent out and TXE will be "High". After the transmitter is enabled, is sent out (refer to Timing Chart of Transmitter Control and Flag Timing).

TXC (Input terminal): This is a clock input signal which determines the transfer speed of transmitted data. In "synchronous mode", the baud rate will be the same as the frequency of TXC. In "asynchronous mode", it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16 or 1/64 the TXC. The falling edge of TXC shifts the serial data out of the 8251.

RXD (Input terminal): This is a terminal which receives serial data.

RXRDY (Output terminal): This is a terminal which indicates that the 8251 contains a character that is ready to READ. If the CPU reads a data character, RXRDY will be reset by the leading edge of RD signal. Unless the CPU reads a data character before the next one is received completely, the preceding data will be lost. In such a case, an overrun error flag status word will be set.

RXC (Input terminal): This is a clock input signal which determines the transfer speed of received data. In "synchronous mode", the baud rate is the same as the frequency of RXC. In "asynchronous mode", it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16, 1/64 the RXC.

SYNDET/BD (Input or Output terminal): This is a terminal whose function changes according to mode. In "internal synchronous mode", this terminal is at high level, if sync characters are received and synchronized. If a status word is read, the terminal will be reset. In "external synchronous mode, this is an input terminal. A "High" on this input forces the 8251 to start receiving data characters.

In "asynchronous mode," this is an output terminal which generates "high-level" output upon the detection of a "break" character if receiver data contains a "low-level" space between the stop bits of two continuous characters. The terminal will be reset, if RXD is at high level. After Reset is active, the terminal will be output at low level.

DSR (Input terminal): This is an input port for MODEM interface. The input status of the terminal can be recognized by the CPU reading status words.

DTR (Output terminal): This is an output port for MODEM interface. It is possible to set the status of DTR by a command.

CTS (Input terminal): This is an input terminal for MODEM interface which is used for controlling a transmit circuit. The terminal controls data transmission if the device is set in "TX Enable" status by a command. Data is transmittable if the terminal is at low level.

RTS (Output terminal): This is an output port for MODEM interface. It is possible to set the status RTS by a command.

5.11 SPECIAL-PURPOSE INTERFACING DEVICES

Special-purpose interfacing devices have been developed to interface special I/O devices such as CRT, floppy disk system, dot matrix printer, keyboard, etc. Brief descriptions of few devices are given below:

(a) **Programmable CRT Controller (Intel 8275H):** The Intel 8275H is a programmable CRT controller. It is a single clip device. It is a 40 pin IC. package and requires + 5 V supply. Its function is to interface CRT raster scan display with the microcomputer. It receives characters from the system memory, displays these characters and refreshes the display. It also keeps the proper track of the display position on the screen.

(b) **Floppy Disk Controllers:** The functions of a floppy disk controller are to interface a floppy disk system to a microprocessor, disk drive selection, track and sector selection, head loading, to issue command to the drive system to perform read/write operation, data separation, serial-to-parallel and parallel-to-serial conversion of data, error checking, etc. The 82078, 82077AA and 82077SL are the latest floppy disk controllers. The 82077AA is a CHMOS single-chip floppy disk controller. It can drive upto 4 floppy disks. It can support both tape and 4 MB floppy drives. It has been provided with integrated tape drive support as well. It has the ability to interface directly perpendicular recording 4 MB drive. It can drive 5.25 inch as well as 3.5 inch floppy drives. The 82078SL is a superset of 82077AA with additional feature of power management which makes it suitable for portable computers. The 82078 is an enhanced floppy disk controller. It supports standard 5 V as well as low-voltage 3.3 V systems. It is compatible with 82077AA, 82077SL and earlier floppy disk controller 8272. It is available in 44 and 64-pin, i.e., it is faster and includes all the features of 82077AA and 82077SL.

(c) **Hard Disk Controller (or Winchester Disk Controller):** The function of a hard disk controller is to interface a hard disk system to the microprocessor. The Intel 82064 is a CHMOS Winchester disk controller. It contains an on-chip error detection and correction circuitry. It uses CHMOS III technology. It is a 40-pin IC and needs + 5 V supply. It is compatible with all Intel and most other microprocessors. It interfaces hard disk drives to the processor. It can control hard disk drives which employ Seagate Technology ST506/ ST412 interface. Its data transfer rate is 5 MB/second. It has 128, 256, 512 or 1024 bytes/sector. It has multiple sector transfer capability.

(d) **Dot Matrix Printer Controller (Intel 8295):** The Intel 8295 is a dot matrix printer controller. It is a 40-pin IC package. Its function is to interface LRC 7040 series dot matrix printers to microprocessor. It can also be used to interface other similar printers. It may be used in serial or parallel mode of operation with the microcomputer. Data are transferred using DMA, interrupts or polling in case of parallel mode of operation. There is an internal buffering of up to 40 characters. The 8295 contains a 7 × 7 matrix character generator to accommodate 64 ASCII characters.

(e) **Memory Controllers:** A dynamic RAM stores data on gate to source capacitor of a transistor. Due to leakage the charge on these stray capacitors leaks away after a few milliseconds. Therefore, dynamic RAMs must be periodically refreshed, usually every 2 milliseconds or less. For this purpose, a dynamic RAM requires refreshing circuitry. Intel has developed dynamic RAM controllers for refreshing dynamic RAMs; some of them will be described in this section. An error detection and correction IC will also be described.

Intel 8203: The 8203 is a 64 K dynamic RAM controller. It can directly address and drive up to 64 devices without external drivers. It generates all signals necessary to control 64 K and 16 K dynamic RAMs. It is compatible with 8085A, 8086 and 8088 family of microprocessors. It operates in two modes; one for 64 K DRAM and other for 16 K DRAM. It is a 40-pin IC and operates at +5 V supply.

Intel 8207: The 8207 is dual-port DRAM controller. It is capable of providing all necessary signals to control 16 K, 64 K and 256 K DRAMS. It can directly address up to 2 MB without external drivers. It can provide necessary signals to control an error detection and correction unit (Intel 8206), if it is in the system. It allows two different buses to access memory independently. It is compatible with 80286, 8086, 8088, 80186 and 80188 microprocessors. It is a 68-pin IC and requires + 5 V supply for its operation.

Intel 82C08: The 82C08 is a CHMOS dynamic RAM controller. It can support 64 K and 256 K DRAMs. It is compatible with 80286, 8086, 8088, 80186 and 80188 microprocessors. It can directly address up to 1 MB without external drivers. It consumes less power and can operate with less battery which solves the problem of non-impact printers. Manufacturers have designed printers which can be used as offline devices to produce additional copies of computer prepared output. The printer takes data from magnetic tapes and produces output.

5.12 PROGRAMMABLE INTERVAL TIMER

5.12.1 Intel 8253/8254

The Intel 8253 is a programmable counter/timer chip designed for use as an Intel microcomputer peripheral. It uses NMOS technology with a single +5 V supply and is packaged in a 24-pin plastic DIP. It is organized as 3 independent 16-bit counters, each with a counter rate up to 2 MHz. All modes of operation are software programmable. The 82C54 is pin compatible with the HMOS 8254, and is a superset of the 8253. Six programmable timer modes allow the 82C54/8253 to be used as an event counter, elapsed time indicator, programmable one-shot, and in many other applications.

5.12.1.1 Features

The timer has three counters, called channels. Each channel can be programmed to operate in one of six modes. Once programmed, the channels can perform their tasks independently. The timer is usually assigned to IRQ-0 (highest priority hardware interrupt request) because of the critical function it performs and because so many devices depend on it.

5.12.1.2 Block Diagram and Pin Configuration

Figure 5.36 (a) shows the block diagram and pin configuration of the 8253 and a general definition of the lines is also defined below:

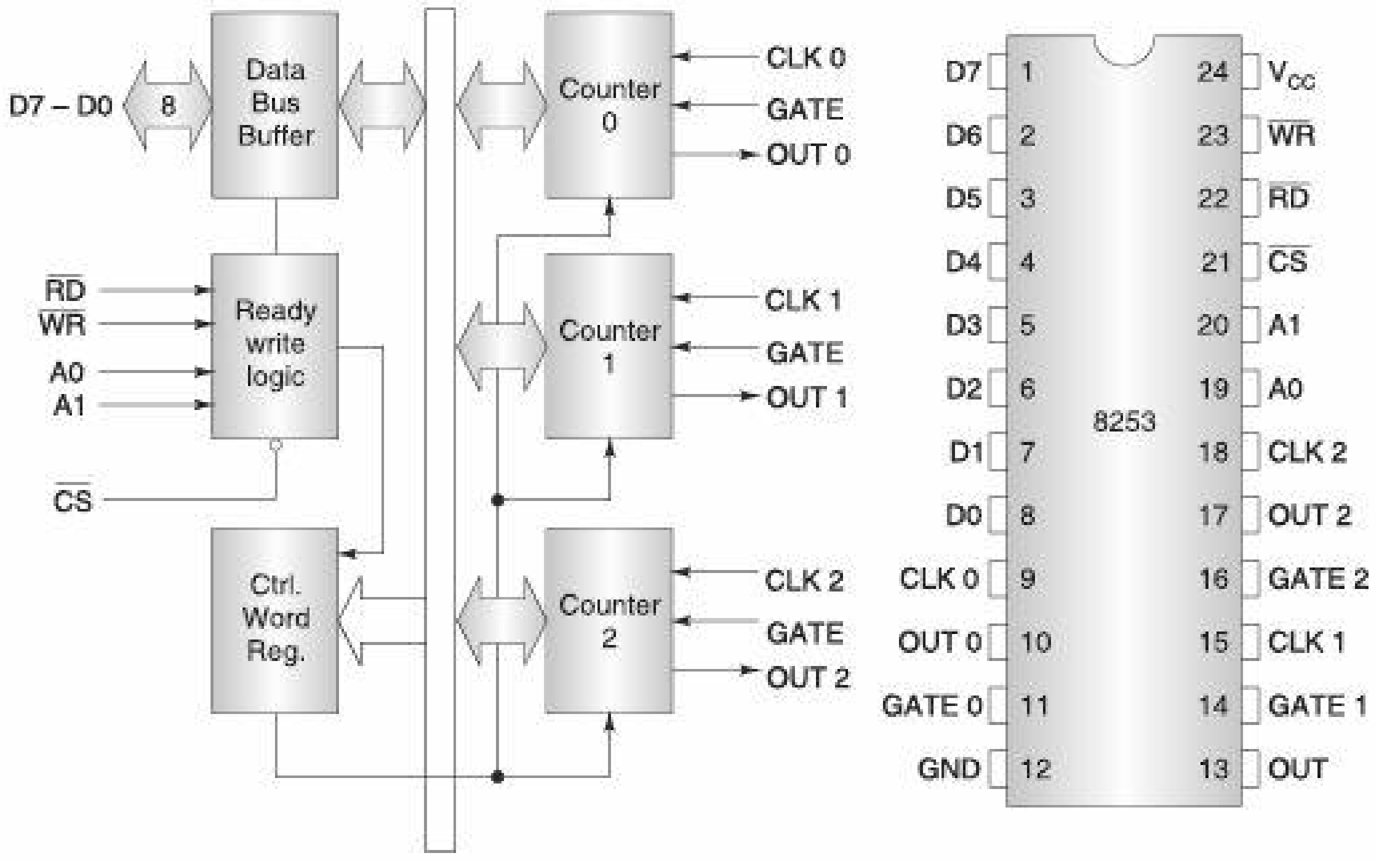


Fig. 5.36(a) Block Diagram of an 8253 Programmable Interval Timer and Pin Description.

5.12.1.2.1 Counter

There are 3 counters (or timers), which are labeled as Counter 0, Counter 1 and Counter 2. Each counter has 2 input pins CLK (clock input) and GATE and 1 pin, OUT, for data output. The 3 counters are 16-bit down counters independent of each other, and can be easily read by the CPU. The timer has three independent, programmable counters and they are all identical.

5.12.1.2.2 Data bus buffer

The first counter (selected by setting $A_1 = A_0 = 0$, see Control Word Register below) helps to generate an 18.2 Hz clock signal. The second counter ($A_1 = 0$, $A_0 = 1$) assists in generating timing, which will be used to refresh the DRAM memory. The last counter contains the logic to buffer the data bus to/from the microprocessor, and to the internal registers.

5.12.1.2.3 Read/write logic

It controls the reading and the writing of the counter registers with following signals:

$\overline{\text{RD}}$ (read signal): It is active low signal used in co-ordination with A_0, A_1 to send data from the counter to the data lines $D_0 - D_7$.

$\overline{\text{WR}}$ (write signal): It is active low signal used in co-ordination with A_0, A_1 to load counter to initialize counter.

$\overline{\text{CS}}$ (chip select signal): It is a chip select signal. The LOW status of this signal enables communication between the CPU and 8253.

A_0, A_1 (address lines): These address lines are used to distinguished different ports of 8254.

5.12.1.2.4 Control word register

This register contains the programmed information which will be sent to the device from the microprocessor. To initialize the counters, the microprocessor must write a control word (CW) in this register. This can be done by setting proper values for the pins of the Read/Write Logic block and then by sending the control word to the Data/Bus Buffer block. Each counter in the block diagram has 3 logical lines connected to it. Two of these lines, clock and gate, are inputs. The third, labeled OUT is an output. The function of these lines changes and depends on how the device is initialized or programmed.

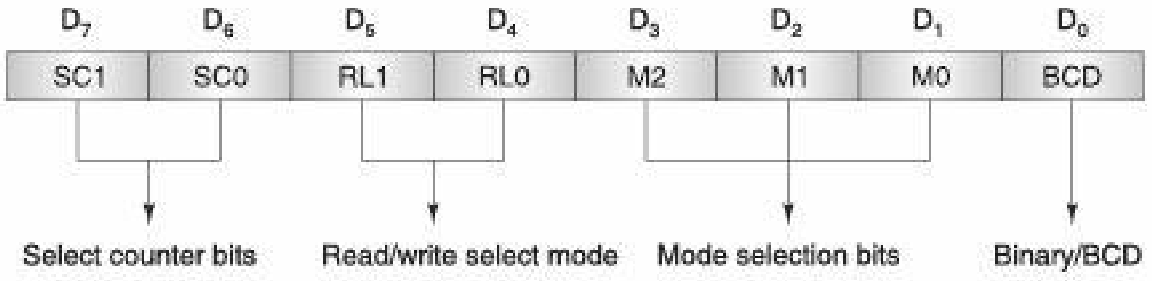


Fig. 5.36(b) Control Word Format of Intel 8253/8254.

Bit D_7, D_6: Bits D_7 and D_6 are labeled SC_1 and SC_0. These bits select the counter to be programmed; it is necessary to define, using the control bits D_7 and D_6, which counter is being set up. Once a counter is set up, it will remain that way until it is changed by another control word.

Table 5.12 Selection of Counter

SC_1	SC_0	Select Counter
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Illegal value

Bit D_5, D_4: Bits D_5 and D_4 (RL_1/RL_0) of the control word shown above are defined as the read/load mode for the register that is selected by bits D_7 and D_6. Bits D_5 and D_4 define how the particular counter is to have data read from or written to it by the CPU.

Table 5.13 Read/Load Mode

RL_1	RL_0	Description
0	0	Count value is latched. This means that the selected counter has its contents transferred into a temporary latch, which can then be read by the CPU.
0	1	Read/load least-significant byte only.
1	0	Read/load most-significant byte only.
1	1	Read/load least-significant byte first, then most significant byte.

Bit D_3, D_2 and D_1 (Mode selection bits): The D_3, D_2, and D_1 bits of the control word set the operating mode of the timer. There are 6 modes in total; for modes 2 and 3, the D_3 bit is ignored, so the missing modes 6 and 7 are aliases for modes 2 and 3 as shown in Table 5.14. Notice that, for modes 0, 2, 3 and 4, GATE must be set to HIGH to enable counting. For modes 1 and 5, the rising edge of GATE starts the count. For details on each mode, see the reference links.

Table 5.14 Mode Selection

M_2	M_1	M_0	Mode Selected
0	0	0	Mode 0 (Interrupt on terminal count)
0	0	1	Mode 1 (Programmable one shot)
X	1	0	Mode 2 (Rate generator)
X	1	1	Mode 3 (Square wave generator)
1	0	0	Mode 4 (Software triggered strobe)
1	0	1	Mode 5 (Hardware triggered strobe)

Bit D_0: Bit D_0 of the control register determines how the register will count. The maximum values for the count in each count mode are 10^4 (10,000 decimal) in BCD, and 2^{16} (65,536 decimal) in binary.

Table 5.15 Binary/BCD Count

D_0	Count down in
0	Binary
1	BCD

5.12.1.3 Clock

This is the clock input for the counter. The counter is 16 bits. The maximum clock frequency is 1/380 nanoseconds or 2.6 megahertz. The minimum clock frequency is DC or static operation.

Out: This single output line is the signal that is the final programmed output of the device. Actual operation of the outline depends on how the device has been programmed.

Gate: This input can act as a gate for the clock input line, or it can act as a start pulse, depending on the programmed mode of the counter.

Each mode of operation for the counter has a different use for the GATE input pin. Table 5.16 shows the different uses of the 8253 gate input pin.

Table 5.16 Gate Input Pin for Different Modes

Signal Status	Low or going low	Rising	High
Mode 0	Disable counting		Enables counting
Mode 1	—	1. Initiates counting 2. Reset output after next clock	—
Mode 2	1. Disable counting 2. Sets output immediately high	1. Reload counter 2. Initiates counting	Enables counting
Mode 3	1. Disable counting 2. Sets output immediately high	Initiates counting	Enables counting
Mode 4	Disable counting	—	Enables counting
Mode 5	—	Initiates counting	—

5.12.1.4 Internal 8253 Registers

Here is a list of the internal 8253 registers that will program the internal counters of the 8253, as shown in Table 5.17.

Table 5.17 Function of the Counters

Counter Type	$\overline{RD}$	$\overline{WR}$	A_1	A_0	Function
Counter 0	1	0	0	0	Load counter 0
	0	1	0	0	Read counter 0
Counter 1	1	0	0	1	Load counter 1
	0	1	0	1	Read counter 1
Counter 2	1	0	1	0	Load counter 2
	0	1	1	0	Read counter 2
Control Word/ Mode Word	1	0	1	1	Write mode word
	0	1	1	1	No operation

5.12.1.4.1 Counter (0-2)

Each counter is identical and each consists of a 16-bit, pre-settable, down counter. Each is fully independent and can be easily read by the CPU. When the counter is read, the data within the counter will not be disturbed. This allows the system or your own program to monitor the counter's value at any time, without disrupting the overall function of the 8253.

5.12.1.4.2 Control word register

This internal register is used to write information to, prior to using the device. This register is addressed when A_0 and A_1 inputs are logical 1's. The data in the register controls the operation mode and the selection of either binary or BCD (binary coded decimal) counting format. The register can only be written to. You can't read information from the register. All the operating modes for the counters are selected by writing bytes to the control register. This is the control word format.

5.12.1.5 Modes of 8253

The following text describes all possible modes. The modes used in the MZ-700 and set by the monitor's startup are mode 0, mode 2, and mode 3.

5.12.1.5.1 Mode 0

Interrupt on Terminal Count: In this mode, the counter will start counting from the initial COUNT value loaded into it, down to 0. Counting rate is equal to the input clock frequency. The OUT pin is set low after the Control Word is written, and counting starts one clock cycle after the COUNT programmed.

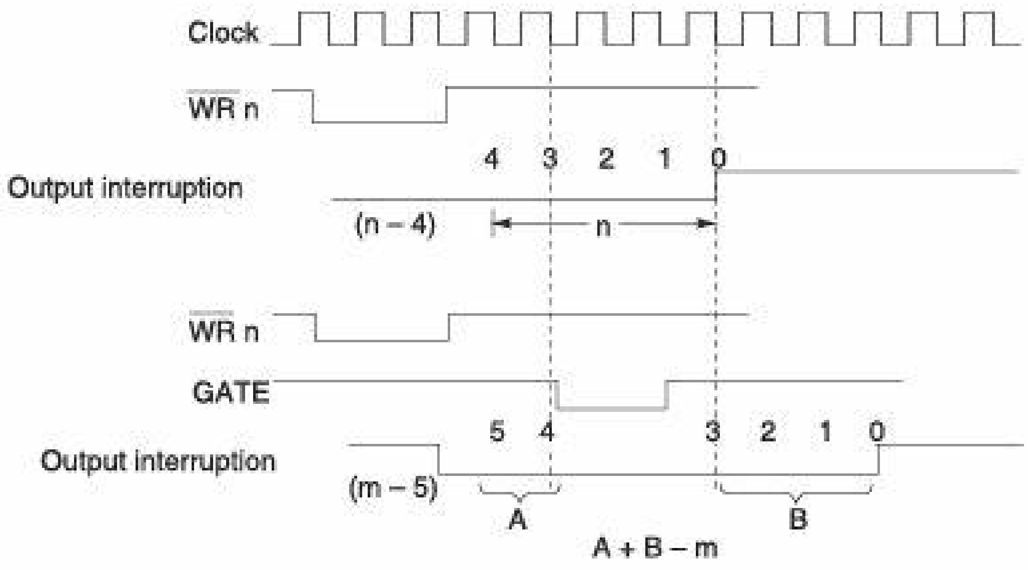


Fig. 5.37(a) Mode 0 Interrupt on Terminal Count.

OUT remains low until the counter reaches 0, at which point OUT will be set high until the counter is reloaded or the Control Word is written. Once the counter starts counting down, the GATE input can disable the internal counting by setting the GATE to a logical 0.

5.12.1.5.2 Mode 1

Programmable One-Shot: This is similar to mode 0, but counting is started by a rising edge on the GATE input instead of immediately after programming. The GATE input is ignored while counting. The output is set high as soon as the Control Word is written. After COUNT is written, the device will wait until the rising edge of the GATE input. One clock cycle after this rising edge is detected; OUT will become and remain low until the counter reaches 0. OUT will then go high, waiting for the next trigger. The one-shot is triggered on the rising edge of the GATE input. If the trigger occurs during the pulse output, the 8253 will be triggered again.

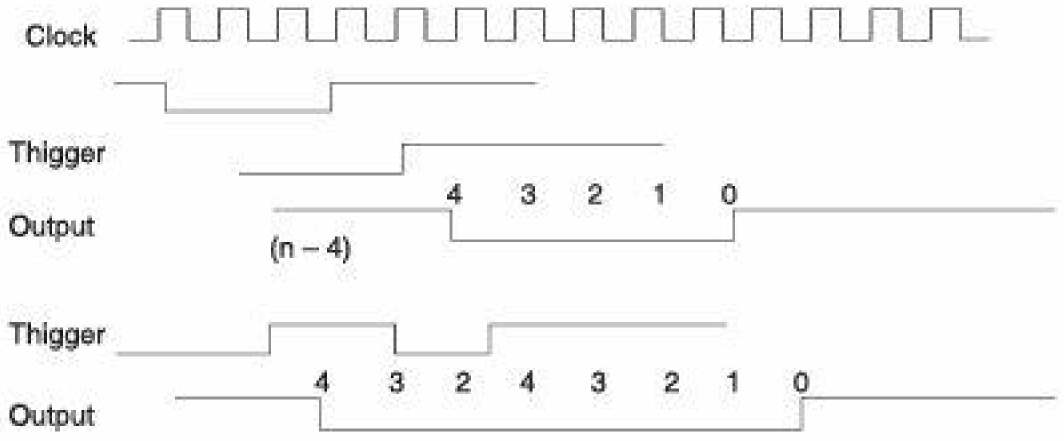


Fig. 5.37(b) Mode 1-Programmable One-Shot.

5.12.1.5.3 Mode 2

Rate Generator: In this mode, the device acts as a divide-by-n counter, which is commonly used to generate a real-time clock interrupt. Like other modes, counting process will start the next clock cycle after COUNT is sent. OUT will then remain high until the counter reaches 1, and will go low for one clock pulse. OUT will then go high again, and the whole process repeats itself. The time between the high pulses depends on the preset count in the counter's register, and is calculated using the following formula:

Value to be loaded into counter = Input frequency/Output frequency

Note that the values in the COUNT register range from n to 1; the register never reaches zero.

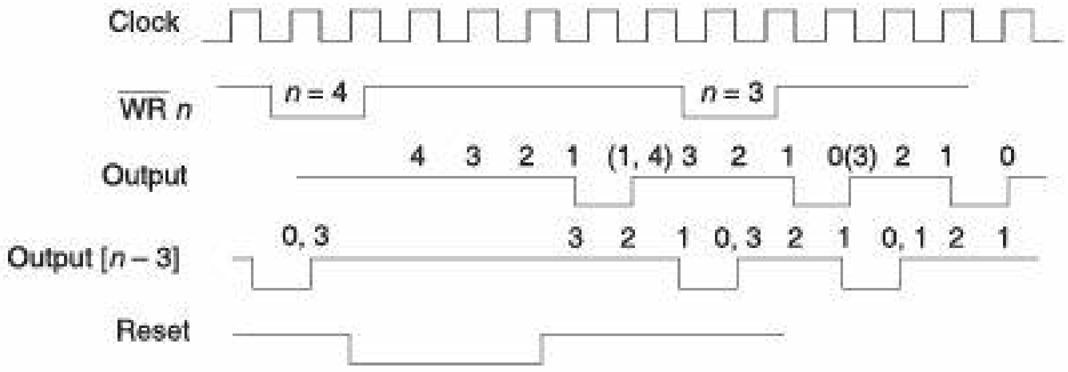


Fig. 5.37(c) Mode 2-Rate Generator.

5.12.1.5.4 Mode 3

Square Wave Generator: This mode is similar to mode 2. However, the duration of the high and low clock pulses of the output will be different. Suppose n is the number loaded into the counter (the COUNT message), a square wave is generated.

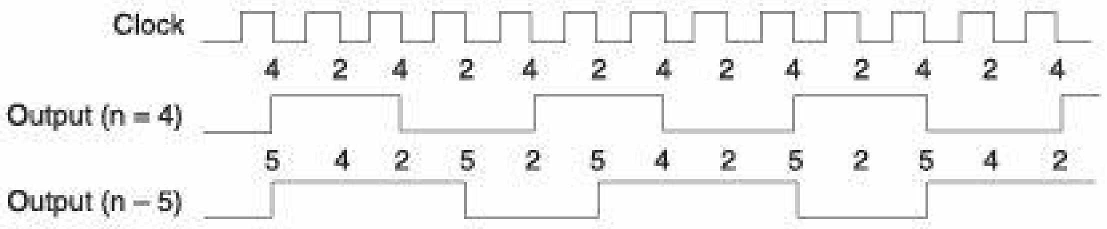


Fig. 5.37(d) Mode 3-Square Wave Generator.

5.12.1.5.5 Mode 4

Software Triggered Strobe: After Control Word and COUNT is loaded, the output will remain high until the counter reaches zero. The counter will then generate a low pulse for 1 clock cycle (a strobe)—after that the output will become high again. A low in the gate disables count and a high enables it. In this mode, the programmer can set up the counter to give an output timeout starting when the register is loaded. On the terminal count, when the counter is equal to 0, the output will go to a logical 0 for one clock period and then returns to a logical 1. First the mode is set; the output will be a logical 1.

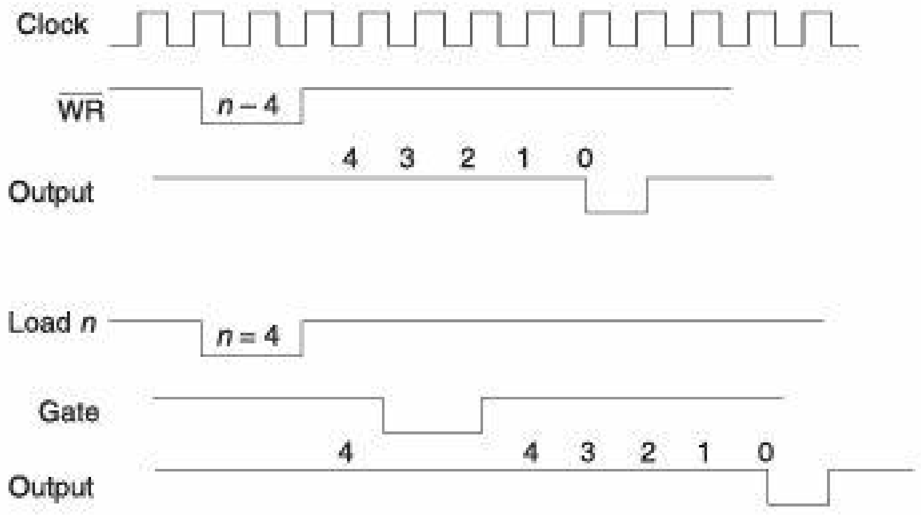


Fig. 5.37(e) Mode 4-Software Triggered Strobe.

5.12.1.5.6 Mode 5

Hardware Triggered Strobe: This mode is similar to mode 4. Count is enabled with the rising edge of gate. However, the counting process is triggered by the GATE input. After receiving the Control Word and COUNT, the output will be set high.

Once the device detects a rising edge on the GATE input, it will start counting. When the counter reaches 0, the output will go low for one clock cycle—after that it will become high again, to repeat the cycle on the next rising edge of GATE.

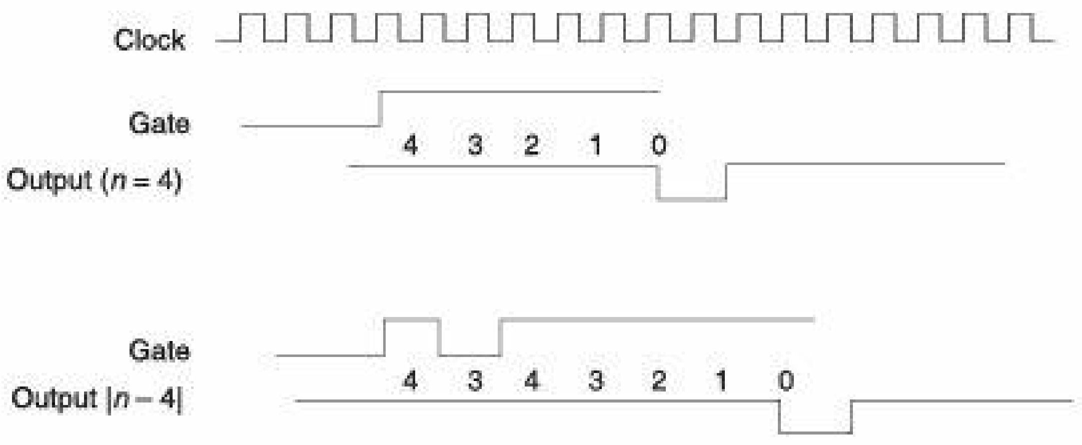


Fig. 5.37(f) Mode 5-Hardware Triggered Strobe.

Program: Write a program to generate a pulse of 20 μs from counter 2 continuously.

Solution: To generate a pulse every 20 μs from counter 2, it should be initialized in Mode 2 and gate signal should be high.

Control Word: It is set as:

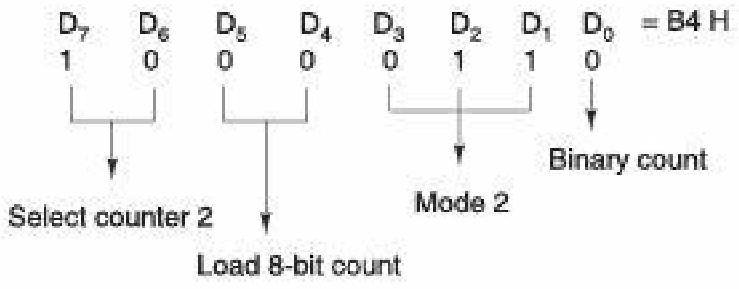


Fig. 5.38

Count: The count value is decremented after every clock period and in the last count, the counter generates a pulse that is equivalent to clock period of the timer.

The clock frequency of 8254 is 2 MHz and pulse should be generated every 20 μs. So the value of count is calculated as:

Program:

```
MVI A, B4H      ; Control word for Mode 2 and Counter 2
OUT 83H         ; write in 8254 control register
MVI A, 40H      ; Low-order byte of the count value
OUT 82H         ; Load Counter 2 with low-order byte
HLT             ; End of the program
```

5.12.2 Read-Back Command (for 8254)

This command allows the user to check the count value, programmed mode, and current states of the OUT pin and Null Count flag of the selected counter(s). The command is written into the Control Word Register and has the format shown in Figure 5.39. The command applies to the counters selected by setting their corresponding bits D_3, D_2, D_1 = 1. The read-back command may be used to latch multiple counter output latches (OL) by setting the COUNT bit $D_5 = 0$ and selecting the desired counter (s).

A0, A1 = 11 $\overline{CS} = 0$ $\overline{RD} = 1$ $\overline{WR} = 0$

D_7	D_6	D_5	D_6	D_3	D_2	D_1	D_0
1	1	$\overline{\text{Count}}$	$\overline{\text{Status}}$	CNT 2	CNT 1	CNT 0	0

D_5: 0 = Latch count of selected counter(s)
D_4: 0 = Latch status of selected counter(s)
D_3: 1 = Select counter 2
D_2: 1 = Select counter 1
D_1: 1 = Select counter 0
D_0: = Reserved for future expansion; must be 0

Fig. 5.39 Read Back Command.

This single command is functionally equivalent to several counter latch commands, one for each counter latched. Each counter's latched count is held until it is read (or the counter is reprogrammed). The counter is automatically unlatched when read, but other counters remain latched until they are read. If multiple count read-back commands are issued to the same counter without reading the count, all but the first are ignored, i.e., the count which will be read is the count at the time the first read-back command was issued. The read-back command may also be used to latch status information of selected counter(s) by setting STATUS bit $D_4 = 0$. Status must be latched to be read; status of a counter is accessed by a read from that counter. The counter status format is shown in Figure 5.40. Bits D_5 through D_0 contain the counter's programmed Mode exactly as written in the last Mode Control Word. OUTPUT bit D_7 contains the current state of the OUT pin. This allows the user to monitor the counter's output via software, possibly eliminating some hardware from a system. NULL COUNT bit D_6 indicates when the last count written to the counter register (CR) has been loaded into the counting element (CE). The exact time this happens depends on the Mode of the counter and is described in the Mode Dentitions, but until the count is loaded into the counting element (CE), it can't be read from the counter.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Output	Null count	RW1	RW0	M2	M1	M0	BCD

D_7	1 = Out pin is 1 0 = Out pin is 0
D_6	1 = Null count 0 = Count available for reading
$D_5 - D_0$	Counter programmed mode

Fig. 5.40 Status Byte.

If the count is latched or read before this time, the count values will not reflect the new count just written. The operation of Null Count is shown in Figure 5.41. If multiple status latch operations of the counter(s) are performed without reading the status, all but the first are ignored, i.e., the status that will be read is the status of the counter at the time the first status read back command was issued.

This action	Causes
A. Write to the control word register, [1]	Null count = 1
B. Wirte to the count register (CR); [2]	Null count = 1
C. New count is loaded into CE (CR ® CE);	Null count = 0

Note:

1. Only the counter specified by the control word will have its null count set to 1. Null count bits of other counters are unaffected.
2. If the counter is programmed for two-byte counts (least significant byte then most significant byte) Null count goes to 1 when the second byte is written.

Fig. 5.41 Null Count Operation.

Both count and status of the selected counter(s) may be latched simultaneously by setting both COUNT and STATUS bits D_5, $D_4 = 0$. This is functionally the same as issuing two separate read-back commands at once, and the above discussions apply here also.

Specifically, if multiple count and/or status read-back commands are issued to the same counter(s) without any intervening reads, all but the first are ignored. This is illustrated in Table 18. If both count and status of a counter are latched, the first read operation of that counter will return latched status, regardless of which was latched first. The next one or two reads (depending on whether the counter is programmed for one or two type counts) return latched count. Subsequent reads return unlatched count.

Table 5.18 Read-Back Command Example

Command								*Description*	*Result*
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
1	1	0	0	0	0	1	0	Read back count and status of Counter 0	Count and Status latched for Counter 0
1	1	1	0	0	1	0	0	Read back status of Counter 1	Status latched for Counter 1
1	1	1	0	1	1	0	0	Read back status of Counter 2,1	Status latched for Counter 2, but not Counter 1
1	1	0	1	1	0	0	0	Read back count of Counter 2	Count latched for Counter 2
1	1	0	0	0	1	0	0	Read back count and status of Counter 1	Count latched for Counter 1, but not Status
1	1	1	0	0	0	1	0	Read back status of Counter 1	Command ignored, status already latched for Counter 1

Table 5.19 Read/Write Operation Summary

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1	A_0	*Operation*
0	1	0	0	0	Write into Counter 0
0	1	0	0	1	Write into Counter 1
0	1	0	1	0	Write into Counter 2
0	1	0	1	1	Write Counter Word
0	0	1	0	0	Read from Counter 0
0	0	1	0	1	Read from Counter 1
0	0	1	1	0	Read from Counter 2
0	0	1	1	1	No Operation (tri-State)
1	X	X	X	X	No Operation (tri-State)
0	1	1	X	X	No Operation tri-State)

5.13 8259 A—PRIORITY INTERRUPT CONTROLLER

The processor 8085 had five hardware interrupt pins. Out of these five interrupt pins, four pins were allotted fixed vector address but the pin INTR was not allotted any vector address, rather than external device was supposed to hand over the type of interrupt, i.e., Type$_0$ to Type$_7$ (for RST$_0$ to RST$_7$) to the CPU. The processor then gets this type and derives the interrupt vector address from that. Consider an application, where more number of I/O devices are connected with CPU, desire to transfer data using interrupt driven mode. In these cases, more interrupt pins are required than available in a CPU. Moreover, in these multiple interrupt systems, the processor will have to take care of priorities for the interrupt, simultaneously occurring at the interrupt request pins. The 8086 has only two interrupt inputs, NMI and INTR. So with two pins, large number of interrupts are not possible.

To overcome all these difficulties, we require a programmable interrupt controller which is able to handle a number of interrupts at a time. This relieves the processor from this entire task. The 8259 was designed to operate only with 8 bits microprocessors like 8085. A modified version, 8259A is compatible with 8-bit as well as 16-bit processors (8086). The 8259A can perform the following tasks:

1. 8259A can manage eight interrupts that are specified in instructions written into 8259A control registers.
2. It can vector an interrupt anywhere in the entire memory space. However, all eight interrupts are spaced at interval of four or eight memory locations.
3. To resolve the eight levels interrupt priorities, it can operate in any one of these three modes:
   - Fully Nested Mode
   - Automatic Rotation Mode
   - Specific Rotation Mode

   8259A can operate on some other modes also, that are explained later (in operating modes of 8259A).

4. It can be used for 64 priority levels by cascading 8259As.
5. It can read the status of pending interrupts, in-service interrupt and masked interrupts.
6. It can also mask each interrupt request individually.
7. It can accept the both level-triggered or edge triggered interrupt requests.

5.13.1 Block Diagram of 8259 A

The block diagram of 8259A is shown in Figure 5.41. It shows all the elements of a programmable device, and some additional blocks. The functions of these blocks are explained below:

5.13.1.1 Interrupt Request Register (IRR)

The interrupts at IRQ (Interrupt Request) input lines are handled by Interrupt Request internally. The IRR contains eight input lines $IR_0 – IR_7$ for interrupts. IRR stores all the interrupt requests in it in order to serve them one by one on priority basis.

5.13.1.2 In-Service register (ISR)

This stores all the interrupt requests those are being served, i.e., ISR keeps a track of the requests being served.

5.13.1.3 Priority Resolver

This unit determines the priorities of the interrupt requests appearing simultaneously. The highest priority is selected and stored into the corresponding bit of ISR during INTA pulse. The IR0 has the highest priority while the IR_7 has the lowest one, normally in fixed priority mode. The priorities, however, may be altered by programming the 8259A in rotating priority mode.

5.13.1.4 Interrupt Mask Register (IMR)

This register stores the bits required to mask the interrupt inputs. IMR operates on IRR at the direction of the Priority resolver.

5.13.1.5 Interrupt Control Logic

This block manages the interrupt and interrupt acknowledge signals to be sent to the CPU for serving one of the eight interrupt requests. This also accepts the interrupt acknowledge (INTA) signal from CPU that causes the 8259A to release vector address onto the data bus.

5.13.1.6 Data Bus Buffer

This tri-state bidirectional buffer interfaces internal 8259A bus to the microprocessor system data bus. Control words, status and vector information pass through data buffer during read or write operations.

5.13.1.7 Read/Write Control Logic

This circuit accepts and decodes commands from the microprocessor. This block also allows the status of the 8259A to be transferred onto the data bus.

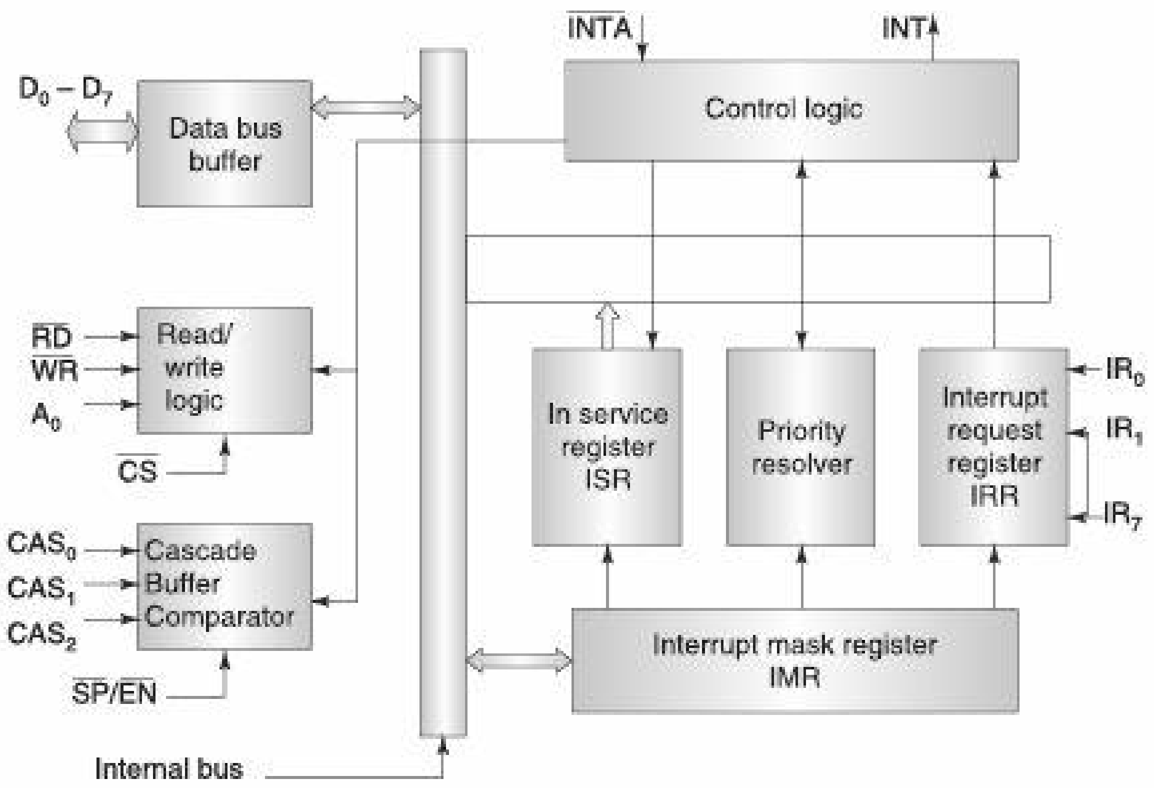


Fig. 5.42 Block Diagram of 8259 A.

5.13.1.8 Cascade Buffer/Comparator

This block expands the number of interrupt levels by cascading two or more 8259 As. It stores and compares the ID's all the 8259A used in system. The three I/O pins CAS_{0-2} pins are used as master/slave. The CAS_{0-2} pins are output pins when the 8259A is used as a master. The same pins act as inputs when the 8259A is in slave mode. The 8259A in master mode sends the ID of the interrupting slave device on these lines. The slave thus selected, will send its preprogrammed vector address on the data bus during the next INTA pulse.

5.13.2 Pin Diagram of 8259A

$\overline{\text{CS}}$: This is an active-low chip select signal for enabling $\overline{\text{RD}}$ and $\overline{\text{WR}}$ operations of 8259A. INTA function is independent of CS.

$\overline{\text{WR}}$: This pin is an active-low write enable input to 8259A. This enables it to accept command words from CPU.

$\overline{\text{RD}}$: This is an active-low read enable input to 8259A. A low on this line enables 8259A to release status onto the data bus of CPU.

D_0-D_7: These pins form a bidirectional data bus that carries 8-bit data either to control word or from status word registers. This also carries interrupt vector information.

CAS_0–CAS_2 Cascade Lines: A signal 8259A provides eight vectored interrupts. If more interrupts are required, the 8259A is used in cascade mode. In cascade mode, a master 8259A along with eight slaves 8259A can provide up to 64 vectored interrupt lines. These three lines act as select lines for addressing the slave 8259A.

$\overline{\text{PS}}/\overline{\text{EN}}$: This pin is a dual purpose pin. When the chip is used in buffered mode, it can be used as buffered enable to control buffer transreceivers. If this is not used in buffered mode, then the pin is used as input to designate whether the chip is used as a master ($\overline{\text{SP}}$ = 1) or slave (SP = 0).

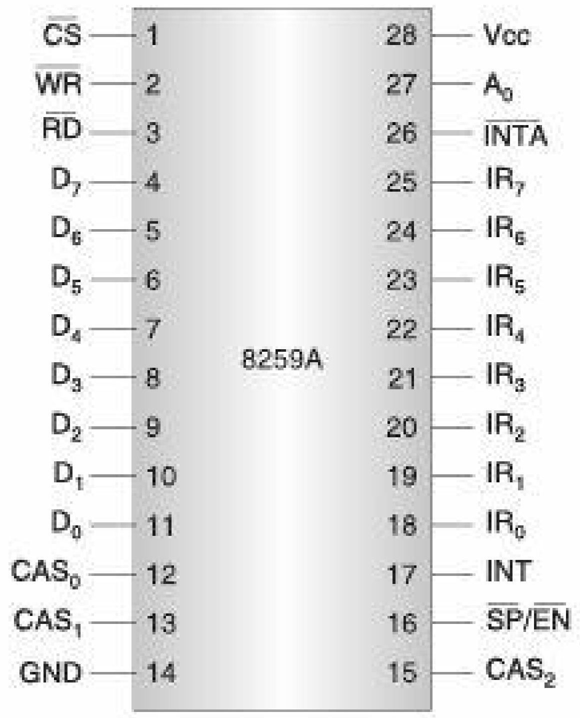


Fig. 5.43 8259 Pin Diagram.

INT: This pin goes high whenever a valid interrupt request is asserted. This is used to interrupt the CPU and is connected to the interrupt input of CPU.

IR_0–IR_7 (Interrupt requests): These pins act as inputs to accept interrupt request to the CPU. In edge triggered mode, an interrupt service is requested by raising an IR pin from a low to a high state and holding it high until it is acknowledged, and just by latching it to high level, if used in level triggered mode.

$\overline{INTA}$ (Interrupt acknowledge): This pin is an input used to strobe-in 8259A interrupt vector data onto the data bus. In conjunction with $\overline{CS}$, $\overline{WR}$ and $\overline{RD}$ pins, this selects the different operations like, writing command word, reading status word, etc.

The device 8259A can be interfaced with any CPU using either polling or interrupt. In polling, the CPU keeps on checking each peripheral device in sequence to ascertain if it requires any service from the CPU. If any such service request is noticed, the CPU serves the request and then goes on to the next device in sequence. After all the peripheral devices are scanned as above, the CPU again starts from the first device. This type of system operation results in the reduction of processing speed because most of the CPU time is consumed in polling the peripheral devices. In the interrupt driven method, the CPU performs the main processing task till it is interrupted by a service requesting peripheral device. The net processing speed of these types of systems is high because the CPU serves the peripheral only if it receives the interrupt request. If more than one interrupt requests are received at a time, all the requesting peripherals are served one by one on priority basis. This method of interfacing may require additional hardware if the number of peripherals to be interfaced is more than the interrupt pins available with the CPU.

5.13.3 Interrupt Sequence in an 8086-8259A System

1. One or more IR lines are raised high, that will set corresponding IRR bits.
2. 8259A resolves priority and sends an INT signal to CPU.

3. The CPU acknowledges with INTA pulse.
4. Upon receiving an INTA signal from the CPU, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not derive data during this period.
5. The 8086 will initiate a second INTA pulse. During this period, 8259A releases 2nd INTA an 8-bit pointer onto a data bus where it is read by the CPU.
6. This completes the interrupt cycle. The ISR bit is reset at the end of the second INTA pulse if automatic end of interrupt (AEOI) mode is programmed. Otherwise, ISR bit remains set until an appropriate EOI command is issued at the end of interrupt subroutine.

5.13.4 Command Words of 8259A

The command words of 8259A are classified in two groups:

1. Initialization command words (ICWs) and
2. Operation command words (OCWs).

5.13.4.1 Initialization Command Words (ICW)

Before it starts functioning, the 8259A must be initialized by writing two to four command words into the respective command word registers. These are called initialized command words. ICW_1 and ICW_2 are compulsory command words in initialization sequence of 8259A while ICW_3 and ICW_4 are optional.

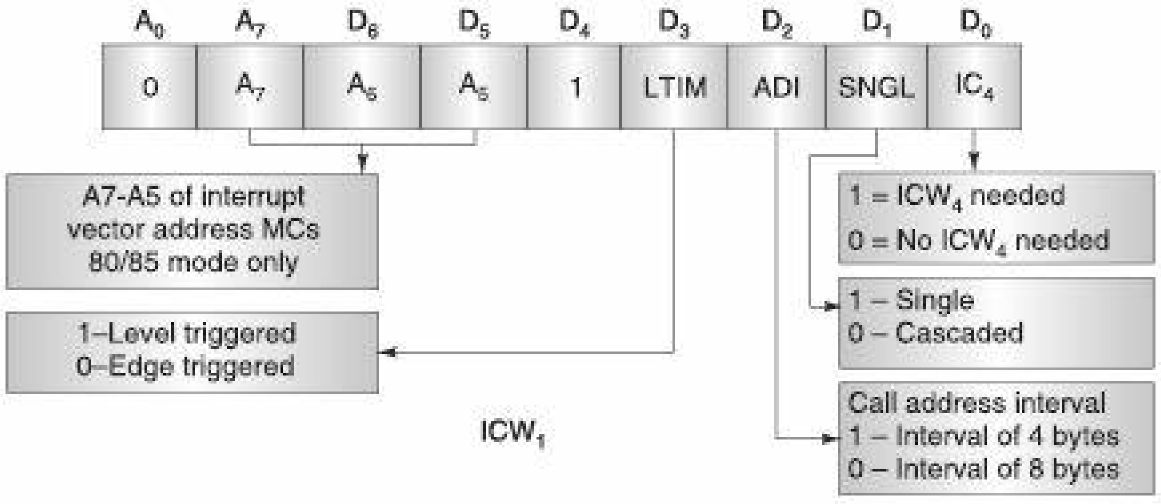


Fig. 5.44 Instruction Command Words ICW_1.

(a) ICW_1

If $A_0 = 0$ and $D_4 = 1$, the control word is recognized as ICW_1. It contains the control bits for edge/level triggered mode, single/cascade mode, call address interval and whether ICW_4 is required or not.

- The edge sense circuit is reset, i.e., by default 8259A interrupts are edge sensitive.
- IMR is cleared.
- Slave mode address is set to 7.
- Special mask mode is cleared and status read is set to IRR.

- If $IC_4 = 0$, all the functions of ICW_4 are set to zero. Master/Slave bit in ICW_4 is used in the buffered mode only.

(b) ICW_2

If $A_0 = 1$, the control word is recognized as ICW_2. The ICW_2 stores details regarding interrupt vector addresses. In an 8085 based system, A_{15}-A_8 of the interrupt vector addresses are the respective bits of ICW_2. In 8086 based system A_{15}-A_{11} of the interrupt vector address are inserted in place of $T_7 - T_3$ respectively and the remaining three bits A_8, A_9, A_{10} are selected depending upon the interrupt level, i.e., from 000 to 111 for IR_0 to IR_7.

$A_0 = 1$ selects ICW_2

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	T_7	T_6	T_5	T_4	T_3	A_{10}	A_9	A_8

Fig. 5.45 Instruction Command Words ICW_2.

- $T_7 - T_3$ are $A_{15} - A_{11}$ of interrupt address
- A_{10}, A_9, A_8 are selected according to interrupt request level. They are not the address lines of microprocessor.

(c) ICW_3

The ICW_3 is read only when there are more than one 8259A in the system, cascading is used (SNGL = 0). The SNGL bit in ICW_1 indicates whether the 8259A in the cascade mode or not. In master mode [SP = 1 or in buffer mode M/S = 1 in ICW_4], the 8-bit slave register will be set bit-wise to 1 for each slave in the system. The requesting slave will then release the second byte of a CALL sequence.

In slave mode [SP = 0 or if BUF = 1 and M/S = 0 in ICW_4] bits D_2 to D_0 identify the slave, i.e., 000 to 111 for slave 1 to slave 8. The slave compares the cascade inputs with these bits and if they are equal, the second byte of the CALL sequence is released by it on the data bus.

(d) ICW_4

This command word depends on the IC_4 bit of ICW_1. If $IC_4 = 1$, ICW_4 is used, otherwise it is neglected.

SFNM: Special fully nested mode is selected, if SFNM = 1

BUF: If BUF = 1, the buffered mode is selected. In the buffered mode, SP/EN acts as enable output and the master/slave is determined using the $M/\overline{S}$ bit of ICW_4.

$M/\overline{S}$ (Master/Slave): If $M/\overline{S} = 1$, 8259A is a master.

If $M/\overline{S} = 0$, 8259A is slave.

If BUF = 0, $M/\overline{S}$ is to be neglected.

AEOI: If AEOI = 1, the automatic end of interrupt mode is selected.

µPM: If the µPM bit is 0, the Mcs-8085 system operation is selected, if µPM = 1, 8086/88 operation is selected.

Master mode ICW_3

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Sn = 1-IR input has a slave
= 0 – IR input does not have a slave

Slave mode ICW_3

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	0	0	ID_2	ID_1	ID_0

$D_2D_1D_0$ – 000 to 111 for IR_0 to IR_7 or slave 1 to slave 8

Fig. 5.46 Instruction Command Words ICW_3.

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	SFNM	BUF	$M/\overline{S}$	AEOI	μPM

Fig. 5.47 Instruction Command Words ICW_4.

5.13.4.2 Operation Command Words

Once 8259A is initialized, it is ready for its normal function, i.e., for accepting the interrupts but 8259A has its own way of handling the received interrupts called modes of operation. These modes of operations can be selected by programming, i.e., writing three internal registers called operation command words registers. The data written into them is called operational command words.

There are three operation command words every bit corresponds to some operational feature of the mode selected, except for a few bits those are either 1 or 0.

- OCW_1
- OCW_2
- OCW_3

(a) OCW_1

OCW_1 is used to mask the unwanted interrupt request if the mask bit is '1'. The corresponding interrupt request is enabled (unmasked) if mask bit is '0'.

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0

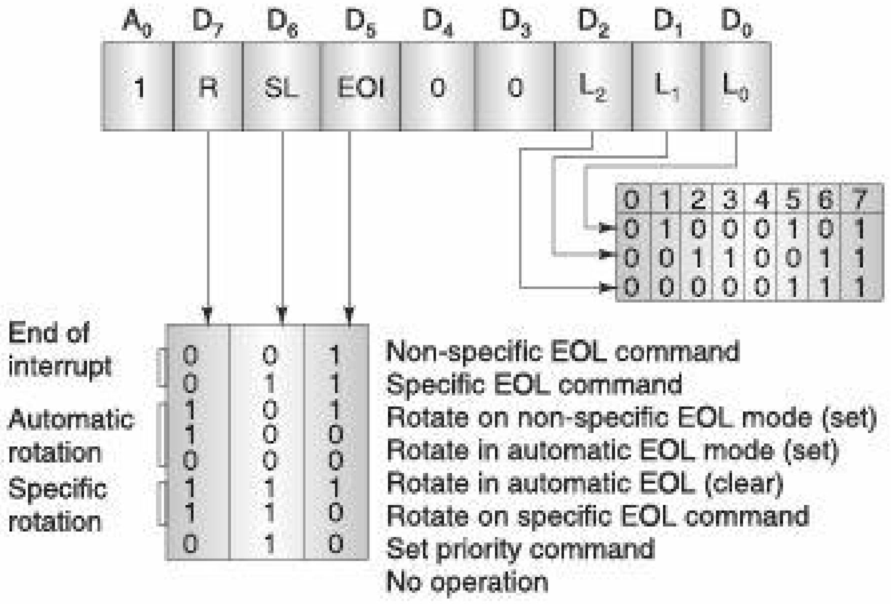
Interrupt mask

1 – Mask set
0 – Mask reset

Fig. 5.48 Operation Command Words OCW_1.

(b) OCW_2

The three bits L_2, L_1 and L_0 in OCW_2 determine the level of interrupt to be selected for operation, if SL (Specific Rotation) bit is active i.e., 1. Level of interrupt is selected by bits D_0-D_2 (L_0-L_2). The D_7 bit is Return (R) and D_5 bit is EOI (end of operation).



A_0 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
1 R SL EOI 0 0 L_2 L_1 L_0
0 1 2 3 4 5 6 7
0 1 0 0 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 0 1 1 1
End of interrupt
Automatic rotation
Specific rotation
0 0 1 Non-specific EOL command
0 1 1 Specific EOL command
1 0 1 Rotate on non-specific EOL mode (set)
1 0 0 Rotate in automatic EOL mode (set)
0 0 0 Rotate in automatic EOL (clear)
1 1 1 Rotate on specific EOL command
1 1 0 Set priority command
0 1 0 No operation

Fig. 5.49 Operation Command Words OCW_2.

(c) OCW_3

In operation command word 3 (OCW_3), if the ESMM bit, i.e., enable special mask mode bit is set to 1, the SMM bit is enabled to select or mask the special mode as shown in Figure 5.50.

When ESMM bit is 0 the SMM bit is neglected. If the SMM bit, i.e., special mask mode bit is 1, the 8259A will enter special mask mode provided ESMM = 1.

If ESMM = 1 and SMM = 0, the 8259A will return to the normal mask mode.

5.13.5 Operating Modes of 8259

The different modes of operation of 8259A can be programmed by setting or resetting the appropriate bits of the ICW or OCW.

The Different Modes of operation of 8259A are:

- Fully Nested Mode
- End of Interrupt (EOI)
- Automatic Rotation Mode
- Automatic EOI Mode
- Specific Rotation Mode
- Specific Mask Mode
- Edge and Level Triggered Mode
- Reading 8259 Status Mode
- Poll Command Mode
- Special Fully Nested Mode

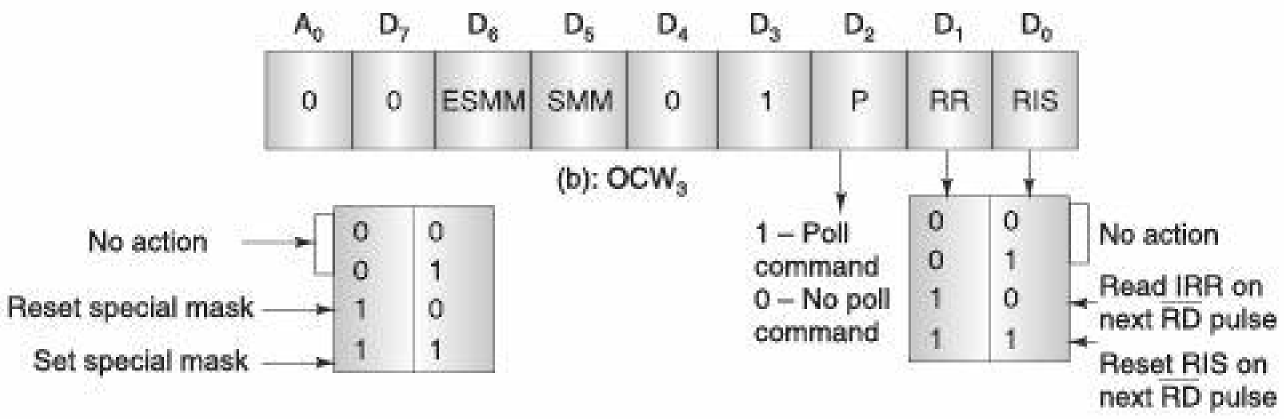

A0 D7 D6 D5 D4 D3 D2 D1 D0
0 0 ESMM SMM 0 1 P RR RIS
(b): OCW3
No action
Reset special mask
Set special mask
1 – Poll command
0 – No poll command
No action
Read IRR on next RD pulse
Reset RIS on next RD pulse

Fig. 5.50 Operation Command Words OCW$_3$.

- Buffered Mode
- Cascade Mode

5.13.5.1 *Fully Nested Mode*

This is the default mode of operation of 8259A. IR$_0$ has the higher priority and IR$_7$ has the lowest one. When interrupt requests are noticed, the highest priority request amongst them is determined and the vector is placed on the data bus. The corresponding bit of ISR is set and remains set till the microprocessor issues an EOI command just before returning from the service routine or the AEOI bit is set.

If the ISR (in service) bit is set, all the same or lower priority interrupts are inhibited but higher levels will generate an interrupt, that will be acknowledged only if the microprocessor interrupt enable flag IF is set. The priorities can afterwards be changed by programming the rotating priority modes.

5.13.5.2 *End of Interrupt (EOI) Mode*

The ISR bit can be reset either with AEOI in ICW$_1$ or by EOI command, issued before returning from the interrupt service routine. There are two types of EOI commands, namely, specific and non-specific. When 8259A is operated in the modes that preserve fully nested structure, it can determine which ISR bit is to be reset on EOI. When non-specific EOI command is issued to 8259A it will automatically reset the highest ISR bit out of those already set.

When a mode that may disturb the fully nested structure is used, the 8259A is no longer able to determine the last level acknowledged. In this case, a specific EOI command is issued to reset a particular ISR bit. An ISR bit that is masked by the corresponding IMR bit, will not be cleared by non-specific EOI of 8259A, if it is in special mask mode.

5.13.5.3 *Automatic Rotation Mode*

This is used in the applications where all the interrupting devices are of equal priority. In this mode, an interrupt request IR level receives priority after it is served while the next device to be served gets the highest priority in sequence. Once all the devices are served like this, the first device again receives the highest priority.

5.13.5.4 *Automatic EOI Mode*

Till AEOI = 1 in ICW_4, the 8259A operates in AEOI mode. In this mode, the 8259A performs a non-specific EOI operation at the trailing edge of the last INTA pulse automatically. This mode should be used only when a nested multilevel interrupt structure is not required with a single 8259A.

5.13.5.5 *Specific Rotation Mode*

In this mode, the bottom priority level can be selected, using L_2, L_1 and L_0 in OCW_2 and R = 1, SL = 1, EOI = 0. The selected bottom priority fixes other priorities. If IR_5 is selected as a bottom priority, then IR_5 will have least priority and IR_4 will have a next higher priority. Thus, IR6 will have the highest priority. These priorities can be changed during an EOI command by programming the rotate on specific EOI command in OCW_2.

5.13.5.6 *Specifc Mask Mode*

In specific mask mode, when a mask bit is set in OCW_1, it inhibits further interrupts at that level and enables interrupt from other levels, which are not masked.

5.13.5.7 *Edge and Level Triggered Mode*

This mode decides whether the interrupt should be edge triggered or level triggered. If bit LTIM of ICW_1 = 0, they are edge triggered, otherwise the interrupts are level triggered.

5.13.5.8 *Reading 8259 Status Mode*

The status of the internal registers of 8259A can be read using this mode. The OCW_3 is used to read IRR and ISR while OCW_1 is used to read IMR. Reading is possible only in no polled mode.

5.13.5.9 *Poll Command Mode*

In polled mode of operation, the INT output of 8259A is neglected, though it functions normally, by not connecting INT output or by masking INT input of the microprocessor. The poll mode is entered by setting P = 1 in OCW_3.

The 8259A is polled by using software execution by microprocessor instead of the requests on INT input. The 8259A treats the next $\overline{RD}$ pulse to the 8259A as an interrupt acknowledge. An appropriate ISR bit is set, if there is a request. The priority level is read and a data word is placed onto data bus, after $\overline{RD}$ is activated. A poll command may give more than 64 priority levels.

5.13.5.10 *Special Fully Nested Mode*

This mode is used in more complicated system, where cascading is used and the priority has to be programmed in the master using ICW_4. This is somewhat similar to the normal nested mode. In this mode, when an interrupt request from a certain slave is in service, this slave can further send request to the master, if the requesting device connected to the slave has higher priority than the one being currently served. In this mode, the master interrupts the CPU only when the interrupting device has a higher or the same priority than the one currently being served. In normal mode, other requests than the one being served are masked out. When entering the interrupt service routine, the software has to check whether this is the only request from the slave. This is done by sending a non-specific EOI to the master, otherwise no EOI should be sent. This mode is important, since in the absence of this mode, the slave would interrupt the master only once and hence the priorities of the slave inputs would have been disturbed.

5.13.5.11 *Buffered Mode*

When the 83259A is used in the systems, where bus driving buffers are used on data buses. The problem of enabling the buffers exists. The 8259A sends buffer enable signal on SP/EN pin, whenever data is placed on the bus.

5.13.5.12 *Cascade Mode*

The 8259A can be connected in a system containing one master and eight slaves (maximum) to handle up to 64 priority levels. The master controls the slaves using CAS0-CAS2 which act as chip select inputs (encoded) for slaves. In this mode, the slave INT outputs are connected with master IR inputs. When a slave request line is activated and acknowledged, the master will enable the slave to release the vector address during second pulse of INTA sequence.

The cascade lines are normally low and contain slave address codes from the trailing edge of the first INTA pulse to the trailing edge of the second INTA pulse. Each 8259A in the system must be separately initialized and programmed to work in different modes. The EOI command must be issued twice, one for master and the other for the slave. A separate address decoder is used to activate the chip select line of each 8259A.

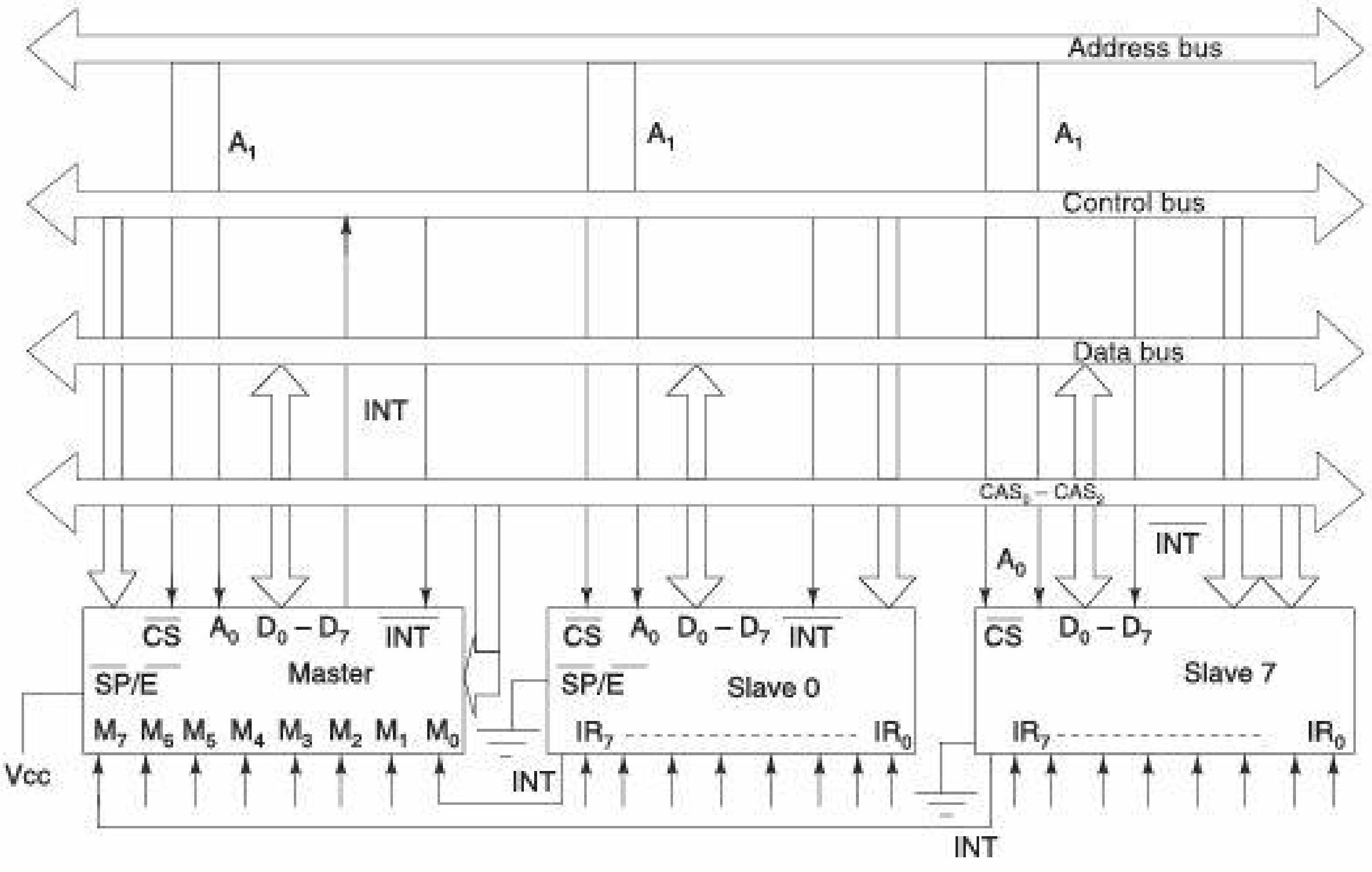
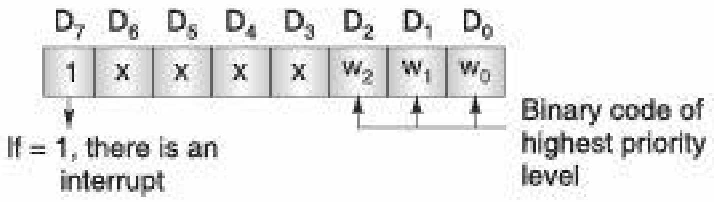

Address bus
A_1
A_1
A_1
Control bus
Data bus
INT
$CAS_0 - CAS_2$
$\overline{INT}$
A_0
$\overline{CS}$ A_0 $D_0 - D_7$ $\overline{INT}$
Master
$\overline{SP/E}$
M_7 M_6 M_5 M_4 M_3 M_2 M_1 M_0
Vcc
$\overline{CS}$ A_0 $D_0 - D_7$ $\overline{INT}$
Slave 0
$\overline{SP/E}$
IR_7 IR_0
INT
$\overline{CS}$ $D_0 - D_7$
Slave 7
IR_7 IR_0
INT

Fig. 5.51 Circuit Connections of 8259A in Cascade Scheme.



D7 D6 D5 D4 D3 D2 D1 D0
1 X X X X W2 W1 W0
If = 1, there is an interrupt
Binary code of highest priority level

Fig. 5.52 Data Word of 8259.

5.14 DIRECT MEMORY ACCESS CONTROLLER (INTEL 8237A)

In this section, we will further discuss a few advanced peripherals and their interfacing techniques with 8085 and 8086. In the applications where the CPU is to transfer bulk data, it may be a waste of time to transfer the data from source to destination using program controlled data transfer or interrupt driven data transfer. The alternate way of transferring the bulk data is the Direct Memory Access (DMA) technique in which the data is transferred under the control of a DMA controller, after it is properly initialized by the CPU. The DMA controller used HOLD and HLDA signals available on the 8085 microprocessor as if it were a peripheral requesting the MPU for the control of the buses. The MPU communicates with the DMA controller by using the chip select line, buses and control signals. Once the controller has gained the control, it acts as a processor for data transfer.

A DMA controller is designed to complete the bulk data transfer task much faster than the CPU. One such application which involves bulk data transfer is the storage of programs or data into secondary memory. The floppy disk is one of the most popular forms of secondary memories. A Floppy Disk Controller (FDC) coordinates the complicated task of interfacing the floppy disk drive mechanism with the CPU, storing the parallel data onto the magnetic disk media in a serial bit string form and also reading the serially stored data and converting it into the parallel form. This data transfer between the CPU and the FDC may be controlled using a DMA controller. A CRT, controller derives all the control and timing signals required for controlling and interfacing a CRT with the CPU. A DMA controller may also be used to transfer data from the system memory to a video RAM of a CRT display. In short, this chapter elaborates DMA controllers and other peripherals which may require DMA for their operation, viz., floppy disk controller and CRT controllers.

The Direct Memory Access or DMA mode of data transfer is the fastest amongst all the modes of data transfer. In this mode, the device may transfer data directly to/from memory without any interference from the CPU. The device requests the CPU (through a DMA controller) to hold its data, address and control bus, so that the device may transfer data directly to/from memory. The DMA data transfer is initiated only after receiving HLDA signal from the CPU. For facilitating DMA type of data transfer between several devices, a DMA controller may be used.

The 8237A Multimode Direct Memory Access (DMA) Controller is a peripheral interface circuit for microprocessor systems. It is designed to improve system performance by allowing external devices to directly transfer information from the system memory. Memory-to-memory transfer capability is also provided. The 8237A offers a wide variety of programmable control features to enhance data throughput and system optimization and to allow dynamic re-configuration under program control. The 8237A is designed to be used in conjunction with an external 8-bit address latch. It contains four independent channels and may be expanded to any number of channels by cascading additional controller chips. The three basic transfer modes allow programmability of the types of DMA service by the user. Each channel can be individually programmed to auto initialize to its original condition following an End of Process (EOP). Each channel has a full 64 K address and word count capability.

The pin diagram of 8237A is shown in Figure 5.53 and explanation of pins is given in Table 5.20.

5.14.1 Pin Description

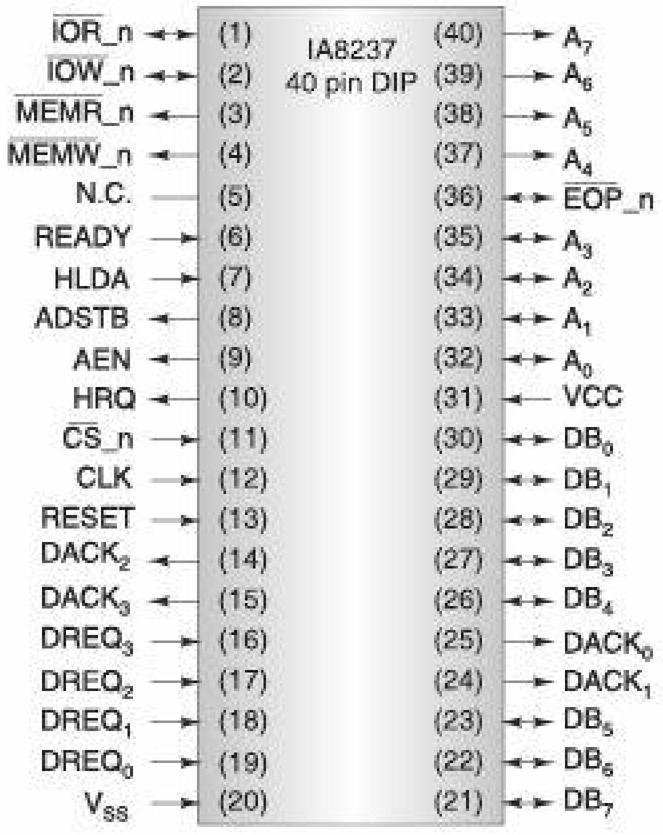


Fig. 5.53 Pin Configuration of 8237A.

Table 5.20 Pin Description of 8237A

Symbol	Type	Name and Function
VCC	I	**Power:** +5V supply
VSS	I	**Ground:** Ground
CLK	I	**CLOCK INPUT:** Clock input controls the internal operations of the 8237A and its rate of data transfers. The input may be driven at up to 5 MHz for the 8237A-5.
$\overline{CS}$	I	**CHIP SELECT:** Chip select is an active low input used to select the 8237A as an I/O device during the idle cycle. This allows CPU communication on the data bus.
RESET	I	**RESET:** Reset is an active high input which clears the Command, Status, Request and Temporary registers. It also clears the first/last flip-flop and sets the Mask register. Following a Reset the device is in the Idle cycle.

Contd...

READY	I	**READY:** Ready is an input used to extend the memory read and write pulses from the 8237A to accommodate slow memories or I/O peripheral devices. Ready must not make transitions during its specified setup/hold time.
HLDA	I	**HOLD ACKNOWLEDGE:** The active high Hold Acknowledge from the CPU indicates that it has relinquished control of the system buses.
$DREQ_0$-$DREQ_3$	I	**DMA REQUEST:** The DMA Request lines are individual asynchronous channel request inputs used by peripheral circuits to obtain DMA service. In fixed priority, DREQ0 has the highest priority and $DREQ_3$ has the lowest priority. A request is generated by activating the DREQ line of a channel. DACK will acknowledge the recognition of DREQ signal. Polarity of DREQ is programmable. Reset initializes these lines to active high. DREQ must be maintained until the corresponding DACK goes active.
DB_0-DB_7	I/O	**DATA BUS:** The Data Bus lines are bidirectional three-state signals connected to the system data bus. The outputs are enabled in the program condition during the I/O Read to output the contents of an Address register, a Status register, the Temporary register or a Word Count register to the CPU. The outputs are disabled and the inputs are read during an I/O Write cycle when the CPU is programming the 8237A control registers. During DMA cycles the most significant 8 bits of the address are output onto the data bus to be strobed into an external latch by ADSTB. In memory-to-memory operations, data from the memory comes into the 8237A on the data bus during the read-from memory transfer. In the write-to-memory transfer, the data bus outputs place the data into the new memory location.
$\overline{IOR}$	I/O	**I/O READ:** I/O Read is a bidirectional active low three-state line. In the idle cycle, it is an input control signal used by the CPU to read the control registers. In the active cycle, it is an output control signal used by the 8237A to access data from a peripheral during a DMA Write transfer.
$\overline{IOW}$	I/O	**I/O WRITE:** I/O Write is a bidirectional active low three-state line. In the idle cycle, it is an input control signal used by the CPU to load information into the 8237A. In the active cycle, it is an output control signal used by the 8237A to load data to the peripheral during a DMA Read transfer.
$\overline{EOP}$	I/O	**END OF PROCESS:** End of process is an active low bidirectional signal. Information concerning the completion of DMA services is available at the bidirectional $\overline{EOP}$ pin. The 8237A allows an external signal to terminate an active DMA service. This is accomplished by pulling the EOP input low with an external EOP signal. The 8237A also generates a pulse when the terminal count (TC) for any channel is reached. This generates an EOP signal which is output through the EOP line. The reception of EOP, either internal or external, will cause the 8237A to terminate the service, reset the request, and, if auto initialize is enabled, to write the base registers to the current registers of that channel. The mask bit and TC bit in the status word will be set for the currently active channel by EOP unless the channel is programmed for auto-initialize. In that case, the mask bit remains unchanged. During memory-to-memory transfers, EOP will be output when the TC for channel 1 occurs. EOP should be tied high with a pull-up resistor if it is not used to prevent erroneous end of process inputs.
A_0-A_3	I/O	**ADDRESS:** The four least significant address lines are bidirectional three-state signals. In the idle cycle, they are inputs and are used by the CPU to address the register to be loaded or read. In the active cycle, they are outputs and provide the lower 4 bits of the output address.

Contd...

Contd...

A_4-A_7	O	**ADDRESS:** The four most significant address lines are three-state outputs and provide 4 bits of address. These lines are enabled only during the DMA service.
HRQ	O	**HOLD REQUEST:** This is the Hold Request to the CPU and is used to request control of the system bus. If the corresponding mask bit is clear, the presence of any valid DREQ causes 8237A to issue the HRQ.
$DACK_0$-$DACK_3$	O	**DMA ACKNOWLEDGE:** DMA Acknowledge is used to notify the individual peripherals when one has been granted a DMA cycle. The sense of these lines is programmable. Reset initializes them to active low.
AEN	O	**ADDRESS ENABLE:** Address Enable enables the 8-bit latch containing the upper 8 address bits onto the system address bus. AEN can also be used to disable other system bus drivers during DMA transfers. AEN is active HIGH.
ADSTB	O	**ADDRESS STROBE:** The active high, Address Strobe is used to strobe the upper address byte into an external latch.
$\overline{\text{MEWR}}$	O	**MEMORY READ:** The Memory Read signal is an active low three-state output used to access data from the selected memory location during a DMA Read or a memory-to-memory transfer.
$\overline{\text{MEWR}}$	O	**MEMORY READ:** The Memory Read signal is an active low three-state output used to access data from the selected memory location during a DMA Read or a memory-to-memory transfer.
PIN-5	I	**PIN_5:** This pin should always be at a logic HIGH level. An internal pull-up resistor will establish logic high when the pin is left floating. It is recommended, however, that PIN5 be connected to VCC.

5.14.2 Block Diagram of 8237 A

Functional Description

The 8237A block diagram includes the major logic blocks and all of the internal registers. The data interconnection paths are also shown in Figure 5.54.

Not shown are the various control signals between the blocks. The 8237A contains 344 bits of internal memory in the form of registers. Table 5.15 lists these registers by name and shows the size of each. A detailed description of the registers and their functions can be found under Register Description.

The 8237A contains three basic blocks of control logic. The Timing Control block generates internal timing and external control signals for the 8237A. The Program Command Control block decodes the various commands given to the 8237A by the microprocessor prior to servicing a DMA Request. It also decodes the Mode Control word used to select the type of DMA during the servicing. The Priority Encoder block resolves priority contention between DMA channels requesting service simultaneously. The Timing Control block derives internal timing from the clock input. In 8237A systems, this input will usually be the w_2 TTL clock from an 8224 or CLK from an 8085AH or 8284A. 33% duty cycle clock generators, however, may not meet the clock high time requirement of the 8237A of the same frequency. For example, 82C84A-5 CLK output violates the clock high time requirement of 8237A-5.

In this case, 82C84A CLK can simply be inverted to meet 8237A-5 clock high and low time requirements. For 8085AH-2 systems above 3.9 MHz, the 8085 CLK (OUT) does not satisfy

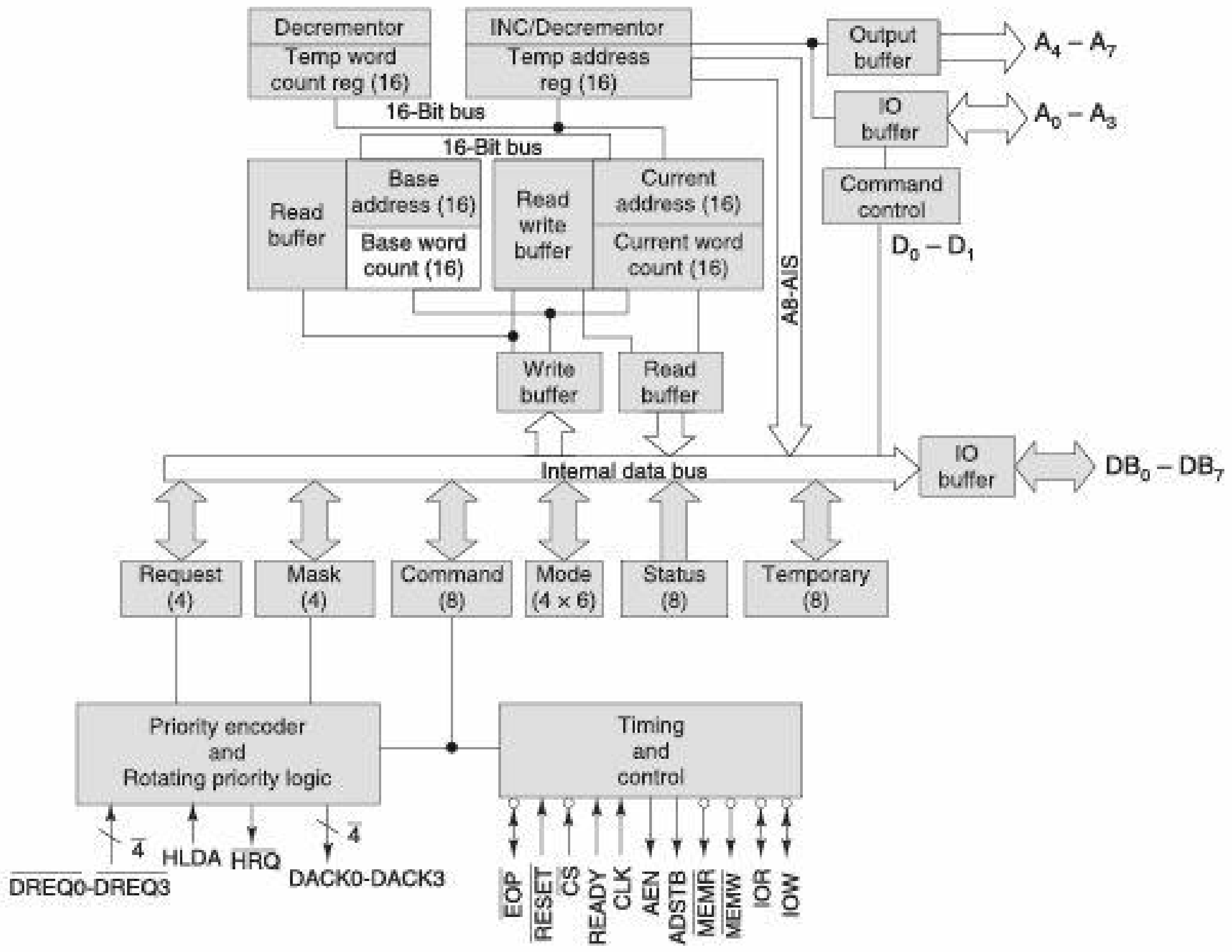


Fig. 5.54 Block Diagram of 8237A.

8237A-5 clock LOW and HIGH time requirements. In this case, an external clock should be used to drive the 8237A-5.

Table 5.21 8237A Internal Registers

Name	*Size*	*Number*
Base Address Registers	16 bits	4
Base Word Count Registers	16 bits	4
Current Address Register	16 bits	4
Current Word Count Address	16 bits	4
Temporary Address Register	16 bits	1
Temporary Word Count Register	16 bits	1
Status Register	8 bits	1
Command Register	8 bits	1
Temporary Register	8 bits	1
Mode Register	6 bits	4
Mask Register	4 bits	1
Request Register	4bits	1

5.14.3 DMA Operation

The 8237A is designed to operate in two major cycles. These are called Idle and Active cycles. Each device cycle is made up of a number of states. The 8237A can assume seven separate states, each composed of one full clock period. State I (SI) is the inactive state. It is entered when the 8237A has no valid DMA requests pending. While in SI, the DMA controller is inactive but may be in the Program Condition, being programmed by the processor. State S0 (S0) is the first state of a DMA service. The 8237A has requested a hold but the processor has not yet returned an acknowledge. The 8237A may still be programmed until it receives HLDA from the CPU. An acknowledgement from the CPU will signal that DMA transfers may begin. S_1, S_2, S_3 and S_4 are the working states of the DMA service. If more time is needed to complete a transfer than is available with normal timing, wait states (SW) can be inserted between S_2 or S_3 and S_4 by the use of the Ready line on the 8237A. Note that the data is transferred directly from the I/O device to memory (or vice versa) with $\overline{\text{IOR}}$ and $\overline{\text{MEMW}}$ (or $\overline{\text{MEMR}}$ and $\overline{\text{IOW}}$) being active at the same time. The data is not read into or driven out of the 8237A in I/O-to-memory or memory-to-I/O DMA transfers.

Memory-to-memory transfers require a read-from and a write-to-memory to complete each transfer. The states, which resemble the normal working states, use two-digit numbers for identification. Eight states are required for a single transfer. The first four states (S_{11}, S_{12}, S_{13}, S_{14}) are used for the read from memory half and the last four states (S_{21}, S_{22}, S_{23}, S_{24}) for the write-to-memory half of the transfer.

5.14.3.1 Idle Cycle

When no channel is requesting service, the 8237A will enter the idle cycle and perform "SI" states. In this cycle the 8237A will sample the DREQ lines every clock cycle to determine if any channel is requesting a DMA service. The device will also sample CS, looking for an attempt by the microprocessor to write or read the internal registers of the 8237A. When $\overline{\text{CS}}$ is low and HLDA is low, the 8237A enters the Program Condition. The CPU can now establish, change or inspect the internal definition of the part by reading from or writing to the internal registers. Address lines A_0-A_3 are inputs to the device and select which registers will be read or written.

The IOR and $\overline{\text{IOW}}$ lines are used to select and time reads or writes. Due to the number and size of the internal registers, an internal flip-flop is used to generate an additional bit of address. This bit is used to determine the upper or lower byte of the 16-bit Address and Word Count registers. The flip-flop is reset by Master Clear or Reset. A separate software command can also reset this flip-flop. Special software commands can be executed by the 8237A in the Program Condition. These commands are decoded as sets of addresses with the $\overline{\text{CS}}$ and $\overline{\text{IOW}}$. The commands do not make use of the data bus. Instructions include Clear First/Last Flip-Flop and Master Clear.

5.14.3.2 Active Cycle

When the 8237A is in the idle cycle and a non-masked channel requests a DMA service, the device will output an HRQ to the microprocessor and enter the active cycle. It is in this cycle that the DMA service will take place, in one of the four modes given below:

(a) Single Transfer Mode: In Single Transfer Mode, the device is programmed to make one transfer only. The word count will be decremented and the address decremented or incremented following each transfer. When the word count "rolls over" from zero to FFFFH, a Terminal Count (TC) will cause an auto-initialize if the channel has been programmed to do so. DREQ must be held active until DACK becomes active in order to be recognized. If DREQ is held active throughout the single transfer, HRQ will go inactive and release the bus to the system. It will again go active and, upon receipt of a new HLDA, another single transfer will be performed. In 8080A, 8085AH, 8088, or 8086 system, this will ensure one full machine cycle execution between DMA transfers. Details of timing between the 8237A and other bus control protocols will depend upon the characteristics of the microprocessor involved.

(b) Block Transfer Mode: In Block Transfer Mode, the device is activated by DREQ to continue making transfers during the service until a TC, caused by word count going to FFFFH, or an external End of Process (EOP) is encountered. DREQ need only be held active until DACK becomes active. Again, an auto initialization will occur at the end of the service if the channel has been programmed for it.

(c) Demand Transfer Mode: In Demand Transfer Mode, the device is programmed to continue making transfers until a TC or external EOP is encountered or until DREQ goes inactive. Thus, transfers may continue until the I/O device has exhausted its data capacity. After the I/O device has had a chance to catch up, the DMA service is re-established by means of a DREQ. During the time between services when the microprocessor is allowed to operate the intermediate values of address and word count are stored in the 8237A Current Address and Current Word Count registers. Only an EOP can cause an auto-initialize at the end of the service.

(d) Cascade Mode: This mode is used to cascade more than one 8237A together for simple system expansion. The HRQ and HLDA signals from the additional 8237A are connected to the DREQ and DACK signals of a channel of the initial 8237A. This allows the DMA requests of the additional device to propagate through the priority network circuitry of the preceding device. The priority chain is preserved and the new device must wait for its turn to acknowledge requests. Since the cascade channel of the initial 8237A is used only for prioritizing the additional device, it does not output any address or control signals of its own. These could conflict with the outputs of the active channel in the added device. The 8237A will respond to DREQ and DACK but all other outputs except HRQ will be disabled. The ready input is ignored. Figure 5.55 shows two additional devices cascaded into an initial device using two of the previous channels. This forms a two-level DMA system. More 8237A's could be added at the second level by using the remaining channels of the first level. Additional devices can also be added by cascading into the channels of the second level device, forming a third level.

5.14.4 Transfer Types

Each of the three active transfer modes can perform three different types of transfers. These are Read, Write and Verify. Write transfers move data from an I/O device to the memory by activating $\overline{\text{MEMW}}$ and $\overline{\text{IOR}}$. Read transfers move data from memory to an I/O device by activating $\overline{\text{MEMR}}$

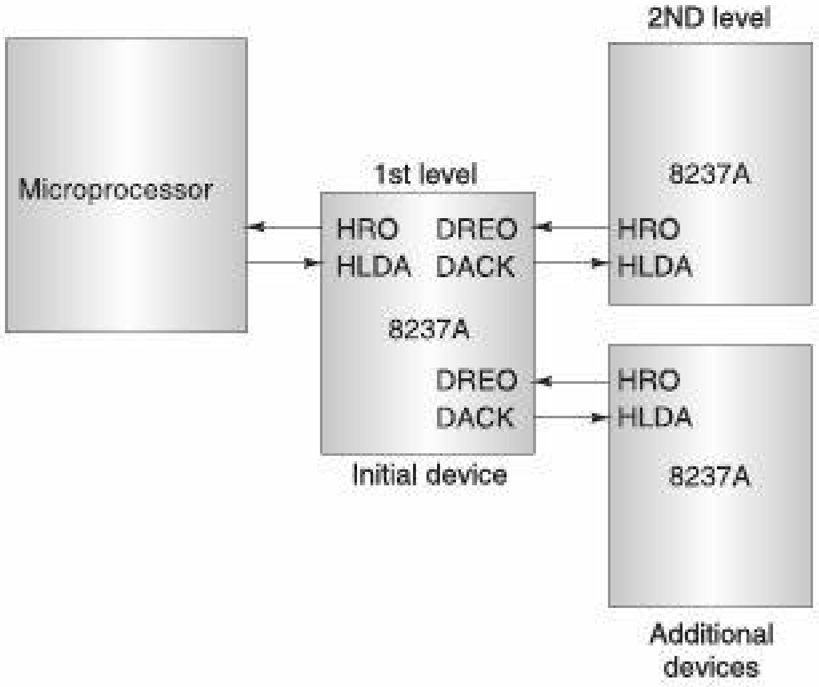


Fig. 5.55 Cascaded 8237As.

and $\overline{IOW}$. Verify transfers are pseudo transfers. The 8237A operates as in Read or Write transfers generating addresses, and responding to $\overline{EOP}$, etc. However, the memory and I/O control lines all remain inactive. The ready input is ignored in verify mode.

5.14.4.1 Memory-to-Memory

To perform block moves of data from one memory address space to another with a minimum of program effort and time, the 8237A includes a memory-to-memory transfer feature. Programming a bit in the Command register selects channels 0 and 1 to operate as memory-to memory transfer channels. The transfer is initiated by setting the software DREQ for channel 0. The 8237A requests a DMA service in the normal manner. After HLDA is true, the device, using four-state transfers in Block Transfer mode, reads data from the memory. The channel 0 Current Address register is the source for the address used and is decremented or incremented in the normal manner. The data byte read from the memory is stored in the 8237A internal Temporary register. Channel 1 then performs a four-state transfer of the data from the temporary register to memory using the address in its Current Address register and incrementing or decrementing it in the normal manner. The channel 1 current word count is decremented. When the word count of channel 1 goes to FFFFH, a TC is generated causing an EOP output terminating the service. Channel 0 may be programmed to retain the same address for all transfers. This allows a single word to be written to a block of memory. The 8237A will respond to external $\overline{EOP}$ signals during memory-to-memory transfers. Data comparators in block search schemes may use this input to terminate the service when a match is found. Memory-to-memory operations can be detected as an active AEN with no DACK outputs.

5.14.4.2 Auto Initialize

By programming a bit in the Mode register, a channel may be set up as an Auto-initialize channel. During Auto initialization, the original values of the Current Address and Current Word

Count registers are automatically restored from the Base Address and Base Word Count registers of that channel following $\overline{EOP}$. The base registers are loaded simultaneously with the current registers by the microprocessor and remain unchanged throughout the DMA service. The mask bit is not altered when the channel is in Auto-initialize. Following Auto-initialize the channel is ready to perform another DMA service, without CPU intervention, as soon as a valid DREQ is detected. In order to auto-initialize both channels in a memory-to-memory transfer, both word counts should be programmed identically. If interrupted externally, $\overline{EOP}$ pulses should be applied in both bus cycles.

5.14.4.3 Priority

The 8237A has two types of priority encoding available as software selectable options. The first is Fixed Priority which fixes the channels in priority order based upon the descending value of their number. The channel with the lowest priority is 3 followed by 2, 1 and the highest priority channel, 0. After the recognition of any one channel for service, the other channels are prevented from interferring with that service until it is completed. After completion of a service, HRQ will go inactive and the 8237A will wait for HLDA to go low before activating HRQ to service another channel. The second scheme is Rotating Priority. The last channel to get service becomes the lowest priority channel with the others rotating accordingly.

With Rotating Priority in a single chip DMA system, any device requesting service is guaranteed to be recognized after no more than three higher priority services have occurred. This prevents any one channel from monopolizing the system.

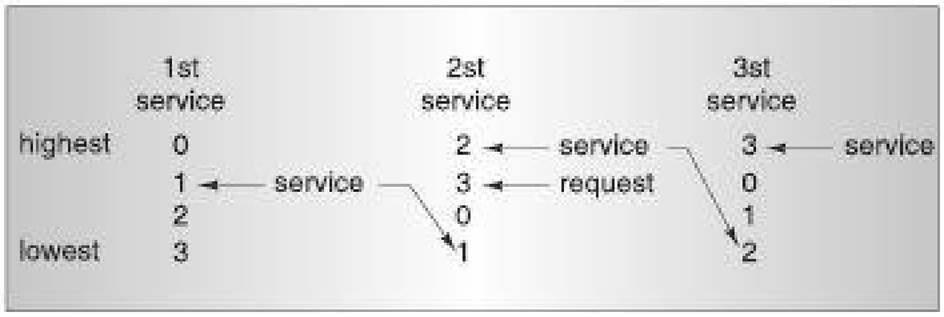


Fig. 5.56 Priority of Channels in 8237A.

5.14.4.4 Compressed Timing

In order to achieve even greater throughput where system characteristics permit, the 8237A can compress the transfer time to two clock cycles. From clock cycles it can be seen that state S_3 is used to extend the access time of the read pulse. By removing state S_3, the read pulse width is made equal to the write pulse width and a transfer consists only of state S_2 to change the address and state S_4 to perform the read/write. S_1 states will still occur when A_8-A_{15} will need updating (see Address Generation).

5.14.4.5 Address Generation

In order to reduce pin count, the 8237A multiplexes the eight higher order address bits on the data lines. State S1 is used to output the higher order address bits to an external latch from which

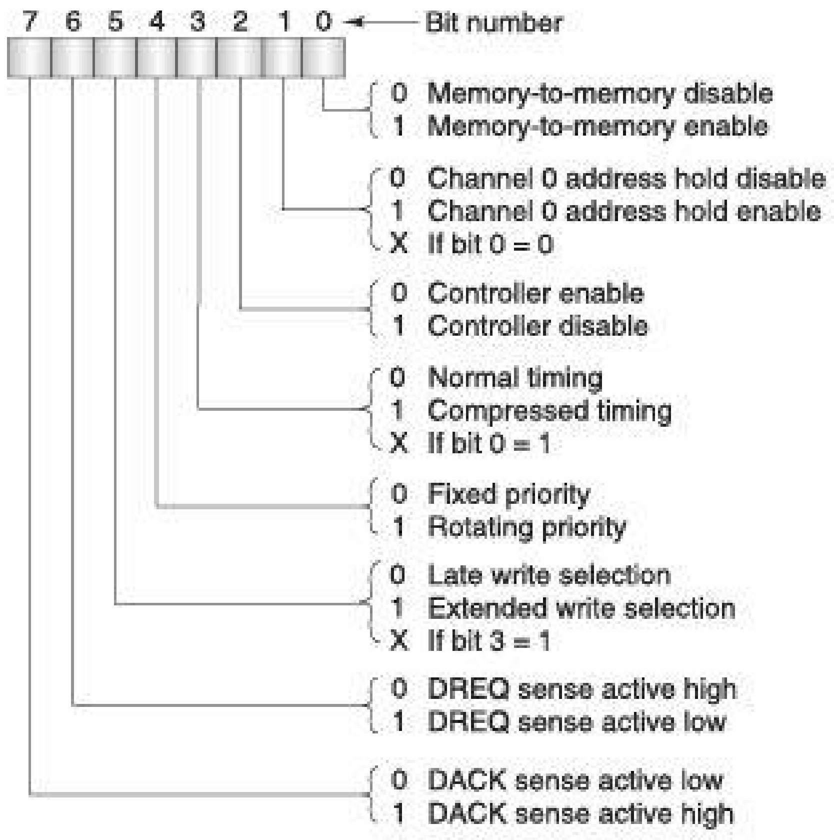


Fig. 5.57 Command Register.

they may be placed on the address bus. The falling edge of Address Strobe (ADSTB) is used to load these bits from the data lines to the latch. Address Enable (AEN) is used to enable the bits onto the address bus through a three-state enable. The lower order address bits are output by the 8237A directly. Lines A_0-A_7 should be connected to the address bus. During Block and Demand Transfer mode services, which include multiple transfers, the addresses generated will be sequential. For many transfers, the data held in the external address latch will remain the same. This data need only change when a carry or borrow from A_7 to A_8 takes place in the normal sequence of addresses. To save time and speed transfers, the 8237A executes S_1 states only when updating of A_8-A_{15} in the latch is necessary. This means for long services, S_1 states and Address Strobes may occur only once every 256 transfers, a savings of 255 clock cycles for each 256 transfers.

5.14.5 Register Description

Current Address register: Each channel has a 16-bit Current Address register. This register holds the value of the address used during DMA transfers. The address is automatically incremented or decremented after each transfer and the intermediate values of the address are stored in the Current Address register during the transfer. This register is written or read by the microprocessor in successive 8-bit bytes. It may also be reinitialized by an Auto-initialize back to its original value. Auto-initialize takes place only after an $\overline{\text{EOP}}$.

5.14.5.1 Current Word Register

Each channel has a 16-bit Current Word Count register. This register determines the number of transfers to be performed. The actual number of transfers will be one more than the number programmed in the Current Word Count register (i.e., programming a count of 100 will result in 101 transfers). The word count is decremented after each transfer. The intermediate value of the word count is stored in the register during the transfer. When the value in the register goes from zero to FFFFH, a TC will be generated. This register is loaded or read in successive 8-bit bytes by the microprocessor in the Program Condition. Following the end of a DMA service it may also be reinitialized by an Auto-initialization back to its original value. Auto-initialize can occur only when an EOP occurs. If it is not Auto-initialized, this register will have a count of FFFFH after TC.

5.14.5.2 Base Address and Base Word Count Registers

Each channel has a pair of Base Address and Base Word Count registers. These 16-bit registers store the original value of their associated current registers. During Auto-initialize, these values are used to restore the current registers to their original values. The base registers are written simultaneously with their corresponding current register in 8-bit bytes in the Program Condition by the microprocessor. These registers cannot be read by the microprocessor.

5.14.5.3 Command Register

This 8-bit register controls the operation of the 8237A. It is programmed by the microprocessor in the Program Condition and is cleared by Reset or a Master Clear instruction. The Figure 5.57(a) shows the function of the command register bits.

5.14.5.4 Mode Register

Each channel has a 6-bit Mode register associated with it. When the register is being written to by the microprocessor in the Program Condition, bits 0 and 1 determine which channel Mode register is to be written as shown in Figure 5.58.

5.14.5.5 Request Register

The 8237A can respond to requests for DMA service which are initiated by software as well as by a DREQ. Each channel has a request bit associated with it in the 4-bit Request register.

These are non-maskable and subject to prioritization by the Priority Encoder network. Each register bit is set or reset separately under software control or is cleared upon generation of a TC or external EOP. The entire register is cleared by a Reset.

To set or reset a bit, the software loads the proper form of the data word for register address coding. In order to make a software request, the channel must be in Block Mode.

5.14.5.6 Mask Register

Each channel has a mask bit associated with it which can be set to disable the incoming DREQ.

Each mask bit is set when its associated channel produces an EOP if the channel is not programmed for Auto-initialize. Mask register may also be set or cleared separately under software control. The entire register is also set by a Reset. This disables all DMA requests until a clear Mask register instruction allows them to occur. The instruction to separately set or clear the mask bits is similar in form to that used with the Request register.

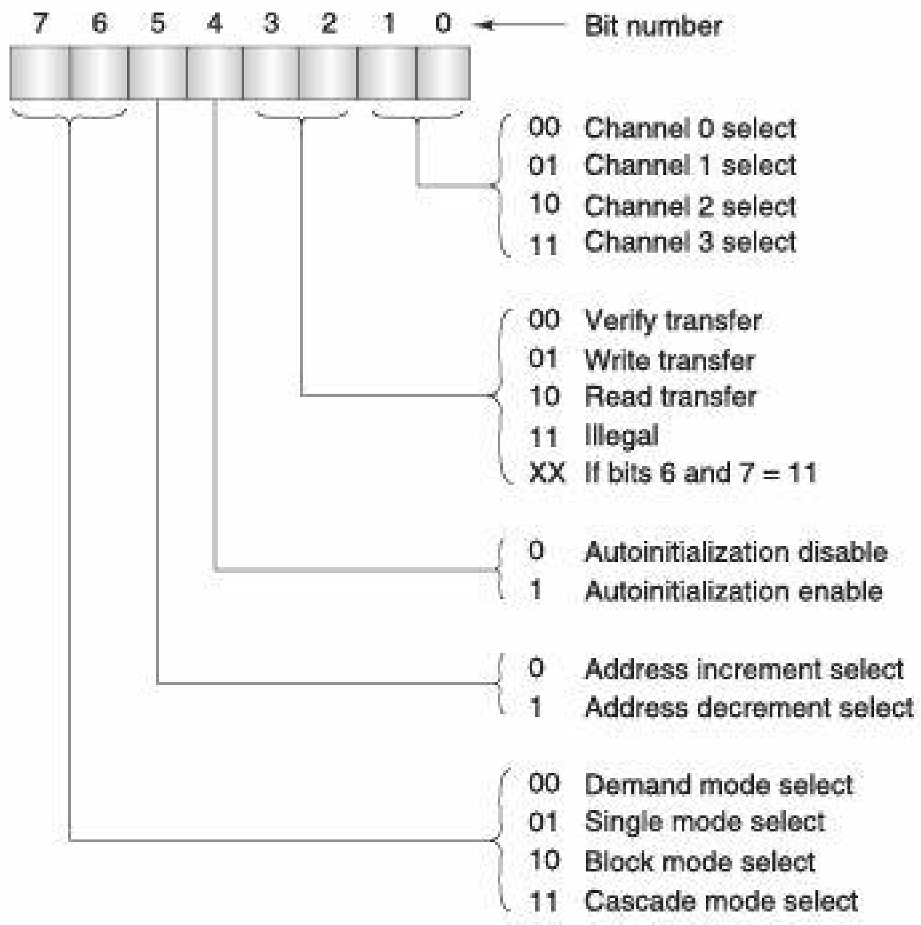


Fig. 5.58 Mode Register.

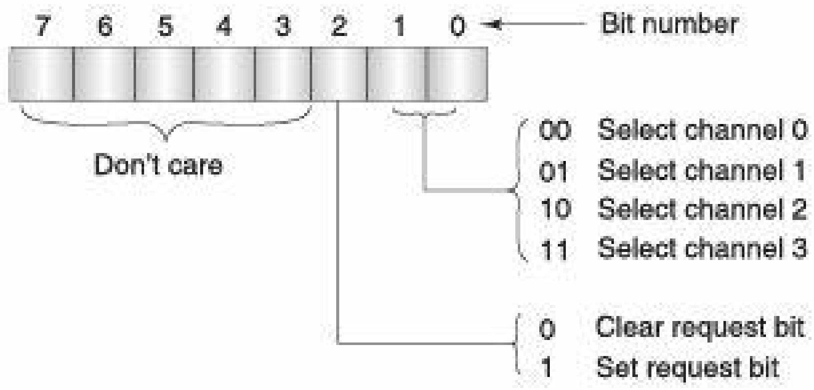
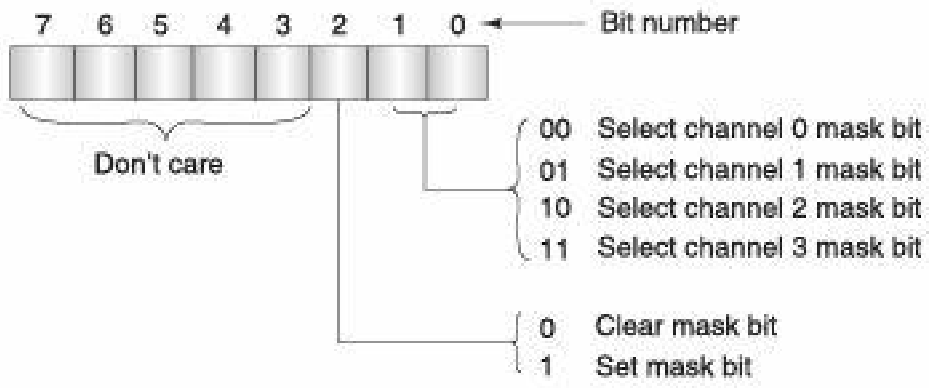


Fig. 5.59 Request Register.



7 6 5 4 3 2 1 0
Bit number
Don't care
00 Select channel 0 mask bit
01 Select channel 1 mask bit
10 Select channel 2 mask bit
11 Select channel 3 mask bit
0 Clear mask bit
1 Set mask bit

Fig. 5.60(a) Mask Register.

All four bits of the Mask register may also be written with a single command.

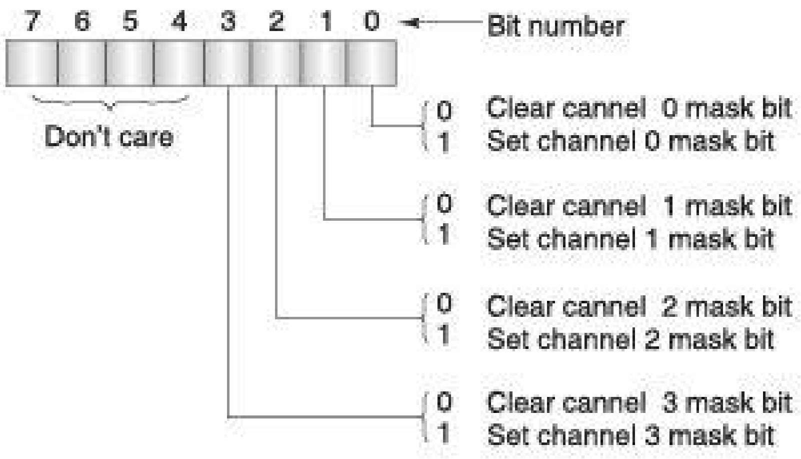


Fig. 5.60(b) Clear/Set the Channels Mask Bit.

5.14.5.7 *Status Register*

The status register is available to be read out of the 8237A by the microprocessor. It contains information about the status of the devices at this point. This information includes which channels have reached a terminal count and which channels have pending DMA requests.

Bits 0-3 are set every time a TC is reached by that channel or an external EOP is applied. These bits are cleared upon Reset and on each Status Read. Bits 4-7 are set whenever their corresponding channel is requesting service.

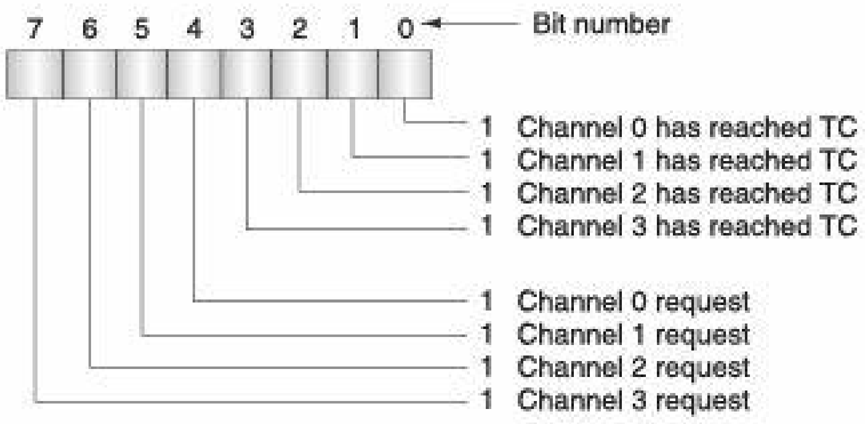

7 6 5 4 3 2 1 0
Bit number
1 Channel 0 has reached TC
1 Channel 1 has reached TC
1 Channel 2 has reached TC
1 Channel 3 has reached TC
1 Channel 0 request
1 Channel 1 request
1 Channel 2 request
1 Channel 3 request

Fig. 5.61 Status Register.

5.14.5.8 *Temporary Register*

The Temporary register is used to hold data during memory-to-memory transfers. Following the completion of the transfers, the last word moved can be read by the microprocessor in the Program Condition. The temporary register always contains the last byte transferred in the previous memory-to-memory operation, unless cleared by a Reset.

5.14.5.9 *Software Commands*

These are additional special software commands which can be executed in the Program Condition. They do not depend on any septic bit pattern on the data bus. The three software commands are:

Table 5.22 Word Count and Address Register Command Codes

Channel	Register	Operation	Signals							Internal	Data Bus
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A_3	A_2	A_1	A_0	Flip-Flop	DB_0-DB_7
0	Base and Current Address	Write	0	1	0	0	0	0	0	0	A_0-A_7
			0	1	0	0	0	0	0	1	A_8-A_{15}
	Current Address	Read	0	0	1	0	0	0	0	0	A_0-A_7
			0	0	1	0	0	0	0	1	A_8-A_{15}
	Base and Current Word Count	Write	0	1	0	0	0	0	1	0	W_0-W_7
			0	1	0	0	0	0	1	1	W_8-W_{15}
	Current Word Count	Read	0	0	1	0	0	0	1	0	W_0-W_7
			0	0	1	0	0	0	1	1	W_8-W_{15}
1	Base and Current Address	Write	0	1	0	0	0	1	0	0	A_0-A_7
			0	1	0	0	0	1	0	1	A_8-A_{15}
	Current Address	Read	0	0	1	0	0	1	0	0	A_0-A_7
			0	0	1	0	0	1	0	1	A_8-A_{15}
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	W_0-W_7
			0	1	0	0	0	1	1	1	W_8-W_{15}
	Current Word Count	Read	0	0	1	0	0	1	1	0	W_0-W_7
			0	0	1	0	0	1	1	1	W_8-W_{15}
2	Base and Current Address	Write	0	1	0	0	1	0	0	0	A_0-A_7
			0	1	0	0	1	0	0	1	A_8-A_{15}
	Current Address	Read	0	0	1	0	1	0	0	0	A_0-A_7
			0	0	1	0	1	0	0	1	A_8-A_{15}
	Base and Current Word Count	Write	0	1	0	0	1	0	1	0	W_0-W_7
			0	1	0	0	1	0	1	1	W_8-W_{15}
	Current Word Count	Read	0	0	1	0	1	0	1	0	W_0-W_7
			0	0	1	0	1	0	1	1	W_8-W_{15}
3	Base and Current Address	Write	0	1	0	1	0	1	0	0	A_0-A_7
			0	1	0	0	0	1	0	1	A_8-A_{15}
	Current Address	Read	0	0	1	0	0	1	0	0	A_0-A_7
			0	0	1	0	0	1	0	1	A_8-A_{15}
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	W_0-W_7
			0	1	0	0	0	1	1	1	W_8-W_{15}
	Current Word Count	Read	0	0	1	0	0	1	1	0	W_0-W_7
			0	0	1	0	0	1	1	1	W_8-W_{15}

Clear First/Last Flip-Flop: This command must be executed prior to writing or reading new address or word count information to the 8237A. This initializes the flip-flop to a known state so that subsequent accesses to register contents by the microprocessor will address upper and lower bytes in the correct sequence.

Master Clear: This software instruction has the same effect as the hardware Reset. The Command, Status, Request, Temporary, and Internal First/Last Flip-Flop registers are cleared and the Mask register is set. The 8237A will enter the idle cycle.

Clear Mask Register: This command clears the mask bits of all four channels, enabling them to accept DMA requests.

5.14.6 Programming

The 8237A will accept programming from the host processor any time that HLDA is inactive; this is true even if HRQ is active. The responsibility of the host is to assure that programming and HLDA are mutually exclusive. Note that a problem can occur if a DMA request occurs, on an unmasked channel while the 8237A is being programmed. For instance, the CPU may be starting to reprogram the two-byte Address register of channel 1 when channel 1 receives a DMA request. If the 8237A is enabled (bit 2 in the command register is 0) and channel 1 is unmasked, a DMA service will occur after only one byte of the Address register has been reprogrammed. This can be avoided by disabling the controller (setting bit 2 in the command register) or masking the channel before programming any other registers. Once the programming is complete, the controller can be enabled/unmasked. After power-up, it is suggested that all internal locations, especially the mode registers, be loaded with some valid value. This should be done even if some channels are unused. An invalid mode may force all control signals to go active at the same time.

5.14.7 Application Information

Figure 5.62 shows a convenient method for configuring a DMA system with the 8237A controller and an 8080A/8085AH microprocessor system. The multimode DMA controller issues a HRQ to the processor whenever there is at least one valid DMA request from a peripheral device. The definitions of register codes are given in Table 5.23.

When the processor replies with a HLDA signal, the 8237A takes control of the address bus, the data bus and the control bus. The address for the first transfer operation comes out in two bytes—the least significant 8 bits on the eight address outputs and the most significant 8 bits on the data bus. The contents of the data bus are then latched into an 8-bit latch to complete the full 16 bits of the address bus. The 8282 is a high-speed, 8-bit, three-state latch in a 20-pin package. After the initial transfer takes place, the latch is updated only after a carry or borrow is generated in the least significant address byte. Four DMA channels are provided when one 8237A is used.

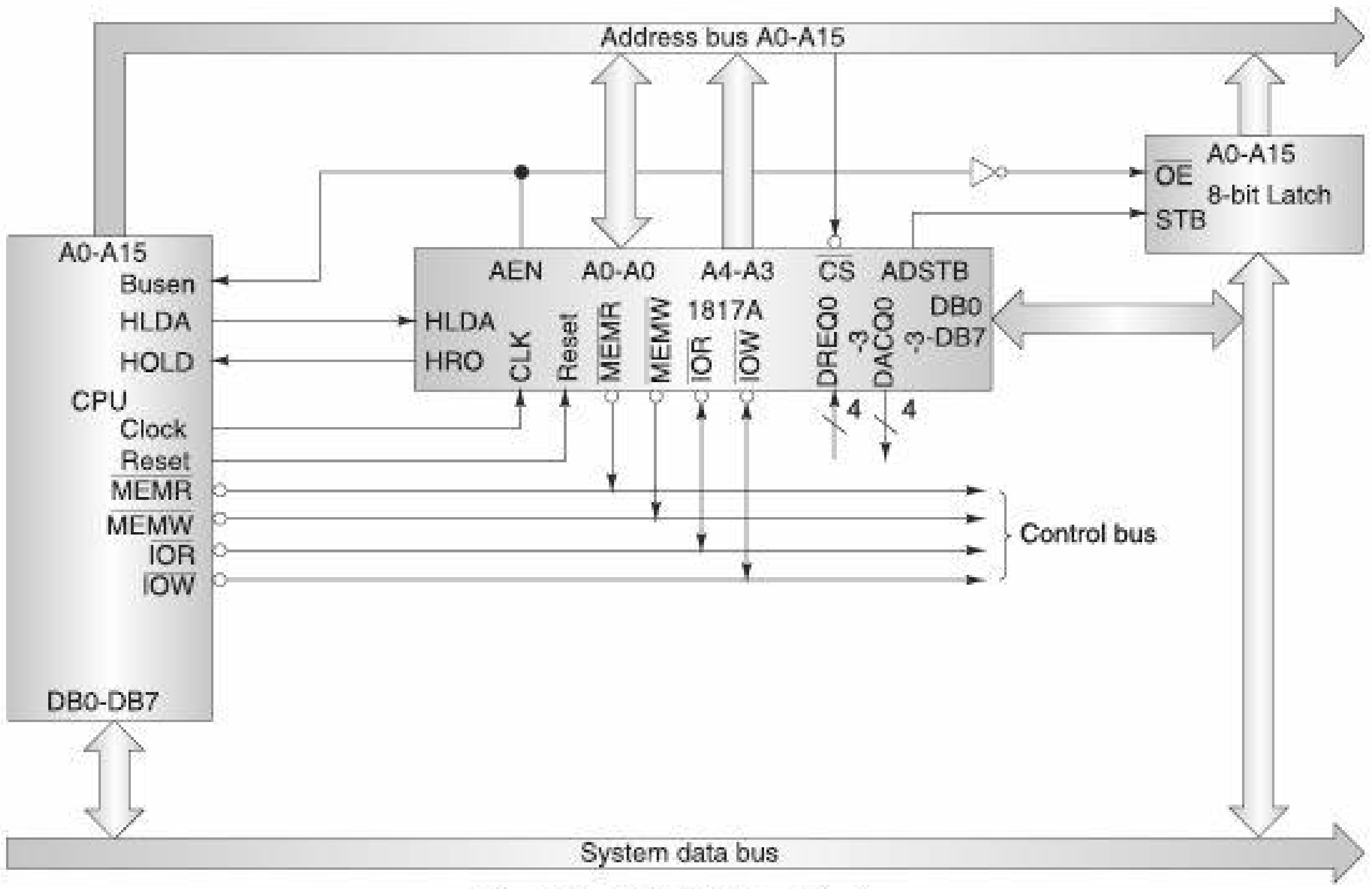


Fig. 5.62 8237A System Interface.

Table 5.23 Definition of Register Codes

Register	*Operation*	*Signals*						
		$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A_3	A_2	A_1	A_0
Command	Write	0	1	0	1	0	0	0
Mode	Write	0	1	0	1	0	1	1
Request	Write	0	1	0	1	0	0	1
Mask	Set/Reset	0	1	0	1	0	1	0
Mask	Write	0	1	0	1	1	1	1
Temporary	Read	0	0	1	1	1	0	1
Status	Read	0	0	1	1	0	0	0

5.15 PROGRAMMABLE DMA CONTROLLER — INTEL 8257

It is a device to transfer the data directly between IO device and memory without through the CPU. So it performs a high-speed data transfer between memory and I/O device.

The Intel 8257 is a 4-channel direct memory access (DMA) controller. It simplifies the transfer of data at high speeds for Intel's microcomputer systems. Its primary function is to generate a sequential memory address, upon a peripheral request, which will allow the peripheral to read or write data directly to or from the memory.

5.15.1 Features of Intel's 8257

- It is a 40 pin IC.
- The 8257 has four channels and so it can be used to provide DMA to four I/O devices.
- Each of the four channels of 8257 has a pair of two 16-bit registers, viz., DMA address register and Terminal Count Register.
- There are two common registers for all the channels; namely, Mode set register and Status register. Thus, there are a total of 10 registers. The CPU selects one of these 10 register using address lines A_0-A_3.
- Each channel can be independently programmable to transfer up to 64 kb of data by DMA.
- Each channel can independently perform read transfer, write transfer and verify transfer.
- The 8257 has chip priority resolver that resolves the peripherals requests (by resolving the priority of channels in fixed or rotating mode, explained later in the chapter) and provide a Hold request signal to CPU.
- Intel's 8257 represent a significant saving in component count for DMA based microcomputer system and greatly simplifies the transfer of data at high speed between the peripherals and memories.

5.15.2 Functional Description of 8257

When the 8257(DMA) device is connected to any I/O port device for data transfer, the 8257 is initialized by software and then transfers a block of data, containing upto 16,384 bytes. Without the intervention of CPU, data is directly transferred between memory and peripheral device. When 8257 receive a DMA transfer request from an enabled peripheral, it performs the following functions:

- It acquires the control of system bus.
- It provides the acknowledgement signal to that peripheral that is connected to highest priority channel.
- Then, outputs the least significant 8-bits of memory address onto the system address lines A_0-A_7 and the most significant 8-bits of memory address to the IO port device (on A_8-A_{15} lines) via the data bus.
- After addressing, the 8257 generates the appropriates memory and IO/read/write signals that cause the peripheral to receive or transfer a data byte directly from or to the memory addressed location.
- The 8257 will retain the control of system bus and repeat the transfer sequence, as long as the peripheral maintains its DMA request. When all the specified data is transferred by 8257, it activates its Terminal Count (TC) output signal, informs the CPU that the data transfer operation is completed.

5.15.3 Pin Description of 8257

The function of most of the pins of 8257 is similar to that of 8237A. The pin diagram of 8257 is shown in Figure 5.63.

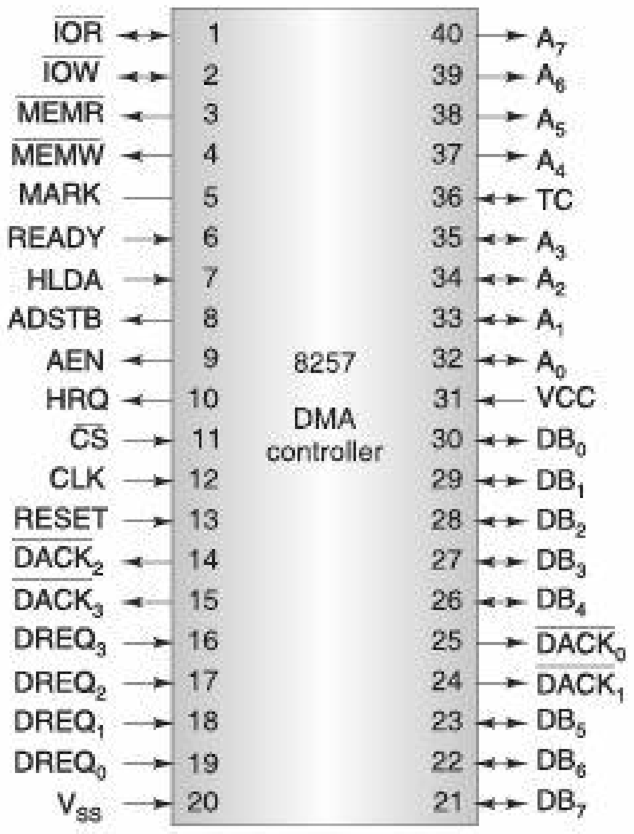


Fig. 5.63 Pin Diagram of 8257.

MARK-Pin no. 5: This output signal always occurs at 128th (or multiples of 128) cycles from the end of data block. It indicates the selected peripheral device that the current DMA cycle is the 128th cycle, since the previous MARK output.

TC (Terminal Count)-Pin no. 36: This output signal notifies the currently selected peripheral that the present DMA cycle should be the last cycle for this data block. TC is activated when the 14-bit value in the terminal count register of the selected channel is zero. The low-order 14-bits of terminal count register loaded with values 'n-1' where n is the desired number of DMA cycles.

$\overline{CS}$ (Chip Select): It is an active low input signal which enables the I/O Read or I/O Write input when the 8257 is being read or programmed in 'Slave mode'. $\overline{CS}$ is automatically disabled to prevent the chip from selecting itself while performing the DMA operation (in Master mode).

5.15.4 Block Diagram of 8257

The block diagram of 8257 is shown in Figure 5.64 (p. 404) and the explanation of each block is given as:

5.15.4.1 Channels (CH_0-CH_3)

The 8257 provides four separate channels. Each channel includes two 16-bit registers: a DMA Address register and other is a Terminal Count register. The DMA address register is loaded with the address of the first memory location to be addressed. In terminal count register, the low-order 14 bits of the value loaded in it, specifies the number of the DMA cycle minus one before the terminal count output is activated. The most significant two bits of terminal count register specify the type of DMA operation for that channel.

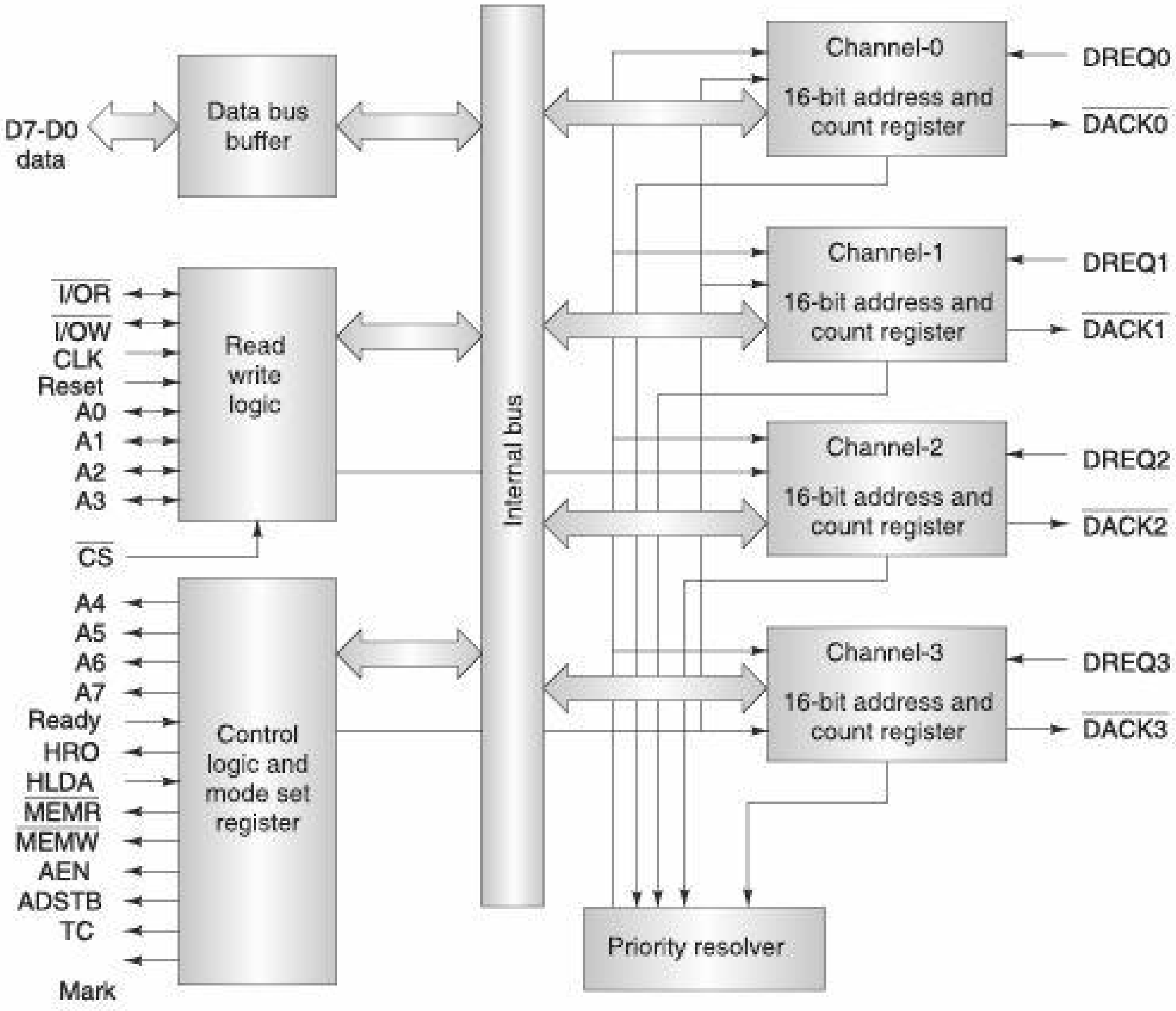


Fig. 5.64 Block Diagram of 8257.

Each channel accepts a DMA Request (DRQn) as Input and provides a DMA Acknowledgement (DACKn) output.

DRQ_0-DRQ_3 (DMA Request): These are the individual asynchronous channel request inputs used by the peripheral I/O devices to take DMA data transmission and obtain a DMA cycle. If fixed priority mode is used then the DRQ_0 has the highest priority and DRQ_3 has the lowest priority.

A request can be generated by raising the request line and holding it high until the DMA acknowledge. For multiple DMA cycle (Burst Mode), the request line is held high until the DMA acknowledge of last cycle arrives.

$DACK_0$-$DACK_3$ (DMA Acknowledge): These are the active low output signal and an active low level on these pins informs the peripheral connected to that channel that it has been selected for a DMA cycle. It acts as a chip select for the peripheral devices that are requesting service. These lines goes active low or high once for each byte transferred.

5.15.4.2 Data Bus Buffer

This is a tri-state, bidirectional, eight-bit buffer that interfaces the 8257 to the system data bus.

D_0-D_7 (Data lines): The eight bits of data for a DMA address register, a terminal count register or the mode set register are received on then data bus, when the 8257 is being programmed by the CPU. When the CPU read the DMA address register, TC register or the status register the data is sent to the CPU over the eight-bit data bus.

During the DMA cycle (8257 is the bus master), the 8257 outputs the most significant eight bits of memory address from one of the DMA address to latch via the data bus. These address bits will be transferred at the beginning of the DMA cycle and then the bus will be released to handle the memory data transfer during the balance of DMA cycle.

5.15.4.3 Read/Write Logic

When the DMA is in slave mode (during read operation), the CPU is programming or reading one of the 8257 registers then the read/write logic accepts the I/O Read or I/O Write signal. It also decodes the least significant four bits (A_0-A_3) and either writes the contents of data bus into address register (if I/OW = 1) or places the contents of address register onto the data bus (if I/OR = 1).

During the DMA cycle (when the 8257 is the bus master), the read/write logic generate the I/O read and memory write (DMA write cycle) or I/O write and memory read (DMA read cycle) signals which controls the data link with the peripheral that granted the DMA cycle.

Note. During the DMA cycle the Non-DMA I/O devices should be de-selected using the AEN signal.

5.15.4.4 Control Logic

This logic controls the sequence of operation during all DMA cycles by generating the appropriate control signals and the 16-bit address that specifies the memory location to be accessed as shown in Table 5.24.

Table 5.24 Addressing of the 8257 Registers

Channel	*Register*	*Address Bits*								*Hex-Address*
		Decoded Input and Enable				*Input to Address Pins*				
		A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Channel-0	DMA Address Register	0	1	1	0	0	0	0	0	60H
	Count Register	0	1	1	0	0	0	0	1	61H
Channel-1	DMA Address Register	0	1	1	0	0	0	1	0	62H
	Count Register	0	1	1	0	0	0	1	1	63H
Channel-2	DMA Address Register	0	1	1	0	0	1	0	0	64H
	Count Register	0	1	1	0	0	1	0	1	65H
Channel-3	DMA Address Register	0	1	1	0	0	1	1	0	66H
	Count Register	0	1	1	0	0	1	1	1	67H
Mode Set Register (Write Only)		0	1	1	0	1	0	0	0	68H
Status Register (Read Only)		0	1	1	0	1	0	0	0	68H

5.15.5 Operating Modes of 8257

Each channel of 8257 has two programmable 16-bit registers named as address register and count register. Address register is used to store the starting address of memory location for DMA data transfer. The address in the address register is automatically incremented after every read/write/ verify transfer. The count register is used to count the number of bytes or words transferred by DMA. The format of count register is,

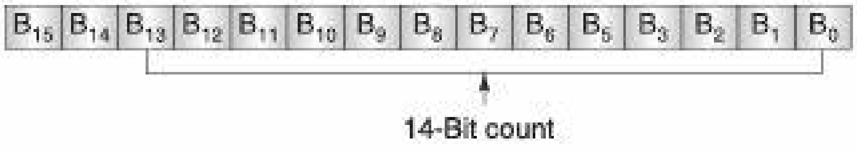

B_{15} B_{14} B_{13} B_{12} B_{11} B_{10} B_9 B_8 B_7 B_6 B_5 B_3 B_2 B_1 B_0
14-Bit count

Fig. 5.65 Count Register Format.

Bit (B_0-B_{13}): These 14 bits are used to count the value.

Bit (B_{14}-B_{15}): These 2 bits are used for indicate the type of DMA transfer (Read/Write/Verify transfer) as shown in table below:

Table 5.25 Description of Bits B_{14}-B_{15}

B_{15}	*B_{14}*	*Transfer Description*
0	0	Verify transfer
0	1	Write transfer
1	0	Read transfer
1	1	Illegal

- In read transfer the data is transferred from memory to I/O device.
- In write operation the data is transferred from I/O device to memory.
- Verification operations generate the DMA addresses without generating the DMA memory and I/O control signals.

5.15.6 Mode Set Register

The Mode set register is usually programmed after the DMA addressing and Terminal Count Register for one or more channels are initialized. It is an eight-bit register and its format is shown in Figure 5.66.

The use of mode set register is:

1. Enable/disable a channel
2. Fixed/rotating priority
3. Stop DMA on terminal count
4. Extended/normal writes time
5. Auto reloading of channel-2

Bit B_0-B_3: The bits B_0, B_1, B_2, and B_3 of mode set register are used to enable/disable channel-0, 1, 2 and 3 respectively. A high logic level (1) in these positions will enable a particular channel and a zero will disable it.

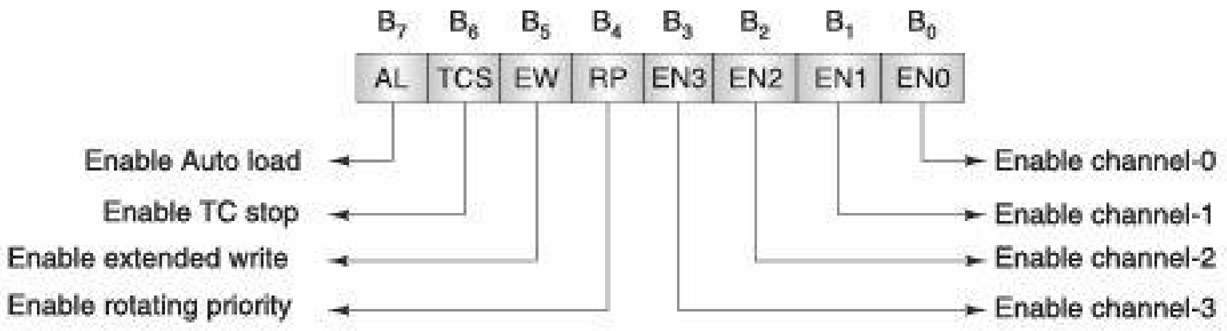


Fig. 5.66 Mode Set Register.

Bit B_4 (Rotating Priority): If the bit B_4 is set to one, then the channels will have rotating priority and if it is zero, then the channels will have fixed priority. In rotating priority after servicing a channel, its priority is made as lowest. And in fixed priority, the channel-0 has highest priority and channel-3 has lowest priority.

Bit B_5 (Extended Write): If the bit B_5 is set to one, then the timing of low write signals (MEMW and IOW) will be extended.

Bit B_6 (Terminal Count Stop): If the bit B_6 is set to one, then the DMA operation is stopped at the terminal count.

Bit B_7 (Auto-load): The bit B_7 is used to select the auto load feature for DMA channel-2. When bit B_7 is set to one, then the content of channel-3 count and address registers are loaded in channel-2 count and address registers respectively whenever the channel-2 reaches terminal count. When this mode is activated, the number of channels available for DMA reduces from four to three. Note that TC stop bit has no effect on channel 2, when the Auto Load bit is set. Channel 3 is still available to the user if the channel 3 enable bit is set, but the use of this channel will change the values to be auto loaded into channel 2 at update time.

5.15.7 Status Register

The format of status register of 8257 is shown in Figure 5.67. The eight-bit status register indicates which channels have reached a terminal count condition.

Bit B_0-B_3 (Terminal Count for Channel 0-4): The TC status bits (TC0-3) are set when the Terminal Count output is activated for that channel. These bits remain set until the status register is read or the 8257 is reset.

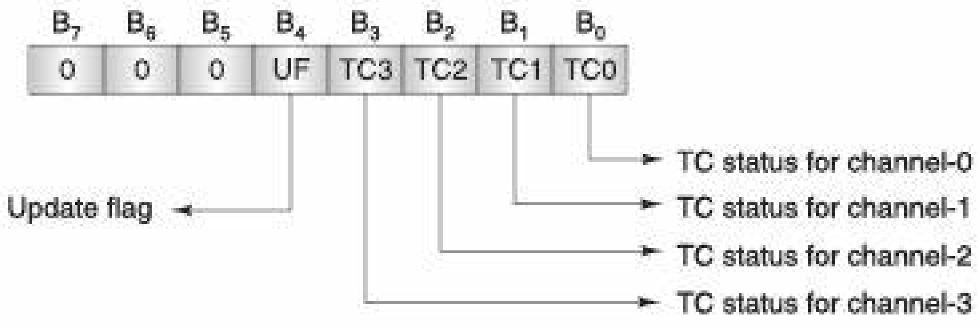


Fig. 5.67 Status Register.

Bit B_4 (Update Flag): A high logic level in this bit position indicates that channel-2 register has been reloaded from channel-3 register in the auto load mode of operation. The Update flag is not affected by a status register read operation and cleared by resetting the 8257 by resetting the Auto Load bit in the Mode Set register or it can be cleared itself at the completion of the update cycle. The purpose of Update flag is to prevent the CPU from in advertently skipping a data block by overwriting a starting address or terminal count in channel-3 registers before those parameters are properly auto-loaded into channel-2. DMA can inhibit by a hardware gate on the HRQ line or by disabling channels with a mode word before reading the TC status.

5.15.8 Priority of Channels

15.5.8.1 Rotating Priority

In Rotating Priority Mode, the priority of the channels has a circular sequence. After each DMA cycle, the priority of each channel changes. The channel which had just been serviced will have the lowest priority as shown in Table 5.26. If the Rotating priority bit is not set (set to zero), each DMA channel has a fixed priority.

Table 5.26 Priority of the Channels

Channel Just Serviced	*Priority Assignment*			
	Highest ←——→ Lowest			
CH-0	CH-1	CH-2	CH-3	CH-0
CH-1	CH-2	CH-3	CH-0	CH-1
CH-2	CH-3	CH-0	CH-1	CH-2
CH-3	CH-0	CH-1	CH-2	CH-3

15.5.8.2 Fixed Priority Mode

In the priority mode, channel 0 has the highest priority and channel 3 has the lowest priority. If the Rotating priority bit is set to one, the priority of each channel changes after each DMA cycle (not each DMA request). Each channel moves up to the next highest priority assignment, while the channel which has just been served moves to the lowest priority assignment.

5.16 INTERFACING ANALOG TO DIGITAL DATA CONVERTERS

In most of the cases, the peripheral I/O- 8255 is used for interfacing the analog to digital converters with microprocessor. We have already studied 8255 interfacing with 8086 as an I/O port, in previous section. In this section, we will only emphasize the interfacing techniques of analog to digital converters with 8255.

The analog to digital converter is treated as an input device by the microprocessor that sends an initializing signal to the ADC to start the analogy to digital data conversion process. The start of conversion signal is a pulse of specific duration. The process of analog 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 ADC to even a few hundred milliseconds in case of slow ADCs.

The available ADC in the market use different conversion techniques for conversion of analog signal to digitals. Successive approximation techniques and dual slope integration techniques are the most popular techniques used in the integrated ADC chip.

5.16.1 Steps for General Algorithm for ADC Interfacing

1. Ensure the stability of analog input, applied to the ADC.
2. Issue start of conversion pulse to ADC.
3. Read end of conversion signal to mark the end of conversion processes.
4. Read digital data output of the ADC as equivalent digital output.
5. Analog 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 analog signal and holds it constant for a specific time duration. The microprocessor may issue a hold signal to the sample and hold circuit.
6. If the applied input changes before the conversion process is over, the digital equivalent of the analog input calculated by the ADC may not be correct.

5.16.2 ADC 0808/0809

The analog to digital converter chips 0808 and 0809 are 8-bit CMOS, successive approximation converters. This technique is one of the fastest techniques for analog to digital conversion. The conversion delay is 100 μs 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 analog multiplexer so that at a time, eight different analog conversions can be done by using address lines.

Table 5.27 Addressing Bits for Analog Input Selection

Analog I/P select	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

Contd...

Contd...

I/P_5	1	0	1
I/P_6	1	1	0
I/P_7	1	1	1

ADD A, ADD B, ADD C. Using these address inputs, multichannel data acquisition system can be designed using a single ADC. The CPU may derive these lines using output port lines in case of multi-channel applications. In case of single input applications, these may be hard-wired to select the proper input. There are unipolar analog to digital converters, i.e., they are able to convert only positive analog input voltage to their digital equivalent. These chips do not contain any internal sample and hold circuit.

If one needs 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 analog inputs.

Vcc: Supply pins +5 V.

GND: GND

Vref +: Reference voltage positive +5 volts maximum.

Vref –: Reference voltage negative 0 volts minimum.

I/P_0–I/P_7: Analog inputs.

ADD A, B, C: Address lines for selecting analog inputs.

O_7–O_0: 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.

Example: 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 analog input is present at I/P_2 of the ADC and a clock input of suitable frequency is available for ADC.

Solution: The analog input I/P_2 is used and therefore address pins A, B, C should be 0, 1, 0 respectively to select I/P_2. The OE and ALE pins are already kept at +5 V 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 a 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 ALP is as follows:

```
MOV AL, 98H        ; Initialize 8255 as discussed above.
OUT CWR, AL
MOV AL, 02H        ; Select I/P2 as analog Input.
```

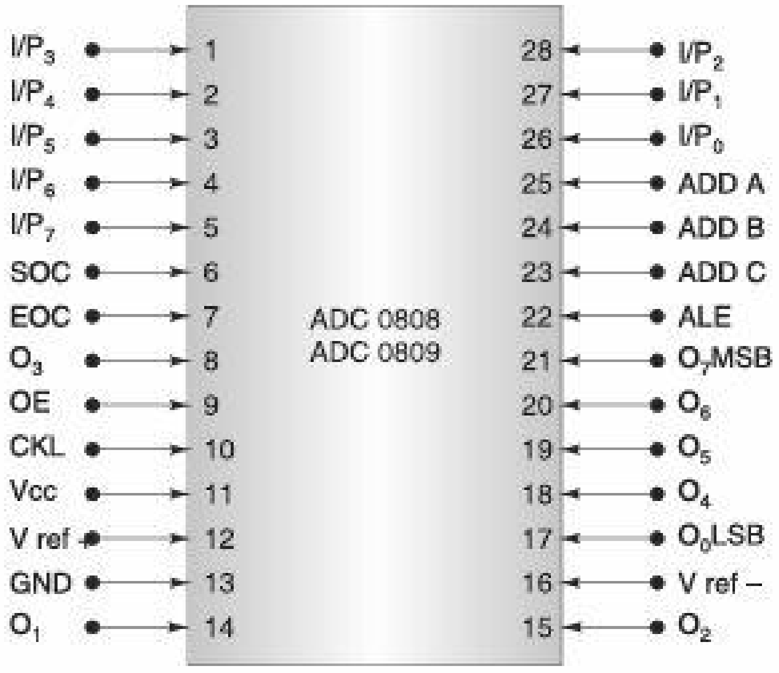


Fig. 5.68 Pin Diagram of ADC 0808/0809.

```
          OUT Port B, AL
          MOV AL, 00H        ; 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 Reading port C upper and
          RCR                ; Rotating through carry.
          JNC WAIT           ; Rotating through carry.
          IN AL, Port A      ; If EOC, read digital equivalent in AL.
          HLT                ; Stop.
```

5.17 INTERFACING DIGITAL TO ANALOG CONVERTERS

The digital to analog converters convert binary numbers into their equivalent voltages. The DACs find applications in areas like digitally controlled gains, motors speed controls, programmable gain amplifiers, etc.

AD7523 8-bit Multiplying DAC: This is a 16-pin DIP, multiplying digital to analog 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. The pin diagram of AD7523 is shown in Figure 5.69.

The supply range is from +5 V to +15 V, while Vref may be anywhere between −10 V and +10 V. The maximum analog output voltage will be anywhere between −10 V and +10 V,

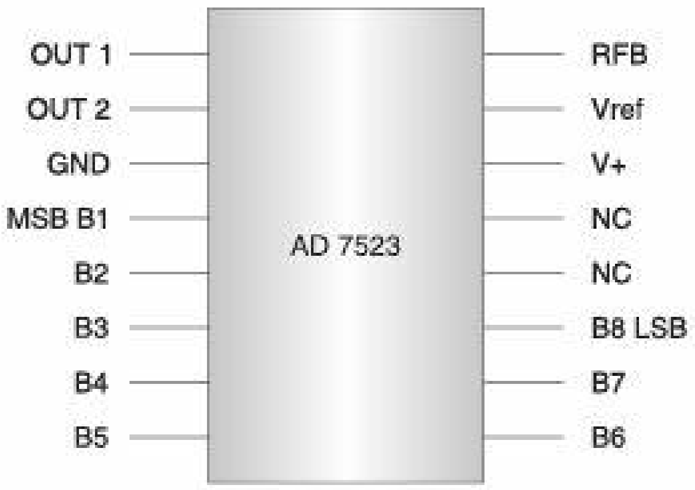


Fig. 5.69 Pin Diagram of AD7523.

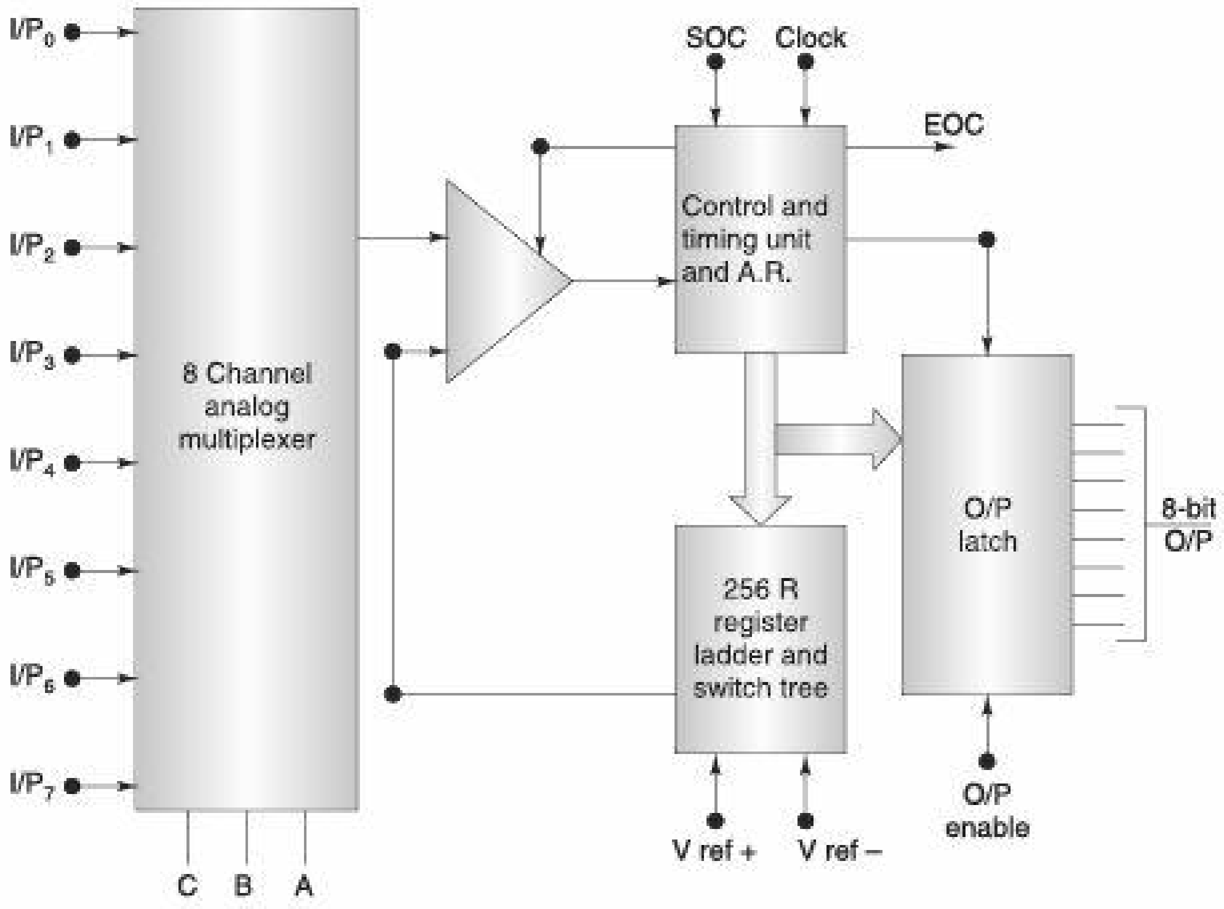

I/P0
I/P1
I/P2
I/P3
I/P4
I/P5
I/P6
I/P7
8 Channel analog multiplexer
C
B
A
SOC
Clock
EOC
Control and timing unit and A.R.
256 R register ladder and switch tree
V ref +
V ref –
O/P latch
8-bit O/P
O/P enable

Fig. 5.70 Block Diagram of ADC 0808/0809.

when all the digital inputs are at logic high state. Usually, a Zener diode is connected between OUT_1 and OUT_2 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 output 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.

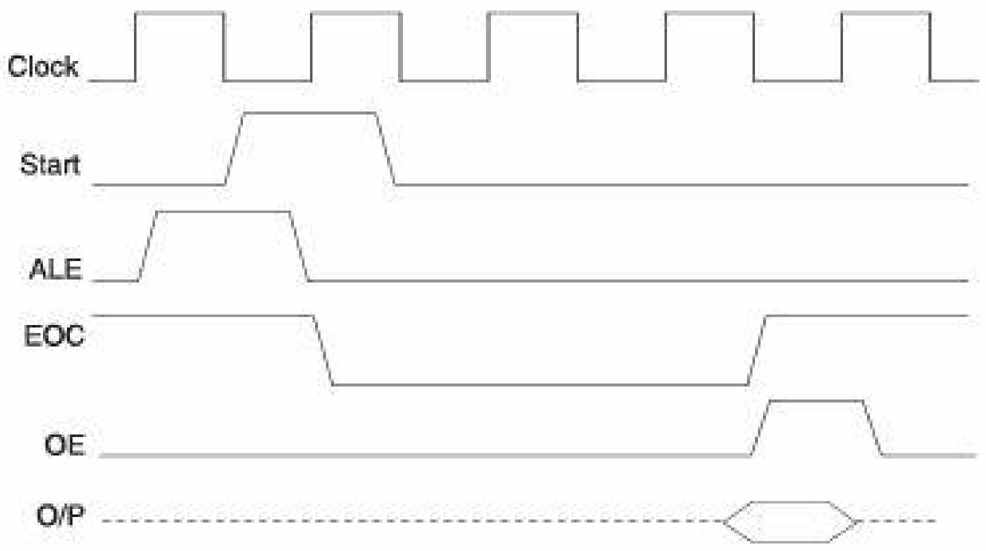


Fig. 5.71 Timing Diagram of ADC 0808.

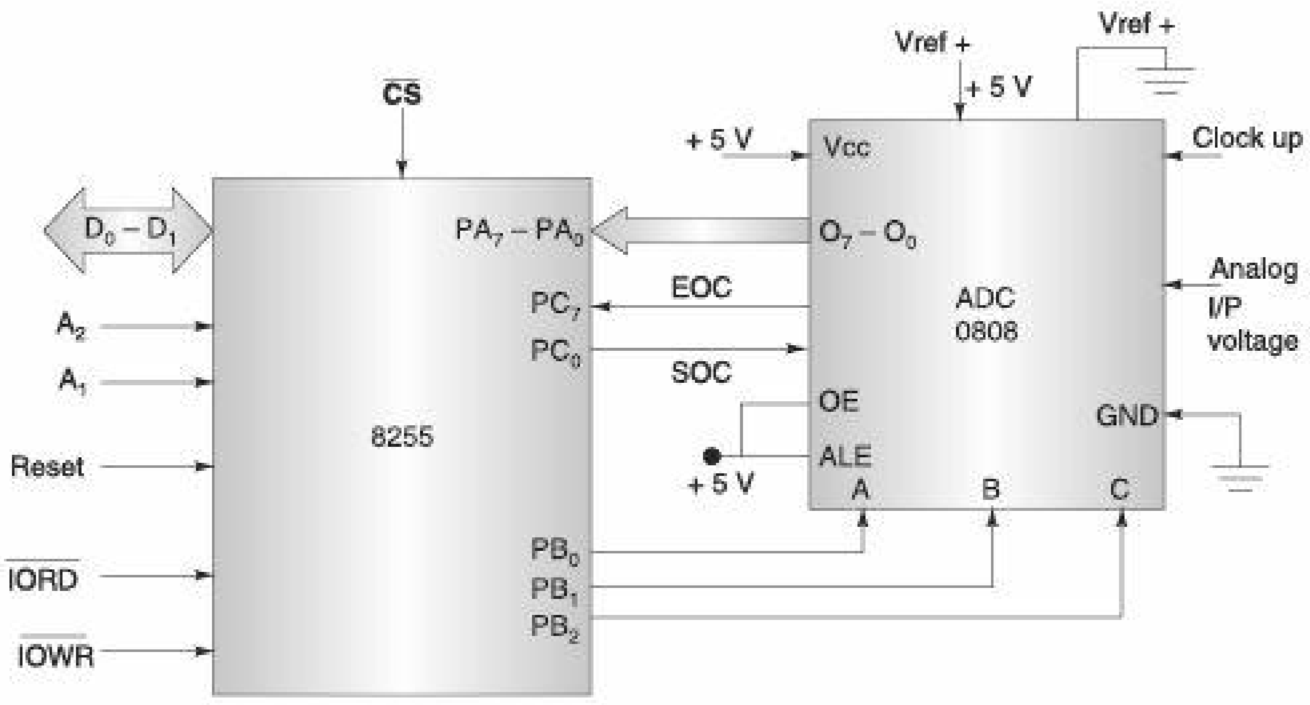


Fig. 5.72 Interfacing 0808 with 8086.

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

Solution: Figure 5.73 shows the interfacing circuit of AD 74523 with 8086 using 8255.

Program gives an ALP to generate a sawtooth waveform using circuit is:

```
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
```

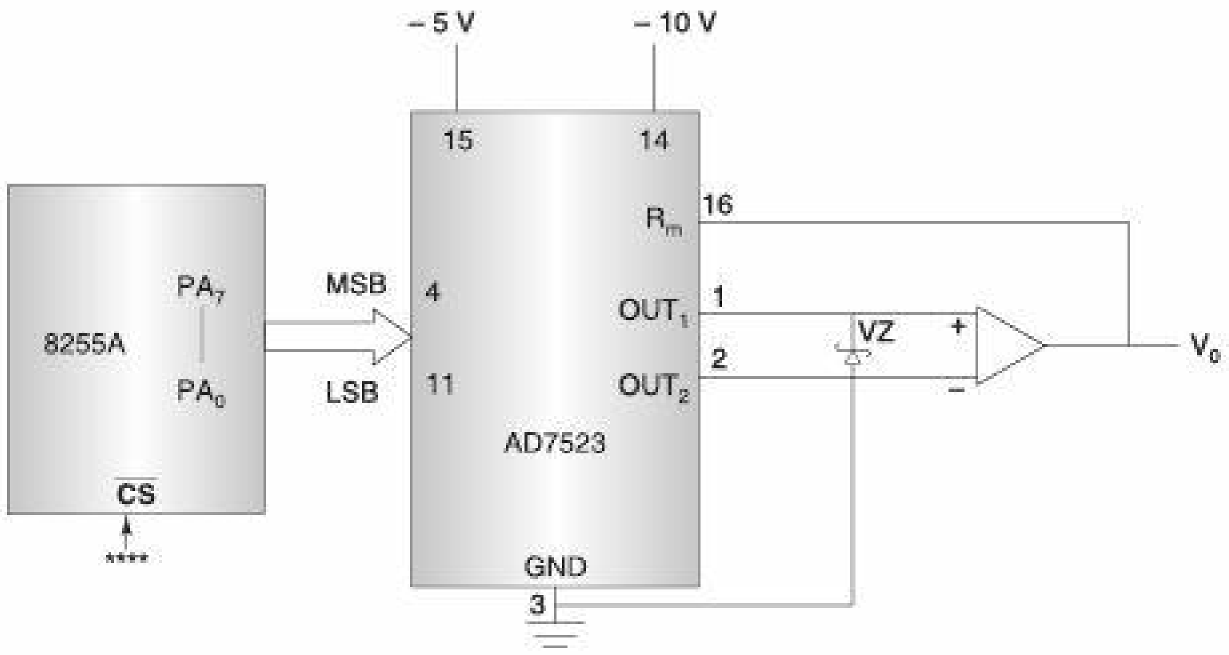


Fig. 5.73 Interfacing of AD7523.

```
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 0 V (analog), 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 sawtooth wave again starts from 00H, i.e., 0 V (analog) and the procedure is repeated.

The ramp period given by this program is precisely 1.00062 ms. Here the count F2H has been calculated by dividing the required delay of 1 ms 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.

Things to Remember

◊ A microprocessor when combined with memory and input/output devices forms a microcomputer.

◊ In I/O mapped I/O, device is identified with a unique device number and data is transferred through IN/OUT instruction.

◊ In memory-mapped I/O, each device is identified with 16-bit address interface control signals to proper handshaking signals, logic for decoding address that appears on the bus.

◊ Handshaking signals are used to determine the direction in which transfer has to take place whether from CPU or to CPU.

◊ In Synchronous Data Transfer scheme, there is synchronization between the device which sends data and the device which receives the data with the same clock input.

◊ Asynchronous Data Transfer scheme is used when the speed of an I/O device does not match the speed of the microprocessor, and the timing characteristic of I/O device is not predictable.

◊ In Interrupt Driven Data Transfer scheme, the microprocessor initiates an I/O device to get ready, and then it executes its main program instead of remaining in a program loop to check the status of the I/O device.

◊ In DMA Data Transfer scheme, CPU does not participate and data is directly transferred from an I/O device to the memory or vice versa. The data transfer is controlled by the I/O device or a DMA controller.

◊ The interfacing circuit converts the data available from an input device into compatible format for the computer.

◊ The Intel 8085 uses a 16-bit wide address bus for addressing memories and I/O devices. Using 16-bit wide address bus it can access 216 = 64 K bytes of memory and I/O devices.

◊ In memory-mapped I/O scheme, there is only one address space which is defined as the set of all possible addresses that a microprocessor can generate.

◊ For the control purpose, 24 lines of I/O ports are divided into two groups, namely, Group A and Group B.

◊ The group A contains the Port A and the Port Cupper. The Group B contains the Port B and the Port Clower.

◊ Control word is written into the control word register which is within 8255. No read operation of the control word register is allowed.

◊ The Intel 8279 is a programmable keyboard interfacing device. Data input and display are the integral part of microprocessor kits and microprocessor-based systems.

◊ A parallel port is that type of socket which is found on personal computers for interfacing with various peripherals. It is also known as a printer port or Centronics port.

◊ RS-232 stands for Recommend Standard number 232 and C is the latest revision of the standard.

◊ A universal asynchronous receiver/transmitter (usually abbreviated UART) is a type of "asynchronous receiver/transmitter", a piece of computer hardware that translates data between parallel and serial interfaces.

◊ Mode instruction is used for setting the function of the 8251. Mode instruction will be in "wait for write" at either internal reset or external reset. That is, the writing of a control word after resetting will be recognized as a "mode instruction."

Questions and Answers

1. **What is interfacing?**

   An interface is a shared boundary between the devices which involves sharing information. Interfacing is the process of making two different systems to communicate with each other.

2. Give the applications of 8251 programmable communication interface chip.

1. 8251 can be used to transmit receive serial data transmission to a CRT terminal, using the 8251 in status check mode 2.
2. A programmable chip designed for synchronous/asynchronous serial data communication.

3. What is DMA data transfer?

DMA implies direct memory access. Either in programmed I/O or interrupt I/O transfers, the data between the I/O devices or external memory is via the accumulator. For voluminous data transfer, there is the time commuting and even through the I/O devices speed matches with speed of UR, so there is direct transfer of data directly between the I/O device and external memory without going through accumulator. This is called DMA.

4. Why the number of O/P ports in peripheral mapped I/O is restricted to 256?

In I/O addressing mode, 8085 has capability of 8 bit I/O address through which it can address 255 I/O ports.

5. What is the need of Direct Memory Access Technique?

Need for DMA: In the applications where the microprocessor/CPU is to transfer bulk data, it may be a waste of time to transfer the data from source to destination using program controlled data transfer or Interrupt driven data transfer.

- The alternate way of transferring the bulk data is the Direct Memory Access (DMA) technique in which the data is transferred under the control of a DMA controller, after it is properly initialized by the microprocessor/CPU.
- A DMA controller is designed to complete the bulk data transfer task much faster than the CPU.
- A DMA controller may also be used to transfer data from the system memory to a video RAM of a CRT display.
- Data transfer between CPU & FDC may be controlled using DMA controller.

6. Name the two modes used by the DMA processor to transfer data.

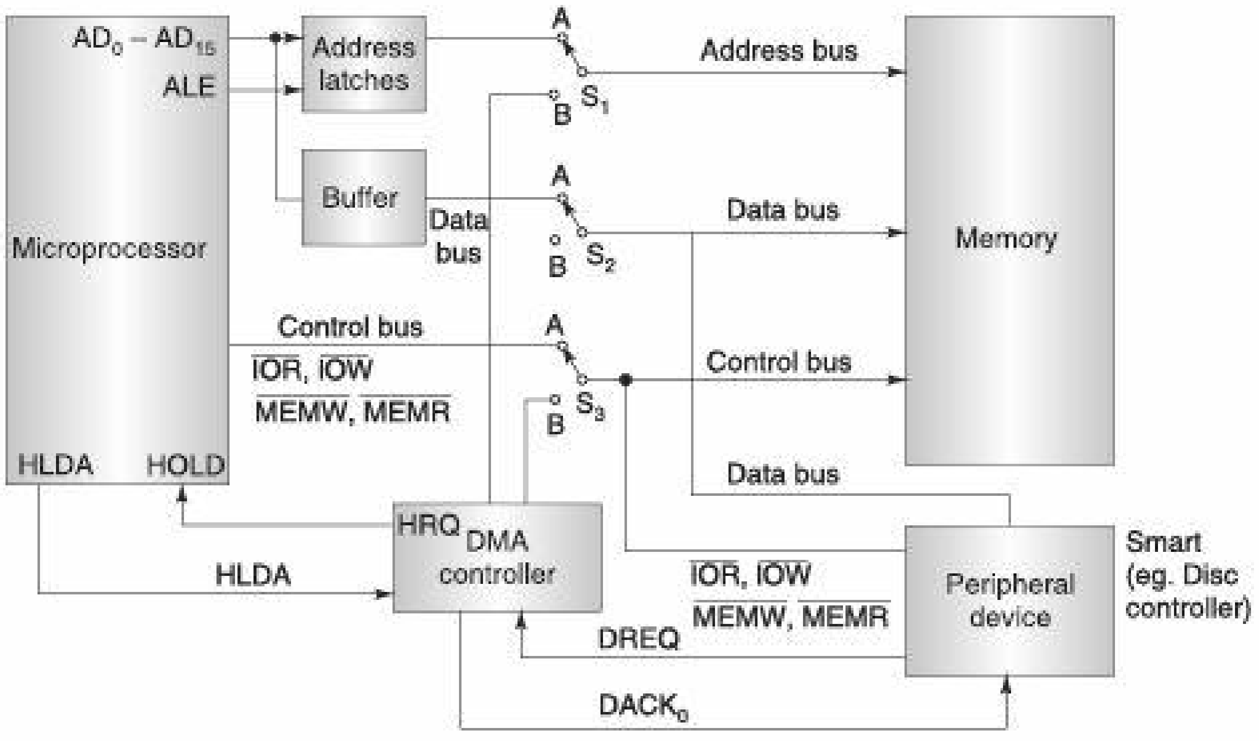
1. Burst or Block Transfer DMA
2. Cycle Steal or Single Byte Transfer DMA.

7. Discuss DMA controller briefly.

In I/O data transfer data is transferred by using microprocessor. The microprocessor will read data from I/O device and then will write data to memory.

- In this case there are two operations for single data transfer.
- If the data is less, then micro process will not waste its time; transferring data from I/O to memory or back. But suppose, data is huge, then the transfer rate from I/O to memory or back will slow down because of microprocessor intervention. In such case, to speed up the process of transferring the data, we can think, Can I/O has direct access to memory and the answer is, yes.
- It can have Direct memory access (DMA), but under Supervision. The device which supervises, data transfer is named as DMA controller.

Now let's have diagrammatic representation of the scheme, which depicts microprocessor, DMA controller, memory and I/O device.



DMA controller scheme

8. List the features of 8251.

Features of 8251

1. It supports both synchronous and asynchronous modes of operation.
2. The synchronous baud rate — DC to 64 K baud.
3. The asynchronous baud rate — DC to 19.2 K baud.
4. The synchronous mode supports 5-8 bits characters.
5. In asynchronous mode it supports 5-8 bits characters.
6. It contains full duplex system.

9. Differentiate UART and USART in data transfer operation and give advantages of USART.

UART (8250): UART stands for Universal Asynchronous Receiver and Transmitter. But nowadays, we generally used USART.

USART (8251): USART is Universal Synchronous/Asynchronous Receiver and Transmitter. Mostly we used USART, because USART supports both synchronous as well as synchronous data transfer operation. On the other hand, UART supports only asynchronous communication.

Advantage of USART

1. 8251 is mostly used for serial communication.
2. It is sending data out, accepting data bits, generation/removing other signals, etc.
3. The 8251 (USART) will convert parallel data into serial stream and transmit on serial output line. But in case of 8250 (USART) is done in parallel mode only.
4. At the same time it can receive serial data on serial input line, converts it to parallel from then transfer to processor.

10. What is the importance of RS232-C in serial communication? Name some application where you see its use.

RS-232 stands for Recommend Standard number 232 and C is the latest revision of the standard. The serial ports on most computers use a subset of the RS-232C standard. The full RS-232C standard specifies a 25-pin "D" connector of which 22 pins are used. Most of these pins are not needed for normal PC communications, and indeed, most new PCs are equipped with male D type connectors having only 9 pins. In the world of serial communications, there are two different kinds of equipment:

- DTE - Data Terminal Equipment
- DCE - Data Communications Equipment

11. What do you mean by "Baud Rate"?

The rate at which the data bits are transmitted known as Baud Rate. It is expressed in bits/second. It is necessary to determine because it defines the speed of data transmission and also called as data transmission rate. In serial communication, one bit is transmitted at a time; therefore, how long the bit stays on or off is determined by the speed at which the bits are transmitted.

12. Give the different types of command words used in 8259A?

The command words of 8259A are classified in two groups.

1. Initialization command words (ICWs)
2. Operation command words (OCWs)

13. Write the operating modes of 8259A?

Operating modes of 8259A are:

(a) Fully Nested Mode
(b) End of Interrupt (EOI)
(c) Automatic Rotation
(d) Automatic EOI Mode
(e) Specific Rotation
(f) Special Mask Mode
(g) Edge and level Triggered Mode
(h) Reading 8259 Status
(i) Poll command
(j) Special Fully Nested Mode
(k) Buffered mode
(l) Cascade mode.

Exercise

1. What do you mean by interface? What should be qualities of a good interface?
2. What is difference between peripheral and memory interfaces?
3. Explain block diagram and working of 8255 PPI. Also explain its working.
4. Explain block diagram and working of 8279 keyboard interface. Also explain its working.

5. Explain working of Centronix parallel interface.
6. Explain block diagram and working of 8251 communication interface. Also explain its working.
7. Explain pin diagram of RS 232 interface.
8. What is difference between asynchronous and synchronous transmission of data and which one is better?
9. Explain the architecture of 8255.
10. Explain mode 2 operation of 8255.
11. What do you mean by DTE & DCE?
12. Write a note on USARTS.
13. What do you mean by baud rate? What is difference between baud rate and bit rate?
14. Define the formats for control register for 8251.
15. Write a note on special purpose interfacing devices: programmable CRT controller, floppy disk controllers, hard disk controllers, dot matrix printer, memory controllers.
16. What is programmable peripheral device?
17. What is synchronous and asynchronous data transfer schemes?
18. What are tasks involved in keyboard interface?
19. How a keyboard matrix is formed in keyboard interface?
20. List the features of Intel 8259?
21. Explain why the number of output ports in the peripheral mapped I/O is restricted to 256 ports.
22. Discuss burst mode of operation of DMA controller.
23. What are the different operating modes of 8255?
24. Calculate the resolution of a 12-bit D/A converter.
25. Explain Keyboard/Display controller with a neat block diagram.

Chapter 6

Microprocessor Applications

6.1 INTRODUCTION

Microprocessors are widely used in instrumentation and control applications. Microprocessor based systems are suitable for dedicated applications in industries such as process control, control of machines and equipment, instrumentations, and so on. A microprocessor-based system (being very small and compact) forms a part of the equipment which is being controlled. Employing multiplexers and demultiplexers, it can perform the task of measurement and automatic control of several physical/electrical quantities or of a few quantities at different points. Nowadays microprocessor-based systems are replacing electromechanical logic. They have numerous applications such as in measurement, display and control of current, voltage, kW, kWh, kVA, kVAR, power factor, phase angle, frequency, temperature, stress, strain, pressure, force, displacement, deflection, vibration, water level, etc.; communication, transportation, traffic control, lift control, military equipment, industrial tool control, robotics, protective relays, excitation and voltage control of generators, and so on. For the measurement and control of physical quantities such as temperature, speed, displacement, etc. transducers are used, which give electrical voltage proportional to physical quantities. The electrical voltage which is obtained as an output of a transducer is an analog (continuous) quantity. It must be converted into digital quantity by an A/D (Analog to Digital) converter before it is applied to a microprocessor. A microprocessor, being very fast can measure, process and control many signals one by one in a short time. To handle multiple signals, an analog multiplexer is employed. An A/D converter, analog multiplexer, sample and hold IC, etc. form a data acquiescing system. In a microprocessor-based data acquisition system, these components operate under the control of a microprocessor.

In this chapter measurement, display and control of some electrical and physical quantities will be discussed. Before the discussion of such quantities, generation of time-delay and interfacing of 7-segment displays will be described. Also interfacing of these devices to microprocessor has been discussed. At the end of this chapter, the interfacing of D/A converter has also been discussed. D/A converter is not a component of data acquisition system but it forms a part of hardware circuitry of many A/D converters. Being cheaper it is also used as an A/D converter.

6.1.1 Applications of Microprocessors

Microprocessors are applicable to a wide range of information processing tasks, ranging from general computing, up to real-time monitoring systems. The microprocessor facilitates new ways to communicate and makes use of the vast information available to us online and offline. Most electronic devices—including everything from computers, remote controls, washing machines, microwaves and cell phones to iPods and more—contain a built-in microprocessor.

(a) **General Computing:** Microprocessors commonly used in general computing tasks include those embedded in your laptop or desktop computers. These microprocessors are responsible for the core computing processes of your computer, such as calculation and data transfer.

(b) **Signal Processing:** Microprocessors are also used as "signal processors" for decoding radio signals and digital signals, such as those used in digital TV sets. Moreover, microprocessors used in signal processing allows people, from different parts of the world, to talk and see each other via their computer screens using small video cameras, which are usually embedded with microphones.

(c) **Real-Time Computing:** In real-time computing systems, microprocessors are embedded in security devices such as the anti-lock braking system (ABS) that are widely used in modern automobiles. The microprocessor detects motions and changes, that are relative to the surrounding or environment of the security device and sends signals that correspond to the changes that it detected.

6.2 SEVEN-SEGMENT LED DISPLAY

The seven-segment LED display is a multiple display. It can display all decimal digits and some letters. It is very popular among multiple displays as it has the smallest number of separately controlled light emitting diodes (LED). Multiple displays of 7-segment LED, 14-segment LED and dot matrix type are available. These displays give better representation of alphanumeric characters be enquired complex circuitry.

In seven-segment display, there are seven light emitting diodes (LED) as shown in Figure 6.1. Each LED can be controlled separately. To display a digit or letter the desired segments are made ON as shown in Figure 6.2.

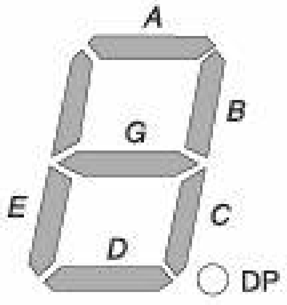


Fig 6.1 Seven-segment Display.

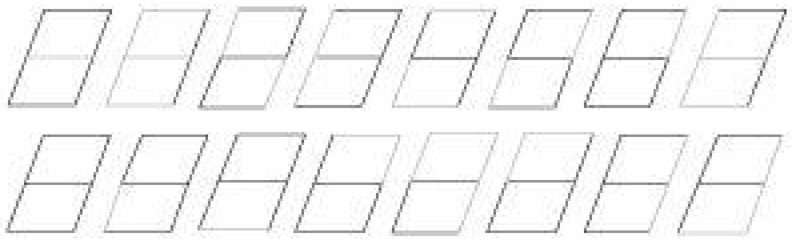


Fig. 6.2 Hexadecimal Digits using 7 Segments.

There are two types of 7-segment display, namely, common cathode type and common anode type as shown in Figure 6.3(a) and (b). When a +5 V d.c. is applied to any segment, the corresponding diode emits light. Thus applying logic '1', i.e. positive logic to the desired segments, the desired letter or decimal number can be displayed. In a common-anode type display all the 7 anodes are tied together and connected to +5 V supply as shown in Figure 6.3(b) particular segment will emit light when 0 logic is applied to it.

The seven-segment displays are not connected to I/O ports directly. They are connected through buffers or drivers/decoders. 7446A, 74L46, 7447, 74LS47 and 74LS47 are decoders/

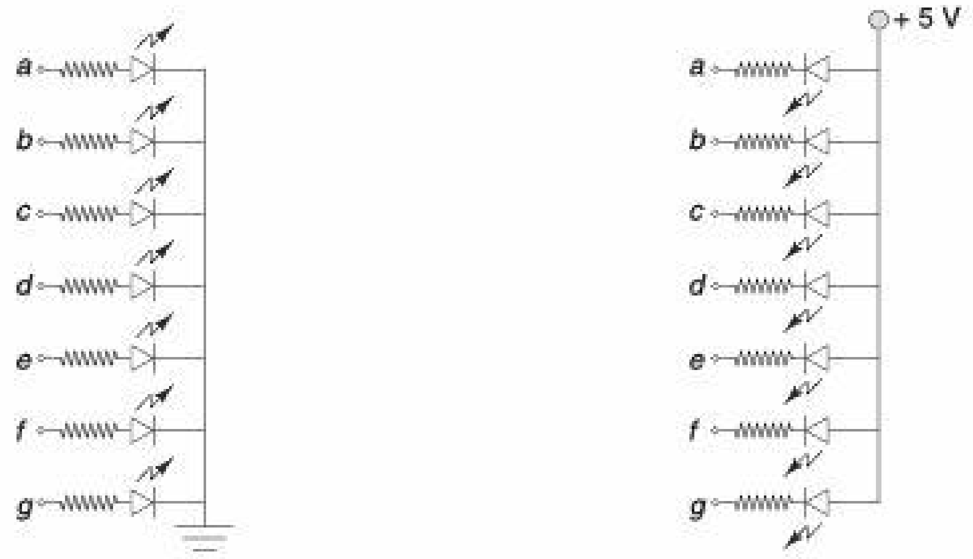


Fig. 6.3(a) Common Cathode 7-Segment Display **(b)** Common Anode 7-Segment Display.

drivers for common anode type seven-segment displays. 7448, 74LS48, 74LS48, 7449 and 74LS49 are decoders/drivers for common cathode type seven-segment displays. Figure 6.4 shows the pin configuration of 74LS48. In Figure 6.3(b) particular segment will emit light when 0 logic is applied to it.

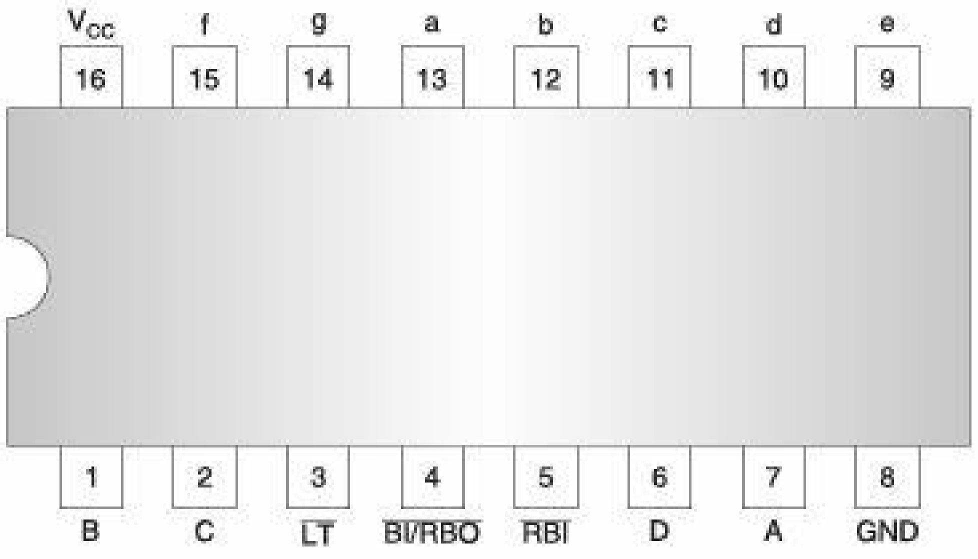


Fig. 6.4 Pin Diagram of 74LS48.

BCD to 7-segment decoder/driver.

FND 500 and FND 503: FND 500 and FND 503 are common-cathode 7-segment displays. Figure 6.5 shows the pin configuration of FND 503.FND 507/510. FND 507 and FND 510 are common-anode 7-segment displays.

MAN 74 A: MAN 74A is a common cathode 7-segment display. The pin configuration of MAN 74A is shown in Figure 6.5. The interfacing of decoder/driver 74LS48 and 7-segment display MAN 74A with microprocessor is shown in Figure 6.7. Table 6.1 is a functional table for 74LS48. The program to display the decimal number is as follows:

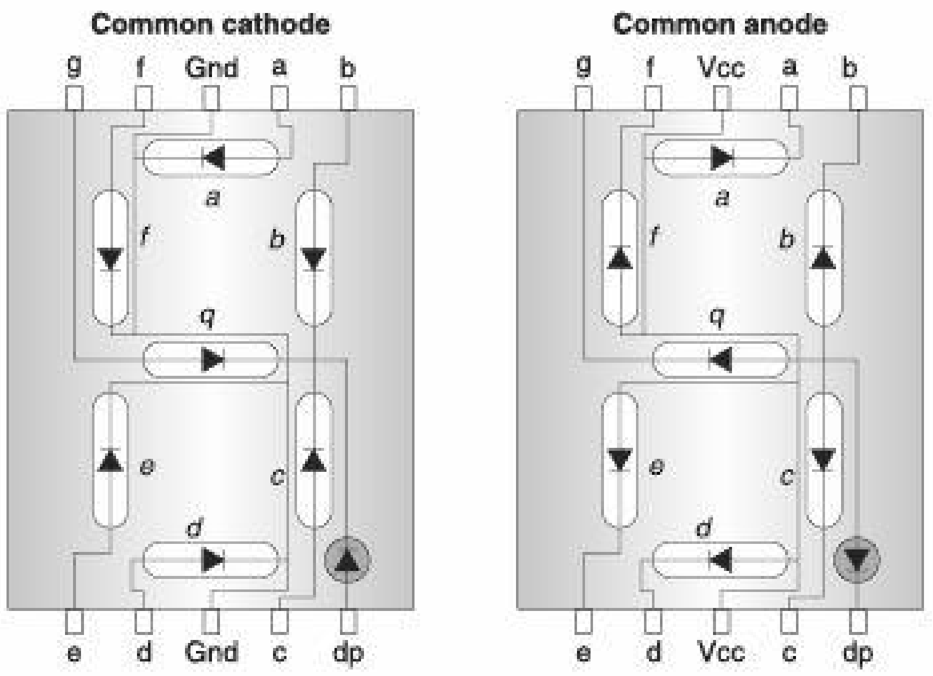


Fig 6.5 Pin Configuration of FND 503.FND 507/510.

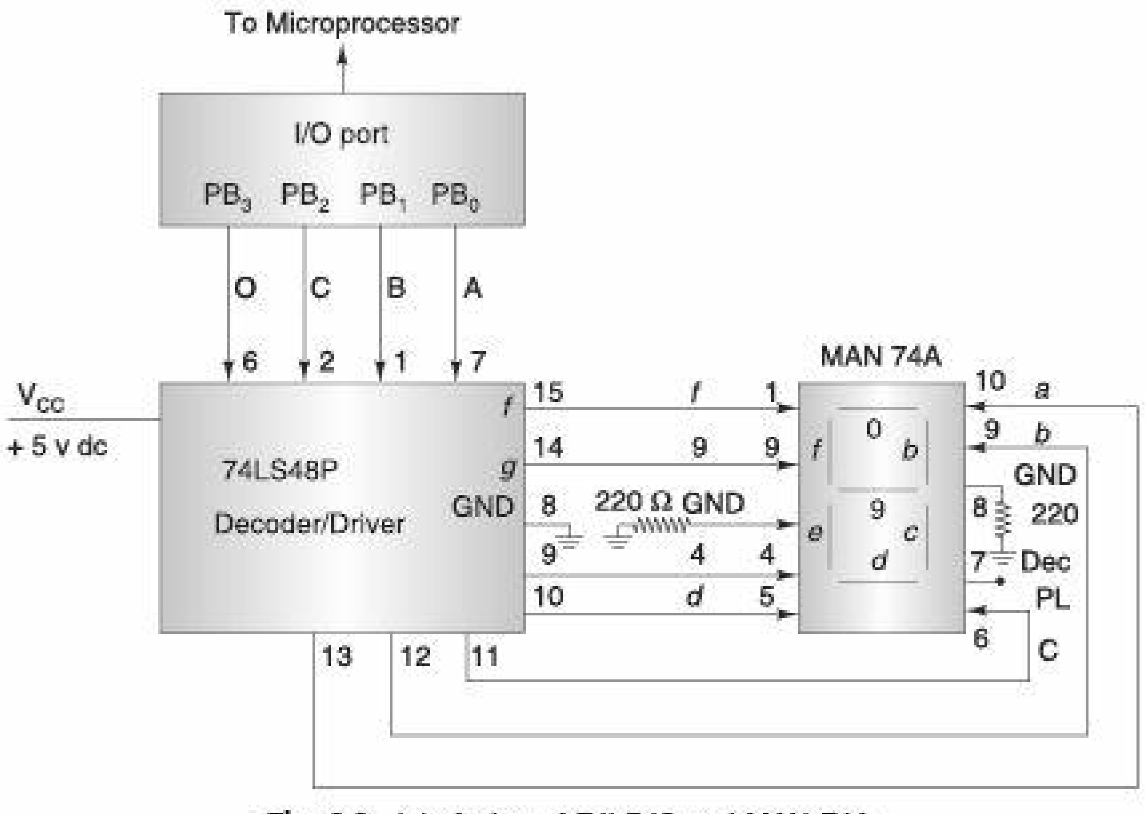


Fig. 6.6 Interfacing of 74LS48 and MAN 74A.

Example 6.1: Write a program to display 5.

Program:

Memory Address	Machine code	Mnemonics	Operands	Comments
2400	3E, 98	MVI	A, 98H	Get control word
2402	D3, 03	OUT	03	Initialize I/O ports
2404	3E, 05	MVI	A,05	Get 05 in accumulator
2406	D3, 01	OUT	01	Send 05 at port B
2408	76	HALT		Stop

The control word 98H makes the Port B an output port. The microprocessor outputs 05 at the Port B. The pins PB0-PB3 of the port B are connected to the decoder/driver 74LS48. Thus, the binary bits corresponding to the decimal number 5 are applied to 74LS48 and it is displayed by the 7-segment display. 0 is the MSB of the decimal number 05. The logic for 0 is output on the pins PB4-PB7. These pins are not connected anywhere and consequently 0 is not displayed. By using 2 decoders/drivers and 7-segment display we can show 2 digits on screen, No. 1 display will display 5 and No. 2 display will show 0. Thus, two units of display will display 2 desired decimal numbers. Suppose, we want to display 89. For this purpose, in the above program the data of the memory location 2405 is changed to 89. Now the microprocessor will output 89 at Port B and 1st unit will display 9 and the 2nd unit 8.

Table 6.1 Functional Table for 74LS48

	INPUTS				OUTPUTS						
DECIMAL	D	C	B	A	a	b	c	d	e	f	g
0	L	L	L	L	H	H	H	H	H	H	L
1	L	L	L	H	L	H	H	L	L	L	L
2	L	L	H	L	H	H	L	H	H	L	H
3	L	L	H	H	H	H	H	H	L	L	H
4	L	H	L	L	L	H	H	L	L	H	H
5	L	H	L	H	H	L	H	H	L	H	H
6	L	H	H	L	L	L	H	H	H	H	H
7	L	H	H	H	H	H	H	L	L	L	L
8	H	L	L	L	H	H	H	H	H	H	H
9	H	L	L	H	H	H	H	L	L	H	H

Note. Pins LT, RBO and RBI may not be used. In case, the seven-segment display is not working properly, it may be tried with keeping LT, RBO and RBI high. If RBI, A, B, C and D are low and LT is high, no segment glows and RBO goes low. If RBO is kept open or made high and LT low, all segments glow. If inputs A, B, C and D correspond to the decimal number 10 or more, unknown figures are displayed.

Seven-Segment Interface with 8085

This section explains the interfacing of common cathode 7 segment in the I/O board. The 7 segments are connected through 8-bit latch (74LS374) and a resistor (range from 100 Ω to 220 Ω) in series to each segment.

The seven-segment display is connected with microprocessor through programmable I/O port 8255. D0-D7 connected to Port A, SSEG to PC0 and C0 to PC1.

Common cathode display pattern from displaying decimal numbers

Digital	A	B	C	D	E	F	G	DP	Hex code
0	1	1	1	1	1	1	0	0	0xFC
1	0	1	1	0	0	0	0	0	0x60
2	1	1	0	1	1	0	1	0	0xDA
3	1	1	1	1	0	0	1	0	0xF2
4	0	1	1	0	0	1	1	0	0x66
5	1	0	1	1	0	1	1	0	0xB6
6	1	0	1	1	1	1	1	0	0xBE
7	1	1	1	0	0	0	0	0	0xE0
8	1	1	1	1	1	1	1	0	0xFF
9	1	1	1	1	0	1	1	0	0xF6

Program Code:

```
PORTA: EQU 80H
PORTB: EQU 81H
PORTC: EQU 82H
CTRLPORT: EQU 83H
          ORG 2000H
          LXI SP, 3FF0H
          MVI A, 80H
          OUT CTRLPORT
START:    MVI B, 10
          LXI H, TABLE
REPEAT:   MOV A, M
          OUT PORTA
          MVI A, 00000000B
          OUT PORTC
          MVI A, 00000001B
          OUT PORTC
          CALL DELAY
          INX H
          DCR B
          JNZ REPEAT
          JMP START
DELAY:    PUSH B
          MVI B, 255
LOOP:      MVI C, 255
LOOP1:    DCR C
          JNZ LOOP1
```

```
              DCR B
              JNZ LOOP
              POP B
              RET
TABLE: DFB 0xFC,0x60,0xDA,0xF2,0x66,0xB6,0xBE,0xE0,0xFF,0xF6
```

To display alphanumeric characters, the segments of a 7-segment LED are controlled independently. Buffers are used to interface 7-segment displays to the microprocessor. Hex buffer 7407 can be used for this purpose. The alphanumeric characters are also displayed by proper interfacing and programming. Figure 6.7 shows the pin diagram of 7407.

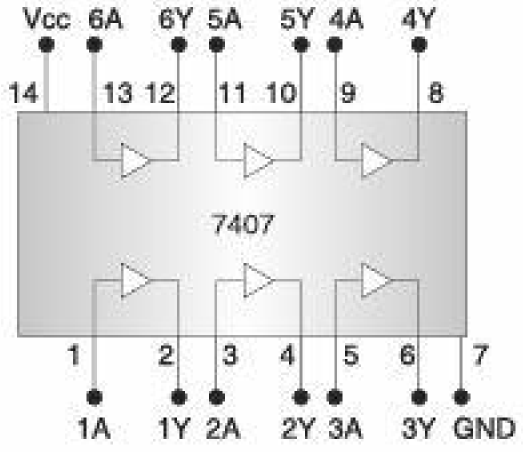


Fig. 6.7 The Pin Diagram of 7407.

6.2.1 Multiple Digit Display

Small number of 7-segment LED displays is interfaced to microprocessor through BCD to 7-segment decoder/driver and I/O ports as discussed in previous sections. When the number increases, the earlier discussed techniques will not be economical. For example, to display 10 digits, 10 decoder/driver and five 8-bit I/O ports are required. To reduce the number of interfacing components, a technique known as multiplexing is used when large number of digits or alphanumeric characters is to be displayed.

A schematic diagram for multidigit display is shown in Figure 6.8. To display a digit or character, the code for seven-segment display is sent to the decoder/driver through the 4 lines of the output port. The microprocessor also sends a low signal to the ground terminal of the 7-segment display throughout of 10 decoder. Each 7-segment LED is turned on and off in a sequence and the process is repeated continuously. In multiplexed technique only one digit is displayed at a time. Due to persistence of vision one can see the desired number on LEDs.

The Intel 8279 has all necessary circuits to operate multidigit multiplexed LED display. This is used in a microprocessor-kit for multidigit display. It can control up to 16 LEDs to operate in multiplexed mode. In addition to multidigit display, it also acts as a keyboard encoder for an 8 × 8 scanned keyboard.

6.3 MICROPROCESSOR-BASED TRAFFIC CONTROL

Figure 6.9(a) shows a simple arrangement and port connections for microprocessor-based traffic light control. All ports of 8255 have been programmed as output ports. The control word to make

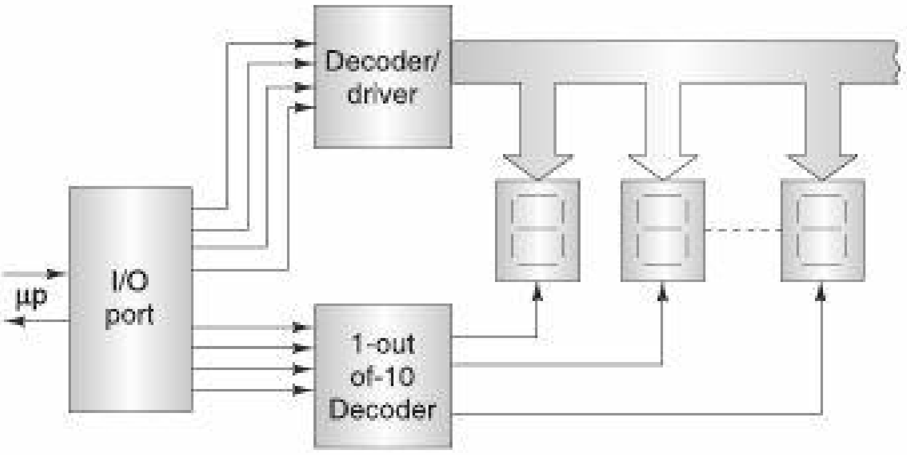


Fig. 6.8 Multiplexed Seven-Segment Displays.

all ports output ports in Mode 0 operation is 80 H. The connection of pins of the ports to LED has been made through buffers (7407). Positive logic has been used to switch on LEDs and the different colors used are red, yellow and green. Green light glows to allow crossing, yellow to make alert, and red does not allow crossing. Green light for right turns has not been shown.

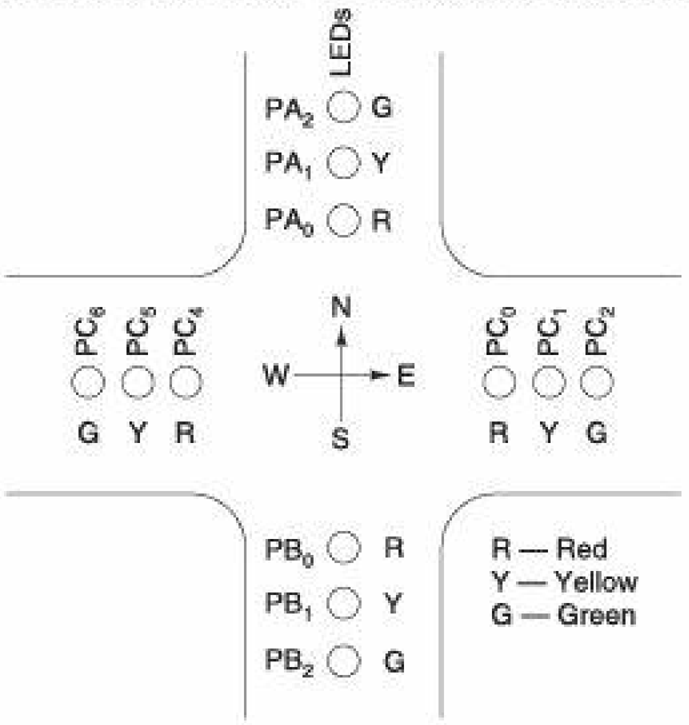


Fig. 6.9(a) Simple Arrangement for Microprocessor Based Traffic Light Control.

One can add some more LEDs— for this purpose connect them to ports and make additions to the program. The program will become a longer one. Once the user understands the circuit and program illustrated in this section; further extension is very easy. In a complex microprocessor-based traffic control sensors may be included to count the number of vehicles crossing the roads. More time can be allowed for the road on which the number of vehicles is more.

Pins	*Light*	*Pins*	*Light*	*Pins*	*Light*	*Pins*	*Light*
PA0	R1	PC0	R2	PC4	R3	PB0	R4
PA1	G1	PC1	G2	PC5	G3	PB1	G4
PA2	Y1	PC2	Y2	PC6	Y3	PB2	Y4

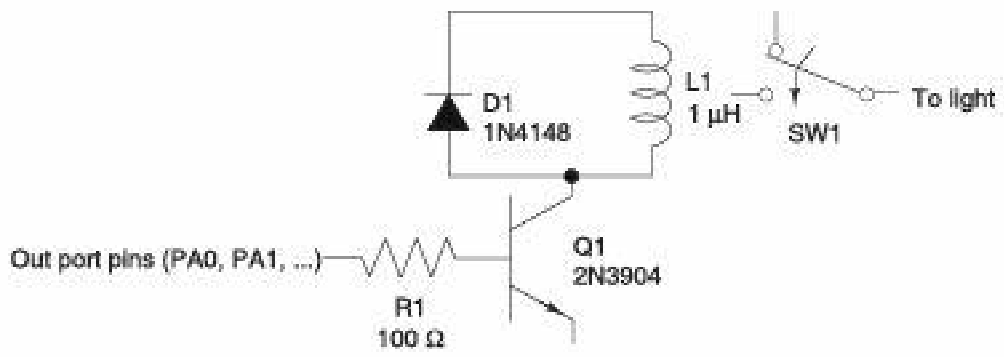


Fig 6.9(b) Basic Circuit for Traffic Light Controlling.

The electric bulbs are controlled by relays. The 8255 pins are used to control relay on-off action with the help of relay driver circuits. The driver circuit includes 12 transistors to drive 12 relays. Fig. 6.10 also shows the interfacing of 8255 to the system.

6.4 MICROPROCESSOR-BASED DATA ACQUISITION SYSTEM

Any data acquisition system consists of two parts, first the measurement of physical quantities and second A/D converters. The data to be acquired may be temperature, pressure, strain and deflection. So this has to be sensed by transducers and then converted into digital by A/D converters.

6.4.1 Measurement of Physical Quantities

Microprocessor-based systems are widely used in industries for the measurement, and display of physical quantities like temperature, pressure, speed, flow, etc. For the measurement of physical quantities, transducers are used to give electrical signal proportional to the input. If the electrical signal is small, it is amplified using amplifiers. The electrical signal is applied to an A/D converter which is connected to a microcomputer. If more than one physical quantity is to be monitored, a multiplexer is included in the interface. A schematic diagram for general interface is shown in Figure 6.10.

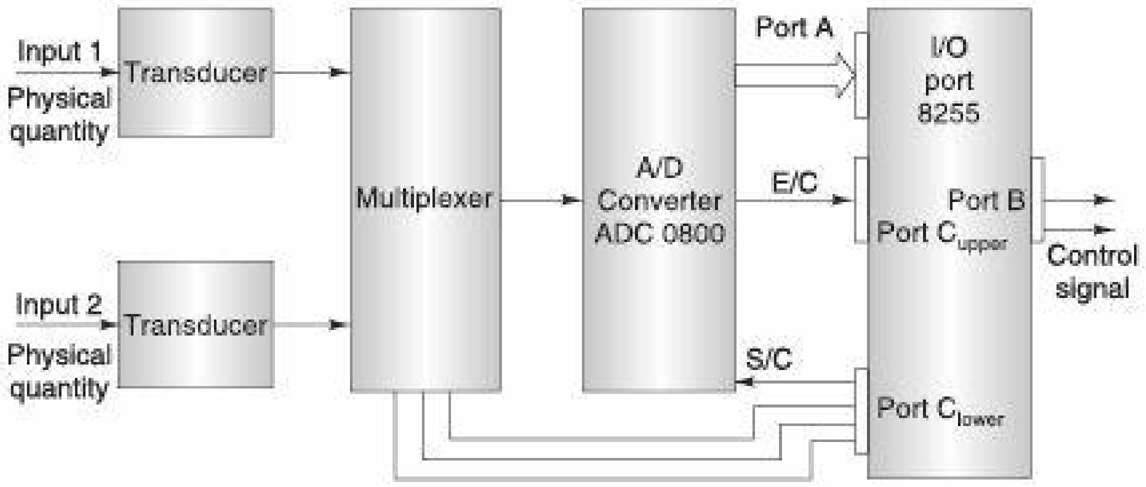


Fig. 6.10 General Interface for Physical Quantity Measurement.

6.4.2 Temperature Measurement and Control

For the measurement of temperature, one of the following devices are used:

Resistance thermometers (−100 to +300°C)

Thermocouples (−250 to +2000°C

Thermistors (−100 to +100°C)

Pyrometers (+100 to +5000°C)

Platinum wires are frequently used in resistance thermometers for industrial applications because of greater resolution, and mechanical and electrical stability as compared to copper or nickel wires. A change in temperature causes a change in resistance. The resistance thermometer is placed in an arm of a Wheatstone bridge to get a voltage proportional to temperature. A thermistor is a semiconductor device fabricated from a sintered mixture of metal alloys, having a large negative temperature coefficient. A thermistor is used in a Wheatstone bridge to get a voltage proportional to temperature. It can be used in the range of −100 to +100°C for greater accuracy as compared to platinum resistance thermometer. A thermocouple is the most widely used transducer to measure temperature. Thermocouple materials for the different range of temperature are as follows:

MATERIAL	*TEMP RANGE C*	*MATERIAL*	*TEMP RANGE C*
Iron constant	-200 to +1300	Chromel-constantan	a-1200
Chromel-alumel	-200 to +1200	Pt-Rh-Platinum	a-1500
Copper-constantan	-200 to +400	Tungsten-menium	a-2000

Microprocessor-Based Scheme

Figure 6.11(a) shows a microprocessor-based scheme for temperature measurement and control the output of a thermocouple proportional to the temperature of the furnace or oven, etc. is in millivolt. It is amplified using multistage amplifier before it is processed by the microprocessor. The amplified voltage is applied to an AID converter. The microprocessor sends a start of conversion signal to the AID converter through the port of 8255 PPI. When AID converter completes conversion, it sends an end of conversion signal to the microprocessor. Having received an end of conversion signal from AID converter the microprocessor reads the output of the AID converter which is a digital quantity proportional to the temperature to be measured. The microprocessor displays the measured temperature. If the temperature of a furnace, oven or water-bath is to be controlled, the microprocessor first measures its temperature, and then compares the measured temperature with a reference temperature at which the temperature is to be maintained. If the measured temperature is higher than the reference temperature, the microprocessor sends control signal to reduce temperature. If the measured temperature is less than the reference temperature, the microprocessor sends a control signal to increase temperature. The temperature of a furnace or oven can be increased or decreased by increasing or decreasing the fuel input to the furnace. If heating is done by electric heaters, current in heating element is controlled. Figure 6.11(b) shows an amplifier circuit to amplify the output of the thermocouple. D.C. level detector is for initial adjustment.

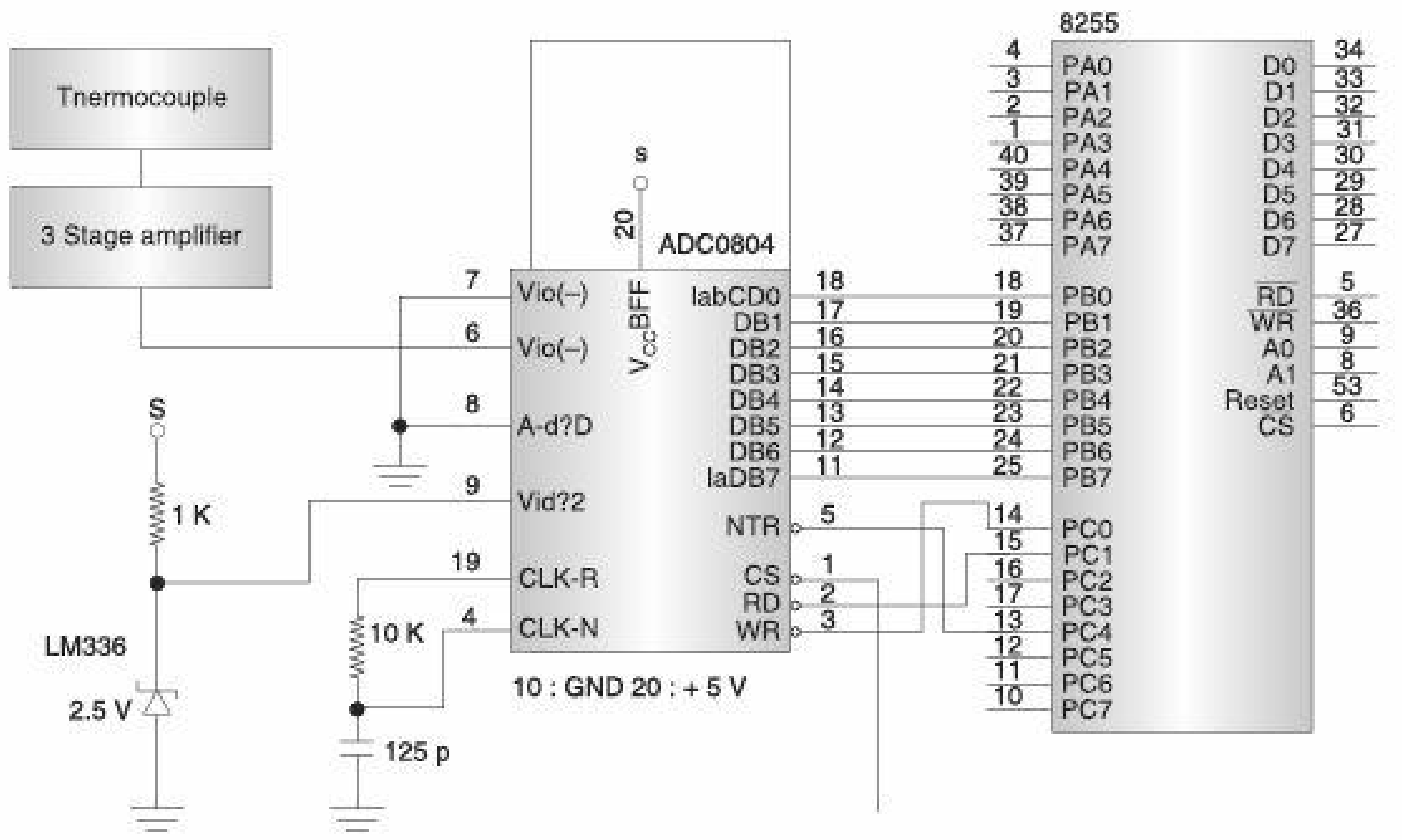


Fig. 6.11(a) Microprocessor-based Scheme for Temperature Measurement.

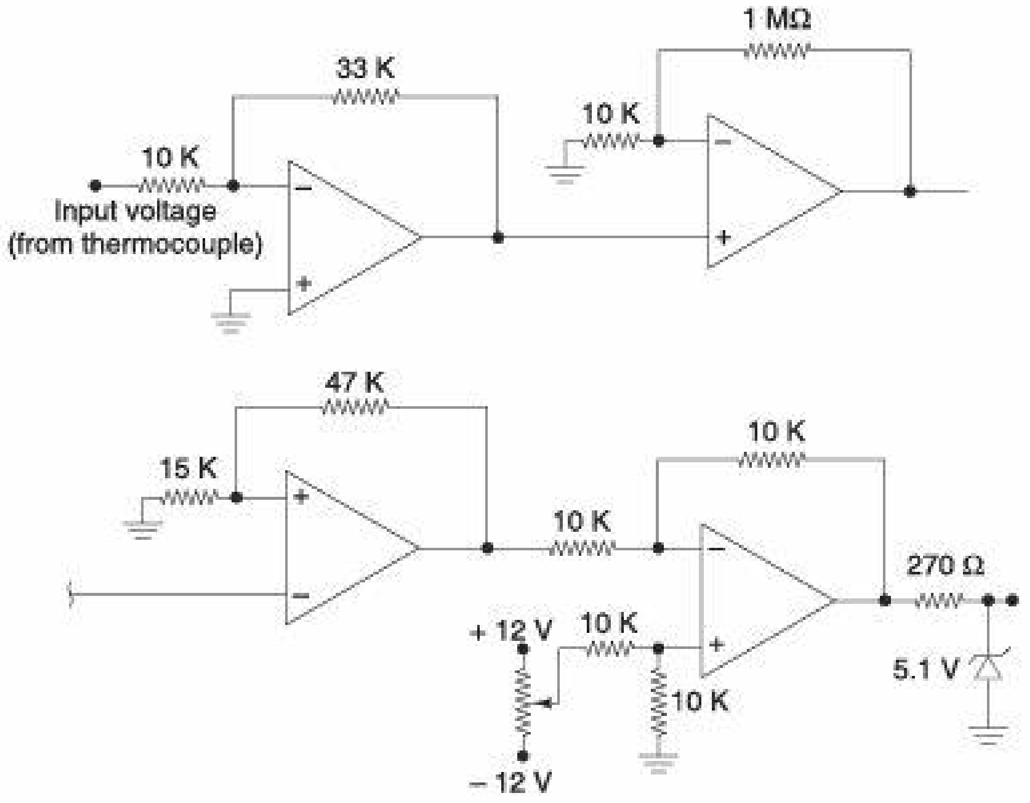


Fig. 6.11(b) 3-Stage Amplifier and DC Level Detector.

6.4.3 Strain Measurement

Strain is a measure of the deformation produced by the application of external forces. In the simplest form the deformation is an elongation (or shortening) L in length. The strain is given by

$$\varepsilon = dL/L$$

A transducer used for strain measurement is called a strain gauge. Strain gauges are used to measure strains and stresses in structures and machines. Copper-nickel alloy wires are used in strain gauges. The grid of fine wires forming a strain gauge is cemented to a thin paper membrane which is further cemented to the surface under test. When the surface is subjected to an external force it is under tension (or compression). The wires of the gauge are elongated (or shortened). This makes a change in the resistance of wire. The change in resistance R is L/R which is utilized to measure strain. A term gauge factor is defined as the ratio of per unit change in resistance to the per unit change in length. It is given by

$$GF = \frac{\frac{\Delta R}{R}}{\varepsilon} = \frac{\frac{\Delta \rho}{\rho}}{\varepsilon} + 1 + 2\nu$$

A gauge was cemented to a cantilever beam as shown in Figure 6.12. There is an arrangement to apply weights which create strain in the beam. Figure 6.13 shows a block diagram for strain measurement. The change in resistance is measured by a bridge as shown in Figure 6.14. The output of the bridge is applied to a differential amplifier. A differential amplifier has been used because there is no ground point in the output of the bridge. The output of the differential amplifier is further amplified by a two-stage amplifier as shown in Figure 6.15 to get sufficient voltage output which can be processed by the microprocessor. It should be in the range of 0-5 V. 5 V is for the maximum strain encountered. Figure 6.16 shows interface connections of ADC to microprocessor through 8255.

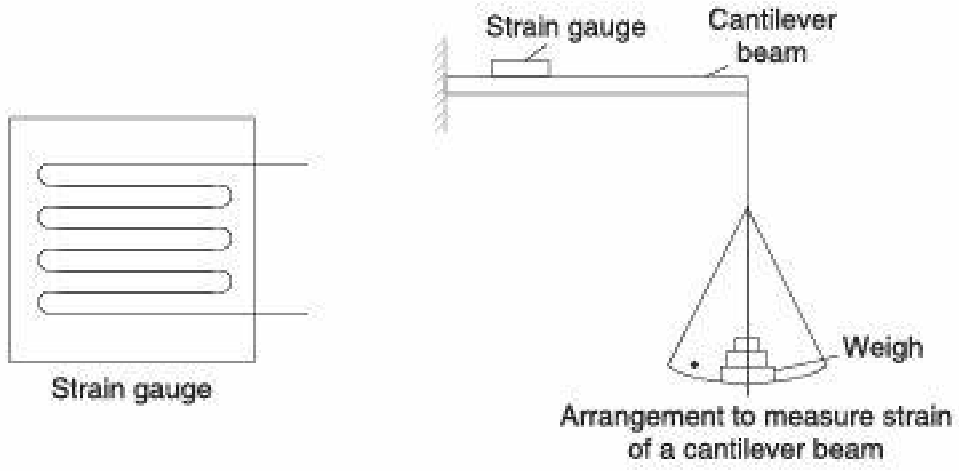


Fig. 6.12 Strain Gauge.

If a strain gauge under tension is placed in one arm of the bridge, the balance of the bridge is upset and an out of balance voltage V proportional to the strain is available at the terminal. To increase the sensitivity of the bridge if another gauge under compression is placed in the adjacent arm, an output of 2 V will be available. Similarly, if four strain gauges are placed in

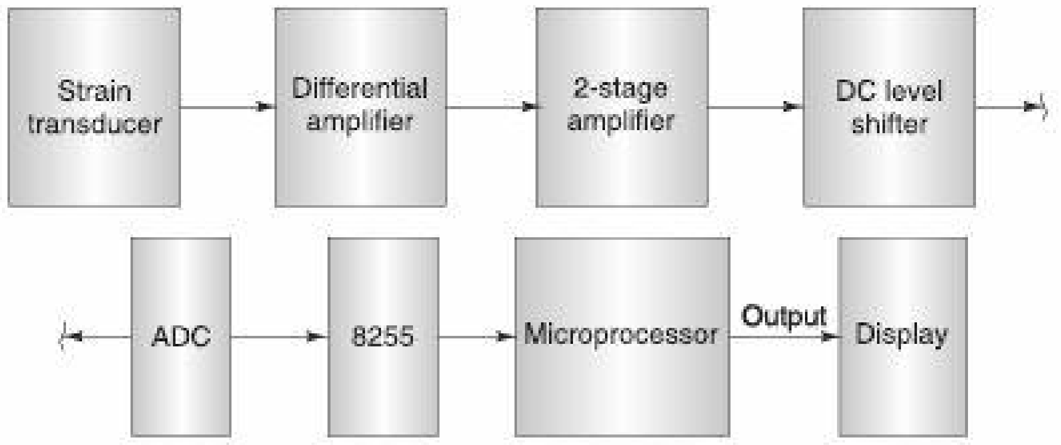


Fig. 6.13 Block Diagram of Strain Measurement.

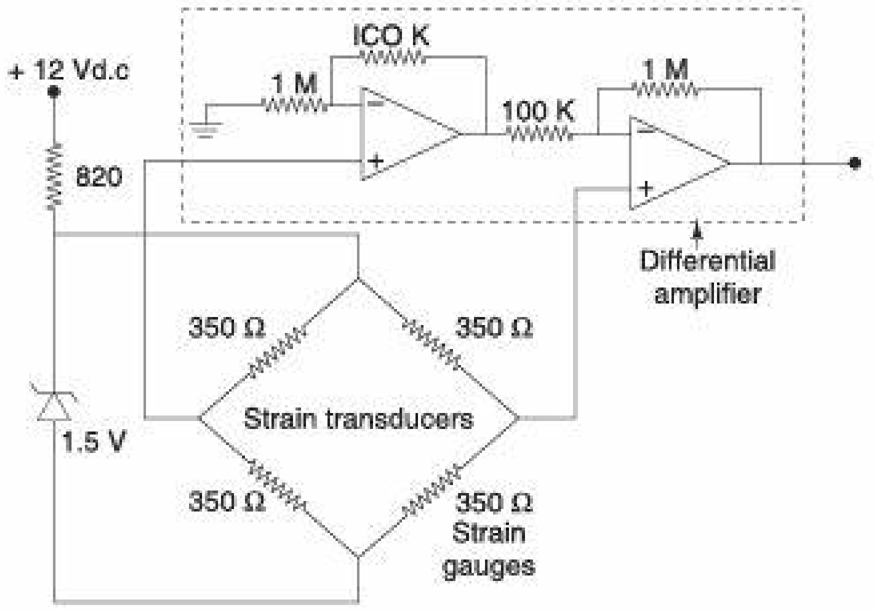


Fig. 6.14 Strain Transducer and Differential Amplifier.

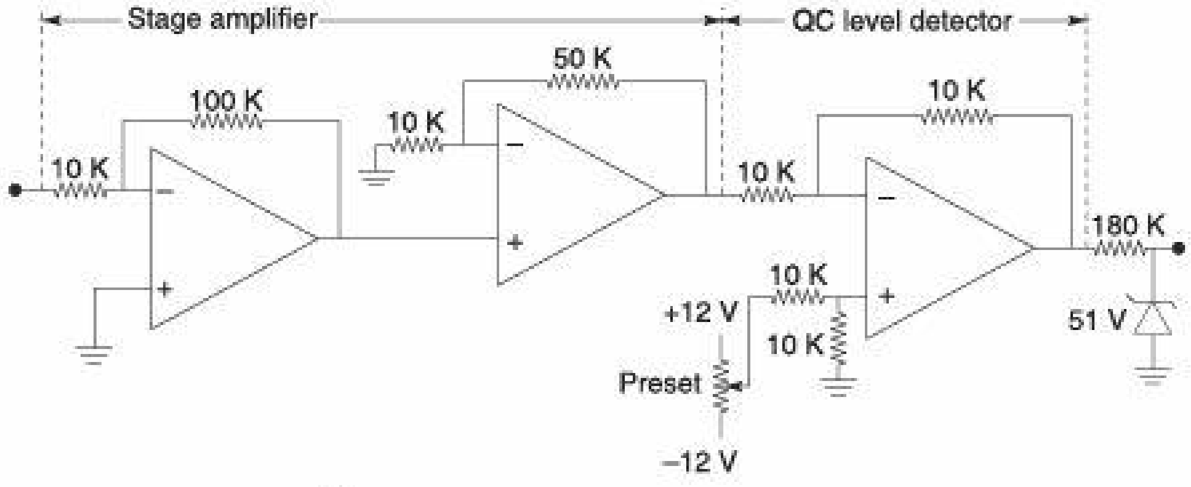


Fig. 6.15 Two-Stage Differential Amplifier.

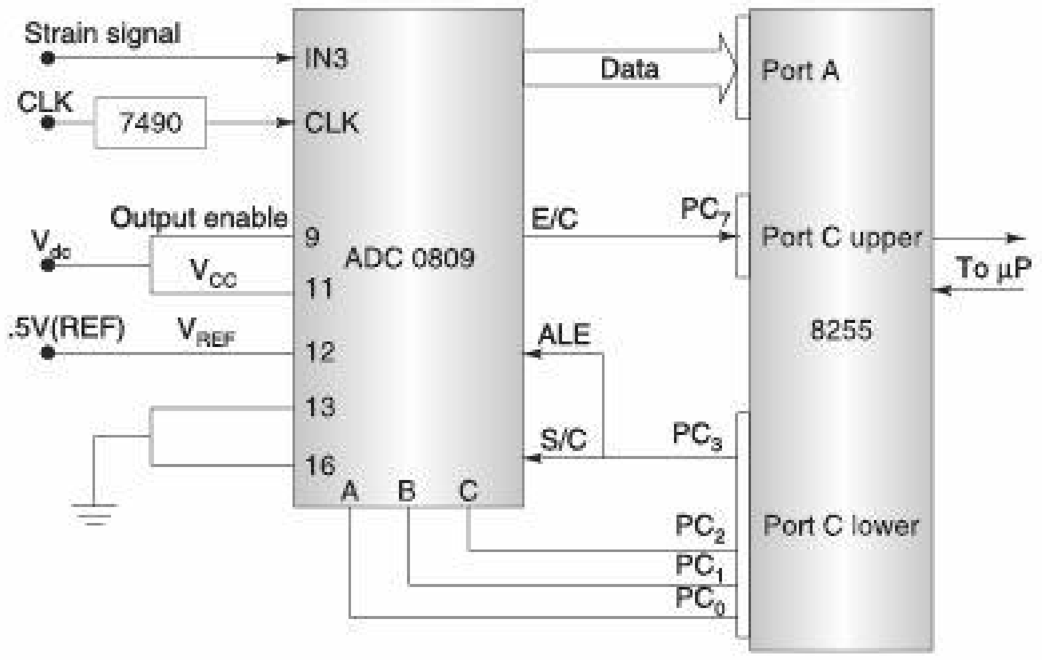


Fig. 6.16 Interface Connections for Strain Measurement.

all arms of the bridge in such a way that all of them cause an increase in the output voltage, the sensitivity is increased 4 times.

6.4.4 Deflection Measurement and Display

Figure 6.17 shows a schematic diagram of the scheme to measure deflection. To sense the deflection of a beam a differential transformer has been used. The core of the transformer is movable and attached to the beam. It is made of nickel-iron and is longitudinally slotted to reduce eddy-current losses. For a particular position of the core, the voltages induced in two secondaries of the differential transformer are equal and the output is zero. If the core is moved either up or down the voltage induced in two secondaries are unequal and there is an output. This output voltage is proportional to the deflection of the beam.

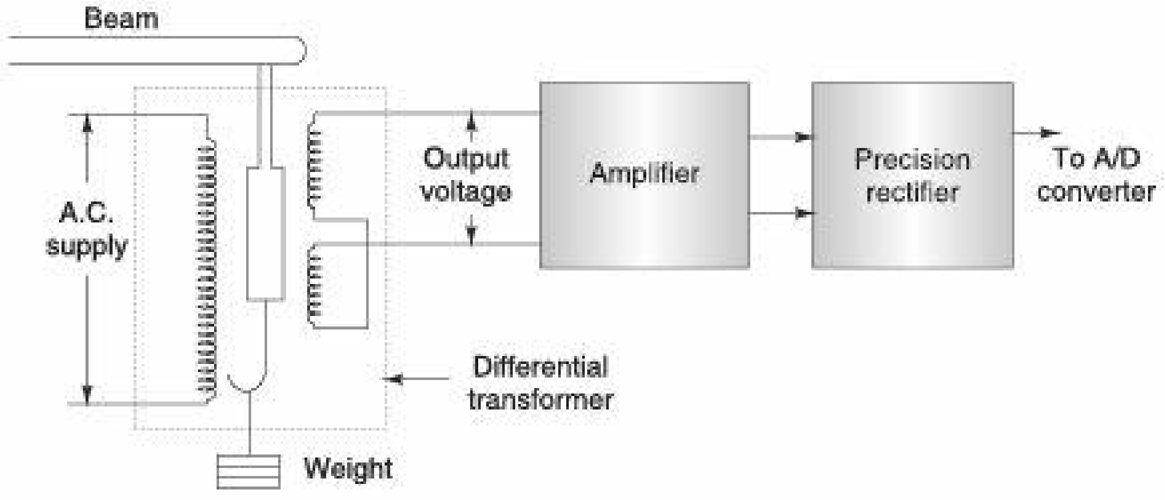


Fig. 6.17 Measurement of Deflection.

A 1.5 V supply at 3 kHz is used to excite the primary winding of the differential transformer. The output of the differential transformer is low, and hence an amplifier is used to amplify its output. After rectification the output voltage is applied to the multiplexer of ADC 0808. The Intel 8253 is employed to generate clock for ADC 0808. For different values of deflection the digital output of the AID converter is 'noted'. A look-up table is prepared using these values to show deflection on LED display. The microprocessor measures deflection and displays the same on LED display. Using elaborate look-up table or computation based on measured digital voltage, more accurate value of deflection can be displayed.

6.5 ANALOG-TO-DIGITAL CONVERTER

Analog refers to physical quantities that vary continuously instead of discretely. Physical phenomena typically involve analog signals. Examples include temperature, speed, position, pressure, voltage, altitude, etc. Microprocessors work with digital quantities (values taken from the discrete domain). For a digital system to interact with analog systems, conversion between analog and digital values is needed. Building blocks to perform the conversions are: (1) Digital to analog converters (DACs); and (2) Analog to digital converters (ADCs). A digital to analog converter has a digital input that specifies an output whose value changes in steps. These step changes are in volts or amperes. The analog to digital converter has an input that can vary from a minimum to a maximum value of volts or amperes. The output is a digital number that represents the input value.

6.5.1 Analog-to-Digital Converter Architectures

The basic ADC function is shown in Figure 6.18. This could also be referred to as a quantizer. Most ADC chips also include some of the support circuitry, such as clock oscillator for the sampling clock, reference (REF), the sample and hold function, and output data latches. In addition to these basic functions, some ADCs have additional built in circuitry. These functions could include multiplexers, sequencers, auto-calibration circuits, programmable gain amplifiers (PGAs), etc.

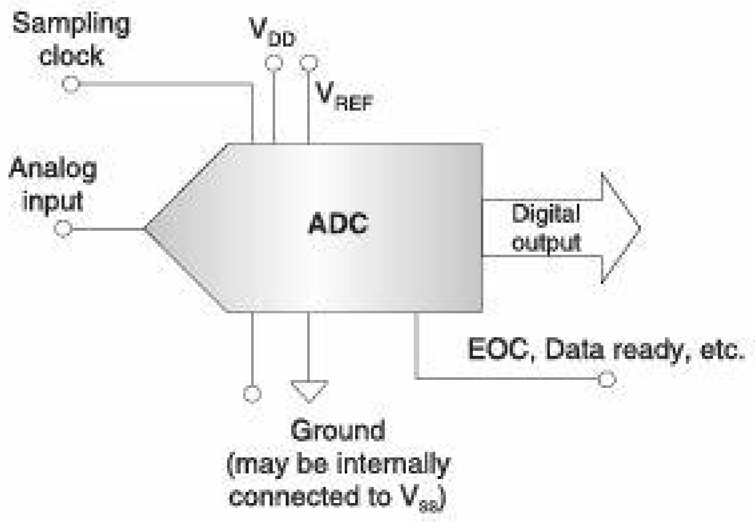


Fig. 6.18 Basic ADC Function.

Some ADCs use external references and have a reference input terminal, while others have an output from an internal reference. In some instances, the ADC may have an internal reference that is pinned out through a resistor. This connection allows the reference to be filtered (using the internal R and an external C) or by allowing the internal reference to be overdriven by an external reference. The simplest ADCs, of course, have neither—the reference is on the ADC chip and has no external connections.

If an ADC has an internal reference, its overall accuracy is specified when using that reference. If such an ADC is used with a perfectly accurate external reference, its absolute accuracy may actually be worse than when it is operated with its own internal reference. This is because it is trimmed for absolute accuracy when working with its own actual reference voltage, not with the nominal value. Twenty years ago it was common for converter references to have accuracies as poor as ±5% since these references were trimmed for low temperature coefficient rather than absolute accuracy, and the inaccuracy of the reference was compensated in the gain trim of the ADC itself. Today the problem is much less severe, but it is still important to check for possible loss of absolute accuracy when using an external reference with an ADC which has a built-in one.

ADCs which have reference terminals must, of course, specify their behaviour and parameters. If there is a reference input the first specification will be the reference input voltage and of course this has two values, the absolute maximum rating, and the range of voltages over which the ADC performs correctly. Most ADCs require that their reference voltage is within quite a narrow range whose maximum value is less than or equal to the ADCs VDD.

The reference input terminal of an ADC may be buffered as shown in Figure 6.19, in which case it has input impedance (usually high) and bias current (usually low) specifications, or it may connect directly to the ADC. In either case, the transient currents developed on the reference input due to the internal conversion process need good decoupling with external low inductance capacitors. Good ADC data sheets recommend appropriate decoupling networks.

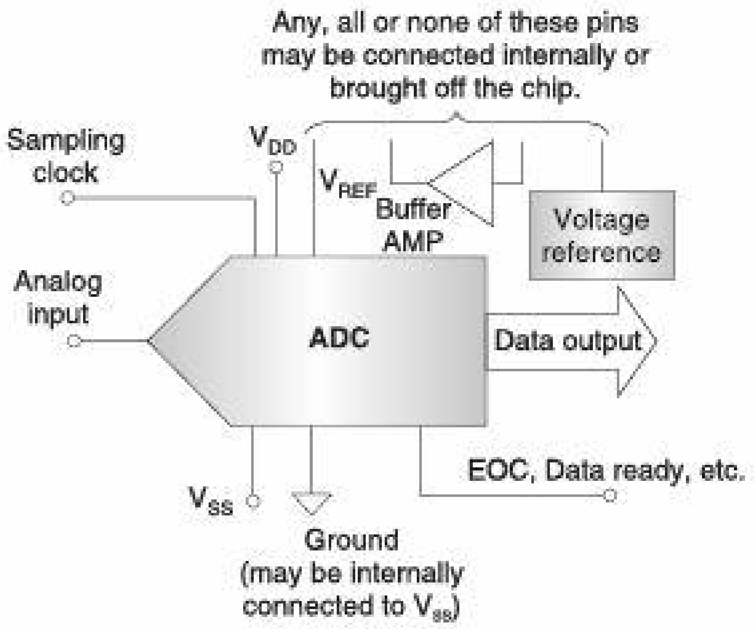

Any, all or none of these pins may be connected internally or brought off the chip.
Sampling clock
V_{DD}
V_{REF}
Buffer AMP
Voltage reference
Analog input
ADC
Data output
EOC, Data ready, etc.
V_{SS}
Ground (may be internally connected to V_{ss})

Fig. 6.19 ADC with Reference and Buffer.

The reference output may be buffered or unbuffered. If it is buffered, the maximum output current will probably be specified. In general, such a buffer will have a unidirectional output stage which sources current but does not allow current to flow into the output terminal. If the buffer does have a push-pull output stage (not as common), the output current will probably be defined as ±(SOME VALUE) mA. If the reference output is unbuffered, the output impedance may be specified, or the data sheet may simply advise the use of a high input impedance external buffer. There are some instances where the power supply is the reference. In these cases it is imperative to make sure the power supply is clean.

There are a couple of general trends in ADCs that should be addressed. The first is the general trend towards lower supply voltages. This is partially due to the processes, particularly CMOS, which are used to manufacture the chips. Increasing demand for speed has driven the feature size of the processes down. This typically results in lower breakdown voltages for the transistors. This, in turn, requires lower supply voltages. Very few new parts are developed with the legacy ±15 V supplies and ±10 V input range.

Since the input signal range of the ADCs is shrinking, there is also a trend towards differential inputs. This helps improve the dynamic range of a converter, typically by 6 dB. There could be even further improvement since the common-mode ground referenced noise is rejected. In many cases the differential input can be driven single-endedly (with the resultant reduction of SNR). Occasionally the REF input might also be differential.

6.5.1.1 The Comparator: A 1-Bit ADC

A comparator is a 1-bit ADC shown in Figure 6.20. If the input is above a threshold, the output has one logic value, below it has another. There is no ADC architecture which does not use at least one comparator of some sort. So while a 1-bit ADC is of very limited use it is a building block for other architectures.

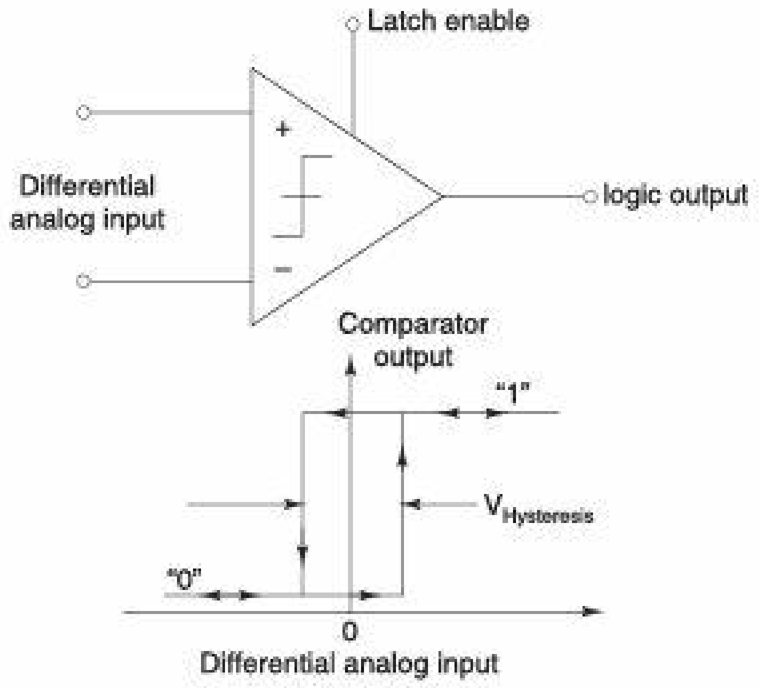


Fig. 6.20 Comparator a 1-Bit ADC.

Comparators used as building blocks in ADCs need good resolution which implies high gain. This can lead to uncontrolled oscillation when the differential input approaches zero. In order to prevent this, hysteresis is often added to comparators using a small amount of positive feedback. Figure 6.21 shows the effects of hysteresis on the overall transfer function. Many comparators have a millivolt or two of hysteresis to encourage "snap" action and to prevent local feedback from causing instability in the transition region. Note that the resolution of the comparator can be no less than the hysteresis, so large values of hysteresis are generally not useful.

6.5.1.2 Successive Approximation ADC

The basic successive approximation ADC is shown in Figure 6.21(a). It performs conversions on command. In order to process ac signals, ADCs must have an input sample-and-hold (SHA) to keep the signal constant during the conversion cycle.

The ADC has a built in DAC. A sample and hold circuit stores an analog input. The ADC logic steps through a sequence of trial-and-error guessing to find the digital equivalent of the input. It begins the sequence by sending a digital signal that is at midrange to the DAC. The analog output V_{AX} from the DAC is compared with the analog input V_A from the sample-and-hold circuit. Thus, the ADC determines whether the analog input is above, at or below half scale. It continues determining to which half of the next range selection the analog input belongs. Figure 6.21(b) shows the algorithm of successive approximation ADC.

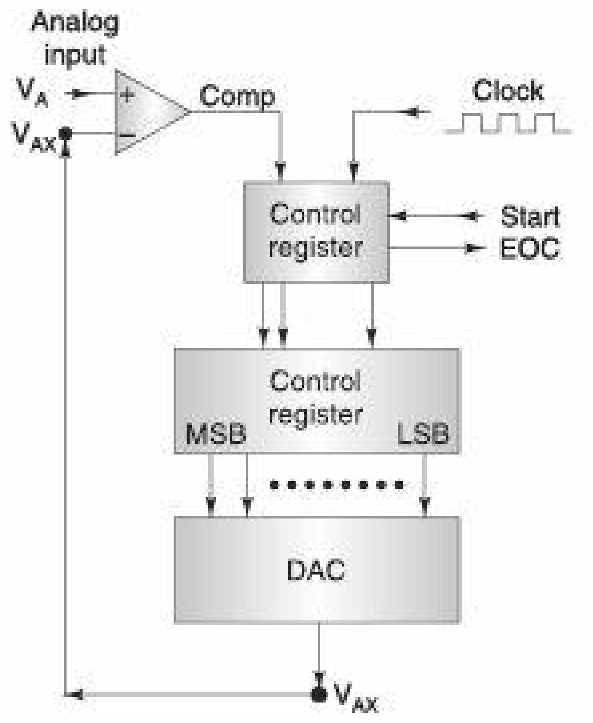


Fig. 6.21(a) Basic Successive Approximation ADC

For example, let consider a working of four-bit SAC operation using a DAC step size of 1 V and V_A = 10.4 V. Figure 6.22 shows block diagram of ADC.

Working

t1: Set MSB = 1 → 1000 → 8V (less than V_A)

t2: Set next bit = 1 → 1100 → 12V (greater than V_A, too high)

t3: Return bit to 0 → 1000

t4: Set next bit = 1 → 1010 → 10V (less than V_A)

t5: Set next bit = 1 → 1011 → 11V (greater than V_A, too high)

t6: Return bit to 0 → 1010

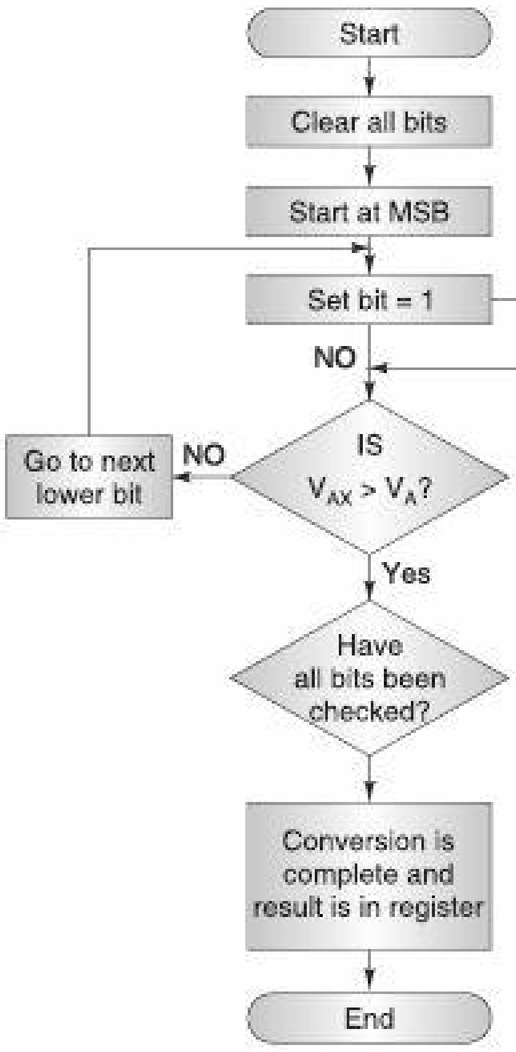
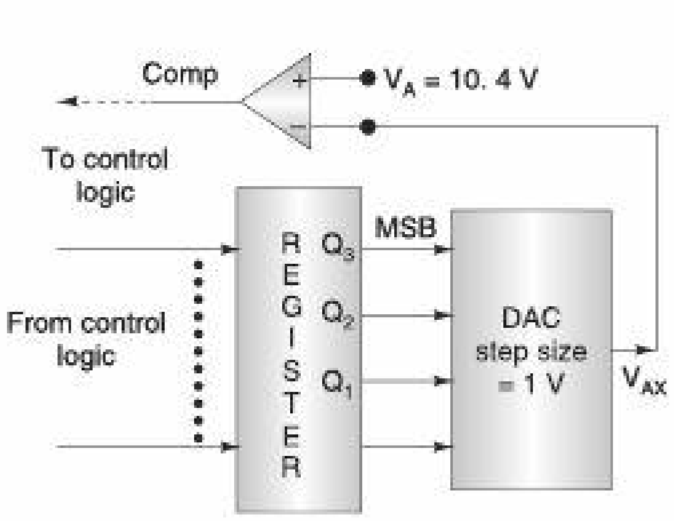


Fig. 6.21(b) Algorithm of basic Successive Approximation ADC.



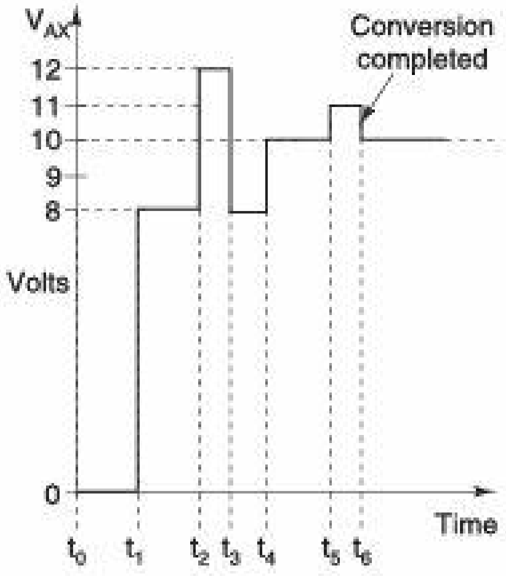


Fig. 6.22(a) 4-bit ADC (b) Timing Diagram.

The fundamental timing diagram for a typical SAR ADC is shown in Figure 6.23. The end of conversion is generally indicated by an end-of-convert (EOC), data-ready (DRDY), or a busy signal (actually, not-BUSY indicates end of conversion).

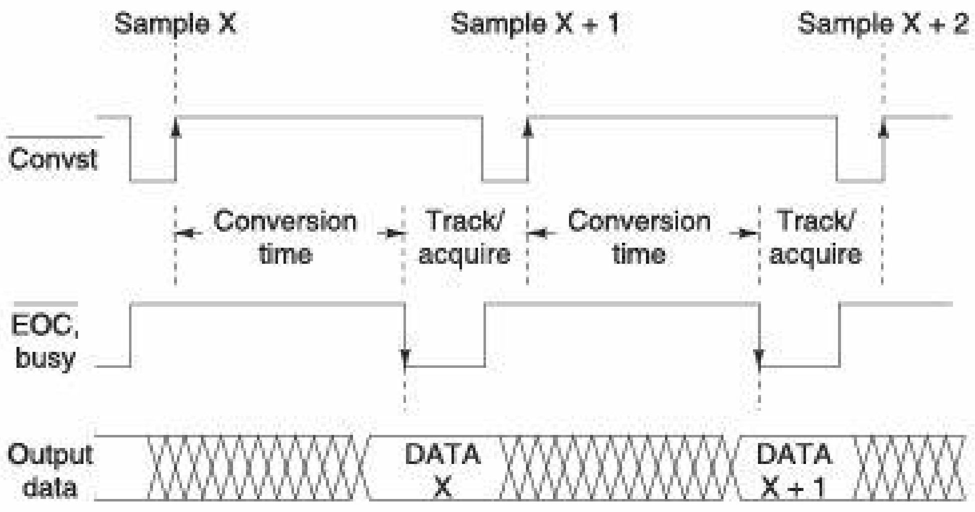


Fig. 6.23 Typical SAR ADC Timing.

The polarities and name of this signal may be different for different SAR ADCs, but the fundamental concept is the same. At the beginning of the conversion interval, the signal goes high (or low) and remains in that state until the conversion is completed, at which time it goes low (or high). The trailing edge is generally an indication of valid output data, but the data sheet should be carefully studied—in some ADCs additional delay is required before the output data is valid.

6.5.1.3 Flash ADC

Flash ADCs (sometimes called parallel ADCs) are the fastest type of ADC and use large number of comparators. An N-bit flash ADC consists of 2^N resistors and 2^N-1 comparators arranged as in Figure 6.24. Each comparator has a reference voltage which is 1 LSB higher than that of the one below it in the chain.

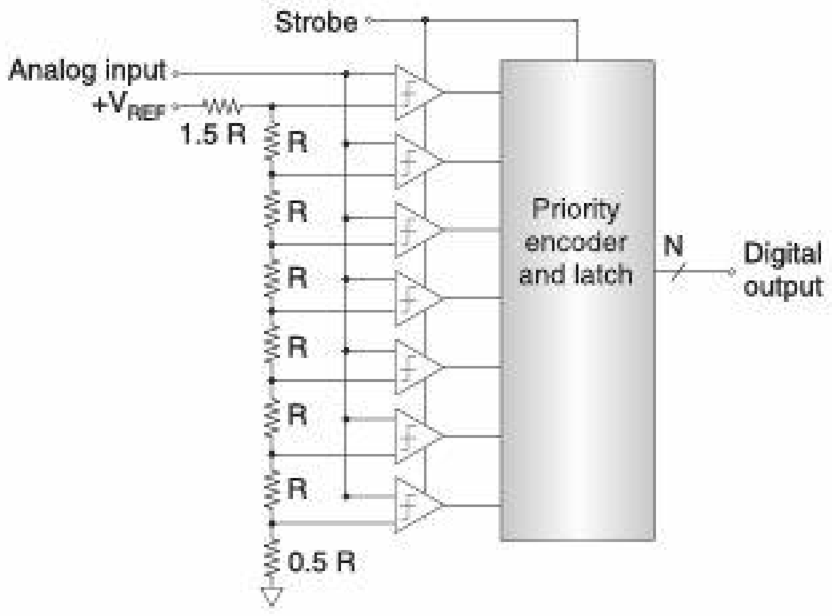


Fig. 6.24 Flash Type ADC.

For a given input voltage, all the comparators below a certain point will have their input voltage larger than their reference voltage and a "1" logic output, and all the comparators above that point will have a reference voltage larger than the input voltage and a "0" logic output. The 2^N–1 comparator outputs therefore behave in a way analogous to a mercury thermometer, and the output code at this point is sometimes called a thermometer code. Since 2^N–1 data outputs are not really practical, they are processed by a decoder to generate an N-bit binary output.

The input signal is applied to all the comparators at once, so the thermometer output is delayed by only one comparator delay from the input, and the encoder N-bit output by only a few gate delays on top of that, so the process is very fast. However, the architecture uses large number of resistors and comparators and is limited to low resolutions, and if it is to be fast, each comparator must run at relatively high power levels. Hence, the problems of flash ADCs include limited resolution, high power dissipation because of the large number of high-speed comparators (especially at sampling rates greater than 50 MSPS), and relatively large (and therefore expensive) chip sizes. In addition, the resistance of the reference resistor chain must be kept low to supply adequate bias current to the fast comparators, so the voltage reference has to source quite large currents (typically > 10 mA).

Figure 6.25 shows a typical 3-bit parallel flash ADC with the truth table (Table 6.2) for input output is relationship.

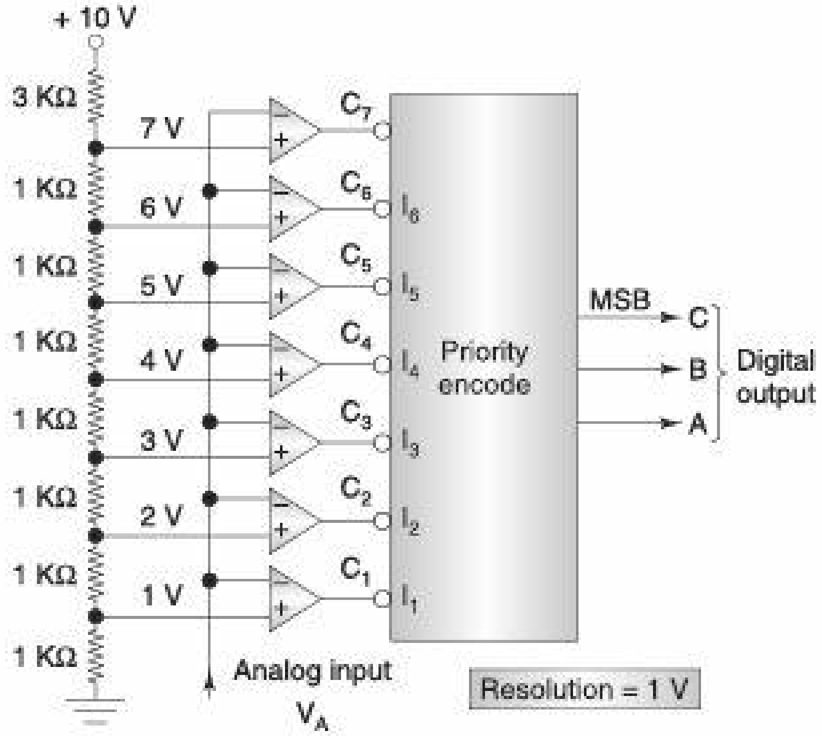


Fig. 6.25 3-bit Flash ADC.

Table 6.2 Truth Table for 3-bit Flash ADC

Analog in	Comparator outputs							Digital outputs		
V_A	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C	B	A
0-1 V	1	1	1	1	1	1	1	0	0	0
1-2 V	0	1	1	1	1	1	1	0	0	1

Contd...

Contd...

2-3 V	0	0	1	1	1	1	1	0	1	0
3-4 V	0	0	0	1	1	1	1	0	1	1
4-5 V	0	0	0	0	1	1	1	1	0	0
5-6 V	0	0	0	0	0	1	1	1	0	1
6-7 V	0	0	0	0	0	0	1	1	1	0
> 7 V	0	0	0	0	0	0	0	1	1	1

6.5.2 ADC 080X Series

The ADC 080x series are CMOS 8-bit successive approximation A to D converters that use a differential potentiometric ladder. These converters are designed to allow operation with control bus with TRI-STATE. The output latches directly drive the data bus. These ADCs appear like memory locations or I/O ports to the microprocessor and no interfacing logic is needed. Differential analog voltage inputs allow increasing the common-mode rejection and offsetting the analog zero input voltage value. In addition to the voltage reference input can be adjusted to allow encoding any smaller analog voltage span to the full 8 bits of resolution.

6.5.2.1 Features of ADC 080X Series

- Compatible with 8080 microprocessors derivatives, no interfacing logic needed-access time - 135 ns
- Easy interface to all microprocessors or operates "stand alone'
- Differential analog voltage inputs
- Logic inputs and outputs meet both MOS and TTL voltage level specifications
- Works with 2.5 V (LM336) voltage reference
- On-chip clock generator
- 0 V to 5 V analog input voltage range with single 5 V supply
- No zero adjust required
- 0.3" standard width 20-pin DIP package
- 20-pin molded chip carrier or small outline package
- Operates ratiometrically or with 5 V_{DC} to 2.5 V_{DC} or analog span adjusted voltage reference

6.5.2.2 Pin Diagram

The pin diagram of ADC 0801 is shown in Figure 6.26(a) and typical application schematic is shown in Figure 6.26(b).

6.5.2.3 Function Description

The functional diagram of the ADC080X series of A/D converters operates on the successive approximation principle. Analog switches are closed sequentially by successive-approximation logic until the analog differential input voltage [$V_{IN}(+) - V_{IN}(-)$] matches a voltage derived from a tapped resistor string across the reference voltage. The most significant bit is tested first and

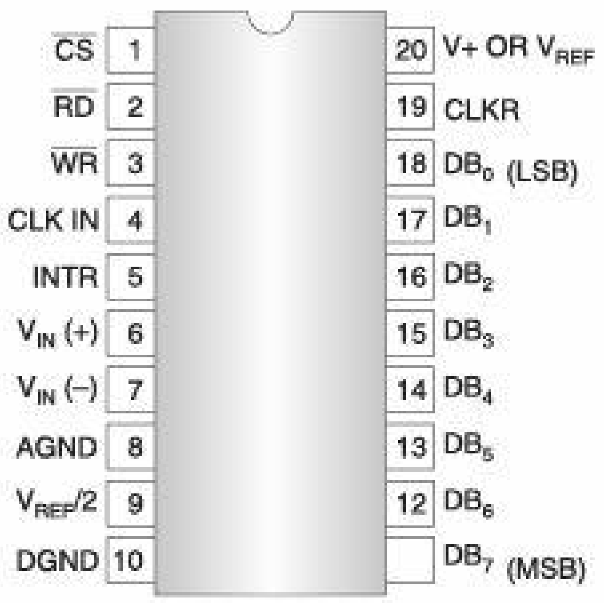


Fig. 6.26(a) ADC 0801 Pin Diagram.

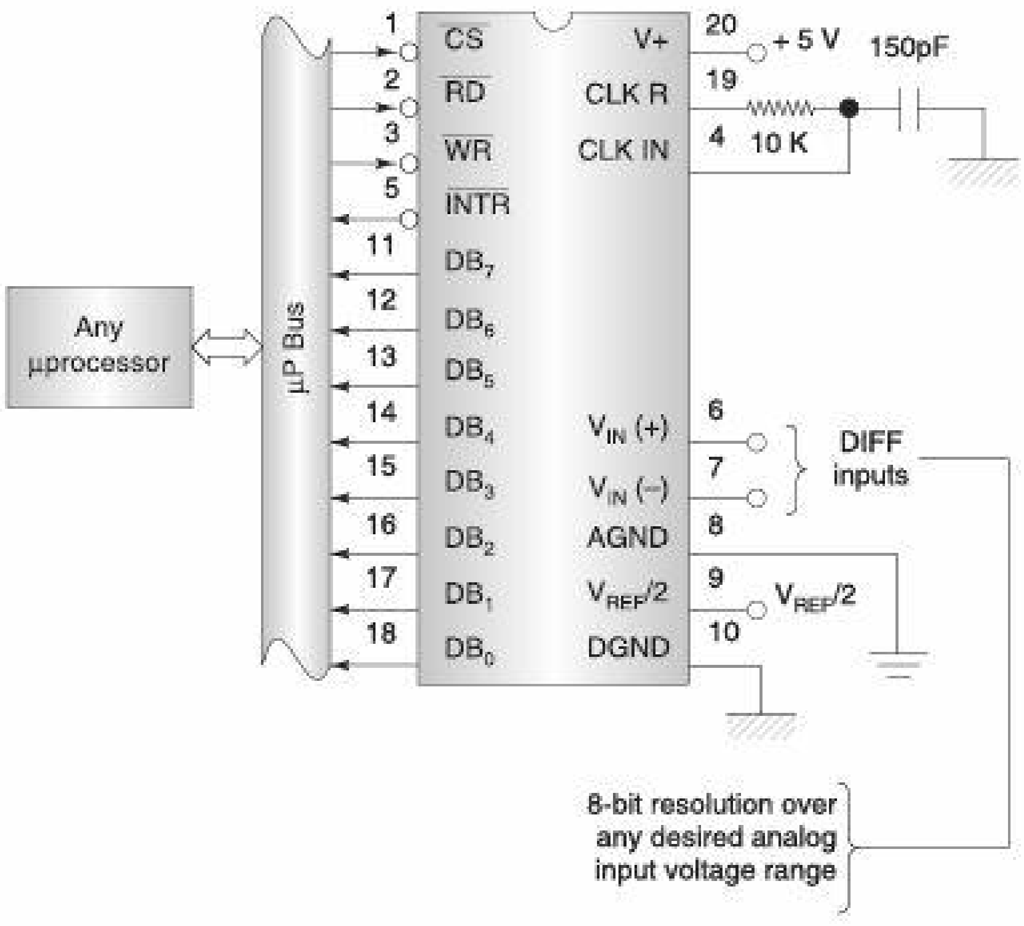


Fig. 6.26(b) ADC 0801 Typical Application Schematic.

after 8 comparisons (64 clock cycles), an 8-bit binary code (1111 1111 = full scale) is transferred to an output latch.

6.5.2.4 *Working*

The normal operation proceeds as follows. On the high-to-low transition of the WR input, the internal SAR latches and the shift-register stages are reset, and the INTR output will be set high. As long as the CS input and WR input remain low, the A/D will remain in a reset state.

Conversion will start from 1 to 8 clock periods after at least one of these inputs makes a low-to-high transition. After the requisite number of clock pulses to complete the conversion, the INTR pin will make a high-to-low transition. This can be used to interrupt a processor, or otherwise signal the availability of a new conversion. An RD operation (with CS low) will clear the INTR line high again. The device may be operated in the free-running mode by connecting INTR to the WR input with CS = 0. To ensure start-up under all possible conditions, an external WR pulse is required during the first power-up cycle. A conversion-in-process can be interrupted by issuing a second start command.

6.5.2.5 *Timing Diagrams of ADC 080X series*

6.5.2.6 *Interfacing ADC 080X with 8085*

Figure 6.27 shows the interfacing of ADC 0801 series of A/D converters with 8085. The start of conversion signal is to be applied to WR. The start of conversion signal is to be first made low and then high. To start conversion CS must be low. When WR goes low, the converter is reset. When returns to the high state, the conversion begins. When both CS and RD are low, the digital output is available on the output lines. INTR goes low when conversion is over. The range of clock frequency is 100 kHz to 800 kHz. The clock may be supplied from an external source or it can be generated by internal clock generator. If the clock is be derived from CPU clock, it is applied to pin 4 (CLK IN).

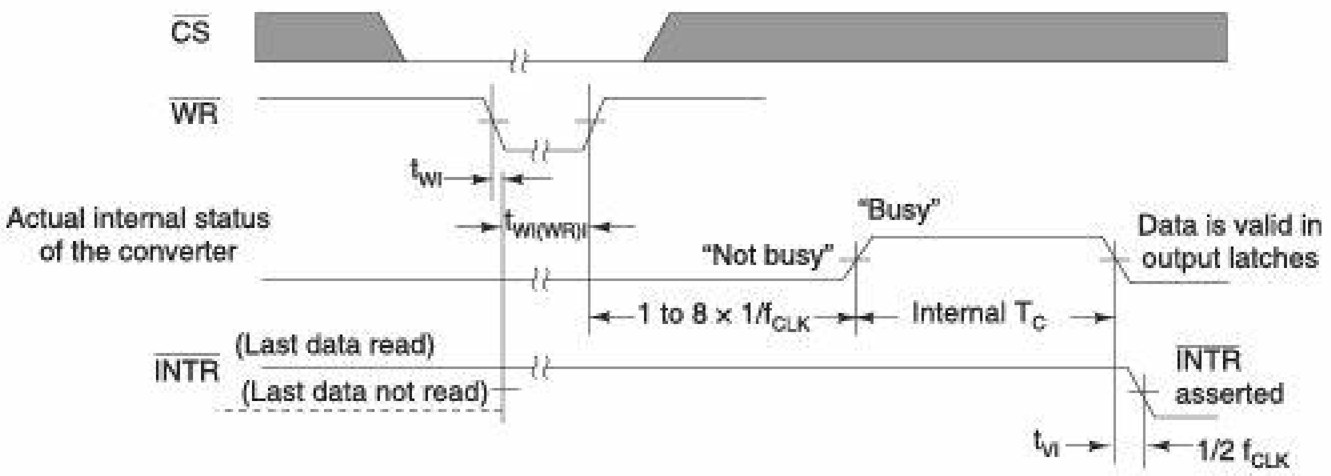


Fig. 6.27(a) Start Conversion.

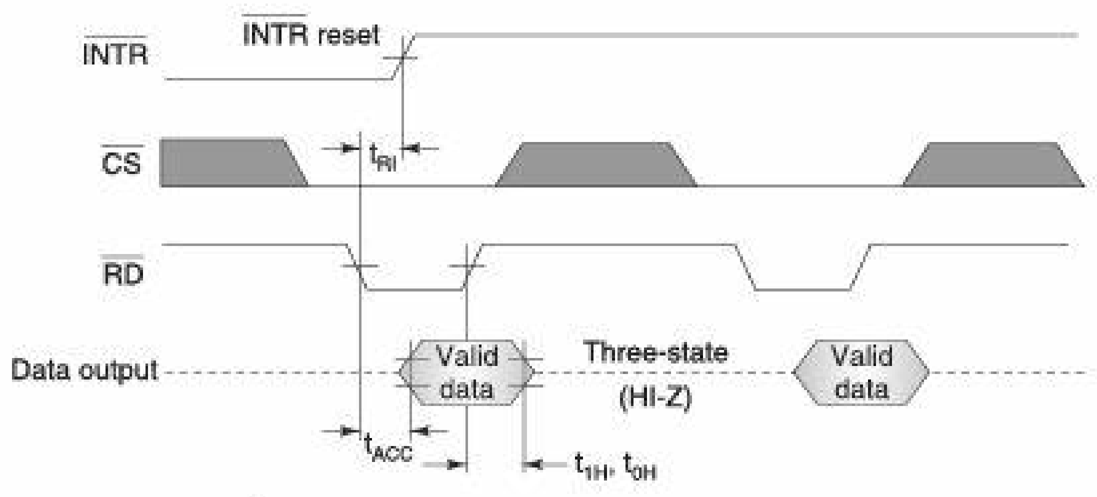


Fig. 6.27(b) Output Enable and RESET INTR.

If CPU clock is of higher frequency, it should be reduced employing counters. If CLK is to be generated by internal clock generator an R-C circuit should be connected to pin 4 and pin 19 as shown. The clock frequency is given by $f = 1/1.1RC$. A typical range of the resistance is 10 K ohms to 50 K ohms. Corresponding to R = 12 K, and C = 120 pF, the clock frequency is 632 kHz.

There are two pins 6 and 7 for analog input voltage. If input voltage to be handled has only positive values, the input can be applied to pin 6, and pin 7 is to be grounded. If the input voltage has only negative values, the input can be applied to pin 7, and pin 6 may be grounded. If differential input is to be applied, both input pins 6 and 7 are used. For a signal having some values positive and some negative, the connection as shown is made. Pin 9 is for V_{REF}. If it is left open, V_{REF} is equal to supply voltage V_{cc}. In some cases a different analog range may be desired. If the maximum analog input of +4 V is required, +2 V is applied to pin 9. If the maximum analog input required is 03 volts, +1.5 V is applied to pin 9. There are two ground pins, pin 8 and pin 10, both are grounded. Figure 6.28 shows interfacing of ADC 0804 for input voltage range of 0 V to + 5 V and the circuit for reference voltage.

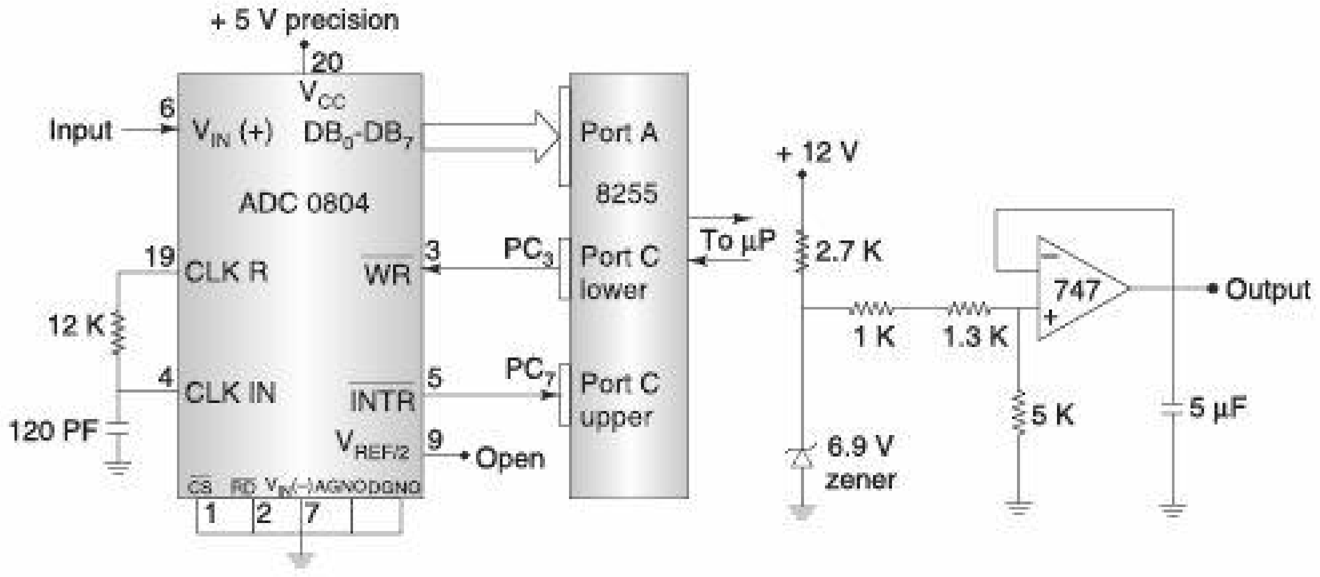


Fig. 6.28 Interfacing of ADC 0804 for Positive Input Voltage and Circuit for VREF.

6.5.2.7 *Interfacing of ADC 080X directly to Microprocessor Bus*

ADC 080X can be directly connected to the microprocessor bus, as shown in Figure 6.29. The microprocessor directly controls the ADC which is connected as an I/O device. Its address is FF. When A8-A15, all high CS becomes low and the device is selected. When microprocessor issues OUT FF, the conversion starts. OUT instruction consists of three machine cycles. In the 1st machine cycle opcode is fetched from the memory. In the 2nd machine cycle the address of the device is fetched from the memory.

In the 3rd machine IO/M goes high. Also, WR of microprocessor bus goes low and again it becomes high. Thus, WR of ADC goes low and then it becomes high. Thus, a start of conversion pulse is applied to AOC. When conversion is over INTR goes low. It is connected to an through inverter. On receiving an interrupt, the microprocessor issues IN FF instruction. IN instruction consists of three machine cycles. The first machine cycle is for encode fetch and the 2nd for memory read cycle.

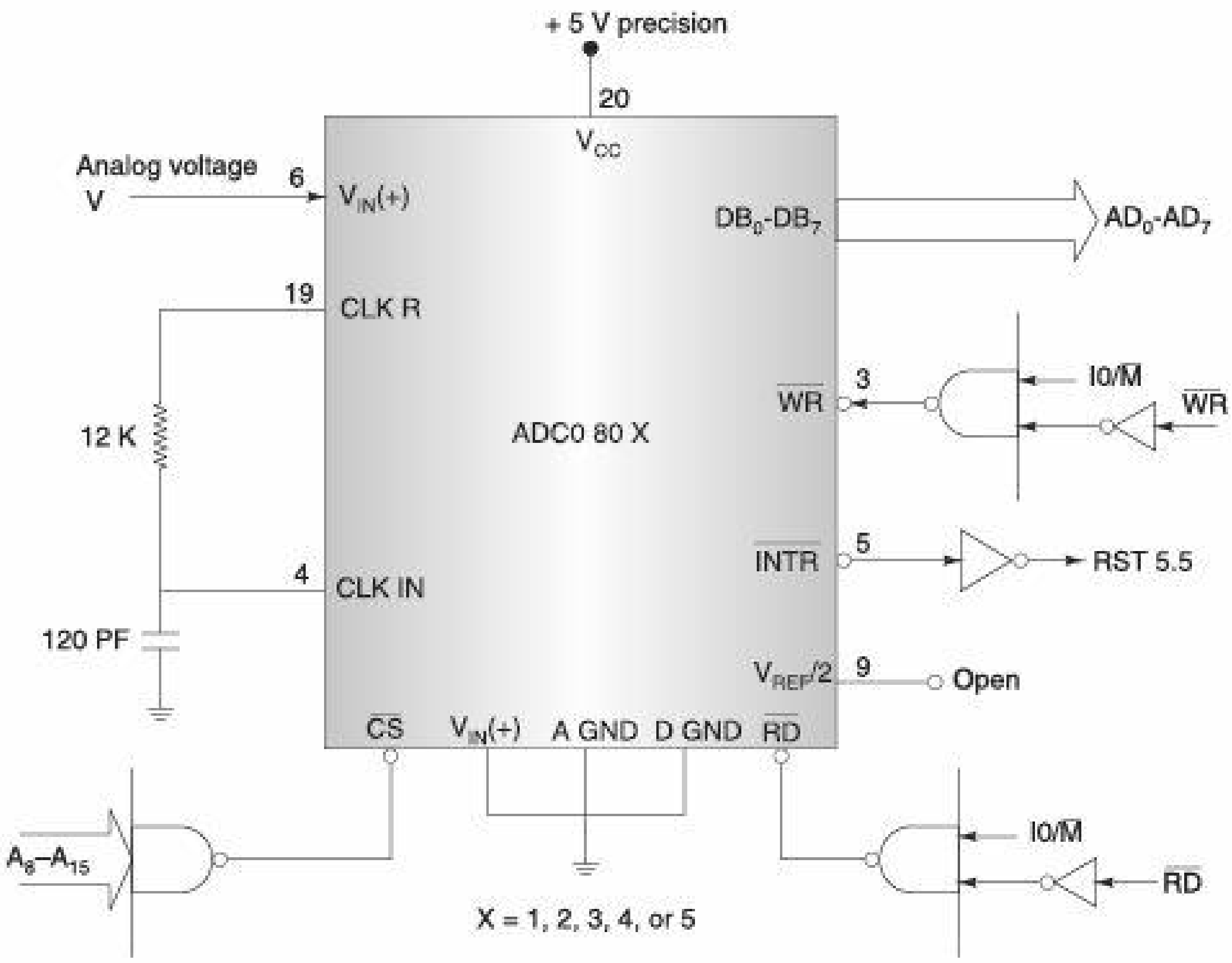


Fig. 6.29(a) Direct Interfacing of ADC with 8085.

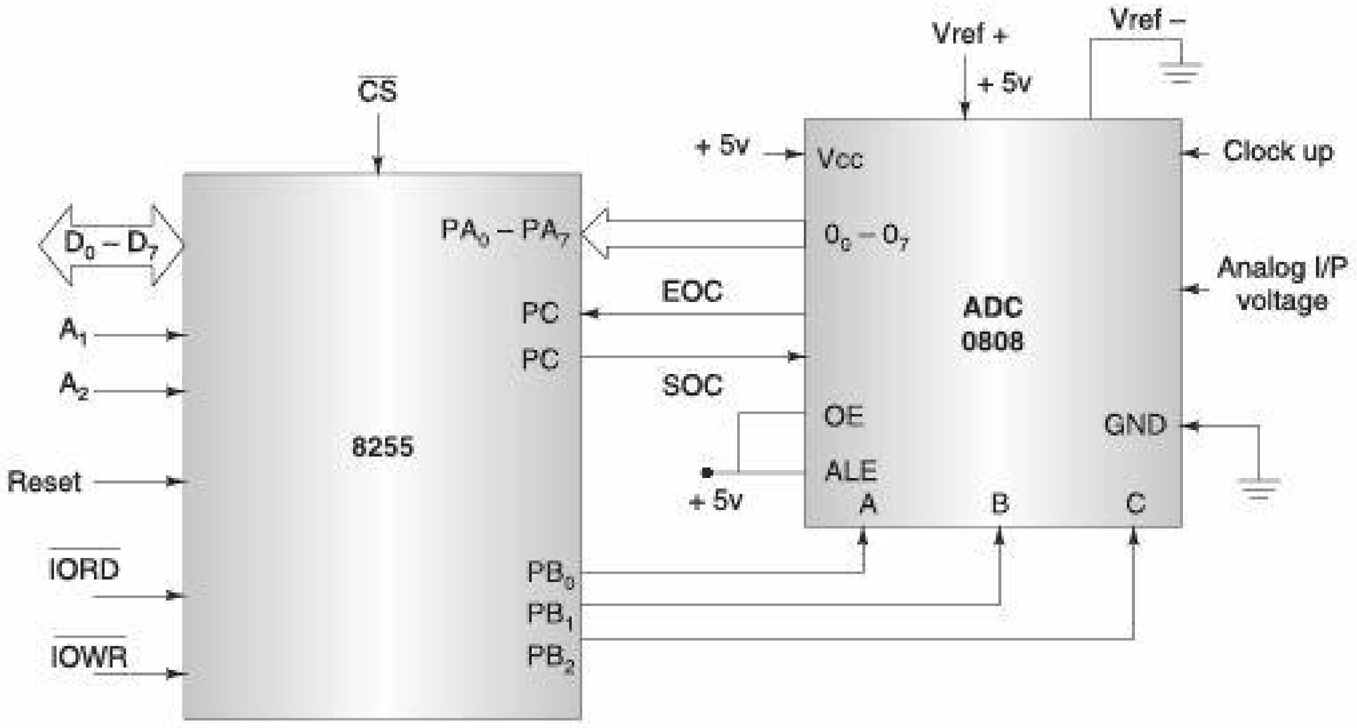


Fig. 6.29(b) Interfacing of 8255 with ADC 0808.

In the 3rd machine cycle RD of microprocessor bus goes low. High IO/M and low RD of microprocessor bus make RD of ADC low. When both CS and RD of ADC become low, digital output is available on output lines of the ADC. The lines are directly connected data lines of the microprocessor. The data is placed in the accumulator for further processing.

6.5.2.8 Interfacing ADC 0808 with 8085 using 8255

Example: 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 analog input is present at I/P2 of the ADC and a clock input of suitable frequency is available for ADC.

Solution: The analog input I/P2 is used and therefore address pins A, B, C should be 0, 1, 0 respectively to select I/P2. The OE and ALE pins are already kept at +5 V 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 a 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

Main Program:

```
MOV AL, 98h;            initialise 8255 as discussed above.
OUT CWR, AL
MOV AL, 02h;            Select I/P2 as analog
OUT Port B, AL;         input.
MOV AL, 00h;            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.
```

6.6 DIGITAL-TO-ANALOG CONVERTERS (DACS OR D/AS)

Introduction

Digital-to-analog converters are used to convert digital quantity to analog quantity. D/A converter produces an output current or voltage proportional to digital quantity (binary word) applied to its input. Nowadays, microcomputers are widely used for industrial control. The output of a microcomputer is a digital quantity. In many applications the digital output of a microcomputer has to be converted into analog quantity which is used for the control of relay, actuator, etc. In communication system digital transmission is faster and convenient but the digital signals have to be converted back to analog signals at the receiving terminal D/A converters are also used as a part of the circuitry of several A/D converters.

A digital-to-analogue converter (DAC) is a chip or circuit that converts a number (digital) into a voltage or current (analogue). The DAC is a useful interface between a computer and an output transducer. For example, DACs are used to control devices that require a continuous range of control voltages or currents such as electro-acoustic transducers (speakers), some types of variable-speed motors, and many other applications where an analog signal output is required. Another common application is to recreate waveforms from digital signals, for example, in CD players.

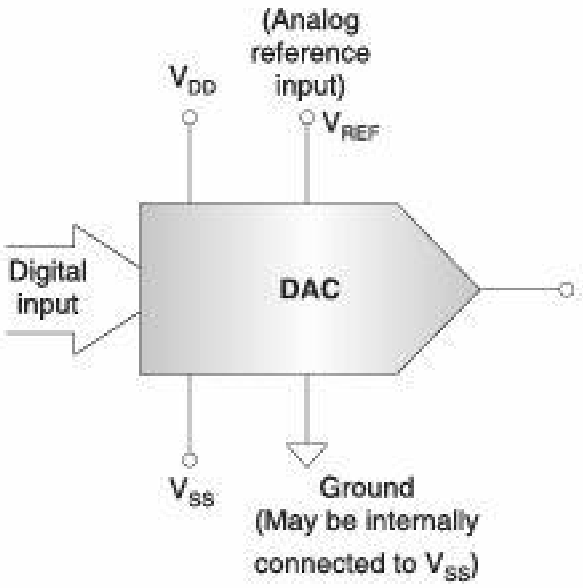


Fig. 6.30(a) The Basic DAC.

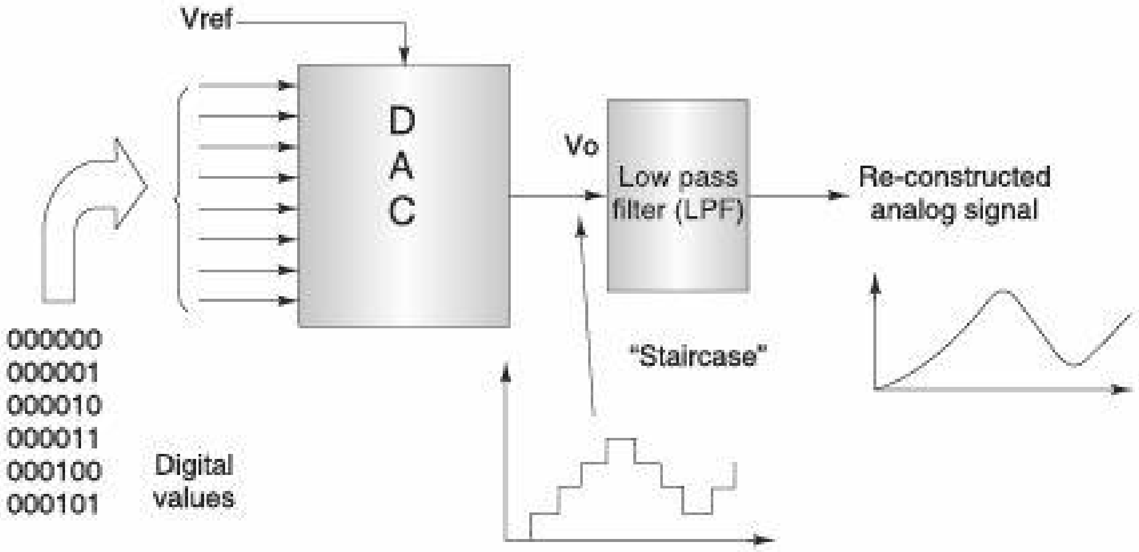


Fig. 6.30(b) Example Digital-to-Analog Converter.

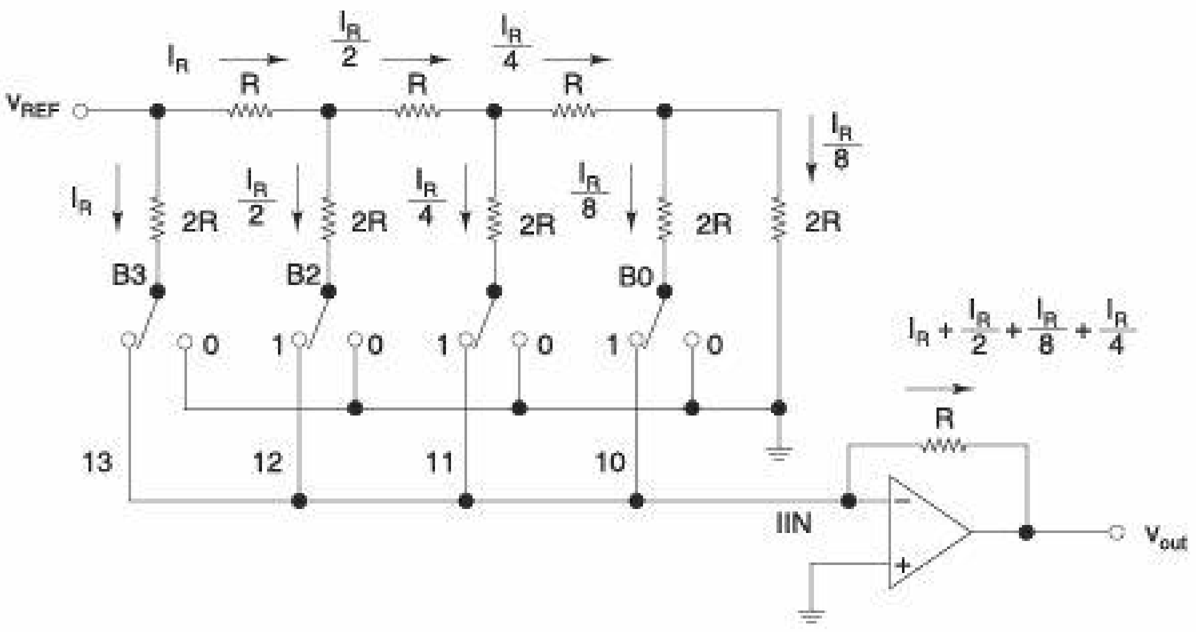


Fig. 6.31 $\lim_{N \approx 0}$ FS.

6.6.1 Working Principle

ADC contains a ladder network. The network has input points for binary bits of the digital word. When MSB of the digital word is 1 it produces an output current, $I_{REF}/2$. The bit next to the MSB produces $I_{REF}/4$, and so on. A D/A converter produces an output current or voltage proportional to the magnitude of the digital word applied to it.

DEFINITIONS

Resolution of the DAC is equal to the number of bits in the applied digital input word.

The full scale (FS):

FS = Analog output when all bits are 1 - Analog output all bits are 0

$$FS = \left(V_{REF} - \frac{V_{REF}}{2^N}\right) - 0 = V_{REF}\left(1 - \frac{1}{2^N}\right)$$

Full scale range (FSR) is defined as

$$\text{FSR} = \lim_{N\text{Ø}} FS = V_{REF}$$

Quantization noise is the inherent uncertainty in digitizing an analog value with a finite resolution converter.

Program:

Memory Address	Machine Codes	Labels	Mnemonics	Operands	Comments
FC00	3E, 98		MVI	A, 98 H	;Get control word for 8255
FC02	D3, 0B		OUT	0B	;Initialize ports
FC04	3E, 80		MVI	A, 80	;Get 80 for digital input to DAC
FC06	D3, 09		OUT	09	;Input 80 to DAC through port B
FC08	76		HLT		;Stop

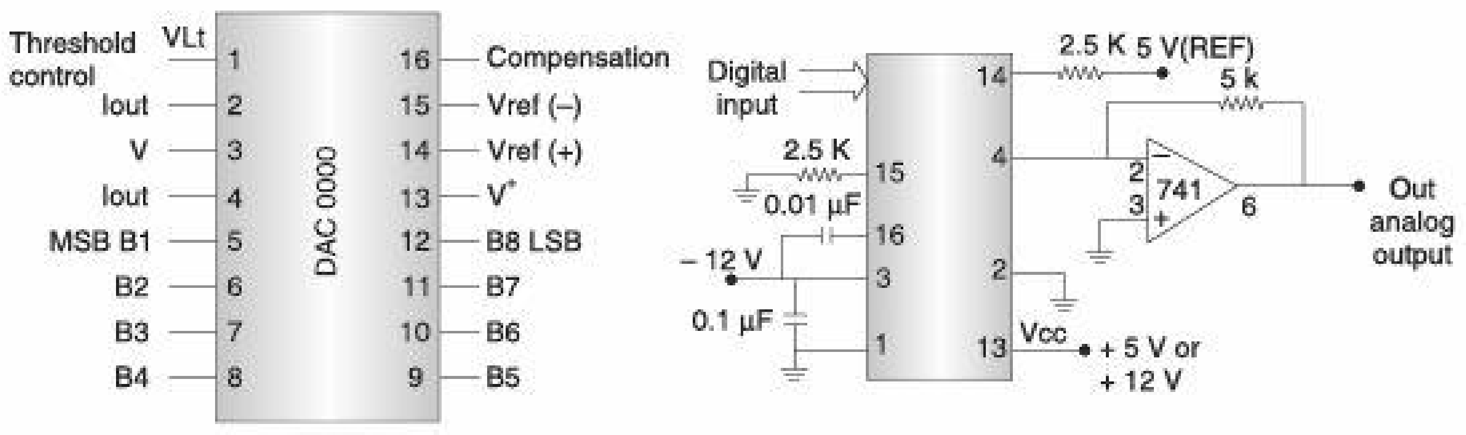


Fig. 6.32 Pin Diagram and Interfacing of DAC 0800.

Corresponding to digital input 80, analog output is 5 V. Now change the digital input to 00, FF, etc. and note down the reading of analog output.

For Bipolar Operation the following Modification in the Circuit has to be done.

Connect pin 2 of DAC to non-inverting terminal of op. amp. This common point is earthed through a 5 K ohm resistor. The connections of pin 4 of DAC and inversing terminal op. amp will remain as before.

Input and output relation for bipolar operations are:

Digital input to D/A Converter	*Analog Output*
00	– 10 V
80	0 V
FF	+ 10 V

6.7 MICROPROCESSOR AS TRAFFIC LIGHT CONTROL SYSTEM

A traffic light control and information transmission device comprising a microprocessor located inside the traffic control box and to control all the circuitries; a traffic light controller connected to and controlled by said microprocessor to send out the stop, go and direction signals; an electronic display board connected to and controlled by the said microprocessor to display characters, patterns and graphic images; a video camera connected to and controlled by said microprocessor to monitor the traffic flow; a compression circuitry connected to the said microprocessor and the said video camera, the said compression circuitry compresses the image data captured by the the said video camera and sends the data to the said microprocessor; an I/O interface connected to the said microprocessor to receive, transmit data and control signals; a traffic flow detector connected to the said I/O interface and gathering the traffic flow information; the traffic flow information is input to the said microprocessor; a DSL connected to the said I/O interface 6 and receiving, transmitting data and control signals; a broadband network linking the said DSL and a central traffic control computer together, the said central traffic control computer sends out and receives the data and controls signals to the said microprocessor through the said broad band network.

The 8085 Microprocessor is a popular microprocessor used in industries for various applications. Such as traffic light control, temperature control, stepper motor control, etc. In this project, the traffic lights are interfaced to the microprocessor system through buffer and ports of programmable peripheral Interface 8255. So the traffic lights can be automatically switched ON/OFF in the desired sequence. The interface board has been designed to work with parallel port of microprocessor system.The hardware of the system consists of two parts. The first part is microprocessor based system with 8085. Microprocessor as CPU and the peripheral devices like EPROM, RAM, keyboard & display

Controller 8279, programmable as peripheral interface 8255, 26-pin parallel port connector, 21 keys hexa key pad and six number of seven-segment LEDs. The second part is the traffic light controller interface board, which consists of 36 LEDs in which 20LEDs are used for vehicle traffic and they are connected to 20 port lines of 8255 through buffer. Remaining LEDs are used for pedestrian traffic. The traffic light interface board is connected to main board using 26 core flat cables to 26-pin port connector. The LEDs can be switched ON/OFF in the specified sequence by the microprocessor in Figure 6.33.

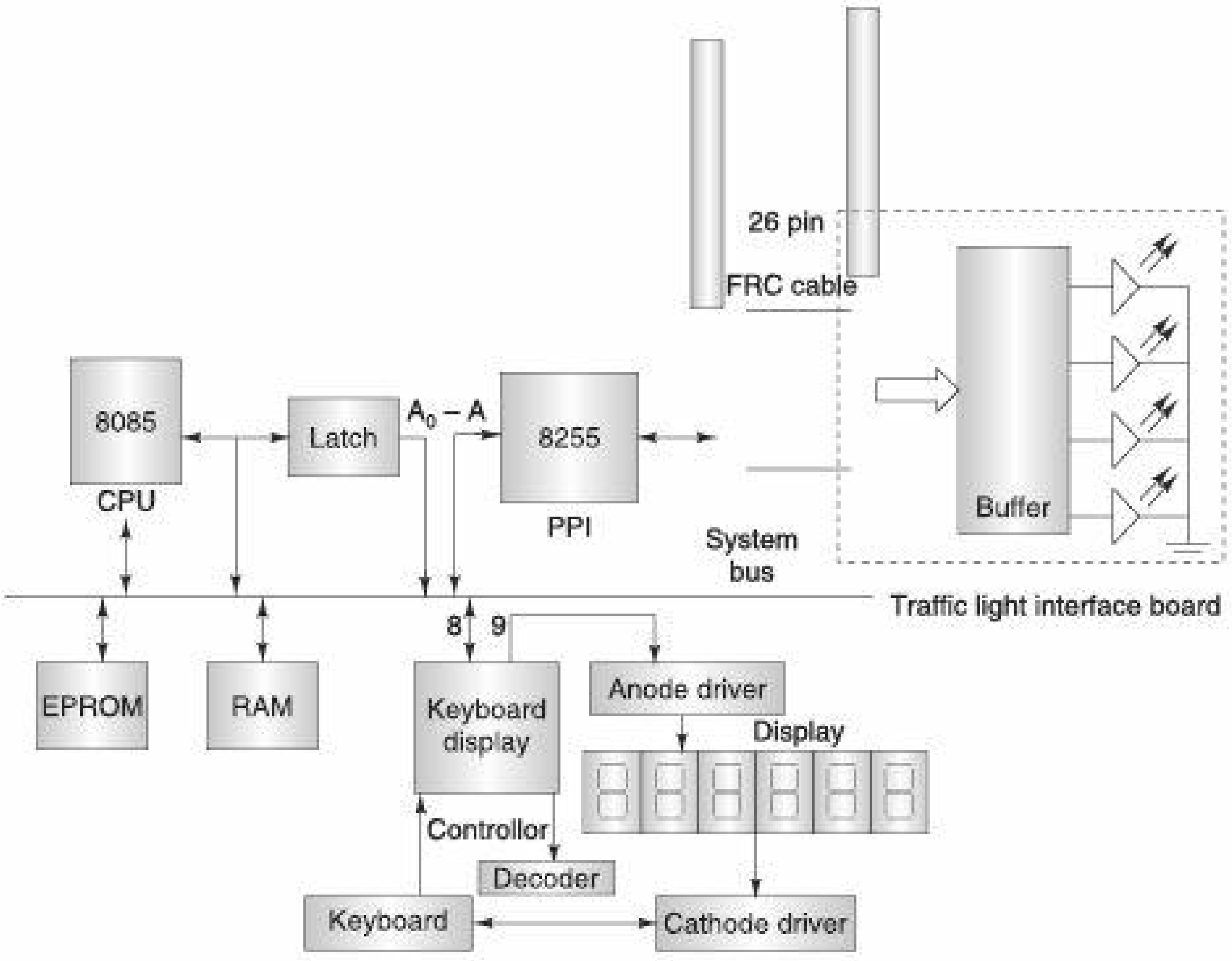


Fig. 6.33 Block Diagram of Traffic Control System Using 8085.

Main Program

```
          LXI SP,FFFFH
          MVI A,80H
          OUT 63H
CALL 1:   MVI A,21H
          OUT 60H
          MVI ,0CH
          OUT 61H
          CALL DELAY1
          MVI A,12H
          OUT 60H
          MVI A,0CH
          OUT 61H
          CALL DELAY2
          MVI A,0CH
          OUT 60H
          MVI A,21H
          OUT 61H
          CALL DELAY1
          MVI A,0CH
          OUT 60H
          MVI A,12H
          OUT 61H
          CALL DELAY2
          JMP CALL1
```

Things to Remember

◊ The seven-segment LED display is a multiple display which can display all decimal digits and some letters. It has the smallest number of separately controlled light emitting diodes (LEDs).

◊ Multiple displays of 7-segment LED, 14-segment LED and dot metrix type are available in the market.

◊ There are two types of 7-segment display, namely, common cathode and common anode display. 7446A, 74L46, 7447, 74LS47 and 74LS47 are decodes/drivers for common anode type seven-segment displays. 7448, 74LS48, 7449 and 74LS49 are decoders/drives for common cathode type seven-segment display.

◊ The seven-segment displays are not connected to I/O ports directly. They are connected through buffers or drivers/decoders.

◊ Hex buffers 7407 are used to interface 7-segment displays to the microprocessor.

◊ In traffic lights, green light glows to allow crossing, yellow to make alert, and red does not allow crossing.

◊ Any data acquisition system will consist of two parts, firstly the measurement of physical quantities and second A/D converters. The data to be acquired may be temperature, pressure, strain and deflection. So this has to be sensed by transducers and then converted into digital by A/D converters.

◊ A thermistor is a semiconductor device fabricated from a sintered mixture of metal alloys, having a large negative temperature coefficient.

◊ Strain is a measure of the deformation produced by the application of external forces.

◊ The acquisition time is the time taken by the S/H circuit to charge the capacitor from the level of holding voltage to the new value of input voltage after the input voltage is applied to the hold capacitor.

◊ The aperture time is the delay between the hold command and the moment at which the input voltage is applied to the hold capacitor. It is very small in nanosecond.

Questions and Answers

1. Give the application of microprocessors.

Answer: Microprocessors are applicable to a wide range of information processing tasks, ranging from general computing, up to real-time monitoring systems. The microprocessor facilitates new ways to communicate and makes use of the vast information available to us online and offline. Most electronic devices–including everything from computers, remote controls, washing machines, microwaves and cell phones to iPods and more–contain a built-in microprocessor.

(a) **General Computing:** Microprocessors commonly used in general computing tasks include those embedded in your laptop or desktop computers. These microprocessors are responsible for the core computing processes of your computer, such as calculation and data transfer.

(b) **Signal Processing:** Microprocessors are also used as "signal processors" for decoding radio signals and digital signals, such as those used in digital TVs. Moreover, microprocessors used in signal processing allows people, from different parts of the world, to talk and see each other via their computer screens using small video cameras, which are usually embedded in microphones.

(c) **Real-Time Computing:** In real-time computing systems, microprocessors are embedded in security devices such as the anti-lock braking system (ABS) that are widely used in modern automobiles. The microprocessor detects motions and changes, that are relative to the surrounding or environment of the security device and sends signals that correspond to the changes that it detected.

2. What is a 7-segment display?

Answer: The seven-segment LED display is a multiple display. It can display all decimal digits and some letters. It is very popular among multiple displays as it has the smallest number of separately controlled light emitting diodes (LEDs). Multiple displays of 7-segment LED, 14-segment LED and dot matrix type are available. These displays give better representation of alphanumeric characters by enquired complex circuitry.

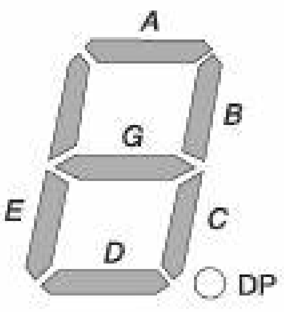


Fig. 2 Seven-Segment Display.

In seven-segment display, there are seven light emitting diodes (LED) as shown in Figure 2. Each LED can be controlled separately. To display a digit or letter the desired segments are made ON as shown in Figure 2(a).

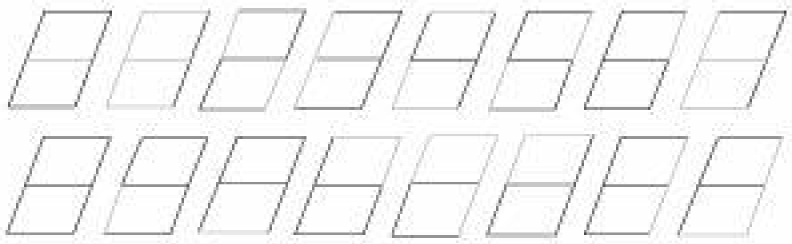


Fig. 2(a) Hexadecimal Digits using 7 Segments.

3. What are various types of seven-segment display?

Answer: There are two types of 7-segment display, namely, common cathode type and common anode type as shown in Figure 3 (a) and (b). When a +5 V d.c.is applied to any segment, the corresponding diode emits light. This applying logic '1', i.e., positive logic to the desired segments, the desired letter or decimal number can be displayed. In a common-anode type display all the 7 anodes are tied together and connected to +5 V supply as shown in Figure 6.3(b) particular segment will emit light when 0 logic is applied to it:

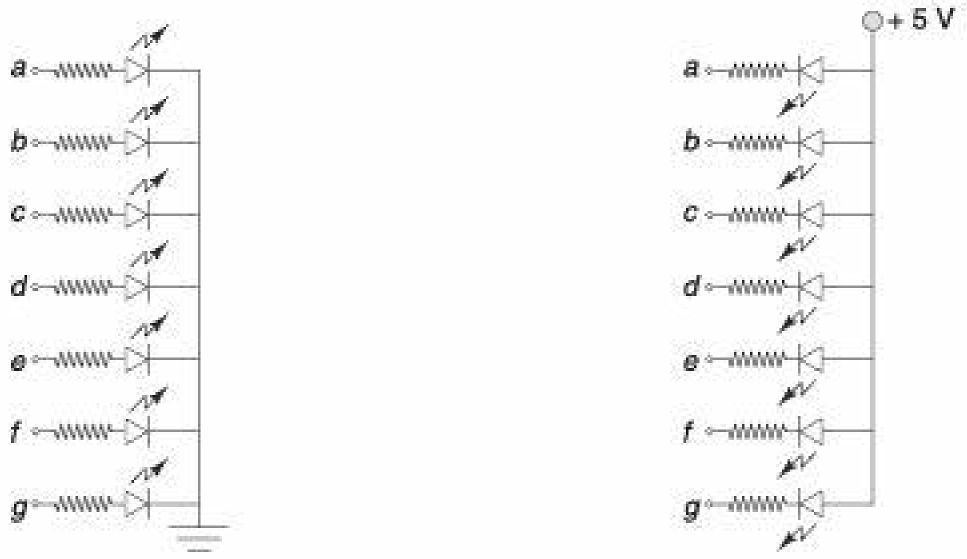
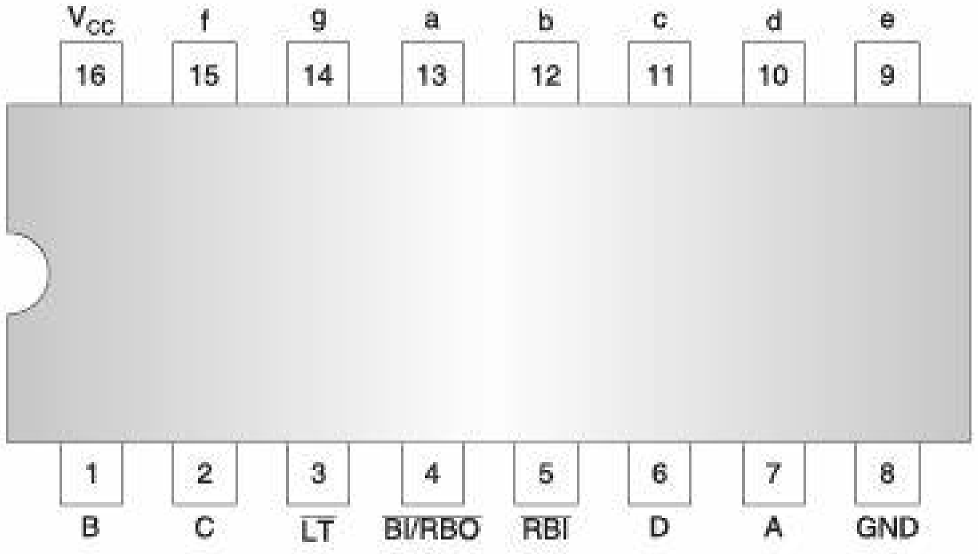


Fig. 3(a) Common Cathode 7-Segment Display (b) Common Anode 7-Segment Display.

4. Give the pin diagram of 7448 Ic.

Answer:



5. How the measurement of physical quantities is done in microprocessor system?

Answer: Microprocessor-based systems are widely used in industries for the measurement, and display of physical quantities like temperature, pressure, speed, flow, etc. For the measurement of physical quantities, transducers are used to give electrical signal proportional to the input. If the electrical signal is small, it is amplified using amplifiers. The electrical signal is applied to an A/D converter which is connected to a microcomputer. If more than one physical quantity is to be monitored, a multiplexer is included in the interface. A schematic diagram for general interface is shown in Figure 5 below.

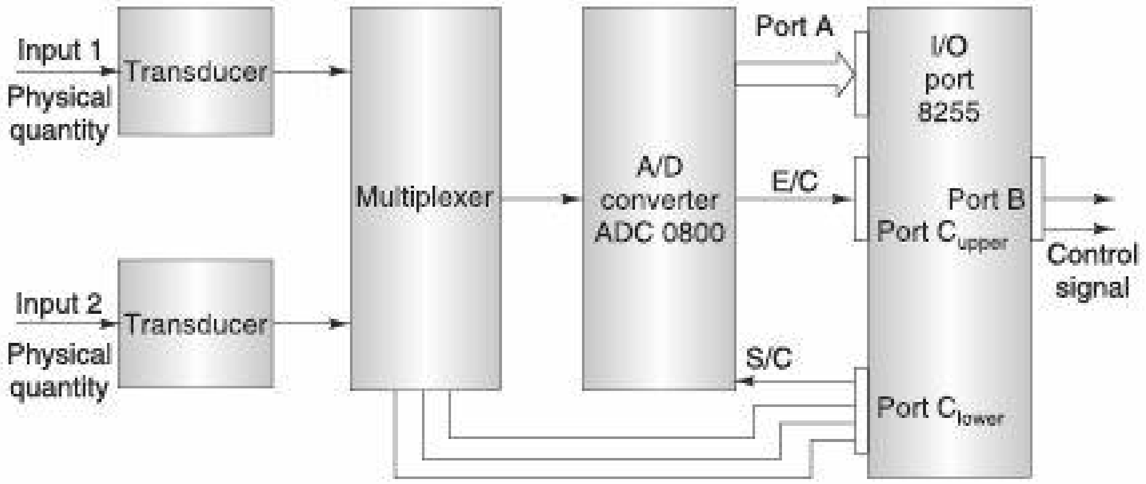
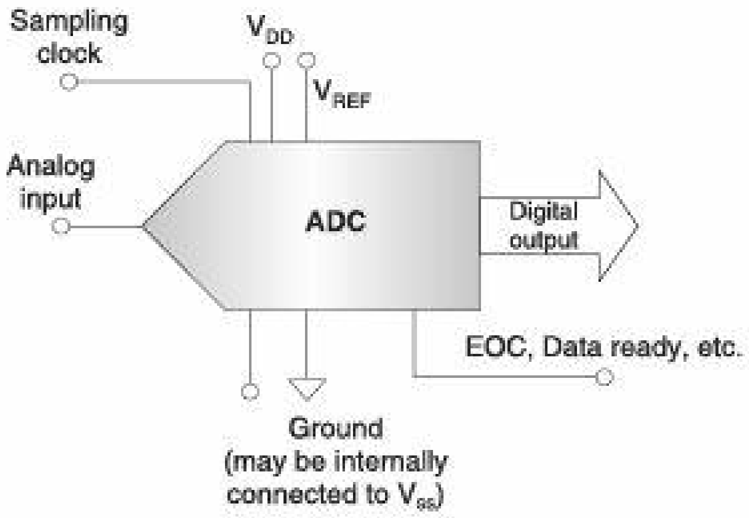


Fig. 5 Measurement of Physical Quantities.

6. What is ADC?

Answer: Analog refers to physical quantities that vary continuously instead of discretely. Physical phenomena typically involve analog signals. Examples include temperature, speed, position, pressure, voltage, altitude, etc. Microprocessors work with digital quantities (values taken from the discrete domain). For a digital system to interact with analog systems, conversion between analog and digital values is needed. Building blocks to perform the conversions are:

(1) Digital to analog converters (DACs), (2) Analog to digital converters (ADCs). A digital to analog converter has a digital input that specifies an output whose value changes in steps. These step changes are in volts or amperes. The analog to digital converter has an input that can vary from a minimum to a maximum value of volts or amperes. The output is a digital number that represents the input value.



7. Give functions of 1 bit comparator?

Answer: A comparator is a 1-bit ADC shown in Figure 6.20. If the input is above a threshold, the output has one logic value, below it has another. There is no ADC architecture which does not use at least one comparator of some sort. So while a 1-bit ADC is of very limited usefulness it is a building block for other architectures.

Exercise

1. What do you mean by 7-segment display?
2. Discuss the interfacing circuit for 7-segment display.
3. Explain the transducer for measuring temperature. How is this interfaced for getting digital output?
4. Explain the transducer for measuring strain. How is this interfaced for getting digital output?
5. Explain the transducer for measuring deflection. How is this interfaced for getting digital output?
6. How will you obtain clock signal for an A/D converter?
7. Discuss the main feature of ADC 0800 and ADC 0808.
8. Which of the following A/D converters accepts negative signals?
   (i) ADC 0800 (ii) ADC 0808/0809 (iii) 08167/0817.
9. Show the interface connections of ADC 0800 to 8085. Show important signals.
10. Show the interface connections of ADC 0808 to 8085. Show important signals.
11. What is the function of a sample and hold circuit? Show the interface connections of ADC 0808 and S/H circuit to a microprocessor.

12. What do you understand by data acquisition system? Show an interface connection of a data acquisition system to a microprocessor.
13. What is the function of an analog multiplexer? Show an interface connection of a multiplexer, S/H circuit and A/D converter to a microprocessor.
14. What is a D/A converter? Discuss its applications.

Chapter 7

Intel 80XXX Series

- The 80186 Microprocessor
- The 80286 Microprocessor
- Introduction to 80386

7.1 THE 80186 MICROPROCESSOR

The 80186 is a very high integration 16-bit microprocessor. It combines 15 ± 5 of the most common microprocessor system components onto one chip while providing twice the performance of the standard 8086. The 80186 is a highly integrated and advanced version, so it is made compatible with the 8086/8088 microprocessors and adds 10 new instruction types to the 8086/8088 instruction set. The 80186 and the 80186XL devices are functionally and register compatible. The processor is provided with an on-chip clock generator for both internal and external clock generation. The clock generator consists of a crystal oscillator, a divide-by-two counter, synchronous and asynchronous ready inputs, and reset circuitry. New additional instructions are introduced with each new 8086 processor as the technology advanced.

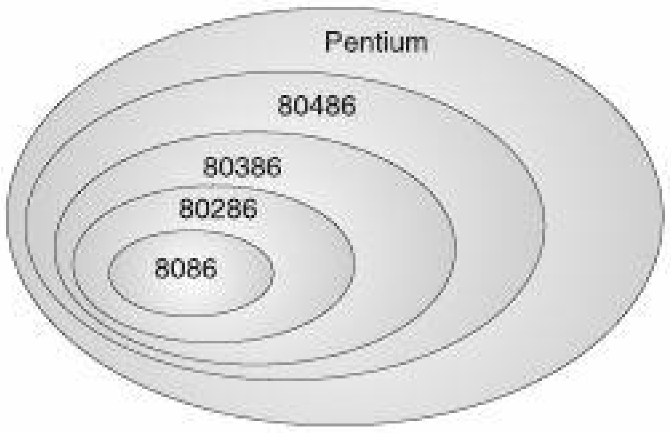


Fig. 7.1 The Intel 80XXX Family.

7.1.1 Architecture

As the 8086 microprocessor contains memory in segments, same as in the 80186. It also contains program, data and stack memories that occupy the same memory space. The total addressable memory size is 1 MB. As most of the processor instructions use 16-bit pointers, the processor can effectively address only 64 KB of memory. To access memory outside of 64 KB, the CPU uses special segment registers to specify where the code, stack and data 64 KB segments are positioned within 1 MB of memory.

In this 16-bit pointer and data register are stored as

Higher order byte	Low order byte
51H	50H

8086 uses physical address for deriving a segment by considering its segment starting address as offset address. Address of segment = 32-bit segment address + physical address.

7.1.1.1 Program Memory

The basic function of microprocessor is to execute a program, and this program can be located in a proper section of memory where it is being called by using instruction (Jump, call). This is called program memory.

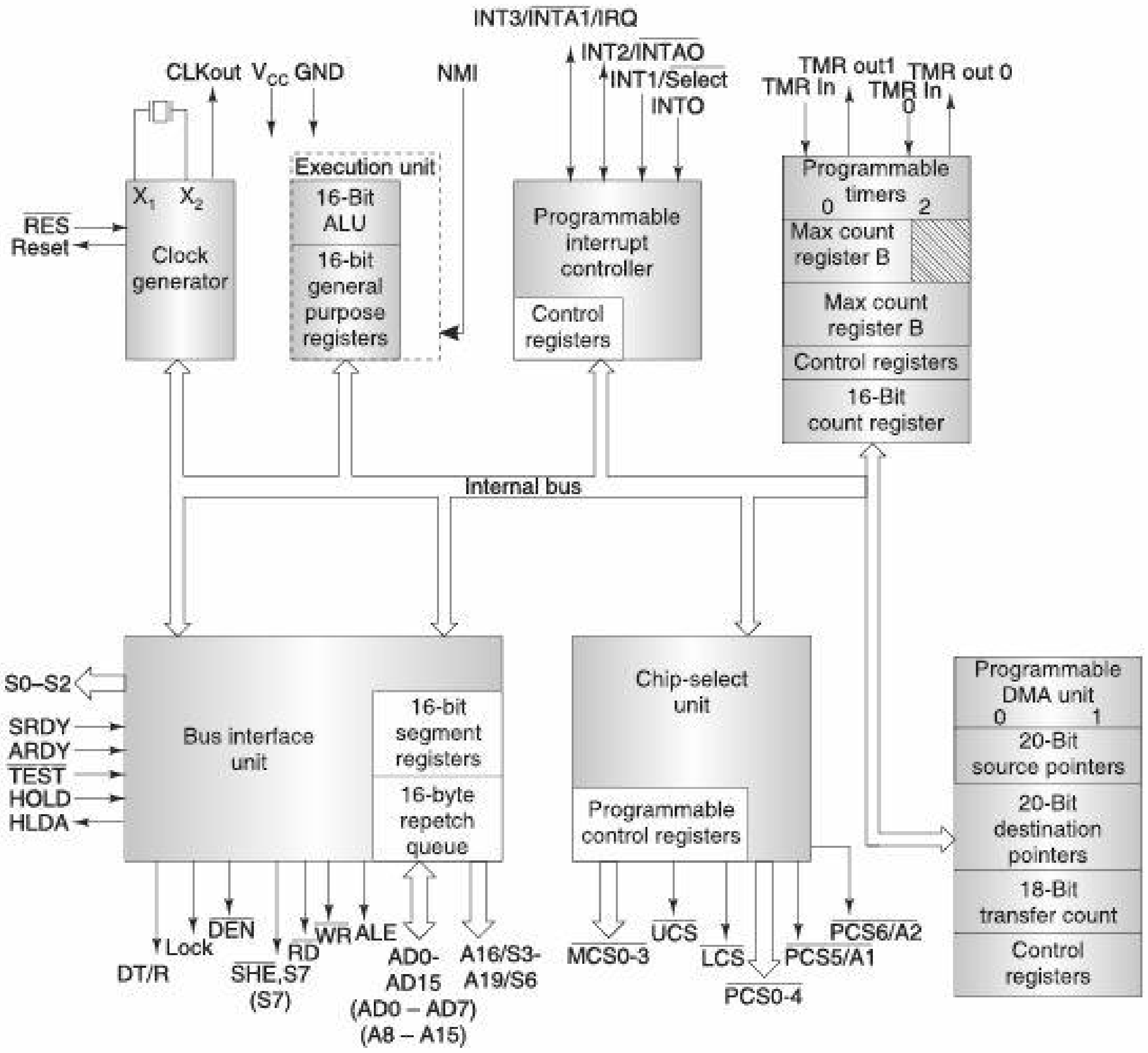


Fig. 7.2 Block Diagram of 80186.

7.1.1.2 Data Memory

80186 contains memory in four segments in which each segment only has accessible memory upto 64 KB. Every segment has its notation for accessing, i.e., DS = Data segment, CS = Code segment, SS = Stack segment, ES = Extra segment. Word data can be located at odd or even byte boundaries. The processor uses two memory accesses to read 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.

7.1.1.3 Stack Memory

Generally, stack memory is used for holding address of next instruction that is being executed. This can be placed anywhere in memory. The stack can be located at odd memory addresses, but it is not recommended for performance reasons.

Few reserved location in memory segment:

- 0000h-03FFh are reserved for interrupt vectors. Each interrupt vector is a 32-bit pointer in format segment: offset.
- FFFF0h-FFFFFh—after RESET, the processor always starts program execution at the FFFF0h address.

7.1.2 Interrupts

Interrupts have already been discussed in the previous section. The 80186 microprocessor has the almost same interrupts as 8086. 80186 contains both hardware and software interrupts.

INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction. When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine, address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction. Integrated 80186 peripherals generate the following hardware interrupts (higher priority interrupts are listed first):

- Timer 0
- Timer 1
- Timer 2
- DMA 0
- DMA 1
- INT0
- INT1
- INT2
- INT3

NMI is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e., the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.

Apart from NMI, the interrupt controller has four interrupts INTO through INT3. INT2/INTAO and INT3/INTAI can be programmed to use as interrupt input as well as interrupt acknowledge outputs. This mode of 80186 is used to interface it with an interrupt controller 8259 A. Software interrupt processing is the same as for the hardware interrupts. Software interrupts have the same priority as the NMI interrupt.

I/O ports

65536 8-bit I/O ports. These ports can also be addressed as 32768 16-bit I/O ports.

Reserved ports

- 00F8h—00FFh
- FF00h—FFFFh: 256-byte control block. This is a default block location after RESET.

If necessary, the block can be remapped to different place in I/O or memory space. For description of the control block please see "Control registers" in the "Registers" section below.

7.1.3 Registers

As described earlier, most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located, the processor uses four-segment registers:

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack segment (SS) is a 16-bit register containing address of 64 KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

Data segment (DS) is a 16-bit register containing address of 64 KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra segment (ES) is a 16-bit register containing address of 64 KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS, DS or ES prefix.

All general registers of the 80186 microprocessor can be used for arithmetic and logic operations. The general registers are:

Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low-order byte of the word, and CH contains the high-order byte. Count register can be used as a counter in string manipulation and shift/rotate instructions.

Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

The following registers are both general and index registers:

Stack Pointer (SP) is a 16-bit register pointing to program stack.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Other registers:

Instruction Pointer (IP) is a 16-bit register.

Flags is a 16-bit register containing nine 1 bit flags:

- Overflow Flag (OF)—set if the result is too large positive number, or is too small negative number to fit into destination operand.
- Direction Flag (DF)—if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.
- Interrupt-enable Flag (IF)—setting this bit enables maskable interrupts.
- Single-step Flag (TF)—if set then single-step interrupt will occur after the next instruction.
- Sign Flag (SF)—set if the most significant bit of the result is set.
- Zero Flag (ZF)—set if the result is zero.
- Auxiliary carry Flag (AF)—set if there was a carry from or borrow to bits 0-3 in the AL register.
- Parity Flag (PF)—set if parity (the number of "1" bits) in the low-order byte of the result is even.
- Carry Flag (CF)—set if there was a carry from or borrow to the most significant bit during last result calculation.

The 80186 has a built-in address decoder for memory and I/O. The unit can be programmed to generate an active low chip select signal when the address is in a particular range. It has six memory address chip select signals; lower chip select (LCS), upper chip select (UCS) and middle chip select (MCSO-3) lines. The LCS is asserted by the addresses 00000H and address specified in the control word. The UCS signal is fixed and will be asserted by the highest address FFFFFH.

There are three 16-bit programmable timer/counters, Timer 0, Timer 1, and Timer 2. The timers/counters 0 and 1 are similar to the 8054 counters. The processor clock can be an input to these counters depending on the control word. The third counter/timer is connected to the processor clock and its output can be directed internally to DMA input, interrupt input or to counters/timers 1 or 2 or both.

Its DMA section has two channels DRQ1 and DRQ2, these follow external devices to request for a DMA transfer. Each DMA channel has source and destination registers of 20-bit transfer count register.

7.1.4 Instruction Set

80186 instruction set consists of the following instructions:

- Data moving instructions.
- Arithmetic—add, subtract, increment, decrement, convert byte/word and compare.
- Logic—AND, OR, exclusive OR, shift/rotate and test.
- String manipulation—load, store, move, compare and scan for byte/word.
- Control transfer—conditional, unconditional, call subroutine and return from subroutine.
- Input/Output instructions.
- Other—setting/clearing flag bits, stack operations, software interrupts, etc.

ADDRESSING MODES

Implied: the data value/data address is implicitly associated with the instruction.

Register: references the data in a register or in a register pair.

Immediate: the data is provided in the instruction.

Direct: the instruction operand specifies the memory address where data is located.

Register Indirect: instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.

Based: 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.

Indexed: 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

Based Indexed: the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

Based Indexed with Displacement: 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

7.1.5 Pin Description

VCC: System Power: +5 V power supply.

VSS: This is system ground pin, it is connected to 0 volt.

RESET: Reset Output indicates that the CPU is being reset, and can be used as a system reset. It is active HIGH, synchronized with the processor clock, and lasts an integer number of clock periods corresponding to the length of the RES signal.

X1, X2: Crystal Inputs X1 and X2 provide external connections for a fundamental mode parallel resonant crystal for the internal oscillator. Instead of using a crystal, an external clock may be applied to X1 while minimizing stray capacitance on X2. The input or oscillator frequency is internally divided by two to generate the clock signal (CLKOUT).

CLKOUT: Clock Output provides the system with a 50% duty cycle waveform. All device pin timings are specified relative to CLKOUT.

RES: An active RES causes the processor to immediately terminate its present activity, clear the internal logic, and enter a dormant state. This signal may be asynchronous to the processor clock. The processor begins fetching instructions approximately 6/2 clock cycles after RES is returned.

TEST: TEST is examined by the WAIT instruction. If the TEST input is HIGH when WAIT execution begins, instruction execution will suspend. TEST will be resampled until it goes LOW, at which time execution will resume. If interrupts are enabled while the processor is waiting for TEST, interrupts will be serviced. During power-up, active RES is required to configure TEST as an input. This pin is synchronized internally.

TMR IN 0, TMR IN 1: Timer inputs are used either as clock or control signals, depending upon the programmed timer mode. These inputs are active HIGH (or LOW-to-HIGH transitions are counted) and internally synchronized.

TMR OUT 0, TMR OUT 1: Timer outputs are used to provide single pulse or continuous waveform generation, depending upon the timer mode selected.

DRQ0, DRQ1: DMA Request is asserted HIGH by an external device when it is ready for DMA Channel 0 or 1 to perform a transfer. These signals are level-triggered and internally synchronized.

NMI: The Non-Maskable Interrupt input causes a Type 2 interrupt. An NMI transition from

LOW to HIGH is latched and synchronized internally, and initiates the interrupt at the next instruction boundary. NMI must be asserted for at least one clock. The Non-Maskable Interrupt cannot be avoided by programming.

INT0, INT1/SELECT, INT2/INTA0, INT3/INTA1/IRQ: Maskable Interrupt Requests can be requested by activating one of these pins. When configured as inputs, these pins are active HIGH. Interrupt Requests are synchronized internally. INT2 and INT3 may be configured to provide active-LOW interrupt-acknowledge output signals. All interrupt inputs may be configured to be either edge- or level-triggered. To ensure recognition, all interrupt requests must remain active until the interrupt is acknowledged. When Slave Mode is selected, the function of these pins changes.

A16-19/S3-S6: Address Bus Outputs (16 ± 19) and Bus Cycle Status (3 ± 6) indicate the four most significant address bits during T1. These signals are active HIGH. During T2, T3, TW, and T4, the S6 pin is LOW to indicate a CPU-initiated bus cycle or HIGH to indicate a DMA initiated bus cycle. During the same T-states, S3, S4, and S5 are always LOW. The status pins float during bus HOLD or RESET.

AD0-AD15: Address/Data Bus signals constitute the time multiplexed memory or I/O address (T1) and data (T2, T3, TW, and T4) bus. The bus is active HIGH. A_0 is analogous to BHE for

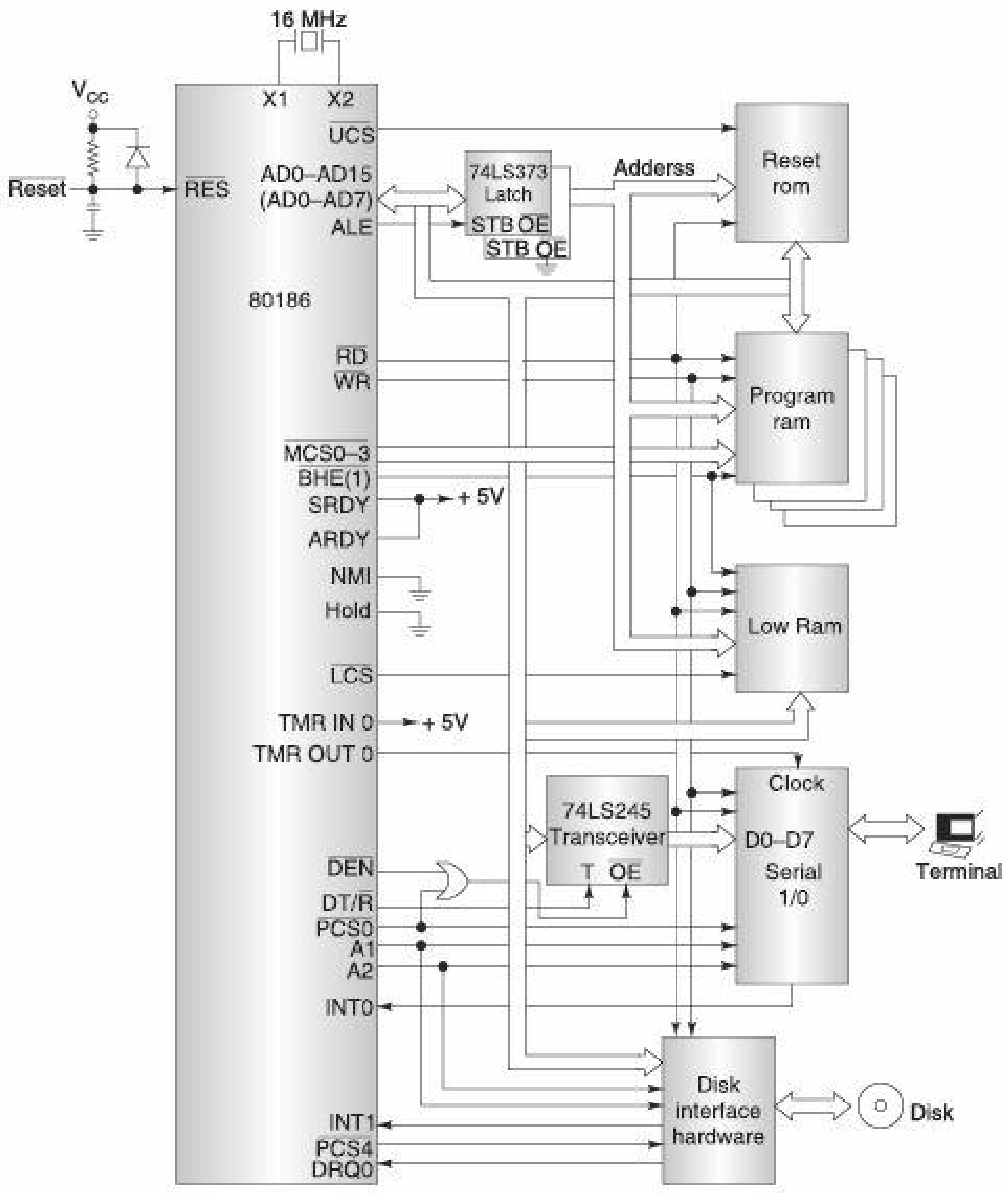


Fig. 7.3 80186 Pin Description.

the lower byte of the data bus, pins D7 through D_0. It is LOW during T1 when a byte is to be transferred onto the lower portion of the bus in memory or I/O operations. BHE does not exist on the 80188, as the data bus is only 8 bits wide.

BHE/S7: This is bus high enable pin which indicates transfer of data on the upper byte of the data bus D_{15}-D_8. For data transfer, BHE and A_0 are encoded.

BHE and A0 Encoding (80186 Only)		
BHE Value	A0 Value	Function
0	0	Word Transfer
0	1	Byte Transfer on upper half of data bus (D15-D8)
1	0	Byte Transfer on lower half of data bus (D7-D0)
1	1	Reserved

ALE/QS0: Address Latch Enable/Queue Status 0 is provided by the processor to latch the address. ALE is active HIGH. Addresses are guaranteed to be valid on the trailing edge of ALE. The ALE rising edge is generated off the rising edge of the CLKOUT immediately preceding T1 of the associated bus cycle, effectively one-half clock cycle earlier than in the 8086. The trailing edge is generated off the CLKOUT rising edge in T1 as in the 8086. Note that ALE is never floated.

WR/QS1: Write Strobe/Queue Status 1 indicates that the data on the bus is to be written into a memory or an I/O device. WR is active for T2, T3 and TW of any write cycle. It is active LOW, and floats during HOLD. When the processor is in queue status mode, the ALE/QS0 and WR/QS1 pins provide information about processor/instruction queue interaction.

QS1	QS0	Queue Operation
0	0	No queue operation
0	1	First opcode byte fetched from the queue
1	1	Subsequent byte fetched from the queue
1	0	Empty the queue

RD/QSMD: Read Strobe is an active LOW signal which indicates that the processor is performing a memory or I/O read cycle. It is guaranteed not to go LOW before the A/D bus is floated. An internal pull up ensures that RD is HIGH during RESET. Following RESET, the pin is sampled to determine whether the processor is to provide ALE, RD, and WR, or queue status information. To enable Queue Status Mode, RD must be connected to GND. RD will float during bus HOLD.

ARDY: Asynchronous Ready informs the processor that the addressed memory space or I/O device will complete a data transfer. The ARDY pin accepts a rising edge that is asynchronous to CLKOUT, and is active HIGH. The falling edge of ARDY must be synchronized to the processor clock. Connecting ARDY HIGH will always assert the ready condition to the CPU. If this line is unused, it should be tied LOW to yield control to the SRDY pin.

SRDY: Synchronous Ready informs the processor that the addressed memory space or I/O device will complete a data transfer. The SRDY pin accepts an active-HIGH input synchronized to CLKOUT. The use of SRDY allows a relaxed system timing over ARDY. This is accomplished by elimination of the one-half clock cycle required to internally synchronize the ARDY input signal. Connecting SRDY high will always assert the ready condition to the CPU. If this line is unused, it should be tied LOW to yield control to the ARDY pin.

LOCK: It is an active low pin. This is a bus lock and indicates that other system bus masters cannot gain control of the system bus following the current bus cycle.

HOLD HLDA: HOLD indicates that another bus master is requesting the local bus. The HOLD input is active HIGH. HOLD may be asynchronous with respect to the processor clock. The processor will issue a HLDA (HIGH) in response to a HOLD request at the end of T4 or Ti. Simultaneous with the issuance of HLDA, the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will lower HLDA. When the processor needs to run another bus cycle, it will again drive the local bus and control lines.

UCS: Upper Memory Chip Select is an active LOW output whenever a memory reference is made to the defined upper portion (1 K ± 256 K block) of memory. This line is not floated during bus HOLD. The address range activating UCS is software programmable.

LCS: Lower Memory Chip Select is active LOW whenever a memory reference is made to the defined lower portion (1 K ± 256 K) of memory. This line is not floated during bus HOLD. The address range activating LCS is software programmable.

MCS0-3: Mid-Range Memory Chip Select signals are active LOW when a memory reference is made to the defined mid-range portion of memory (8 K ± 512 K). These lines are not floated during bus HOLD. The address ranges activating MCS0 ± 3 are software programmable.

PCS0-4: Peripheral Chip Select signals 0 ± 4 are active LOW when a reference is made to the defined peripheral area (64 KB I/O space). These lines are not floated during bus HOLD. The address ranges activating PCS0 ± 4 are software programmable.

PCS5/A1: Peripheral Chip Select 5 or latched A1 may be programmed to provide a sixth peripheral chip select, or to provide an internally latched A1 signal. The address range activating PCS5 is software-programmable. PCS5/A1 does not float during bus HOLD. When programmed to provide latched A1, this pin will retain the previously latched value during HOLD.

PCS6/A2: Peripheral Chip Select 6 or latched A2 may be programmed to provide a seventh peripheral chip select, or to provide an internally latched A2 signal. The address range activating PCS6 is software programmable. PCS6/A2 does not float during bus HOLD. When programmed to provide latched A2, this pin will retain the previously latched value during HOLD.

DT/R: Data Transmit/Receive controls the direction of data flow through an external data bus trans-receiver. When LOW, data is transferred to the processor.When HIGH, the processor places write data on the data bus.

DEN: Data Enable is provided as a data bus transreceiver output enable. DEN is active LOW during each memory and I/O access. DEN is HIGH whenever DT/R changes state. During RESET, DEN is driven HIGH for one clock, then floated. DEN also floats during HOLD.

7.2 THE 80286 MICROPROCESSOR

The 80286 is an advanced version of 8086. It is designed for multiprogramming/multiuser/multitasking environments. An 80286 has memory management. Capabilities that make it capable of addressing 1 GB virtual memory and 16 MB of physical memory. It provides separation between code and data modules. In fact, 80286 is an 8086 that is optimized to execute instructions in fewer clock cycles, that makes 80286 faster. The 80286 operates in two modes: Read Address mode and protected virtual address mode.

(PVAM). In both modes, the processor operates with full performance. The 80286 provides special operations to support the efficient implementation and execution of operating systems. It means one instruction can end execution of one task, save its state, switch to a new task, load its state, and start execution of the new task. The 80286 also supports virtual memory systems by providing a segment not present and restartable instructions. The 80286 provided the first glimpse into the world of the protection mechanisms then exclusive to the world of mainframes and minicomputers which would pave the way for the x86 and the IBM PC architecture to extend from the personal computer all the way to high-end servers, drive the market for other architectures all the way down to only the highest-end servers and mainframes, a fact which presumably gave the *IBM PC/AT* its name. The 80286 is an advanced high performance microprocessor with specially optimized capabilities for multiuser and multitasking systems. The 80286 has built in memory protection that supports operating systems and task isolation as well as program and data privacy within tasks. A 25 MHz 80286 microprocessor is capable of providing up to twenty times the maximum throughput that can be achieved on a 5 MHz 8086 processor based system. The 80286 microprocessor, however, does not have a multiplexed address data bus as in the 8086. As 80286 has 16 data lines and 24 address lines. The 80286 can thus directly address 40 MB of physical memory. However, due to the enhanced memory management unit (MMU) the 80286 has the capability of addressing 1G byte of memory.

7.2.1 The 80286 Architecture

Unlike 80186, the 16-bit 80286 microprocessor does not contain any additional peripheral control circuits such as DMA controller, etc. Physically the 80286 can be divided into four functional units.

- **Address Unit (AU):** This unit obtains the 20-bit physical address based on the 16-bit content of a segment register and 16-bit offset similar to 8086 microprocessor. This is called real address mode, the 80286 address 1 MB of physical memory. Hence, in this mode only 20 address A_0-A_{19} lines are used, and rest four (A_{20}-A_{23}) lines are ignored. In protected virtual address mode, all the 24 address lines (A_0-A_{23}) are used. In this mode AU operates as Memory Management Unit (MMU) and can address 2^{24} ~ 16 MB physical memory. The BU outputs the memory or I/O devices connected to the 80286 after it receives the address from AU.
- **Instruction Unit (IU):** This instruction unit of 80286 translates/decodes up to three pre-fetched instructions and places them in queue for execution by execution unit.
- **Bus Unit (BU):** The bus unit provides all memory and I/O read and write operations and performs data transfer between 80286 and the coprocessor such as 80287. It has 6-byte prefetch queue, which pre-fetches the instructions up to six bytes and places them in queue for the instruction unit.
- **Execution Unit (EU):** The EU executes the instructions received from the instruction unit in a sequential manner. The execution unit includes the CPU registers and Arithmetic Logic Unit (ALU). The EU contains flag registers, general-purpose registers, pointer and index registers and a 16-bit Machine Status Word (MSW) register.

7.2.2 Pin Description of 80286

Pin Name/ Symbol	Description
CLK	It provides the fundamental timing for 80286 system. It is divided by two inside the 80286 to generate the processor clock. The internal divide by two, circuitry can be synchronized to an external clock generator by a LOW to HIGH transition on the RESET input.
D_{15}-D_0	It inputs data during memory, I/O and interrupt acknowledge read cycles; outputs data during memory and I/O write cycles. The data bus is active HIGH and is held at high impedance to the last valid logic level during bus hold acknowledge.
A_{23}-A_0	It outputs physical memory and I/O port addresses. A_{23}-A_{16} are LOW during I/O transfers. A_0 is LOW when data is to be transferred on pins D_7-D_0. The address bus is active high and floats to three-state off during bus hold acknowledge.
BHE	BUS HIGH ENABLE—It indicates transfer of data on the upper byte of the data bus, D_{15}-D_8. Eight-bit oriented devices assigned to the upper byte of the data bus would normally use BHE to condition chip select functions. BHE is active LOW and floats to three-state OFF during bus hold acknowledge.

BHE	A_0	Function
0	0	Word transfer
0	1	Byte transfer on upper half (D_{15}-D_8)
1	0	Byte transfer on lower half (D_7-D_0)
1	1	Reserved

Pin Name/ Symbol	Description
S_1, S_0	BUS CYCLE STATUS—It indicates initiation of a bus cycle and along with M/IO and COD/INTA, defines the type of bus cycle. The bus is in a T8 state whenever one or both are LOW. S1 and S0 is active LOW and are held at a high impedance logic one during bus hold acknowledges.

COD/INTA	M/IO	S1	S0	**Bus Cycle Initiated**
0	0	0	0	Interrupt acknowledge
0	0	0	1	Reserved
0	0	1	0	Reserved
0	0	1	1	Not a status cycle
0	1	0	0	If A1 = 1 then halt; else shutdown
0	1	0	1	Memory data read
0	1	1	0	Memory data write
0	1	1	1	Not a status cycle
0	0	0	0	Reserved
1	0	0	1	I/O read
1	0	1	0	I/O write
1	0	1	1	Not a status cycle
1	1	0	0	Reserved
1	1	0	1	Memory instruction read
1	1	1	0	Reserved
1	1	1	1	Not a status cycle

Pin Name/ Symbol	Description
NMI	The request on non-maskable interrupt cannot be masked. No interrupt acknowledge cycles are performed. The interrupt enable bit in the 80286 flag word does not affect this input. The NMI input which is active high may be asynchronous to the system clock, and is edge triggered after internal synchronization.
READY	This is ready input which terminates the bus cycles. The bus cycles are extended without limit until terminated by READY low. The READY is an active low asynchronous input requiring setup and hold times relative to the system clock to be met for correct operation.

Contd...

Contd...

LOCK	This is bus lock and indicates that other system bus masters cannot gain control of the system bus following the current bus cycle.
HOLD/HLDA	These are hold request and hold acknowledge respectively. The hold input allows another local bus master to request control of the local bus. When control is granted, the 8086 floats its bus drivers to three-state off and then activates HLDA, thus entering the bus hold acknowledge condition.
INTR	The general interrupting requests come through this interrupt request pin. The 80286 has to suspend its current program execution and service a pending external request. These interrupt requests can be masked whenever a higher priority interrupt request arises.
PREQ	The processor extension request is used by the coprocessor such as 80287 to request the 80286 to transfer a data operand.
BUSY	The processor extension busy signal is sent by the coprocessor to indicate that it is busy. The low on 80286 causes to wait during a WAIT or ESC instruction.
PEACK	The processor extension acknowledges is used by the 80286 to acknowledge that the request operand is being transferred. This is an active low output and may be asynchronous to the system clock.
RESET	This pin is used for cleaning the internal logic of the 80286 and it is active high.
ERROR	This is processor extension error pin. The coprocessor uses this signal to cause 80286 to perform a type 16 interrupt when a wait or escape instruction is executing.
CAP	This is substrate filter capacitor, it must be connected between CAP pin and ground. This capacitor filters the output of the internal substrate bias generator.
VSS	This is system ground pin, it is connected to 0 volt.
VCC	This is system power supply pin, it should have 5 volt power supply.

7.2.3 Operating Modes of 80286

The Intel's 80286 operates in two modes: Real Address Mode and Protected Virtual Address Mode (PVAM). Upto some extent the operation of 80286 in real address mode is similar to that of 8086 operation, though 80286 has a few additional features and instructions which make it faster. In PVAM the functioning of 80286 is much different and uses memory management unit (MMU) for its memory management.

7.2.3.1 The Real Address Mode

The Intel's 80286 starts executing in its real address mode, soon after its boot up/or reset. The real address mode is so called because physical memory addresses are generated simply by adding an offset to a segment base, as in 8086. In real address mode 80286 can address upto 1 MB of physical memory. The 80286 directly executes 8086 machine code programs with a slight modification to them. However, 80286 executes most of the programs much faster, because of the extensive pipelining and other hardware enhancements and improvements. In real address mode 80286 executes all the 8086 instructions; the additional instructions of the Intel's 80186 like ENTER, LEAVE and BOUND; and few additional instructions are used to switch the 80286 from real address mode to PVAM.

There are several built-in interrupt types in 80286. Like in 8086, on the occurrence of an interrupt the 80286 multiplies the interrupt type number by 4 and goes to the resulting address in the interrupt vector table to get the code segment (CS) and instruction pointer (IP) values for the interrupt procedure. In this mode, the interrupt vector table is in the first 1 KB of memory.

7.2.3.2 The Protected Virtual Address Mode

The functioning of the 80286 in Protected Virtual Address Mode (PVAM) is much different from the Real Address Mode. In PVAM, the 80286 possesses memory management, protection, task switching, and interrupt processing. After boot/up/or reset, the 80286 starts operating in Real Address Mode. The real address mode is used to initialize peripheral devices, load the main part of the OS from disk into memory, load some registers, enable interrupt, and enter the PVAM. The 80286 enters into the PVAM by setting the protection enable bit of the Machine Status Word (MSW).

Figure 7.4 represents the format for the MSW. The MSW is a 16-bit register. The bit 0 is the protection enable (PE) bit. Bits 1, 2 and 3 indicate whether a processor extension (such as coprocessor) is present or not. The bits are changed in the machine status word (MSW) by loading the desired word in a register or memory location and executing the Load Machine Status Word (LMSW) instruction. Once the PVAM is entered by executing the LMSW instruction, resetting the system can only get it back to the real address mode. This processor is so designed that a programmer should not switch the system back into real address mode to defeat the protection scheme in PVAM.

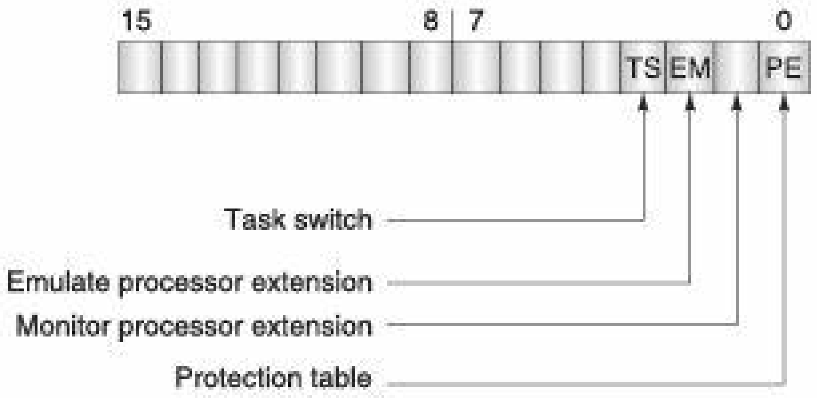


Fig. 7.4 Protected Virtual Address Mode PVAM.

The original 286 (more properly, 80286) was a vast power increase over the 8086/8088, particularly the 8/16-bit hybrid 8088. The 8 MHz ones were very rare, as by the time most people could afford to buy 286s they were buying 12 and 16 MHz ones.

7.3 INTRODUCTION TO 80386

The 80386 is an advanced 32-bit microprocessor designed for applications requiring high performance. This processor is a logical extension of 80286 and is optimized for multitasking operating system. The 32-bit registers and data paths support 32-bits and data types. The processor addresses upto 4 GB of physical memory and 64 terabytes of virtual memory. The integrated memory management and protection architecture includes address translation registers, advanced multitasking hardware and a protection mechanism to support the operating system. Additionally, the 80386 allows the simultaneous running of multiple operating systems. The instruction pipelining, on chip address translation, and high bus bandwidth ensure short average instruction execution time and high system throughout. The 80386 offers new testability and debugging features. Testability features include a self test and direct access to the page translation cache. The model of memory organization seen by application programmers is determined by systems-software designers. The architecture of the 80386 gives designers the freedom to choose a model for each task. The model of memory organization can range between the following extremes:

- A "flat" address space consisting of a single array of up to 4 GB.
- A segmented address space consisting of a collection of up to 16,381 segments of up to 4 GB each.

7.3.1 The "Flat" Model

In a "flat" model of memory organization, the application programmer sees a single array of up to 232 bytes (4 gigabytes). While the physical memory can contain up to 4 gigabytes, it is usually much smaller; the processor maps the 4 gigabytes flat space onto the physical address space by the address translation mechanisms. Applications programmers do not need to know the details of the mapping. A pointer into this flat address space is a 32-bit ordinal number that may range from 0 to 232-1. Relocation of separately-compiled modules in this space must be performed by systems software (e.g., linkers, locators, binders, loaders).

7.3.2 The Segmented Model

In a segmented model of memory organization, the address space as viewed by an applications program (called the logical address space) is a much larger space of up to 246 bytes (64 terabytes). The processor maps the 64 terabyte logical address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms. Application programmers do not need to know the details of this mapping. Applications programmers view the logical address space of the 80386 as a collection of up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space. Segment sizes may range from one byte up to a maximum of 232 bytes (4 gigabytes).

7.3.3 Registers

The 80386 contains a total of 7 registers that are of interest to the applications programmer. These registers may be grouped into these basic categories:

1. General-purpose register
2. Segment register
3. Instruction pointer and flags
4. Control registers
5. System address register

6. Debug register
7. Test register
   1. ***General registers:*** These eight 32-bit general-purpose registers are used primarily to contain operands for arithmetic and logical operations.
   2. ***Segment registers:*** These special-purpose registers permit system software designers to choose either a flat or segmented model of memory organization. These six registers determine, at any given time, which segments of memory are currently addressable.
   3. ***Status and instruction registers:*** These special-purpose registers are used to record and alter certain aspects of the 80386 processor state.

General registers

The 80386 has eight 32-bit general-purpose registers to hold data or addresses. The general registers of the 80386 are named EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used interchangeably to contain the operands of logical and arithmetic operations. They may also be used interchangeably for operands of address computations (except that ESP cannot be used as an index operand). The low-order word of each of these eight registers has a separate name and can be treated as a unit. This feature is useful for handling 16-bit data items and for compatibility with the 8086 and 80286 processors. The word registers are named AX, BX, CX, DX, BP, SP, SI, and DI. Each byte of the 16-bit registers AX, BX, CX, and DX has a separate name and can be treated as a unit. This feature is useful for handling characters and other 8-bit data items. The byte registers are named AH, BH, CH, and DH (high bytes), and AL, BL, CL, and DL (low bytes). All of the general-purpose registers are available for addressing calculations and for the results of most arithmetic and logical calculations; however, a few functions are dedicated to certain registers. By implicitly choosing registers for these functions, the 80386 architecture can encode instructions more compactly. The instructions that use specific registers include: double-precision multiply and divide, I/O, string instructions, translate , loop, variable shift and rotate, and stack operations.

Segment registers

The segment registers of the 80386 give system software designers the flexibility to choose among various models of memory organization. Complete programs generally consist of many different modules, each consisting of instructions and data. The 80386 architecture takes advantage of this by providing mechanisms to support direct access to the instructions and data of the current module's environment, with access to additional segments on demand. At any given instant, six segments of memory may be immediately accessible to an executing 80386 program. The segment registers CS, DS, SS, ES, FS, and GS are used to identify these six current segments. Each of these registers specifies a particular kind of segment, as characterized by the associated mnemonics ("code," "data," or "stack"). Each register uniquely determines one particular segment, from among the segments that make up the program that is to be immediately accessible at highest speed.

The segment containing the currently executing sequence of instructions is known as the current code segment; it is specified by means of the CS register. The 80386 fetches all

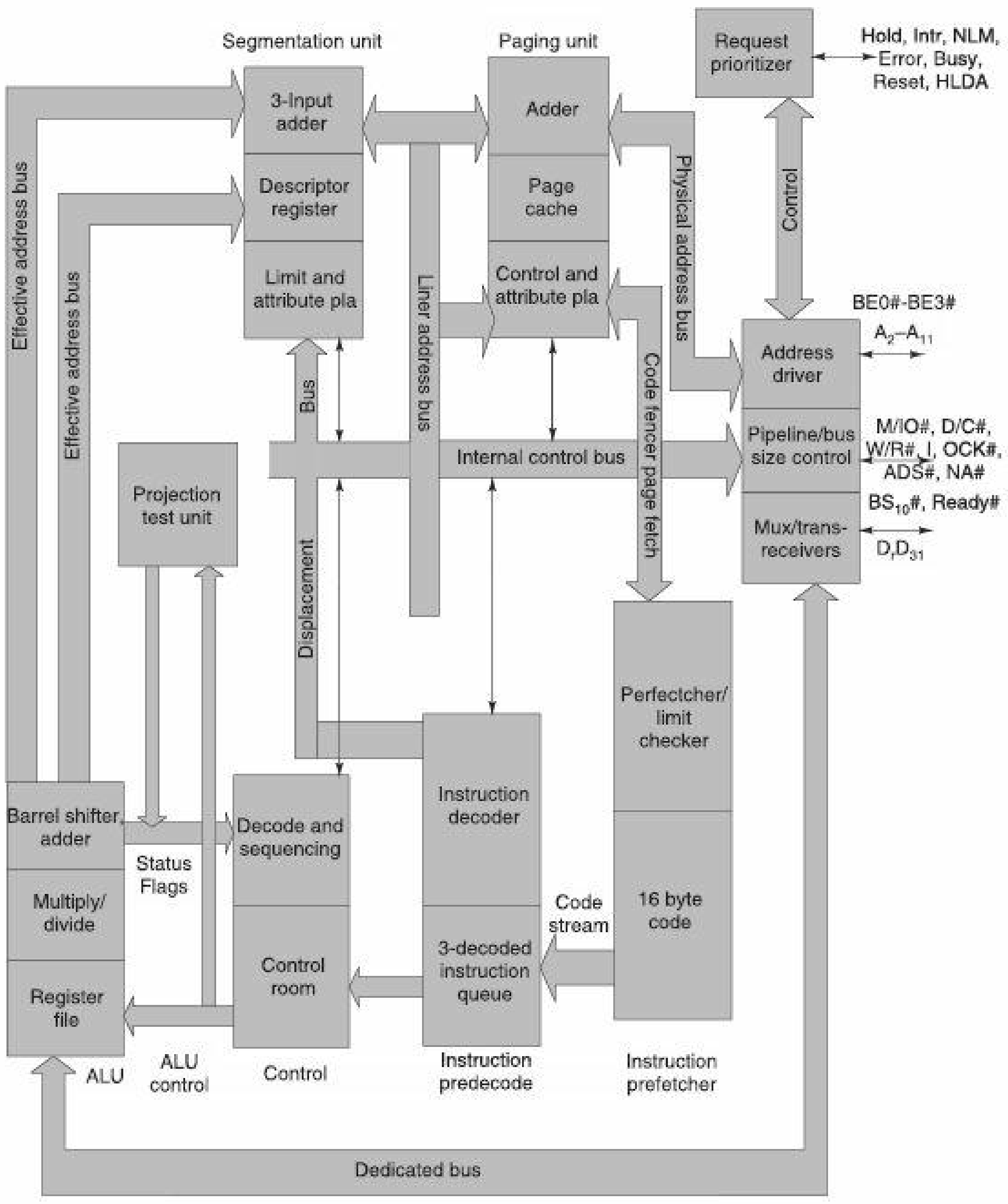

Segmentation unit
Paging unit
Request prioritizer
Hold, Intr, NLM, Error, Busy, Reset, HLDA
3-Input adder
Descriptor register
Limit and attribute pla
Adder
Page cache
Control and attribute pla
Effective address bus
Effective address bus
Liner address bus
Physical address bus
Control
BE0#-BE3#
A_2–A_{11}
Address driver
Pipeline/bus size control
Mux/trans-receivers
M/IO#, D/C#, W/R#, I, OCK#, ADS#, NA#
BS_{10}#, Ready#
$D_,D_{31}$
Bus
Internal control bus
Code fencer page fetch
Projection test unit
Displacement
Perfectcher/ limit checker
Barrel shifter, adder
Multiply/ divide
Register file
Status Flags
Decode and sequencing
Control room
Instruction decoder
3-decoded instruction queue
Code stream
16 byte code
ALU
ALU control
Control
Instruction predecode
Instruction prefetcher
Dedicated bus

Fig. 7.5

instructions from this code segment, using as an offset the contents of the instruction pointer. CS is changed implicitly as the result of intersegment control-transfer instructions (for example, CALL and JMP), interrupts, and exceptions. All stack operations use the SS register to locate the stack. Unlike CS, the SS register can be loaded explicitly, thereby permitting programmers to define stacks dynamically.

The DS, ES, FS, and GS registers allow the specification of four data segments, each addressable by the currently executing program. The processor associates a base address with each segment selected by a segment register.

To address an element within a segment, a 32-bit offset is added to the segment's base address. Once a segment is selected (by loading the segment selector into a segment register), a data manipulation instruction only needs to specify the offset. Simple rules define which segment register is used to form an address when only an offset is specified.

Stack Implementation

Stack operations are facilitated by three registers:

1. *The Stack Segment (SS) register:* Stacks are implemented in memory. A system may have a number of stacks that is limited only by the maximum number of segments. A stack may be up to 4 gigabytes long, the maximum length of a segment. One stack less is directly addressable at a time—the one located by SS. This is the current stack, often referred to as "the" stack. SS is used automatically by the processor for all stack operations.
2. *The Stack Pointer (ESP) register:* ESP points to the Top of the push-down Stack (TOS). It is referenced implicitly by PUSH and POP operations, subroutine calls and returns, and interrupt operations. When an item is pushed onto the stack, the processor decrements ESP, then writes the item at the new TOS. When an item is popped off the stack, the processor copies it from TOS, then increments ESP. In other words, the stack grows down in memory towards lesser addresses.
3. *The Stack-Frame Base Pointer (EBP) register:* The EBP is the best choice of register for accessing data structures, variables and dynamically allocated work space within the stack. EBP is often used to access elements on the stack relative to a fixed point on the stack rather than relative to the current TOS. It typically identifies the base address of the current stack frame established for the current procedure. When EBP is used as the base register in an offset calculation, the offset is calculated automatically in the current stack segment (i.e., the segment currently selected by SS). Because SS does not have to be explicitly specified, instruction encoding in such cases is more efficient. EBP can also be used to index into segments addressable via other segment registers.

Flags register

The flags register is a 32-bit register named EFLAGS. The flags control certain operations and indicate the status of the 80386. The low-order 16 bits of EFLAGS is named FLAGS and can be treated as one unit. This feature is useful when executing 8086 and 80286 code, because this part of FLAGS-EFLAGS is identical to the FLAGS register of the 8086 and the 80286. The flags may be considered in three groups: the status flags, the control flags, and the systems flags as shown in Figure 7.6.

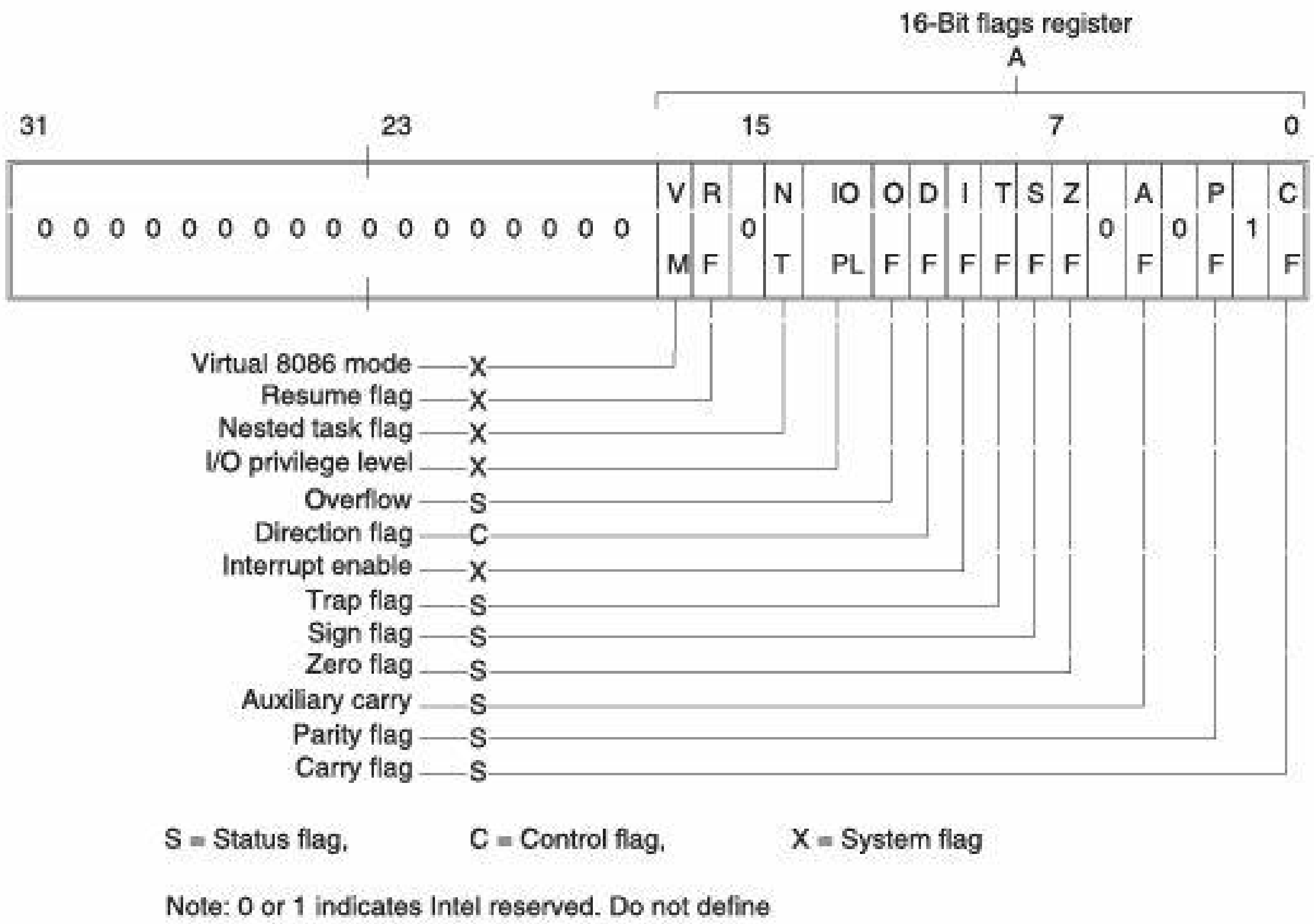


Fig. 7.6 Flag Status of 80386.

Status flags

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before the execution of an arithmetic instruction.

Control flag

The control flag DF of the EFLAGS register controls string instructions. DF (Direction Flag, bit 10) setting causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses. Clearing DF causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

Instruction pointer

The instruction pointer register (EIP) contains the offset address, relative to the start of the current code segment, of the next sequential instruction to be executed. The instruction pointer is not directly visible to the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions.

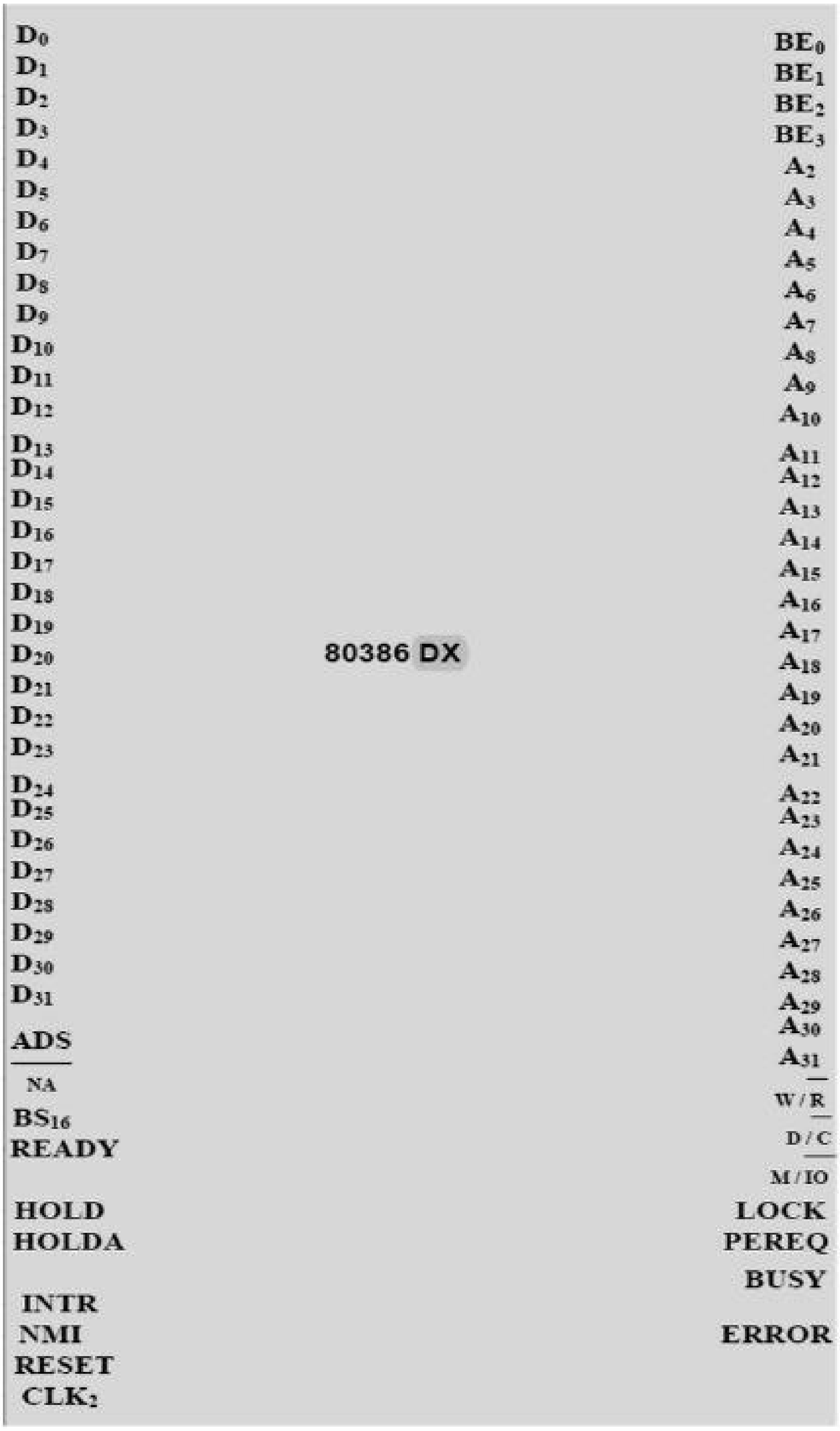


Fig. 7.7 Pin Diagram of 80386

The desription of additional pins which were not present in the 8086 are given below

- **BE0 toBE3:** The 32-bit data bus supported by 80386 and the memory system of 80386 can be viewed as a 4-byte wide memory access mechanism. The 4 byte enable lines BE0 to BE3 , may be used for enabling these 4 blanks. Using these 4 enable signal lines, the CPU may transfer 1 byte /2/3/4 byte of data simultaneously.
- **ADS:** The address status output pin indicates that the address bus and bus cycle definition pins are carrying the respective valid signals. The 80383 does not have any ALE signals and so this signals may be used for latching the address to external latches.
- **READY:** The ready signals indicates to the CPU that the previous bus cycle has been terminated and the bus is ready for the next cycle. The signal is used to insert WAIT states in a bus cycle and is useful for interfacing of slow devices with CPU.
- **BS16:** The bus size 16 input pin allows the interfacing of 16 bit devices with the 32 bit wide 80386 data bus. Successive 16 bit bus cycles may be executed to read a 32 bit data from a peripheral.
- **BUSY:** The busy input signal indicates to the CPU that the coprocessor is busy with the allocated task.
- **ERROR:** The error input pin indicates to the CPU that the coprocessor has encountered an error while executing its instruction.

  PEREQ: The processor extension request output signal indicates to the CPU to fetch a data word for the coprocessor.
- **INTR:** This interrupt pin is a maskable interrupt, that can be masked using the IF of the flag register.

  NMI: A valid request signal at the non-maskable interrupt request input pin internally generates a non- maskable interrupt of type2.

  N/C: No connection pins are expected to be left open while connecting the 80386 in the circuit.

7.3.3.1 Protected Virtual Addressing Mode (PVAM)

The 80386 operates in two memory management modes in PVAM. They are given as follows:

1. Non Paged mode: MMU operates similar to 80286. Virtual addresses are represented with a selector component and an offset component. The selector component is used to index a descriptor in a descriptor table. The descriptor contains the 32-bit physical base address for the segment. The offset part of the virtual address is added to the base address to produce the actual physical address. The offset part of a virtual address can be 16 or 32 bits so segment can be as large as 4 gigabytes.

Hence, the virtual memory size is

$2^{13} * 2 * 2^{32} = 2^{46}$ bytes or 64 Terabytes.

Advantage of segmentation of memory: Segments correspond to code and data structures in the program. Hence, segmentation is useful.

Limitation of segmentation of memory: If we need only a part of memory, even then we have to swap the whole segment content. This will increase the time for execution.

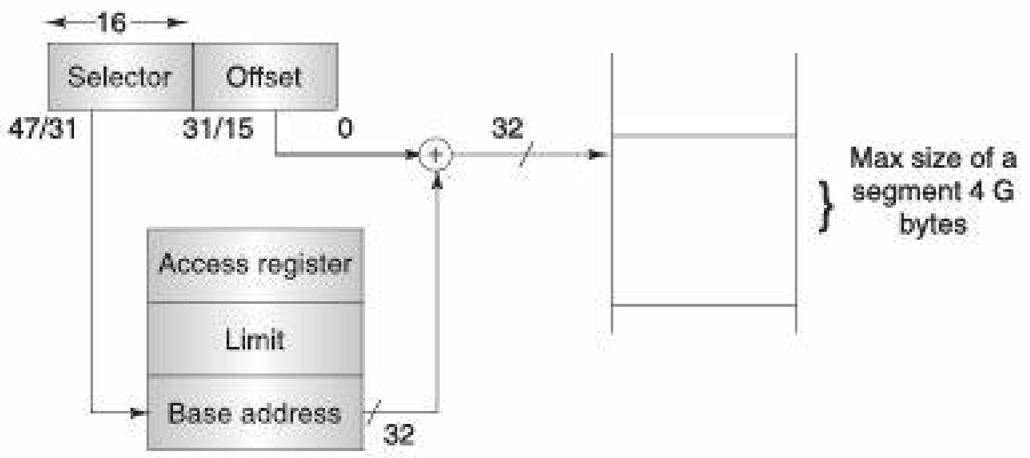


Fig. 7.8(a) Non Paged Mode.

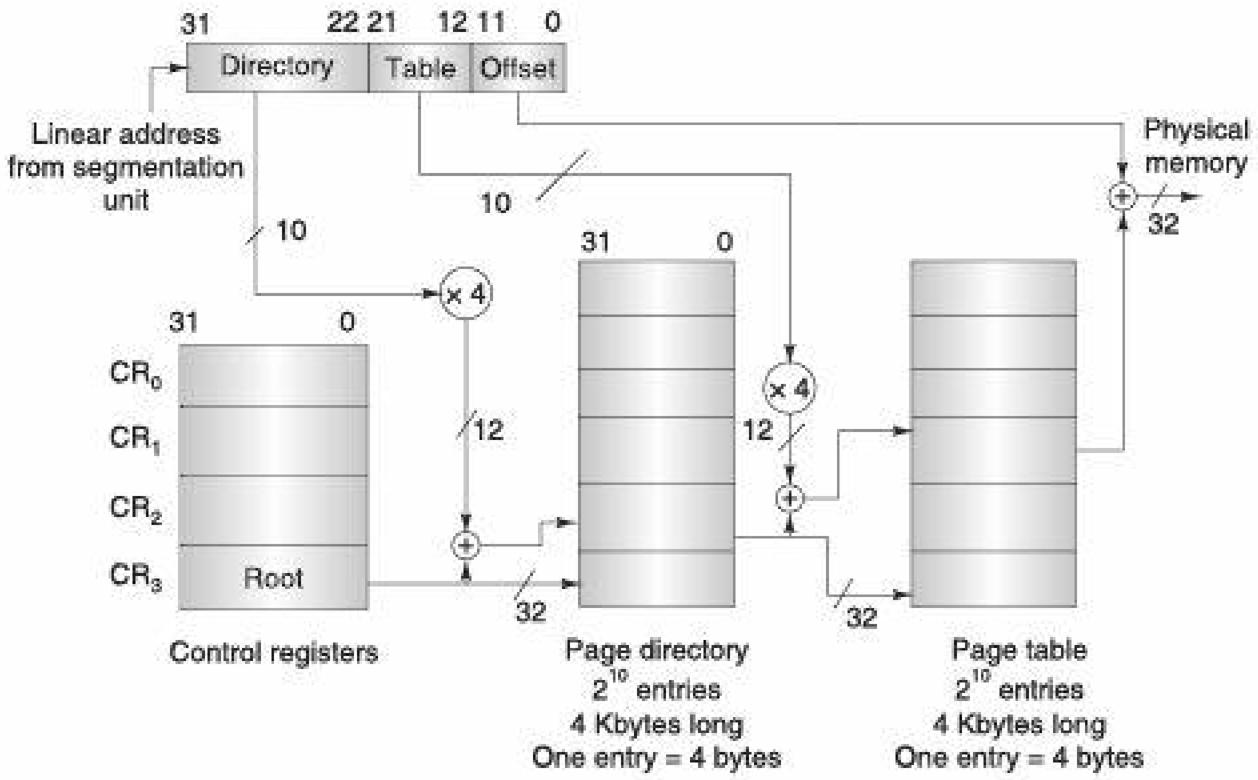


Fig. 7.8(b) Address Translation Mechanism in 80386 Paging Unit.

Paged mode: In this mode, instead of segments, 4 Kbytes of fixed page length are used.

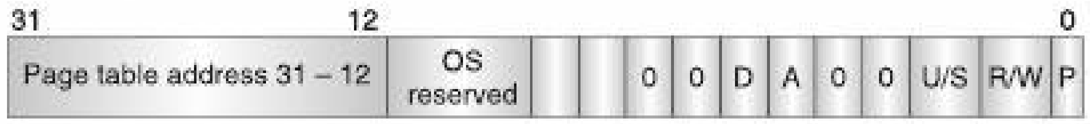
Limitation: Pages do not correspond to the logical structure of the program.

Advantage: Pages can be quickly swapped.

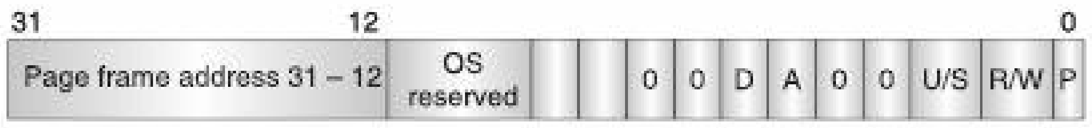
Conversion of linear address into physical address:

Page directory

Page directory entry



Page table entry



P = entry can be used in address translation

P = 1 Yes

P = 0 No

A = accessed

A = 1 page is accessed

A = 0 page is unaccessed

D = dirty bit

Dirty bit is set before any write operation to the page.

Dirty bit is undefined for page directory entries.

U/S and R/W bits are used to provide protection.

U/S	R/W	Permitted for level 3	Permitted for levels 2, 1, 0
0	0	None	Read-write
0	1	None	Read-write
1	0	Read only	Read-write
1	1	Read-write	Read-write

7.3.4 Instruction Format

The information encoded in an 80386 instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction must also select, explicitly or implicitly, which of the currently addressable segments contains the operand. 80386 instructions are composed of various elements and have various formats. Among these instruction elements, only one, the opcode, is always present. The other elements may or may not be present, depending on the particular operation involved and on the location and type of the operands. The elements of an instruction, in order of occurrence are as follows:

1. **Prefixes:** one or more bytes preceding an instruction that modify the operation of the instruction. The following types of prefixes can be used by application programs:
   (a) Segment override—explicitly specifies which segment register an instruction should use, thereby overriding the default segment-register selection used by the 80386 for that instruction.
   (b) Address size—switches between 32-bit and 16-bit address generation.
   (c) Operand size—switches between 32-bit and 16-bit operands.
   (d) Repeat—used with a string instruction to cause the instruction to act on each element of the string.

2. Opcode—specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different variant of the operation.
3. Register specifier: an instruction may specify one or two register operands. Register specifiers may occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.
4. Addressing-mode specifier—when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.
5. SIB (scale, index, base) byte—when the addressing-mode specifier indicates that an index register will be used to compute the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.
6. Displacement—when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16 or 8 bits. The 8-bit form is used in the common case when the displacement is sufficiently small. The processor extends an 8-bit displacement to 16 or 32 bits, taking into account the sign.
7. Immediate operand—when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide. In cases where an 8-bit immediate operand is combined in some way with a 16- or 32-bit operand, the processor automatically extends the size of the 8-bit operand, taking into account the sign.

7.3.5 Operand Selection

An instruction can act on zero or more operands, which are data manipulated by the instruction. An example of a zero-operand instruction is NOP (no operation). An operand can be in any of these locations:

- In the instruction itself (an immediate operand)
- In a register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP in the case of 32-bit operands; AX, BX, CX, DX, SI, DI, SP, or BP in the case of 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL in the case of 8-bit operands; the segment registers; or the EFLAGS register for flag operations)
- In memory
- At an I/O port

Immediate operands and operands in registers can be accessed more rapidly than operands in memory since memory operands must be fetched from memory. Register operands are available in the CPU. Immediate operands are also available in the CPU, because they are pre-fetched as part of the instruction.

For most instructions, one of the two explicitly specified operands—either the source or the destination—can be either in a register or in memory. The other operand must be in a register or be an immediate source operand. Thus, the explicit two-operand instructions of the 80386 permit operations of the following kinds:

- Register-to-register
- Register-to-memory
- Memory-to-register
- Immediate-to-register
- Immediate-to-memory

Certain string instructions and stack manipulation instructions, however, transfer data from memory to memory. Both operands of some string instructions are in memory and are implicitly specified. Push and pop stack operations allow transfer between memory operands and the memory-based stack.

7.3.5.1 Immediate Operands

Certain instructions use data from the instruction itself as one (and sometimes two) of the operands. Such an operand is called an immediate operand. The operand may be 32, 16, or 8-bit long. For example: SHR PATTERN, 2 One byte of the instruction holds the value 2, the number of bits by which to shift the variable PATTERN. TEST PATTERN, 0FFFF00FFH

A double word of the instruction holds the mask that is used to test the variable PATTERN.

7.3.5.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, or DL). The 80386 has instructions for referencing the segment registers (CS, DS, ES, SS, FS, GS). These instructions are used by applications programs only if systems designers have chosen a segmented memory model.

The 80386 also has instructions for referring to the flag register. The flags may be stored on the stack and restored from the stack. Certain instructions change the commonly modified flags directly in the EFLAGS register.

7.3.5.3 Memory Operands

Data-manipulation instructions that address operands in memory must specify (either directly or indirectly) the segment that contains the operand and the offset of the operand within the segment. However, for speed and compact instruction encoding, segment selectors are stored in the high-speed segment registers. Therefore, data-manipulation instructions need to specify only the desired segment register and an offset in order to address a memory operand.

An 80386 data-manipulation instruction that accesses memory uses one of the following methods for specifying the offset of a memory operand within its segment:

1. Most data-manipulation instructions that access memory contain a byte that explicitly specifies the addressing method for the operand. A byte, known as the modR/M byte, follows the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

When an index register is used, the modR/M byte is also followed by another byte that identifies the index register and scaling factor. This addressing method is the most flexible.

2. A few data-manipulation instructions implicitly use specialized addressing methods:
   (a) For a few short forms of MOV that implicitly use the EAX register, the offset of the operand is coded as a doubleword in the instruction. No base register, index register, or scaling factor are used.
   (b) String operations implicitly address memory via DS:ESI, (MOVS, CMPS, OUTS, LODS, SCAS) or via ES:EDI (MOVS, CMPS, INS, STOS).
   (c) Stack operations implicitly address operands via SS:ESP registers; e.g., PUSH, POP, PUSHA, PUSHAD, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, RET, IRET, IRETD, exceptions, and interrupts.

7.3.6 Interrupts and Exceptions

The 80386 has two mechanisms for interrupting program execution:

1. Exceptions are synchronous events that are the responses of the CPU to certain conditions detected during the execution of an instruction.
2. Interrupts are asynchronous events typically triggered by external devices needing attention. Interrupts and exceptions are alike in that both cause the processor to temporarily suspend its present program execution in order to execute a program of higher priority. The major distinction between these two kinds of interrupts is their origin. An exception is always reproducible by re-executing with the program and data that caused the exception, whereas an interrupt is generally independent of the currently executing program.
   (a) A divide error exception results when the instruction DIV or IDIV is executed with a zero denominator or when the quotient is too large for the destination operand.
   (b) The debug exception may be reflected back to an applications program if it results from the trap flag (TF).
   (c) A breakpoint exception results when the instruction INT 3 is executed. This instruction is used by some debuggers to stop program execution at specific points.
   (d) An overflow exception results when the INTO instruction is executed and the OF (overflow) flag is set (after an arithmetic operation that set the OF flag).
   (e) A bounds check exception results when the BOUND instruction is executed and the array index it checks falls outside the bounds of the array.
   (f) Invalid opcodes may be used by some applications to extend the instruction set. In such a case, the invalid opcode exception presents an opportunity to emulate the opcode.
   (g) The "coprocessor not available" exception occurs if the program contains instructions for a coprocessor, but no coprocessor is present in the system.
   (h) A coprocessor error is generated when the coprocessor detects an illegal operation.

7.3.7 Instruction Set

7.3.7.1 Data Movement Instructions

These instructions provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture. They fall into the following classes:

1. General-purpose data movement instructions.
2. Stack manipulation instructions.
3. Type conversion instructions.

7.3.7.2 General-Purpose Data Movement Instructions

MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand. The MOV instruction is useful for transferring data along any of these paths. There are also variants of MOV that operate on segment registers. The MOV instruction cannot move from memory to memory or from segment register to segment register are not allowed. Memory-to-memory moves can be performed, however, by the string move instruction MOVS. XCHG (Exchange) swaps the contents of two operands. This instruction takes the place of three MOV instructions. It does not require a temporary location to save the contents of one operand while the other is being loaded. XCHG is especially useful for implementing semaphores or similar data structures for process synchronization. The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands. The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand. When used with a memory operand, XCHG automatically activates the LOCK signal.

7.3.7.3 Stack Manipulation Instructions

PUSH (Push) decrements the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP. PUSH is often used to place parameters on the stack before calling a procedure; it is also the basic means of storing temporary variables on the stack. The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers).

PUSHA (Push All Registers) saves the contents of the eight general registers on the stack. This instruction simplifies procedure calls by reducing the number of instructions required to retain the contents of the general registers for use in a procedure. The processor pushes the general registers on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI. PUSHA is complemented by the POPA instruction.

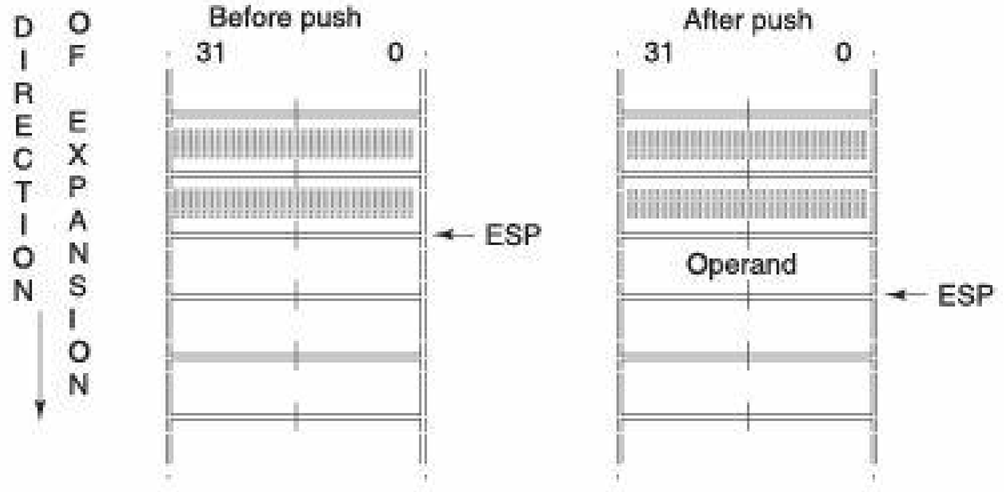


Fig. 7.9 PUSH Instruction.

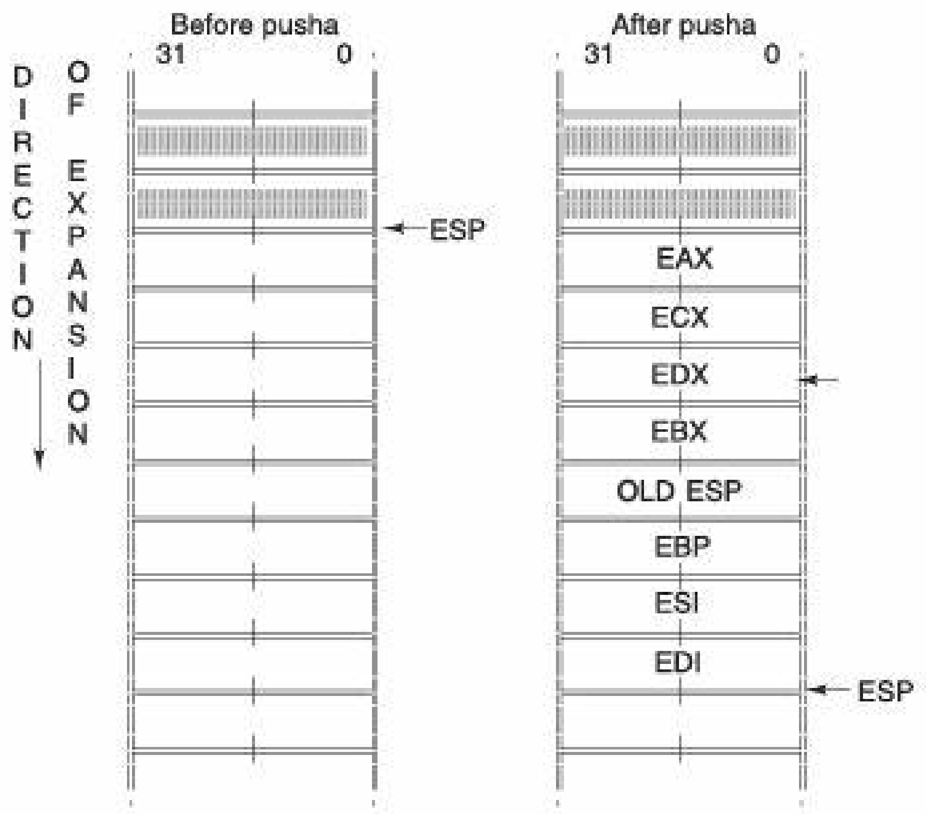


Fig. 7.10 PUSH A Instruction.

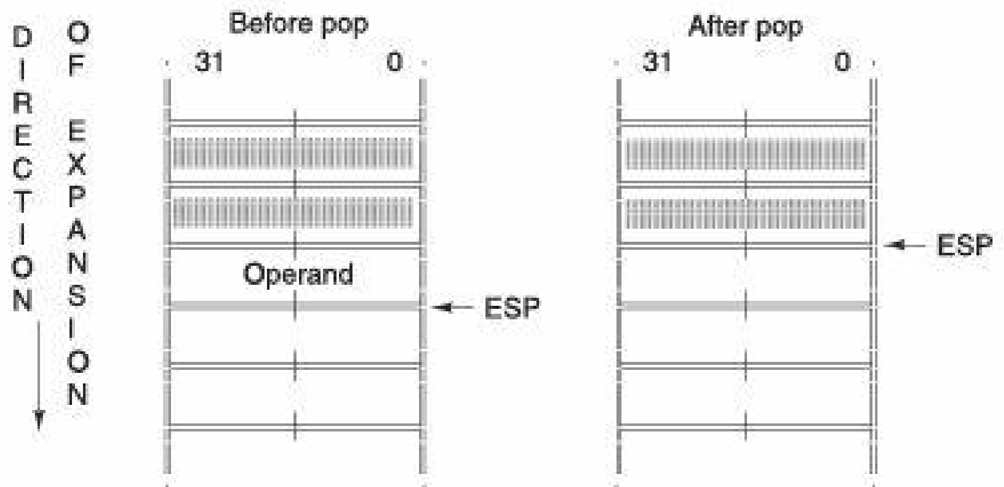


Fig. 7.11 POP Instruction.

POP (Pop) transfers the word or doubleword at the current top of stack (indicated by ESP) to the destination operand, and then increments ESP to point to the new top of stack. POP moves information from the stack to a general register, or to memory. There is also a variant of POP that operates on segment registers. POPA (Pop All Registers) restores the registers saved on the stack by PUSHA, except that it ignores the saved value of ESP.

7.3.7.4 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into 64-bit items (quad-words). These instructions are especially useful for converting signed integers, because they automatically fill the extra bits of the larger item with the value of the sign bit of the smaller item. This kind of conversion is called sign extension. There are two classes of type conversion instructions:

1. The forms CWD, CDQ, CBW, and CWDE which operate only on data in the EAX register.
2. The forms MOVSX and MOVZX, which permit one operand to be in any general register while permitting the other operand to be in memory or in a register.

CWD (Convert Word to Doubleword) and CDQ (Convert Doubleword to Quad-Word) double the size of the source operand. CWD extends the sign of the word in register AX throughout register DX. CDQ extends the sign of the doubleword in EAX throughout EDX. CWD can be used to produce a doubleword dividend from a word before a word division, and CDQ can be used to produce a quad-word dividend from a doubleword before doubleword division.

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout AX.

CWDE (Convert Word to Doubleword Extended) extends the sign of the word in register AX throughout EAX.

MOVSX (Move with Sign Extension) sign-extends an 8-bit value to a 16-bit value and a 8 or 16-bit value to 32-bit value.

MOVZX (Move with Zero Extension) extends an 8-bit value to a 16-bit value and an 8 or 16-bit value to 32-bit value by inserting high-order zeros.

7.3.7.5 Binary Arithmetic Instructions

The arithmetic instructions of the 80386 processor simplify the manipulation of numeric data that is encoded in binary. Operations include the standard add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign. Both signed and unsigned binary integers are supported. The binary arithmetic instructions may also be used as one step in the process of performing arithmetic operations on decimal integers.

Many of the arithmetic instructions operate on both signed and unsigned integers. These instructions update the flags ZF, CF, SF, and OF in such a manner that subsequent instructions can interpret the results of the arithmetic operation as either signed or unsigned. CF contains information relevant to unsigned integers; SF and OF contain information relevant to signed integers. ZF is relevant to both signed and unsigned integers; ZF is set when all bits of the result are zero. If the integer is unsigned, CF may be tested after one of these arithmetic operations to determine whether the operation required a carry or borrow of a one-bit in the high-order position of the destination operand. CF is set if a one-bit was carried out of the high-order position (addition instructions ADD, ADC, AAA, and DAA) or if a one-bit was carried (i.e., borrowed) into the high-order bit (subtraction instructions SUB, SBB, AAS, DAS, CMP, and NEG). If the integer is signed, both SF and OF should be tested. SF always has the same value as the sign bit of the result. The most significant bit (MSB) of a signed integer is the bit next to the sign bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword. OF is set in either of these cases:

1. A one bit was carried out of the MSB into the sign bit but no one bit was carried out of the sign bit (addition instructions ADD, ADC, INC, AAA, and DAA). In other words,

the result was greater than the greatest positive number that could be contained in the destination operand.

2. A one bit was carried from the sign bit into the MSB but no one bit was carried into the sign bit (subtraction instructions SUB, SBB, DEC, AAS, DAS, CMP, and NEG). In other words, the result was smaller than the smallest negative number that could be contained in the destination operand.

These status flags are tested by executing one of the two families of conditional instructions: JCC (jump on condition cc) or SET CC (byte set on condition).

7.3.7.6 Addition and Subtraction Instructions

ADD (Add Integers) replaces the destination operand with the sum of the source and destination. ADC (Add Integers with Carry) sums the operands, adds one if CF is set, and replaces the destination operand with the result. If CF is cleared, ADC performs the same operation as the ADD instruction. An ADD followed by multiple ADC instructions can be used to add numbers longer than 32 bits.

INC (Increment) adds one to the destination operand. INC does not affect CF. Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed. SUB (Subtract Integers) subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. The operands may be signed or unsigned bytes, words, or doublewords integers are supported. The binary arithmetic instructions may also be used as one step in the process of performing arithmetic operations on decimal integers.

Many of the arithmetic instructions operate on both signed and unsigned integers. These instructions update the flags ZF, CF, SF, and OF in such a manner that subsequent instructions can interpret the results of the arithmetic operation as either signed or unsigned. CF contains information relevant to unsigned integers; SF and OF contain information relevant to signed integers. ZF is relevant to both signed and unsigned integers; ZF is set when all bits of the result are zero. If the integer is unsigned, CF may be tested after one of these arithmetic operations to determine whether the operation required a carry or borrow of a one-bit in the high-order position of the destination operand. CF is set if a one-bit was carried out of the high-order position (addition instructions ADD, ADC, AAA, and DAA) or if a one-bit was carried (i.e., borrowed) into the high-order bit (subtraction instructions SUB, SBB, AAS, DAS, CMP, and NEG). If the integer is signed, both SF and OF should be tested. SF always has the same value as the sign bit of the result. The most significant bit (MSB) of a signed integer is the bit next to the sign bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword. OF is set in either of these cases:

1. A one bit was carried out of the MSB into the sign bit but no one bit was carried out of the sign bit (addition instructions ADD, ADC, INC, AAA, and DAA). In other words, the result was greater than the greatest positive number that could be contained in the destination operand.
2. A one bit was carried from the sign bit into the MSB but no one bit was carried into the sign bit (subtraction instructions SUB, SBB, DEC, AAS, DAS, CMP, and NEG). In other

words, the result was smaller than the smallest negative number that could be contained in the destination operand.

These status flags are tested by executing one of the two families of conditional instructions: JCC (jump on condition cc) or SETCC (byte set on condition).

7.3.7.7 Comparison and Sign Change Instruction

CMP (Compare) subtracts the source operand from the destination operand. It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands. A subsequent Jcc or SETcc instruction can test the appropriate flags.

NEG (Negate) subtracts a signed integer operand from zero. The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.

7.3.7.8 Multiplication Instructions

The 80386 has separate multiply instructions for unsigned and signed operands. MUL operates on unsigned numbers, while IMUL operates on signed integers as well as unsigned. MUL (Unsigned Integer Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, the processor multiplies it by the contents of AL and returns the double-length result to AH and AL. If the source operand is a word, the processor multiplies it by the contents of AX and returns the double-length result to DX and AX. If the source operand is a doubleword, the processor multiplies it by the contents of EAX and returns the 64-bit result in EDX and EAX. MUL sets CF and OF when the upper half of the result is nonzero; otherwise, they are cleared. IMUL (Signed Integer Multiply) performs a signed multiplication operation. IMUL has three variations:

1. **A one-operand form:** The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.
2. **A two-operand form:** One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.
3. **A three-operand form:** Two are source and one is the destination operand. One of the source operands is an immediate value stored in the instruction; the second may be in memory or in any general register. The product may be stored in any general register. The immediate operand is treated as signed. If the immediate operand is a byte, the processor automatically sign extends it to the size of the second operand before performing the multiplication. The three forms are similar in most respects:
   (a) The length of the product is calculated to twice the length of the operands.
   (b) The CF and OF flags are set when significant bits are carried into the high-order half of the result. CF and OF are cleared when the high-order half of the result is the sign-extension of the low-order half.

7.3.7.9 Division Instructions

The 80386 has separate division instructions for unsigned and signed operands. DIV operates on unsigned numbers, while IDIV operates on signed integers as well as unsigned. In either case, an exception (interrupt zero) occurs if the divisor is zero or if the quotient is too large for AL, AX, or EAX.

DIV (Unsigned Integer Divide) performs an unsigned division of the accumulator by the source operand. The dividend (the accumulator) is twice the size of the divisor (the source operand); the quotient and remainder have the same size as the divisor, as the following table shows. Size of Source Operand (divisor) Dividend Quotient Remainder.

Byte AX	AL	AH	
Word DX:	AX	AX	EDX
Doubleword EDX:	EAX	EAX	EDX

Non-integral quotients are truncated to integers towards 0. The remainder is always less than the divisor. For unsigned byte division, the largest quotient is 255. For unsigned word division, the largest quotient is 65,535.

IDIV (Signed Integer Divide) performs a signed division of the accumulator by the source operand. IDIV uses the same registers as the DIV instruction. For signed byte division, the maximum positive quotient is +127, and the minimum negative quotient is −128. For signed word division, the maximum positive quotient is +32, 767 and the minimum negative quotient is −32,768. For signed doubleword division the maximum positive quotient is 231 − 1, the minimum negative quotient is −231. Non-integral results are truncated towards 0. The remainder always has the same sign as the dividend and is less than the divisor in magnitude.

7.3.7.10 Decimal Arithmetic Instructions

Decimal arithmetic is performed by combining the binary arithmetic instructions (already discussed in the prior section) with the decimal arithmetic instructions. The decimal arithmetic instructions are used in one of the following ways:

- To adjust the results of a previous binary arithmetic operation to produce a valid packed or unpacked decimal result.
- To adjust the inputs to a subsequent binary arithmetic operation so that the operation will produce a valid packed or unpacked decimal result. These instructions operate only on the AL or AH register. Most utilize the AF flag.

7.3.7.11 Packed BCD Adjustment Instructions

DAA (Decimal Adjust after Addition) adjusts the result of adding two valid packed decimal operands in AL. DAA must always follow the addition of two pairs of packed decimal numbers (one digit in each half-byte) to obtain a pair of valid packed decimal digits as results. The carry flag is set if carry was needed.

DAS (Decimal Adjust after Subtraction) adjusts the result of subtracting two valid packed decimal operands in AL. DAS must always follow the subtraction of one pair of packed decimal numbers (one digit in each half-byte) from another to obtain a pair of valid packed decimal digits as results. The carry flag is set if a borrow was needed.

7.3.7.12 Unpacked BCD Adjustment Instructions

AAA (ASCII Adjust after Addition) changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAA must always follow the addition of two unpacked decimal operands in AL. The carry flag is set and AH is incremented if a carry is necessary.

AAS changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAS must always follow the subtraction of one unpacked decimal operand from another in AL. The carry flag is set and AH decremented if a borrow is necessary. AAM (ASCII Adjust after Multiplication) corrects the result of a multiplication of two valid unpacked decimal numbers. AAM must always follow the multiplication of two decimal numbers to produce a valid decimal result. The high-order digit is left in AH, the low-order digit in AL. AAD (ASCII Adjust before Division) modifies the numerator in AH and AL to prepare for the division of two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH should contain the high-order digit and AL the low-order digit. This instruction adjusts the value and places the result in AL. AH will contain zero.

7.3.7.13 Logical Instructions

The group of logical instructions includes:

- The Boolean operation instructions
- Bit test and modify instructions
- Bit scan instructions
- Rotate and shift instructions
- Byte set on condition

7.3.7.14 Boolean Operation Instructions

The logical operations are AND, OR, XOR, and NOT. NOT (Not) inverts the bits in the specified operand to form a one's complement of the operand. The NOT instruction is a unary operation that uses a single operand in a register or memory. NOT has no effect on the flags. The AND, OR, and XOR instructions perform the standard logical operations "and", "(inclusive) or", and "exclusive or". These instructions can use the following combinations of operands:

- Two register operands
- A general register operand with a memory operand
- An immediate operand with either a general register operand or a memory operand. AND, OR, and XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.

7.3.7.15 Bit Scan Instructions

These instructions scan a word or doubleword for a one bit and store the index of the first set bit into a register. The bit string being scanned may be either in a register or in memory. The ZF flag is set if the entire word is zero (no set bits are found); ZF is cleared if one bit is found. If no set bit is found, the value of the destination register is undefined. BSF (Bit Scan Forward) scans from low-order to high-order (starting from bit index zero). BSR (Bit Scan Reverse) scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

7.3.7.16 Shift and Rotate Instructions

The shift and rotate instructions reposition the bits within the specified operand. These instructions fall into the following classes:

- Shift instructions
- Double shift instructions
- Rotate instructions

7.3.7.17 Test Instruction

TEST (Test) performs the logical "and" of the two operands, clears OF and CF, leaves AF undefined, and updates SF, ZF, and PF. The flags can be tested by conditional control transfer instructions or by the byte-set-on-condition instructions. The operands may be doublewords, words, or bytes. The difference between TEST and AND is that TEST does not alter the destination operand. TEST differs from BT in that TEST is useful for testing the value of multiple bits in one operation, whereas BT tests a single bit.

7.3.7.18 Control Transfer Instructions

The 80386 provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

7.3.7.19 Unconditional Transfer Instructions

JMP, CALL, RET, INT and IRET instructions transfer control from one code segment location to another. These locations can be within the same code segment (near control transfers) or in different code segments (far control transfers). The variants of these instructions that transfer control to other segments are discussed in a later section of this chapter. If the model of memory organization used in a particular 80386 application does not make segments visible to applications programmers, intersegment control transfers will not be used.

7.3.7.20 Flag Control Instructions

The flag control instructions provide a method for directly changing the state of bits in the flag register.

7.3.7.21 Carry and Direction Flag Control Instructions

The carry flag instructions are useful in conjunction with rotate-with-carry instructions RCL and RCR. They can initialize the carry flag, CF, to a known state before execution of a rotate that moves the carry bit into one end of the rotated operand. The direction flag control instructions are specifically included to set or clear the direction flag, DF, which controls the left-to-right or right-to-left direction of string processing. If DF = 0, the processor automatically increments the string index registers, ESI and EDI, after each execution of a string primitive. If DF = 1, the processor decrements these index registers. Programmers should use one of these instructions before any procedure that uses string instructions to ensure that DF is set properly.

Flag Control Instruction	Effect
STC (Set Carry Flag)	CF ← 1
CLC (Clear Carry Flag)	CF ← 0
CMC (Complement Carry Flag)	CF ← NOT (CF)
CLD (Clear Direction Flag)	DF ← 0
STD (Set Direction Flag)	DF ← 1

7.3.7.22 Segment-Register Transfer Instructions

The MOV, POP, and PUSH instructions also serve to load and store segment registers. These variants operate similarly to their general-register counterparts except that one operand can be a segment register. MOV cannot move segment register to a segment register. Neither POP nor MOV can place a value in the code-segment register CS; only the far control-transfer instructions can change CS.

7.3.7.23 No-Operation Instruction

NOP (No Operation) occupies a byte of storage but affects nothing but the instruction pointer, EIP.

7.3.7.24 Translate Instruction

XLAT (Translate) replaced a byte in the AL register with a byte from a user-coded translation table. When XLAT is executed, AL should have the unsigned index to the table addressed by EBX. XLAT changes the contents of AL from table index to table entry. EBX is unchanged. The XLAT instruction is useful for translating from one coding system to another such as from ASCII to EBCDIC. The translate table may be up to 256 bytes long. The value placed in the AL register serves as an index to the location of the corresponding translation value.

7.3.8 Systems Architecture

The systems-level features of the 80386 architecture include:

Memory Management
Protection
Multitasking
Input/Output
Exceptions and Interrupts
Initialization
Coprocessing and Multiprocessing
Debugging

These features are implemented by registers and instructions.

7.3.8.1 Systems Registers

The registers designed for use by systems programmers fall into these classes:

EFLAGS
Memory-Management Registers

Control Registers
Debug Registers
Test Registers

7.3.8.2 Systems Flags

The systems flags of the EFLAGS register control I/O, maskable interrupts, debugging, task switching, and enabling of virtual 8086 execution in a protected, multitasking environment. These flags are highlighted in Figure 7.12.

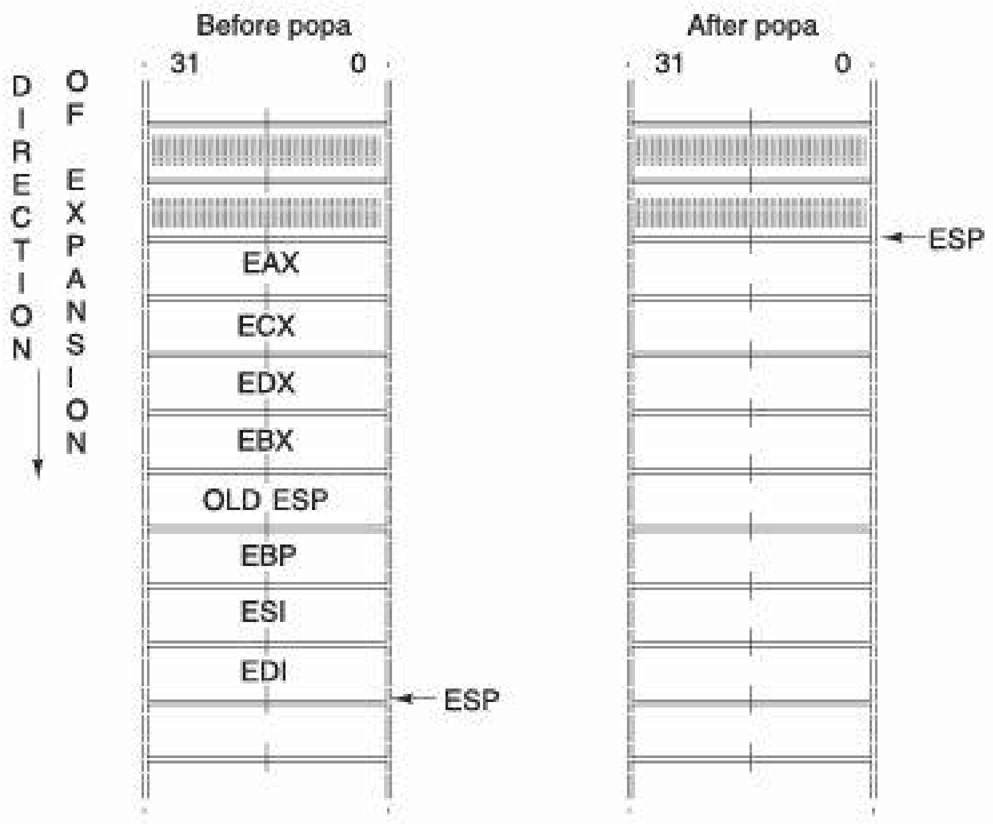


Fig. 7.12 POP A Instruction.

IF (Interrupt-Enable Flag, bit 9): Setting IF allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no effect on either exceptions or non-maskable external interrupts.

NT (Nested Task, bit 14): The processor uses the nested task flag to control chaining of interrupted and called tasks. NT influences the operation of the IRET instruction.

RF (Resume Flag, bit 16): The RF flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception.

TF (Trap Flag, bit 8): Setting TF puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an exception after each instruction, allowing a program to be inspected as it executes each instruction. Single-stepping is just one of the several debugging features of the 80386.

VM (Virtual 8086 Mode, bit 17): When set, the VM flag indicates that the task is executing an 8086 program.

7.3.8.3 Memory-Management Registers

Four registers of the 80386 locate the data structures that control segmented memory management:

GDTR Global Descriptor Table Register

LDTR Local Descriptor Table Register

These registers point to the segment descriptor tables GDT and LDT.

IDTR Interrupt Descriptor Table Register

This register points to a table of entry points for interrupt handlers (the IDT).

TR Task Register

This register points to the information needed by the processor to define the current task.

7.3.8.4 Control Registers

These registers are accessible to systems programmers only via variants of the MOV instruction, which allow them to be loaded from or stored in general registers; for example:

MOV EAX, CR0

MOV CR3, EBX

CR0 contains system control flags, which control or indicate conditions that apply to the system as a whole, not to an individual task.

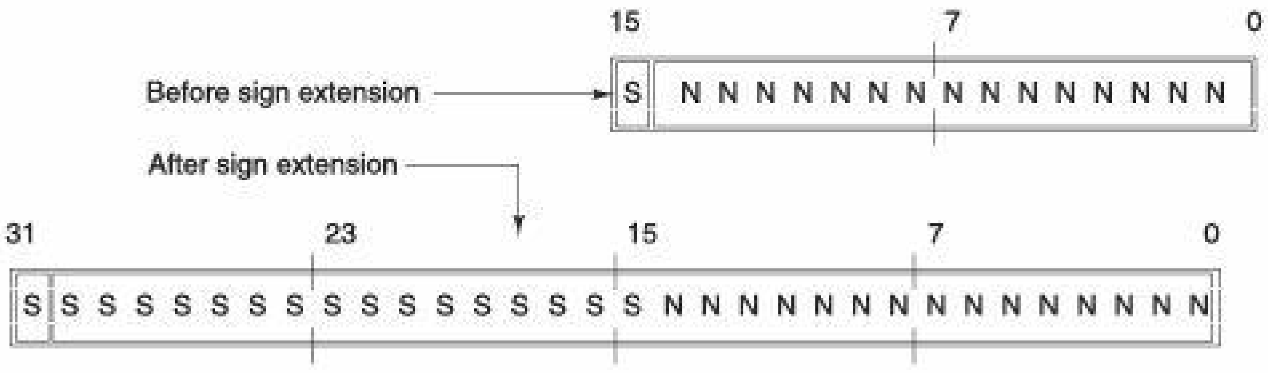

15
7
0
Before sign extension
S N N N N N N N N N N N N N N N
After sign extension
31
23
15
7
0
S S S S S S S S S S S S S S S S S N N N N N N N N N N N N N N N

Fig. 7.13 SIGN Extension.

EM (Emulation, bit 2): EM indicates whether coprocessor functions are to be emulated.

ET (Extension Type, bit 4): ET indicates the type of coprocessor present in the system (80287 or 80387).

MP (Math Present, bit 1): MP controls the function of the WAIT instruction, which is used to coordinate a coprocessor.

PE (Protection Enable, bit 0): Setting PE causes the processor to begin executing in protected mode. Resetting PE returns to real-address mode.

PG (Paging, bit 31): PG indicates whether the processor uses page tables to translate linear addresses into physical addresses.

TS (Task Switched, bit 3): The processor sets TS with every task switch and tests TS when interpreting coprocessor instructions.

CR2 is used for handling page faults when PG is set. The processor stores in CR2 the linear address that triggers the fault. CR3 is used when PG is set. CR3 enables the processor to locate the page table directory for the current task.

7.3.8.5 Debug Register

The debug registers bring advanced debugging abilities to the 80386, including data breakpoints and the ability to set instruction breakpoints without modifying code segments.

7.3.8.6 Test Registers

The test registers are not a standard part of the 80386 architecture. They are provided solely to enable confidence testing of the translation look aside buffer (TLB), the cache is used for storing information from page tables.

7.3.9 Systems Instructions

Systems instructions deal with such functions as:

1. Verification of Pointer Parameters
   ARPL—Adjust RPL
   LAR— Load Access Rights
   LSL—Load Segment Limit
   VERR—Verify for Reading
   VERW—Verify for Writing
2. Addressing Descriptor Tables
   LLDT—Load LDT Register
   SLDT—Store LDT Register
   LGDT—Load GDT Register
   SGDT—Store GDT Register
3. Multitasking
   LTR—Load Task Register
   STR—Store Task Register
4. Coprocessing and Multiprocessing
   CLTS—Clear Task-Switched Flag
   ESC—Escape instructions
   WAIT—Wait until Coprocessor not Busy
   LOCK—Assert Bus-Lock Signal
5. Input and Output
   IN—Input
   OUT—Output

INS—Input String
OUTS—Output String

6. Interrupt Control
   CLI—Clear Interrupt-Enable Flag
   STI—Set Interrupt-Enable Flag
   LIDT—Load IDT Register
   SIDT—Store IDT Register
7. Debugging (refer to Chapter 12)
   MOV—Move to and from debug registers
8. *TLB testing (refer to Chapter 10)*
   MOV—Move to and from test registers
9. *System Control*
   SMSW—Set MSW
   LMSW—Load MSW
   HLT—Halt Processor
   MOV—Move to and from control registers

The instructions SMSW and LMSW are provided for compatibility with the 80286 processor. 80386 programs access the MSW in CR0 via variants of the MOV instruction. HLT stops the processor until receipt of an INTR or RESET signal.

7.3.10 Memory Management

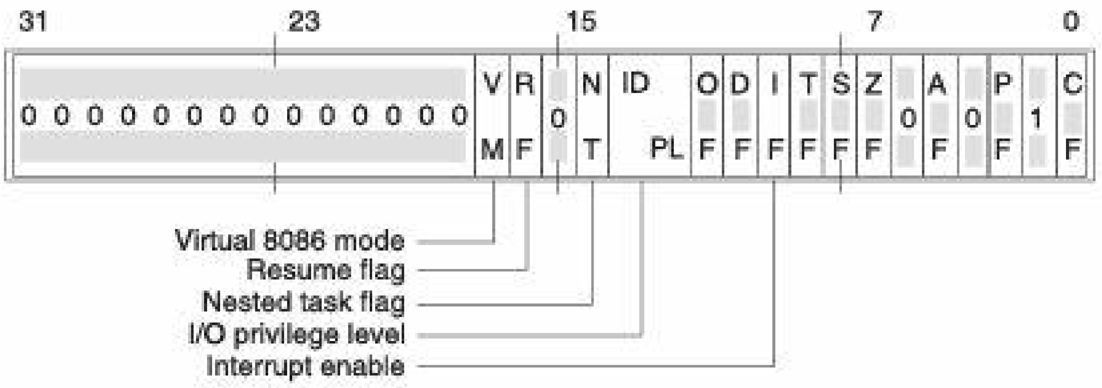
The 80386 transforms logical addresses (i.e., addresses as viewed by programmers) into physical address (i.e., actual addresses in physical memory) in two steps:

- Segment translation, in which a logical address (consisting of a segment selector and segment offset) are converted into a linear address.
- Page translation, in which a linear address is converted into a physical address. This step is optional, at the discretion of systems-software designers.

These translations are performed in a way that is not visible to applications programmers. Figure 7.14 presents a simplified view of the 80386 addressing mechanism. In reality, the addressing mechanism also includes memory protection features.

7.3.11 "Flat" Architecture

When the 80386 is used to execute software designed for architectures that don't have segments, it may be expedient to effectively "turn off" the segmentation features of the 80386. The 80386 does not have a mode that disables segmentation, but the same effect can be achieved by initially loading the segment registers with selectors for descriptors that encompass the entire 32-bit linear address space. Once loaded, the segment registers don't need to be changed. The 32-bit offsets used by 80386 instructions are adequate to address the entire linear-address space.



Note
0 or 1 Indicates intel reserved. Do not define_

Fig. 7.14 Flags in 80386.

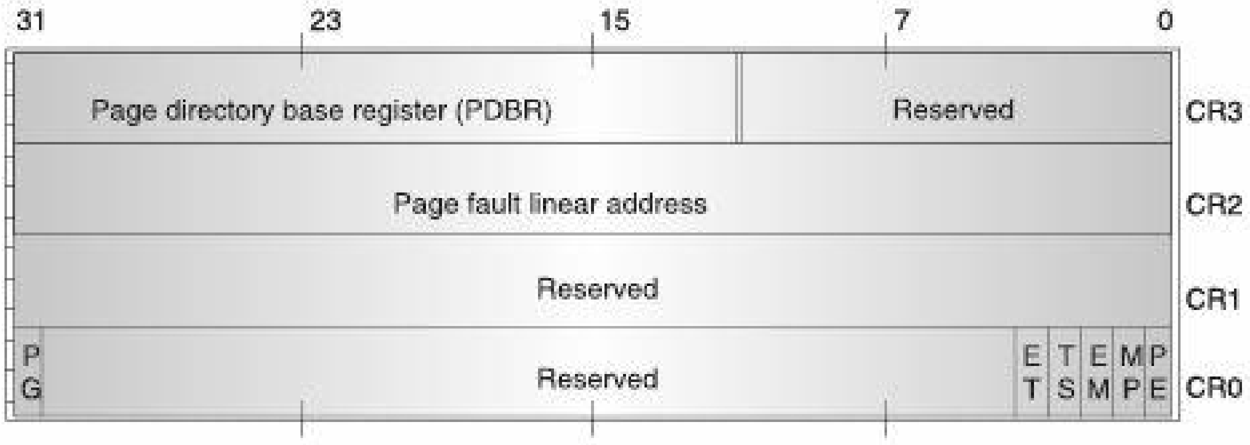

31
23
15
7
0
Page directory base register (PDBR)
Reserved
CR3
Page fault linear address
CR2
Reserved
CR1
PG
Reserved
ET
TS
EM
MP
PE
CR0

Fig. 7.15 Control Registers.

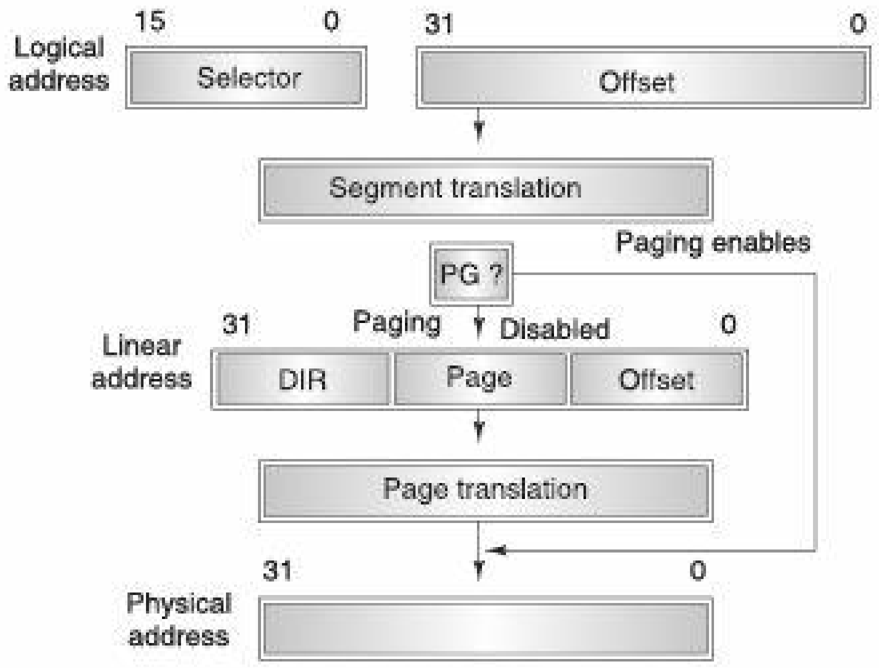


Fig. 7.16 Address Translation Overview.

7.3.12 Multitasking

To provide efficient, protected multitasking, the 80386 employs several special data structures. It does not, however, use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures. The registers and data structures that support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later. In addition to the simple task switch, the 80386 offers two other task-management features:

1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.
2. With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus, each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

Exception Conditions: This classification provides information needed by system programmers for restarting the procedure in which the exception occurred:

Faults: The CS and EIP values saved when a fault is reported point to the instruction causing the fault.

Traps: The CS and EIP values stored when the trap is reported point to the instruction dynamically after the instruction causing the trap. If a trap is detected during an instruction that alters program flow, the reported values of CS and EIP reflect the alteration of program flow. For example, if a trap is detected in a JMP instruction, the CS and EIP values are pushed onto the stack point to the target of the JMP, not to the instruction after the JMP.

Aborts: An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Interrupt 0—Divide Error: The divide-error fault occurs during a DIV or an IDIV instruction when the divisor is zero.

Interrupt 1—Debug Exceptions: The processor triggers this interrupt for any number of conditions; whether the exception is a fault or a trap depends on the condition:

- Instruction address breakpoint fault.
- Data address breakpoint trap.
- General detect fault.

- Single-step trap.
- Task-switch breakpoint trap.

The processor does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception.

Interrupt 3—Breakpoint: The INT 3 instruction causes this trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in an executable segment with the breakpoint opcode. The operating system or a debugging subsystem can use a data-segment alias for an executable segment to place an INT 3 anywhere it is convenient to arrest normal execution so that some sort of special processing can be performed. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task. The saved CS:EIP value points to the byte following the breakpoint. If a debugger replaces a planted breakpoint with a valid opcode, it must subtract one from the saved EIP value before returning.

Interrupt 4—Overflow: This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor cannot determine which is intended and, therefore, does not cause overflow exceptions automatically. Instead it merely sets OF when the results, if interpreted as signed numbers, would be out of range. When doing arithmetic on signed operands, careful programmers and compilers either test OF directly or use the INTO instruction.

Interrupt 5—Bounds Check: This fault occurs when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A program can use the BOUND instruction to check a signed array index against signed limits defined in a block of memory.

Interrupt 6—Invalid Opcode This fault occurs when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e., prefetching an invalid opcode does not cause this exception.) No error code is pushed on the stack. The exception can be handled within the same task. This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP referencing a register operand, or an LES instruction with a register source operand.

Interrupt 7—Coprocessor Not Available This exception occurs in either of two conditions:

1. The processor encounters an ESC (escape) instruction, and the EM (emulate) bit of CR0 (control register zero) is set.
2. The processor encounters either the WAIT instruction or an ESC instruction, and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.

7.3.13 Differences between 80286 and 80386

The few differences that do exist primarily affect operating system code.

Wraparound of 80286 24-Bit Physical Address Space: With the 80286, any base and offset combination that addresses beyond 16 MB wraps around to the first megabyte of the 80286 address space. With the 80386, since it has a greater physical address space, any such address falls into the 17th megabyte. In the unlikely event that any software depends on this anomaly, the same effect can be simulated on the 80386 by using paging to map the first 64 KB of the 17th megabyte of logical addresses to physical addresses in the first megabyte.

Reserved Word of Descriptor: Because the 80386 uses the contents of the reserved word (last word) of every descriptor, 80286 programs that place values in this word may not execute correctly on the 80386.

New Descriptor Type Codes Operating-system code that manages space in descriptor tables often uses an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for both the 80286 and 80386. Other values that were invalid on for the 80286 may be valid for the 80386 because of the additional descriptor types defined by the 80386.

Restricted Semantics of LOCK: The 80286 processor implements the bus lock function differently from the 80386. Programs that use forms of memory locking specific to the 80286 may not execute properly when transported to a specific application of the 80386. The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the results from using LOCK before any other instruction.

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

Additional exceptions

The 80386 defines new exceptions that can occur even in systems designed for the 80286.

- Exception #6—invalid opcode

This exception can result from improper use of the LOCK instruction.

- Exception #14—page fault

This exception may occur in an 80286 program if the operating system enables paging. Paging can be used in a system with 80286 tasks as long as all tasks use the same page directory. Because there is no place in an 80286 TSS to store the PDBR, switching to an 80286 task does not change the value of PDBR. Tasks ported from the 80286 should be given 80386 TSSs so they can take full advantage of paging.

7.3.14 Differences from 8086

In general, the 80386 in real-address mode will correctly execute ROM-based software designed for the 8086, 8088, 80186, and 80188. Following is a list of the minor differences between 8086 execution on the 80386 and on an 8086.

1. *Instruction clock counts:* The 80386 takes fewer clocks for most instructions than the 8086/8088. The areas most likely to be affected are:
   - Delays required by I/O devices between I/O operations.
   - Assumed delays with 8086/8088 operating in parallel with an 8087.
2. *Divide exceptions point to the DIV instruction:* Divide exceptions on the 80386 always leave the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS: IP value points to the next instruction.

3. *Undefined 8086/8088 opcodes:* Opcodes that were not defined for the 8086/8088 will cause exception 6 or will execute one of the new instructions defined for the 80386.
4. *Value written by PUSH SP:* The 80386 pushes a different value on the stack for PUSH SP than the 8086/8088. The 80386 pushes the value of SP before SP is incremented as part of the push operation; the 8086/8088 pushes the value of SP after it is incremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

   ```
   PUSH BP
   MOV BP, SP
   XCHG BP, [BP]
   ```

   This code functions as the 8086/8088 PUSH SP instruction on the 80386.
5. *Shift or rotate by more than 31 bits*. The 80386 masks all shift and rotate counts to the low order five bits. This MOD 32 operation limits the count to a maximum of 31 bits, thereby limiting the time that interrupt response is delayed while the instruction is executing.
6. *Redundant prefixes*. The 80386 sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. Exception 13 occurs if the limit on instruction length is violated. The 8086/8088 has no instruction length limit.
7. *Operand crossing offset 0 or 65,535:* On the 8086, an attempt to access a memory operand that crosses offset 65,535 (e.g., MOV a word to offset 65,535) or offset 0 (e.g., PUSH a word when SP = 1) causes the offset to wrap around modulo 65,536. The 80386 raises an exception in these cases—exception 13 if the segment is a data segment (i.e., if CS, DS, ES, FS, or GS is being used to address the segment), exception 12 if the segment is a stack segment (i.e., if SS is being used).
8. *Sequential execution across offset 65,535:* On the 8086, if sequential execution of instructions proceeds past offset 65,535, the processor fetches the next instruction byte from offset 0 of the same segment. On the 80386, the processor raises exception 13 in such a case.
9. *LOCK is restricted to certain instructions:* The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80386 always asserts the LOCK signal during an XCHG instruction with memory (even if the LOCK prefix is not used). LOCK may only be used with the following 80386 instructions when they update memory:

   BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT, and NEG. An undefined-opcode exception (interrupt 6) results from using LOCK before any other instruction.
10. *Single-stepping external interrupt handlers:* The priority of the 80386 single-step exception is different from that of the 8086/8088. The change prevents an external interrupt handler from being single-stepped if the interrupt occurs while a program is being single-stepped. The 80386 single-step exception has higher priority than any external

interrupt. The 80386 will still single-step through an interrupt handler invoked by the INT instructions or by an exception.

11. *IDIV exceptions for quotients of 80H or 8000H:* The 80386 can generate the largest negative number as a quotient for the IDIV instruction. The 8086/8088 causes exception zero instead.
12. *Flags in stack:* The setting of the flags stored by PUSHF, by interrupts, and by exceptions is different from that stored by the 8086 in bit positions 12 through 15. On the 8086, these bits are stored as ones, but in 80386 real-address mode bit 15 is always zero, and bits 14 through 12 reflect the last value loaded into them.
13. *NMI interrupting NMI handlers:* After an NMI is recognized on the 80386, the NMI interrupt is masked until an IRET instruction is executed.
14. *Coprocessor errors vector to interrupt 16:* Any 80386 system with a coprocessor must use interrupt vector 16 for the coprocessor error exception. If an 8086/8088 system uses another vector for the 8087 interrupt, both vectors should point to the coprocessor-error exception handler.
15. *Numeric exception handlers should allow prefixes:* On the 80386, the value of CS:IP saved for coprocessor exceptions points at any prefixes before an ESC instruction. On 8086/8088 systems, the saved CS:IP points to the ESC instruction.
16. *Coprocessor does not use interrupt controller:* The coprocessor error signal to the 80386 does not pass through an interrupt controller (an 8087 INT signal does). Some instructions in a coprocessor error handler may need to be deleted if they deal with the interrupt controller.
17. *Six new interrupt vectors:* The 80386 adds six exceptions that arise only if the 8086 program has a hidden bug.

    Exceptions

    - Processor detected: These are further classified as faults, traps, and aborts.
    - Programmed: The instructions INTO, INT 3, INT n, and BOUND can trigger exceptions. These instructions are often called "software interrupts", but the processor handles them as exception.

    It is recommended that exception handlers be added that treat these exceptions as invalid operations. This additional software does not significantly affect the existing 8086 software because the interrupts do not normally occur. These interrupt identifiers should not already have been used by the 8086 software, because they are in the range reserved by Intel.
18. *One megabyte wraparound:* The 80386 does not wrap addresses at 1 megabyte in real-address mode. On members of the 8086 family, it is possible to specify addresses greater than one megabyte. For example, with a selector value 0FFFFH and an offset of 0FFFFH, the effective address would be 10FFEFH (1 Mbyte + 65519). The 8086, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to 0FFEFH. However, the 80386, which can form addresses up to 32 bits long does not truncate such an address.

Things to Remember

◊ The 80186 is a very high integration 16-bit microprocessor.

◊ The 80186 is object code compatible with the 8086/8088 microprocessors and adds 10 new instruction types to the 8086/8088 instruction set.

◊ **INTR** is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/ CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.

◊ INT instruction – breakpoint interrupt. This is a type 3 interrupt.

◊ INT <interrupt number> instruction—any one interrupt from available 256 interrupts.

◊ INTO instruction – interrupt on overflow. This is a type 4 interrupt.

◊ **Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions.

◊ **Stack segment** (SS) is a 16-bit register containing address of 64 KB segment with program stack.

◊ **Data segment** (DS) is a 16-bit register containing address of 64 KB segment with program data.

◊ **Extra segment** (ES) is a 16-bit register containing address of 64 KB segment, usually with program data.

◊ **Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.

◊ **Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.

◊ **Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.

◊ **Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.

◊ **Stack Pointer** (SP) is a 16-bit register pointing to program stack.

◊ **Control registers** are used to control 80186 integrated peripherals. These 16-bit registers are part of 256-byte control block that can be mapped into system RAM or I/O space.

◊ **Based Indexed with displacement**—8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

◊ The 80286 has built-in memory protection that supports operating systems and task isolation as well as program and data privacy within tasks. A 25 MHz 80286 microprocessor is capable of providing upto twenty times the maximum throughput that can be achieved on a 5 MHz 8086 processor based system.

◊ The intel's 80286 operates in two modes: Real Address Mode and Protected Virtual Address Mode (PVAM).

◊ In a "flat" model of memory organization, the applications programmer sees a single array of up to 232 bytes (4 gigabytes).

◊ An instruction can act on zero or more operands, which are the data manipulated by the instruction. An example of a zero-operand instruction is NOP (no operation).

◊ MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand.

◊ XCHG (Exchange) swaps the contents of two operands. This instruction takes the place of three MOV instructions.

◊ The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands.

◊ **Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions.

◊ **Stack segment** (SS) is a 16-bit register containing address of 64 KB segment with program stack.

◊ **Data segment** (DS) is a 16-bit register containing address of 64 KB segment with program data.

◊ **Extra segment** (ES) is a 16-bit register containing address of 64 KB segment, usually with program data.

◊ **Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.

◊ **Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.

◊ **Count**register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.

◊ **Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.

◊ **Stack Pointer** (SP) is a 16-bit register pointing to program stack.

◊ **Control registers** are used to control 80186 integrated peripherals. These 16-bit registers are part of 256-byte control block that can be mapped into system RAM or I/O space.

◊ **Based Indexed with displacement**—8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

◊ The 80286 has built-in memory protection that supports operating systems and task isolation as well as program and data privacy within tasks. A 25 MHz 80286 microprocessor is capable of providing upto twenty times the maximum throughput that can be achieved on a 5 MHz 8086 processor based system.

◊ The Intel's 80286 operates in two modes: Real Address Mode and Protected Virtual Address Mode (PVAM).

◊ In a "flat" model of memory organization, the applications programmer sees a single array of up to 232 bytes (4 gigabytes).

◊ An instruction can act on zero or more operands, which are the data manipulated by the instruction. An example of a zero-operand instruction is NOP (no operation).

◊ MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand.

◊ XCHG (Exchange) swaps the contents of two operands. This instruction takes the place of three MOV instructions.

◊ The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands.

Questions and Answers

1. Draw the architecture of 80186 and give a brief introduction.

Answer: It also contains program, data and stack memories that occupy the same memory space. The total addressable memory size is 1 MB. As most of the processor instructions use 16-bit pointers, the processor can effectively address only 64 KB of memory. To access memory outside of 64 KB, the CPU uses special segment registers to specify where the code, stack and data 64KB segments are positioned within 1 MB of memory.

Data memory

80186 contains memory in four segments in which each segment only has accessible memory upto 64 KB. Every segment has its notation for accessing, i.e., DS = Data segment, CS = Code segment, SS = Stack segment, ES = Extra segment. Word data can be located at odd or even byte boundaries. The processor uses two memory accesses to read 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.

Interrupts

80186 contains both hardware and software interrupts. Integrated 80186 peripherals generate the following hardware interrupts (higher priority interrupts are listed first):

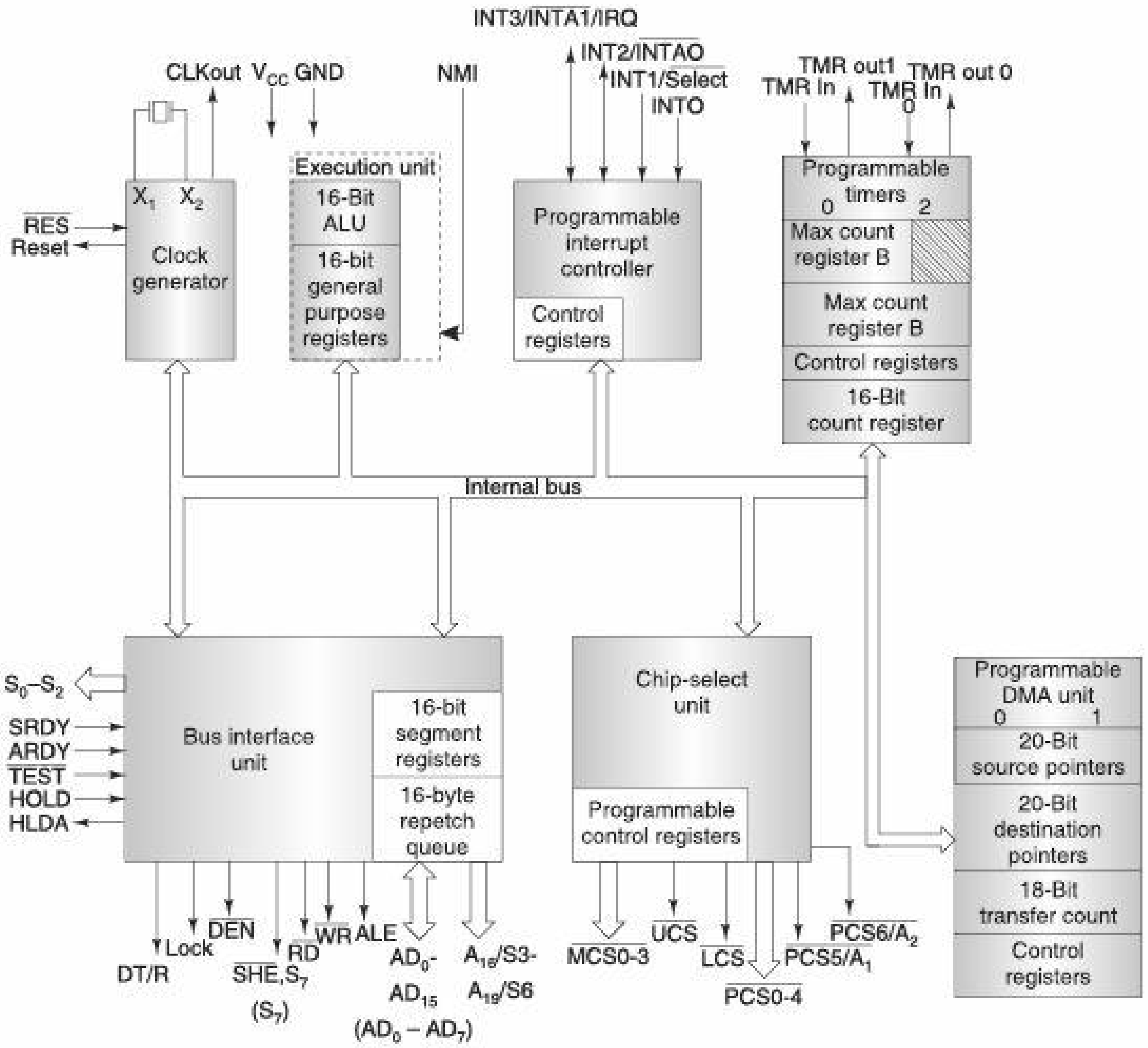
- Timer 0
- Timer 1
- Timer 2
- DMA 0
- DMA 1
- INT0
- INT1
- INT2
- INT3

I/O ports

The 80186 has 8-bit I/O ports. These ports can also be addressed as 32768 16-bit I/O ports.

Reserved ports:

- 00F8h—00FFh
- FF00h—FFFFh: 256-byte control block. This is a default block location after RESET. If necessary, the block can be remapped to different place in I/O or memory space.



Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low-order byte of the word, and CH contains the high-order byte. Count register can be used as a counter in string manipulation and shift/rotate instructions.

Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

2. Explain the various registers of 86XXX series microprocessor.

Answer: The following registers are both general and index registers:

Stack Pointer (SP) is a 16-bit register pointing to program stack.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Other registers:

Instruction Pointer (IP) is a 16-bit register.

Flags is a 16-bit register containing nine 1-bit flags:

- Overflow Flag (OF)—set if the result is too large positive number, or is too small negative number to fit into destination operand.
- Direction Flag (DF)—if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.
- Interrupt-enable Flag (IF)—setting this bit enables maskable interrupts.
- Single-step Flag (TF)—if set then single-step interrupt will occur after the next instruction.
- Sign Flag (SF)—set if the most significant bit of the result is set.
- Zero Flag (ZF)—set if the result is zero.
- Auxiliary carry Flag (AF)—set if there was a carry from or borrow to bits 0-3 in the AL register.
- Parity Flag (PF)—set if parity (the number of "1" bits) in the low-order byte of the result is even.
- Carry Flag (CF)—set if there was a carry from or borrow to the most significant bit during last result calculation.

3. Give the pin description of 80186 and explain the function of each pin

Answer: Various pins are as follows:

VCC: System Power: +5 V power supply.

VSS: This is system ground pin, it is connected to 0 volt.

RESET: Reset Output indicates that the CPU is being reset, and can be used as a system reset. It is active HIGH, synchronized with the processor clock, and lasts an integer number of clock periods corresponding to the length of the RES signal.

X1, X2: Crystal Inputs X1 and X2 provide external connections for a fundamental mode parallel resonant crystal for the internal oscillator.

CLKOUT: Clock Output provides the system with a 50% duty cycle waveform. All device pin timings are specified relative to CLKOUT.

RES: An active RES causes the processor to immediately terminate its present activity, clear the internal logic, and enter a dormant state.

TEST: TEST is examined by the WAIT instruction. If the TEST input is HIGH when WAIT execution begins, instruction execution will suspend.

TMR IN 0, TMR IN 1: Timer inputs are used either as clock or control signals, depending upon the programmed timer mode.

TMR OUT 0, TMR OUT 1: Timer outputs are used to provide single pulse or continuous waveform generation, depending upon the timer mode selected.

DRQ0, DRQ1: DMA Request is asserted HIGH by an external device when it is ready for DMA Channel 0 or 1 to perform a transfer. These signals are level-triggered and internally synchronized.

NMI: The Non-Maskable Interrupt input causes a Type 2 interrupt. An NMI transition from LOW to HIGH is latched and synchronized internally, and initiates the interrupt at the next instruction boundary.

INT0, INT1/SELECT, INT2/INTA0, INT3/INTA1/IRQ: Maskable Interrupt Requests can be requested by activating one of these pins. When configured as inputs, these pins are active HIGH. Interrupt Requests are synchronized internally. INT2 and INT3 may be configured to provide active-LOW interrupt-acknowledge output signals.

A16-19/S3-S6: Address Bus Outputs (16 ± 19) and Bus Cycle Status (3 ± 6) indicate the four most significant address bits during T_1.

AD_0-AD_{15}: Address/Data Bus signals constitute the time multiplexed memory or I/O address (T_1) and data (T_2, T_3, T_W, and T_4) bus.

BHE/S_7: This is bus high enable pin which indicates transfer of data on the upper byte of the data bus D_{15}-D_8. For data transfer, BHE and A_0 are encoded.

BHE and A0 Encoding (80186 Only)		
BHE Value	A0 Value	Function
0	0	Word Transfer
0	1	Byte Transfer on upper half of data bus (D15-D8)
1	0	Byte Transfer on lower half of data bus (D7-D0)
1	1	Reserved

ALE/QS0: Address Latch Enable/Queue Status 0 is provided by the processor to latch the address. ALE is active HIGH. Addresses are guaranteed to be valid on the trailing edge of ALE.

WR/QS1: Write Strobe/Queue Status 1 indicates that the data on the bus is to be written into a memory or an I/O device. WR is active for T_2, T_3 and T_W of any write cycle. 8

QS1	QS0	Queue Operation
0	0	No queue operation
0	1	First opcode byte fetched from the queue
1	1	Subsequent byte fetched from the queue
1	0	Empty the queue

RD/QSMD: Read Strobe is an active LOW signal which indicates that the processor is performing a memory or I/O read cycle.

ARDY: Asynchronous Ready informs the processor that the addressed memory space or I/O device will complete a data transfer. The ARDY pin accepts a rising edge that is asynchronous to CLKOUT, and is active HIGH.

SRDY: Synchronous Ready informs the processor that the addressed memory space or I/O device will complete a data transfer. The SRDY pin accepts an active-HIGH input synchronized to CLKOUT.

LOCK: It is an active low pin. This is a bus lock and indicates that other system bus masters cannot gain control of the system bus following the current bus cycle.

HOLD and HLDA: HOLD indicates that another bus master is requesting the local bus. The HOLD input is active HIGH. HOLD may be asynchronous with respect to the processor clock. The processor will issue a HLDA (HIGH) in response to a HOLD request at the end of T4 or Ti. Simultaneous with the issuance of HLDA, the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will lower HLDA. When the processor needs to run another bus cycle, it will again drive the local bus and control lines.

UCS: Upper Memory Chip Select is an active LOW output whenever a memory reference is made to the defined upper portion (1 K ± 256 K block) of memory. This line is not floated during bus HOLD. The address range activating UCS is software programmable.

LCS: Lower Memory Chip Select is active LOW whenever a memory reference is made to the defined lower portion (1 K ± 256 K) of memory. This line is not floated during bus HOLD. The address range activating LCS is software programmable.

MCS0-3: Mid-Range Memory Chip Select signals are active LOW when a memory reference is made to the defined mid-range portion of memory (8 K ± 512 K). These lines are not floated during bus HOLD. The address ranges activating MCS0 ± 3 are software programmable.

PCS0-4: Peripheral Chip Select signals 0 ± 4 are active LOW when a reference is made to the defined peripheral area (64 KB I/O space). These lines are not floated during bus HOLD. The address ranges activating PCS0 ± 4 are software programmable.

PCS5/A1: Peripheral Chip Select 5 or latched A1 may be programmed to provide a sixth peripheral chip select, or to provide an internally latched A1 signal. The address range activating PCS5 is software-programmable. PCS5/A1 does not float during bus HOLD. When programmed to provide latched A1, this pin will retain the previously latched value during HOLD.

PCS6/A2: Peripheral Chip Select 6 or latched A2 may be programmed to provide a seventh peripheral chip select, or to provide an internally latched A2 signal.

DT/R: Data Transmit/Receive controls the direction of data flow through an external data bus trans-receiver.

DEN: Data Enable is provided as a data bus transreceiver output enable. DEN is active LOW during each memory and I/O access. DEN is HIGH whenever DT/R changes state.

4. What is the function of Protected Virtual Address Mode in 80286?

Answer: The functioning of the 80286 in Protected Virtual Address Mode (PVAM) is much different from the Real Address Mode. In PVAM, the 80286 possesses memory management, protection, task switching, and interrupt processing. After boot/up/or reset, the 80286 starts operating in Real Address Mode. The real address mode is used to initialize peripheral devices, load the main part of the OS from disk into memory, load some registers, enable interrupt, and enter the PVAM. The 80286 enters into the PVAM by setting the protection enable bit of the Machine Status Word (MSW).

The Figure 7.17 represents the format for the MSW. The MSW is a 16-bit register. The bit 0 is the protection enable (PE) bit. Bits 1, 2 and 3 indicate whether a processor extension (such as coprocessor) is present or not. The bits are changed in the machine status word (MSW) by loading the desired word in a register or memory location and executing the Load Machine Status Word (LMSW) instruction. Once the PVAM is entered by executing the LMSW instruction, resetting the system can only get it back to the real address mode. This processor is so designed that a programmer should not switch the system back into real address mode to defeat the protection scheme in PVAM.

The original 286 (more properly, 80286) was a vast power increase over the 8086/8088, particularly the 8/16-bit hybrid 8088. The 8 MHz ones were rare, as by the time most people could afford to buy 286s they were buying 12 and 16 MHz ones.

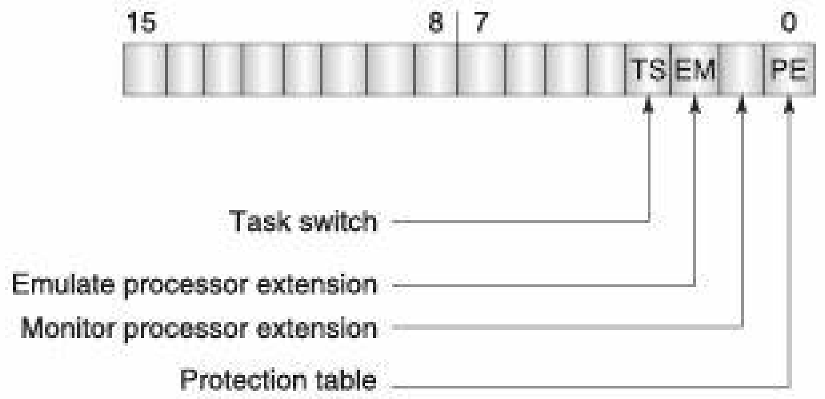


Fig. 7.17 Protected Virtual Address Mode PVAM.

5. Explain the flat mode and segmented mode in 80386.

Answer:

The "Flat" Model: In a "flat" model of memory organization, the application programmer sees a single array of up to 232 bytes (4 gigabytes). While the physical memory can contain up to 4 gigabytes, it is usually much smaller; the processor maps the 4 gigabytes flat space onto the physical address space by the address translation mechanisms. Applications programmers do not need to know the details of the mapping. A pointer into this flat address space is a 32-bit ordinal number that may range from 0 to 232-1. Relocation of separately-compiled modules in this space must be performed by systems software (e.g., linkers, locators, binders, loaders).

The Segmented Model: In a segmented model of memory organization, the address space as viewed by an applications program (called the logical address space) is a much larger space of up to 246 bytes (64 terabytes). The processor maps the 64 terabyte logical address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms. Application programmers do not need to know the details of this mapping. Applications programmers view the logical address space of the 80386 as a collection of up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space. Segment sizes may range from one byte up to a maximum of 232 bytes (4 gigabytes).

6. What are the various registers of 80386?

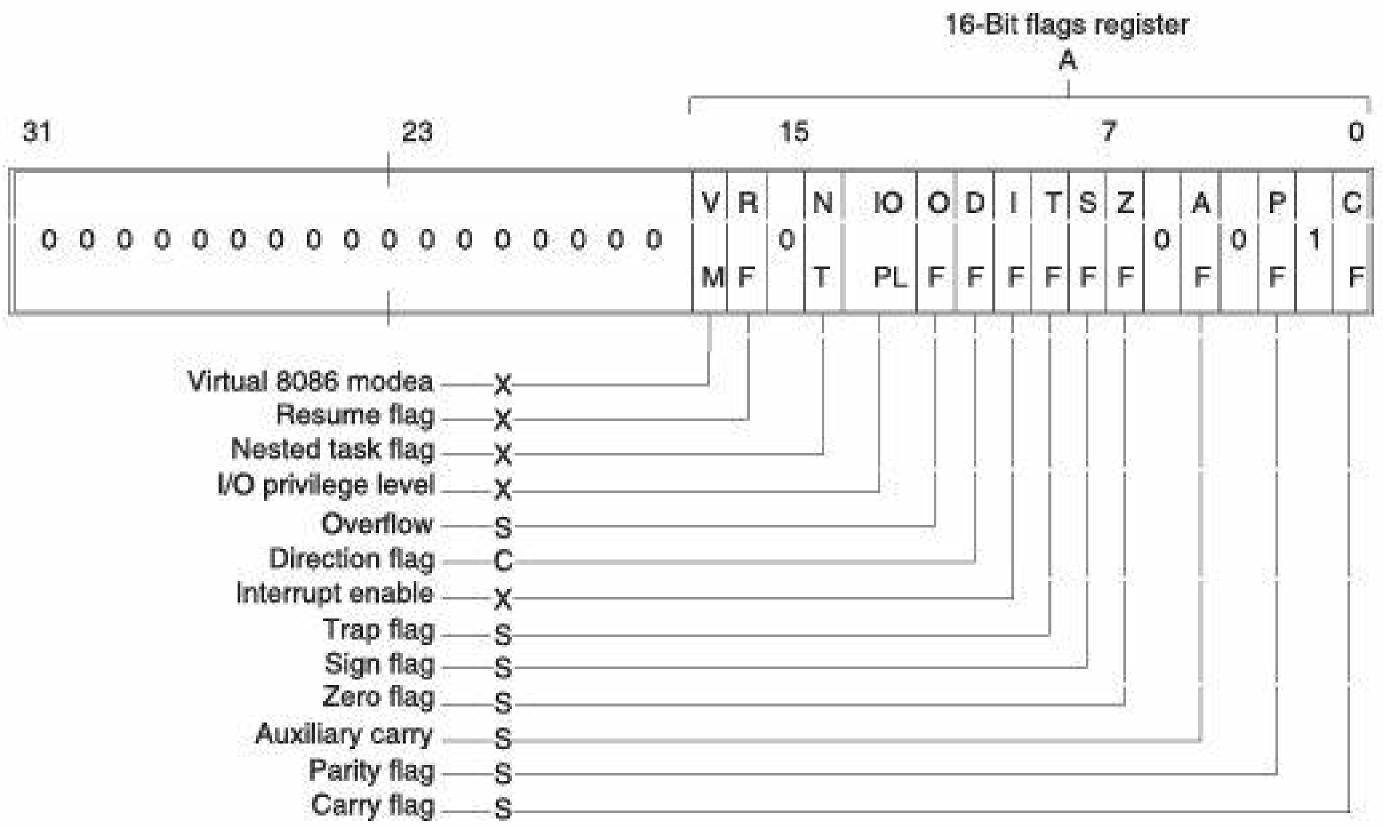
Answer: The 80386 contains a total of 7 registers that are of interest to the applications programmer. These registers may be grouped into these basic categories:

1. General-purpose register
2. Segment register
3. Instruction pointer and flags
4. Control registers
5. System address register
6. Debug register
7. Test register
    1. *General registers:* These eight 32-bit general-purpose registers are used primarily to contain operands for arithmetic and logical operations.
    2. *Segment registers:* These special-purpose registers permit system software designers to choose either a flat or segmented model of memory organization. These six registers determine, at any given time, which segments of memory are currently addressable.
    3. *Status and instruction registers:* These special-purpose egisters are used to record and alter certain aspects of the 80386 processor state.

7. Explain the flags register of 80386.

Flags Register: The flags register is a 32-bit register named EFLAGS. The flags control certain operations and indicate the status of the 80386. The low-order 16 bits of EFLAGS is named FLAGS and can be treated as one unit. This feature is useful when executing 8086 and 80286 code, because this part of FLAGS-EFLAGS is identical to the FLAGS register of the 8086 and the 80286. The flags may be considered in three groups: the status flags, the control flags, and the systems flags as shown in Figure 7.18.

Status Flags: The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that



S = Status flag, C = Control flag, X = System flag

Note: 0 Or 1 indicates intel reserved. do not define

Fig. 7.18 Flag Status of 80386.

their operations are complete. There are instructions to set, clear, and complement CF before the execution of an arithmetic instruction.

Control Flag: The control flag DF of the EFLAGS register controls string instructions. DF (Direction Flag, bit 10) Setting causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses. Clearing DF causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

8. What is the instruction format of 80386?

Answer:

Instruction Format: The information encoded in an 80386 instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction must also select, explicitly or implicitly, which of the currently addressable segments contains the operand. 80386 instructions are composed of various elements and have various formats. These instruction elements, only one, the opcode, is always present. The other elements may or may not be present, depending on the particular operation involved and on the location and type of the operands. The elements of an instruction, in order of occurrence are as follows:

1. Prefixes: one or more bytes preceding an instruction that modify the operation of the instruction. The following types of prefixes can be used by application programs:

   (a) Segment override—explicitly specifies which segment register an instruction should use, thereby overriding the default segment-register selection used by the 80386 for that instruction.
   (b) Address size—switches between 32-bit and 16-bit address generation.
   (c) Operand size—switches between 32-bit and 16-bit operands.
   (d) Repeat—used with a string instruction to cause the instruction to act on each element of the string.
2. Opcode—specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different variant of the operation.
3. Register specifier—an instruction may specify one or two register operands. Register specifiers may occur either in the same byte as the opcode or in the same byte as the addressing mode specifier.
4. Addressing-mode specifier—when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.
5. SIB (scale, index, base) byte—when the addressing-mode specifier indicates that an index register will be used to compute the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.
6. Displacement—when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16 or 8 bits. The 8-bit form is used in the common case when the displacement is sufficiently small. The processor extends an 8-bit displacement to 16 or 32 bits, taking into account the sign.
7. Immediate operand—when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide. In cases where an 8-bit immediate operand is combined in some way with a 16- or 32-bit operand, the processor automatically extends the size of the 8-bit operand, taking into account the sign.

Exercise

1. What is the function of an accumulator in 80186?
2. How many address lines are used to identify an I/O port in the peripheral I/O and in memory-mapped I/O methods in 80386?
3. While executing a program, when the 80386 completes the fetching of the machine code located at the memory address 2057H, what is the content of a program counter?
4. Explain the need to demultiplex the bus AD_7-AD_0.
5. Explain the functions of ALE and IO/M signals of the 80386 microprocessor.
6. Explain why a latch is used for an output port, but a tri-state buffer can be used for an input port.
7. What are the control signals necessary in the memory-mapped I/O?
8. Can the microprocessor differentiate whether it is reading from a memory-mapped input port or from memory?

9. Write a program using the ADI instruction to add two hexadecimal numbers 3AH and 69H and to display the answer at an output port.
10. Write instructions to
    (a) Load 15H in the accumulator
    (b) Decrement the accumulator
    (c) Display the answer

    Specify the answer you would except at the output port.
11. Write a program to
    (a) Clear the accumulator
    (b) Add 47H
    (c) Subtract 92H
    (d) Add 64H (using ADI instruction)
    (e) Display the results after subtracting 92H and after adding 47H.
12. Explain architecture of 80286 with pin description.
13. Draw the real mode diagram of 80286.
14. What are the differences between 80186 and 80286?
15. What are the differences between 80186 and 80386?
16. Explain the pin description of 80186.
17. What are the main interrupts in 80386.
18. What does NMI stand for? Explain its function in 80386.
19. What is the function of accumulator in 80386?
20. Draw the table of all pin descriptions of 80386.
21. Explain the segment register of 80386.
22. Compare microprocessors 8085, 8086, 80186, 80286, 80386.

Chapter 8

Microcontrollers 8051

8.1 INTRODUCTION

The rapidly expanding technology with in microcontrollers have contributed to many development in computer applications, industrial instrumentation and controls. With the advancement in the semiconductor technology, the whole of CPU of a digital computer is now fabricated on a single chip using LSI and VLSI technology.

8.2 MICROPROCESSOR V/S MICROCONTROLLER

A microcontroller differs from a microprocessor in many ways. Early name for a microcontroller was microcomputer. The main difference between a microprocessor and microcontroller is the completeness of the machine each represents. In order to put a microprocessor into use the designer required memory peripheral chips and serial and parallel ports to make completely functional computer. On the other hand a complete computer based system could be built using single chip microcontrollers, with a minimum of external components. Figure 8.1 shows a simplified block diagram of a microprocessor, which consists of an arithmetic and logic unit (ALU), general-purpose registers, stack pointer (SP), program counter (PC), clock timing circuits and interrupt circuits.

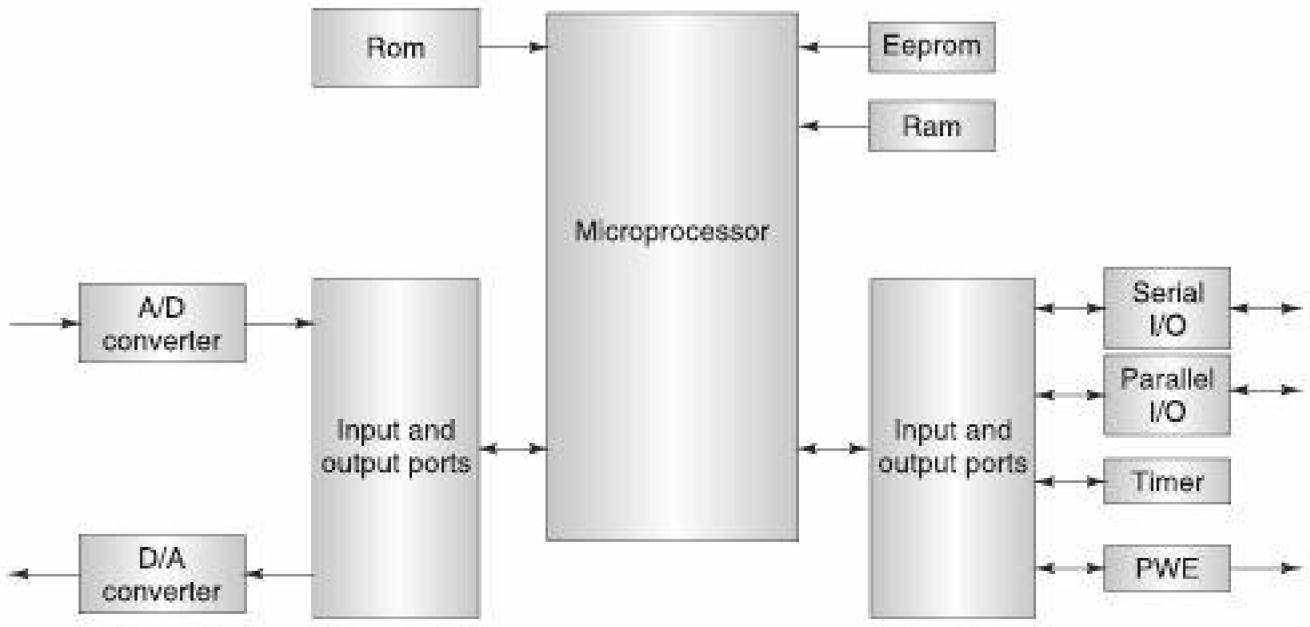


Fig. 8.1 Block Diagram of Microprocessor.

To make a complete microcomputer system only microprocessor is not sufficient, but it is required to add other peripherals such as read only memory (ROM), read/write memory (RAM), decoders, drivers, number of input/output devices to make a complete microcomputer system. In addition special purpose devices, such as interrupt controller, programmable timers, programmable I/O devices, DMA controllers may be added to improve the capacity, performance and flexibility of a microcomputer system.

A microcontroller incorporates all the features that are found in a microprocessor, except it also has added features to make a complete microcomputer system on its own. The microcontroller has built-in ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit. Figure 8.2 shows the simplified block diagram of a microcontroller.

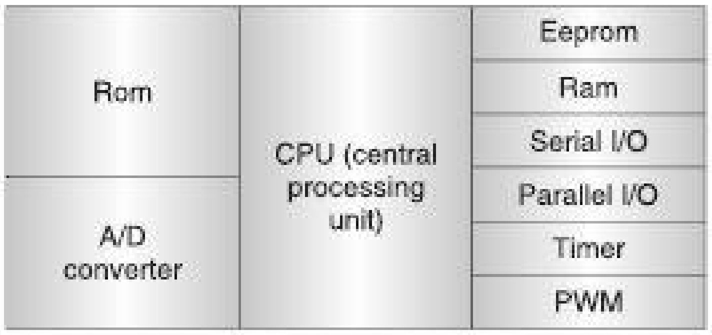


Fig. 8.2 Block Diagram of Microcontroller.

Now the Table 8.1 gives few comparisons between microprocessor an microcontroller as below:

Table 8.1 Comparison between Microprocessor and Microcontroller

Sr. No.	*Microprocessor*	*Microcontroller*
1.	Microprocessor contains ALU, general-purpose register, stack pointer, program counter, clock timing circuit and interrupt circuit.	Microcontroller contains the circuitry of micro-processor and in addition it has built-in ROM, RAM, I/O devices, timers and counters.
2.	It has many instructions to move data between memory and CPU.	It has one or two instructions to move data between memory and CPU.
3.	It has one or two-bit handling instructions.	It has many bit handling instructions.
4.	Access times for memory and I/O devices are more.	Less access times for built-in memory and I/O devices.
5.	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing the reliability.
6.	Microprocessor based system is more flexible in design point of view.	Less flexible in design point of view.
7.	It has single memory map for data and code.	It has separate memory map for data and code.
8.	Less number of pins are multi-functioned.	Number of pins are multi-functioned.

8.3 8051 FAMILY OVERVIEW

In this section, we first look at the various members of the 8051 family of microcontrollers and their internal features. Plus we see who the different manufacturers of the 8051 are and what kind of products they offer.

8.3.1 8051 Microcontroller

In 1981, Intel Corporation introduced an 8-bit microcontroller called the 8051. This microcontroller had 128 bytes of RAM, 4 Kbytes of on-chip ROM, two timers, one serial port, and four ports (each 8-bits wide) all on a single chip. At the time it was also referred to as a “system on a chip.” The 8051 is an 8-bit processor, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU. The 8051 has a total of four I/O ports, each 8 bits wide. Although the 8051 can have a maximum of 64 Kbytes of on-chip ROM, many manufacturers have put only 4 Kbytes on the chip.

Features of 8051

The features of the 8051 family are as follows:

1. 4096 bytes on-chip program memory.
2. 128 bytes on-chip data memory on-chip.
3. Four register banks.
4. 128 user-define software flags.
5. 64 kilobytes each program and external RAM addressability.
6. One microsecond instruction cycle with 12 MHz crystal.
7. 32 bidirectional, I/O lines organized as four 8-bits ports (16 lines on 8031).
8. Two multiple mode, 16-bit Timers/counters.
9. Multiple modes, high-speed programmable serial port.
10. Two-level prioritized interrupt structure.
11. Full depth stack for subroutine return linkage and data storage.
12. Direct byte and bit addressability.
13. Binary or Decimal arithmetic.
14. Signed-overflow detection and parity computation.
15. Hardware Multiple and Divide in 4 use.
16. Integrated Boolean Processor for control applications.
17. Upwardly compatible with existing 8084 software.

8.3.2 Other Members of the 8051 Family

There are two other members in the 8051 family of microcontrollers. They are the 8052 and the 8031.

8.3.3 8052 Microcontroller

The 8052 is another member of the 8051 family. The 8052 has all the standard features of the 8051 as well as extra 128 bytes of RAM and an extra timer. In other words, the 8052 has 256 bytes of RAM and 3 timers. It also has 8 Kbytes of on-chip program ROM instead of 4 Kbytes.

The 8051 is a subset of the 8052; therefore, all programs written for the 8051 will run on the 8052, but the reverse is not true.

8.3.4 8031 Microcontroller

Another member of the 8051 family is the 8031 chip. This chip is often referred to as a ROM-less 8051 since it has 0 Kbytes of on-chip ROM. To use this chip you must add external ROM to it. This external ROM must contain the program that the 8031 will fetch and execute. Contrast that to the 8051 in which the on-chip ROM contains the program to be fetched and executed but is limited to only 4 Kbytes of code. The ROM containing the program attached to the 8031 can be as large as 64 Kbytes. In the process of adding external ROM to the 8031, you lose two ports. That leaves only 2 ports (of the 4 ports) for I/O operations. To solve this problem, you can add external I/O to the 8031. Interfacing the 8031 with memory and I/O ports such as the 8255 chip is discussed in Chapter 5. There are also various speed versions of the 8031 available from different companies.

8.3.5 Various 8051 Microcontrollers

Although the 8051 is the most popular member of the 8051 family, you will not see "8051" in the part number. This is because the 8051 is available in different memory types, such as UV-EPROM, flash, and NV-RAM, all of which have different part numbers. A discussion of the various types of ROM will be given in Chapter 14. The UV-EPROM version of the 8051 is the 8751. The flash ROM version is marketed by many companies including Atmel Corp. and Dallas Semiconductor. The Atmel Flash 8051 is called AT89C51, while Dallas Semiconductor calls theirs DS89C4xO (DS89C420/430/440). The NV-RAM version of the 8051 made by Dallas Semiconductor is called DS5000. There is also an OTP (one-time programmable) version of the 8051 made by various manufacturers. Next we discuss briefly each of the above chips and describe applications where they are used.

8.3.6 Comparison of 8051 Family Members

The comparison between 8051, 8052 and 8031 microcontroller is shown in Table 8.2.

Table 8.2 Comparison of 8051 Family Members

FEATURES	8051	8052	8031
ROM (on-chip program space in bytes)	4 K	8 K	0 K
RAM(bytes)	128	256	128
Timers	2	3	2
I/O Pins	32	32	32
Serial Port	1	1	1
Interrupt sources	6	8	6

8.3.7 8751 Microcontroller

This 8751 chip has only 4 Kbytes of on-chip UV-EPROM. Using this chip for development requires access to a PROM burner, as well as a UV-EPROM eraser to erase the contents of UV-EPROM inside the 8751 chip before you can program it again. Because the on-chip ROM for the 8751 is UV-EPROM, it takes around 20 minutes to erase the 8751 before it can be programmed again. This has led many manufacturers to introduce flash and NV-RAM versions of the 8051, as we will discuss next. There are also various speed versions of the 8751 available from different manufacturers.

8.3.8 DS89C4xO from Dallas Semiconductor (Maxim)

Many popular 8051 chips have on-chip ROM in the form of flash memory. The AT89C51 from Atmel Corp. is one example of an 8051 with flash ROM. This is ideal for fast development since flash memory can be erased in seconds compared to the twenty minutes or more needed for the 8751. For this reason the AT89C51 is used in place of the 8751 to eliminate the waiting time needed to erase the chip and thereby speed up the development time. Using the AT89C51 to develop a microcontroller-based system requires a ROM burner that supports flash memory; however, a ROM eraser is not needed. Notice that in flash memory you must erase the entire contents of ROM in order to program it again. This erasing of flash is done by the PROM burner itself, which is why a separate eraser is not needed. To eliminate the need for a PROM burner, Dallas Semiconductor, now part of the Maxim Corp., has a version of the 8051/52 called DS89C4xO (DS89C420/430/...) that can be programmed via the serial COM port of an IBM.

The DS89C4xO (420/430/440/450) comes with an on-chip loader, which allows the program to be loaded into the on-chip flash ROM while it is in the system. This can be done via the serial COM port of an IBM PC. This in-system program loading of the DS89C4xO via a PC serial COM port makes it an ideal home development system. Dallas Semiconductor also has an NV-RAM version of the 8051 called DS5000. The advantage of NV-RAM is the ability to change the ROM contents one byte at a time. The DS5000 also comes with a loader, allowing it to be programmed via the PC's COM port. From Table 8.3, notice that the DS89C4xO is a really an 8052 chip since it has 256 bytes of RAM and 3 timers.

Table 8.3 Versions of 8051/52/Microcontroller from Dallas Semiconductor (Maxim)

Part Number	ROM	RAM	I/O Pins	Timers	Interrupts	VCC
DS89C420/30	16 K(flash)	256	32	3	6	5V
DS89C440	32 K(flash)	256	32	3	6	5V
DS89C450	64 K(flash)	256	32	3	6	5V
DS5000	8 K(NVRAM)	128	32	3	6	5V
DS80C320	0 K	256	32	3	6	5V
DS87520	16 K(UVRAM)	256	32	3	6	5V

8.3.9 OTP Version of the 8051

There are also OTP (one-time-programmable) versions of the 8051 available from different sources. Flash and NV-RAM versions are typically used for product development. When a product is designed and absolutely finalized, the OTP version of the 8051 is used for mass production since it is much cheaper in terms of price per unit.

8.3.10 8051 Family from Philips

Another major producer of the 8051 family is Philips Corporation. Indeed, they have one of the largest selections of 8051 microcontrollers. Many of their products include features such as A-to-D converters, D-to-A converters, extended I/O, and both OTP and flash.

8.4 APPLICATIONS OF MICROCONTROLLERS

There are many benefits of using a microcontroller. Because the entire standard features of a microcomputer embedded on a single chip with its performance equal to that of a multiple chip devices.

There are many applications using microcontroller, in different fields, such as:

1. Hand-held Instruments
   - Small pagers
   - Electronic plane meter
   - Digital level meter
   - IC Tester
2. Peripheral devices
   - Keyboard controller
   - Modem
   - Printer buffer
   - Color plotter
3. Stand-alone devices
   - Color copier
   - Typewriter
   - Cable TV terminal
   - Fish finder
   - Lawn sprinkle
   - Charge card phones
4. Instrument subfunctions
   - Front panel handler for a digitalizing oscilloscope
   - Touch sensitive liquid crystal display
   - High performance multimeter
   - Interface option for a microwave counter
   - Modular spectrum analyzer
5. Automotive applications
   - Engine control modules
   - Antilock braking system
   - Dynamic ride control
   - Electronic instrument panels
6. Other applications
   - CRT displays
   - Navigation systems
   - Transmission controllers
   - Multiplexed wiring systems
   - Cellular telephones
   - Electronic entry and security systems
   - Electronic power storing

8.5 8051 MICROCONTROLLER ARCHITECTURE

There are different types of microcontroller architecture designed for embedded system development. These are:

1. **RISC:** RISC stands for Reduced Instruction Set Computer. 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. Therefore, fewer, simpler and faster instructions would be better, than the large, complex and slower CISC instructions. However, more instructions are needed to accomplish a task. Atmel's AVR microcontroller based on RISC architecture.
2. **CISC:** CISC stands for Complex Instruction Set Computer. Most PCs use CPU based on this architecture. For instance, Intel and AMD CPUs are based on CISC architectures. Typically CISC chips have a large amount of different and complex instructions. In common CISC chips are relatively slow (compared to RISC chips) per instruction, but use little (less than RISC) instructions. MCS-51 family microcontrollers are based on CISC architecture.

Von-Neumann Architecture: Microcontrollers based on the Von-Neumann architecture have a single data bus that is used to fetch both instructions and data. Program instructions and data are stored in a common main memory. When such a controller addresses main memory, it first fetches an instruction and then it fetches the data to support the instruction. The two separate fetches slow the controller's operation.

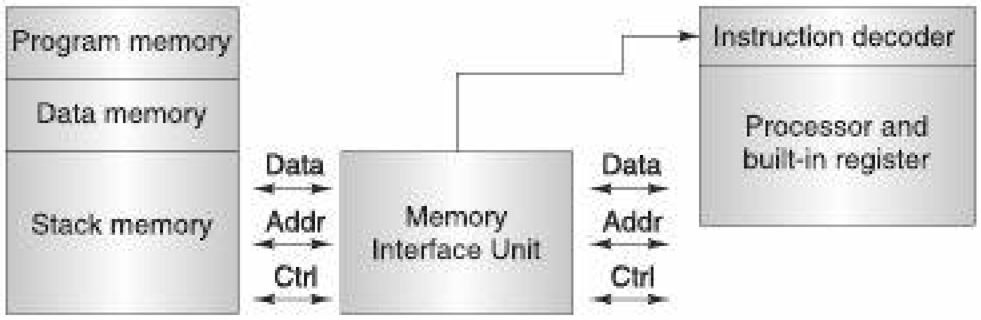


Fig. 8.3 Von-Neumann Architecture.

Harvard Architecture: Harvard architecture computers have separate memory areas for program instructions and data. There are two or more internal data buses, which allow simultaneous access to both instructions and data. The CPU fetches program instructions on the program memory bus. Since Harvard machines have an explicit memory space for data, using program memory for data storage is trickier. For example, a data value declared as a C constant must be stored in ROM as a constant value. Some chips have special instructions allowing the retrieval of information from program memory space. These instructions are always more complex or expensive than the equivalent instructions for fetching data from data memory. Others simply do not have them; data must be loaded by the side effect of a return instruction, for instance.

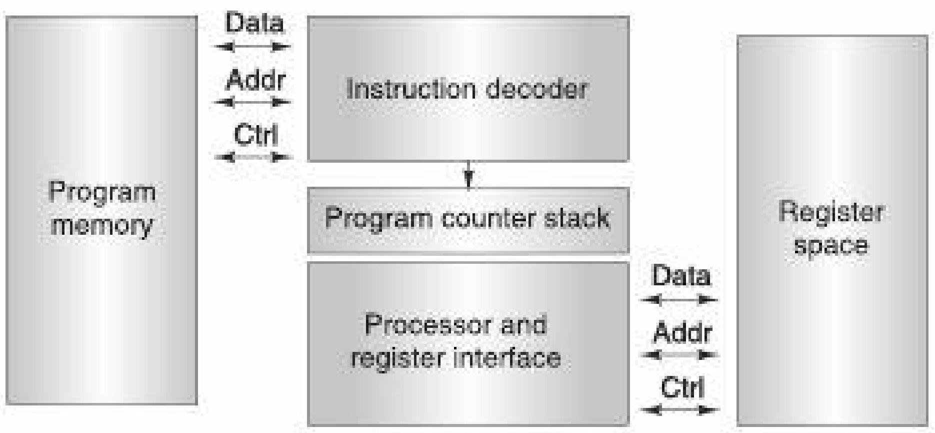


Fig. 8.4 Harvard Architecture.

8.5.1 8051 Architecture

The Intel 8051 is a Harvard architecture single chip microcontroller (μC) which was developed by Intel in 1980 for use in embedded systems. It was extremely popular in the 1980s and early 1990s, but today it has largely been superseded by a vast range of enhanced devices with 8051-compatible processor cores that are manufactured by more than 20 independent manufacturers including Atmel, Maxim IC (via its Dallas Semiconductor subsidiary), NXP (formerly Philips Semiconductor), Winbond, Silicon Laboratories, Texas Instruments and Cypress Semiconductor. Intel's official designation for the 8051 family of μCs is MCS 51.

Intel's original 8051 family was developed using NMOS technology, but later versions, identified by a letter "C" in their name, e.g., 80C51, used CMOS technology.

- **A Register (Accumulator):** A register is a general-purpose register used for storing intermediate results obtained during operation. Prior to executing an instruction upon any number or operand it is necessary to store it in the accumulator first. All results obtained from arithmetical operations performed by the ALU are stored in the accumulator. Data to be moved from one register to another must go through the accumulator. In other words, the A register is the most commonly used register and it is impossible to imagine a microcontroller without it. More than half instructions used by the 8051 microcontroller use somehow the accumulator.

ACC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

- **B Register:** Multiplication and division can be performed only upon numbers stored in the A and B registers. All other instructions in the program can use this register as a spare accumulator (A).
- **Flags and Program Status Word (PSW):** Flags are 1-bit registers provided to store the result of certain program instruction. In order that the flags may be conveniently addressed, they are grouped inside the program status word (PSW) and program control (PCON) registers. The 8051 has four math flags that respond automatically to outcomes of the mathematical operations and three general-purpose user flags that can be set to 1 or cleared to 0 by the programmer. The math flag includes carry (C), auxiliary carry (AC), overflow (OV) and parity (P). User flags named F0, GF0 and GF1.

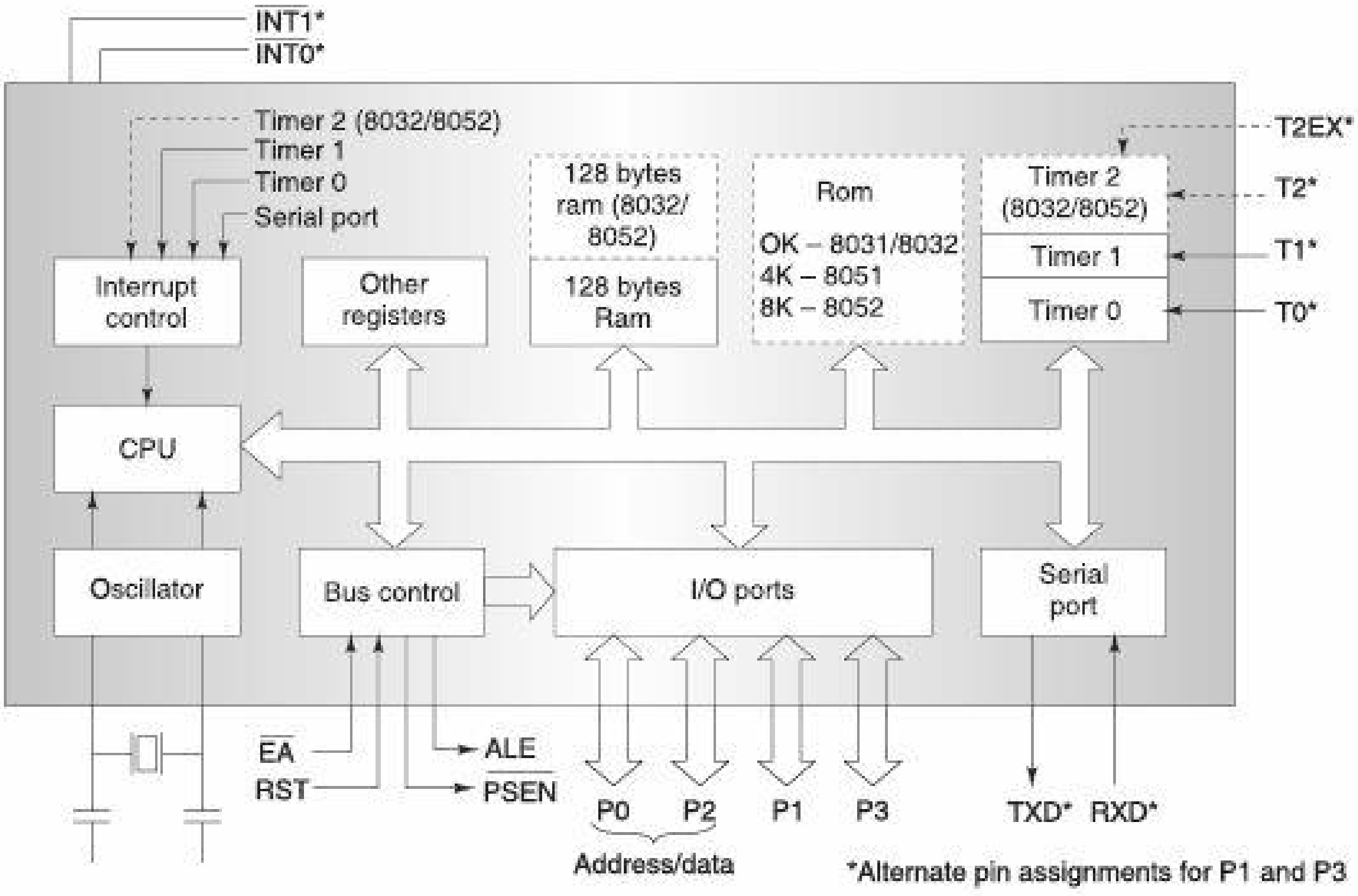


Fig. 8.5 8051 Architecture.

CY	AC	F0	RS1	RS0	OV	–	P

The program status word contains the math flags, user flag F0, and the register select bits this identify which of the four general-purpose register banks is currently in use by the programmer. The remaining two user flags, GF0 and GF1, are stored in PCON.

- **CY, the carry flag:** This flag is set whenever there is a carry out from the D7 bit. This flag bit is affected after an 8-bit addition or subtraction. It can also be set to 1 or 0 directly by an instruction "SETB C" and "CLR C", where "SETB C" stands for set bit carry and "CLR C" for clear carry.
- **AC, the auxiliary carry flag:** If there is a carry from D3 to D4 during an ADD or SUB operation, this bit is set otherwise it is cleared. This flag is used by instruction that performs BCD arithmetic.
- **P, the parity flag:** The parity flag reflects the number of 1s in the accumulator only. If the A register contains an odd number of 1s, then P = 1 and P = 0 for even.
- **OV, overflow flag:** This flag is set whenever the result of a signed number operation is too large, causing the higher order bit to overflow into the sign bit. This flag is used to detect the errors in signed arithmetic operation.
- **The stack and the stack pointer (SP):** The stack refers to an area of internal RAM is used in conjunction with certain opcodes to store and retrieve data quickly. It is an

8-bit register. The address held in SP register is the location in internal RAM where the last byte of data was stored by the stack operation. When data is to be stored on SP, it increments before storing data on the stack so that the stack grows up as data is stored, as the data is retrieved from the stack, the SP decrements to point to the next available byte of stored data.

- **Register bank selection bits, RS1-RS0:** These are the register bank selection bits.

Table 8.4 PSW Register

RS1	*RS0*	*Register Bank Selected*
0	0	Register bank 0
0	1	Register bank 1
1	0	Register bank 2
1	1	Register bank 3

- **Input/output pins, ports and circuits:** One of the major features of microcontroller is its versatility built in the input/output circuits that connects the 8051 to the external world. The 8051 DIP has 40 pins and the success of the design in the marketplace was determined by the flexibility built into the use of these pins. For this reason, 24 of these pins may each be used for one of two entirely different functions, yielding total pin configuration of 64. The function of pins performs at any given instant depends, first, what is physically connected to it and then, upon what software commands are used to program the pin. There are 4 8-bit ports: P0, P1, P2, and P3.
- **Oscillator and clock circuits:** The heart of the 8051 is the circuitry is that generates the clock pulses by which all internal operations are synchronized. Pins XTAL1 and XTAL2 are provided for providing a resonant network to form an oscillator. The crystal frequency is basic internal clock of the 8051 microcontroller typically 1 MHz to 16 MHz frequency is required for the operation.

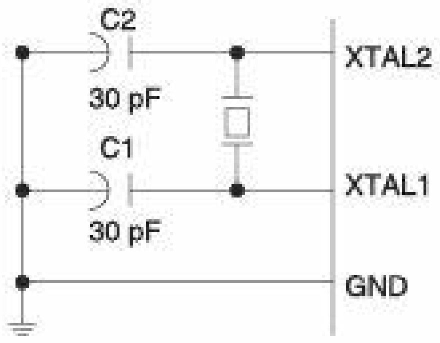


Fig. 8.6 Crystal Circuit.

- **Data Pointer (DTPR):** DPTR register is not a true one because it doesn't physically exist. It consists of two separate registers: DPH (Data Pointer High) and (Data Pointer Low). For this reason, it may be treated as a 16-bit register or as two independent 8-bit registers. Their 16 bits are primarily used for external memory addressing. Besides, the DPTR Register is usually used for storing data and intermediate results.

- **ALU:** The arithmetic and logic unit performs 8-bit arithmetic and logical operations over the operands held by the temporary registers TMPI and TMP2. Users cannot access these temporary registers.
- **Instruction Register:** This register decodes the opcode of an instruction to be executed and gives information to the timing and control unit to generate necessary signals for the execution of the instruction.

8.6 MEMORY ORGANISATION OF 8051

The 8051 has two types of memory and these are Program Memory and Data Memory. Program Memory (ROM) is used to permanently save the program being executed, while Data Memory (RAM) is used for temporarily storing data and intermediate results created and used during the operation of the microcontroller. All 8051 microcontrollers have a 16-bit addressing bus and are capable of addressing 64 KB memory.

On-Chip memory

The 8051 includes a certain amount of on-chip memory. On-chip memory is really one of two types: Internal RAM and Special Function Register (SFR) memory. On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of several types.

The layout of the 8051's internal memory is presented in the following memory map:

1. **Special Function Register (SFR) Memory:** Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. For example, four SFRs permit access to the 8051's 32 input/output lines. Another SFR allows a program to read or write to the 8051's serial port. SFR is a part of Internal Memory. This is not the case.

   When using this method of memory access (it's called direct address), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address; any instruction with an address of 80h through FFh refers to an SFR control register.

Table 8.5 SFR Address

SFR	Direct Address (HEX)	Bit Address (HEX)
A	0E0	0E0–0E7
B	0F0	0F0–0F7
IE	0A8	0A8–0AF
IP	0B8	0B8–0BF
P0	80	80–87
P1	90	90–97
P2	0A0	0A0–0A7
P3	0B0	0B0–0B7
PSW	0D0	0D0–0D7
TCON	88	88–8F
SCON	98	98–9F

Registers: The 8051 contains 34 general-purpose registers. Two of these A and B comprise the mathematical core of the 8051 CPU. The other 32 are arranged in B0-B3 bank, each containing 8 registers, each named, R0-R7.

2. **Internal RAM:** A general-purpose RAM area above bit area, 30h to 7Fh, addressable as bytes. A bit addressable area of 16 bytes occupies RAM byte addresses 20h to 2Fh, forming a total of 128 bits addressable. Addressable bits are useful when the program need only remember a binary unit. Internal RAM is in short supply as it is. Bits RS0 and RS1 in the PSW determine which bank of the registers is currently in use at any time when the program is running.

   Internal ROM- The 8051 is organized so that data memory and program memory can be in two entirely different physical memory entities. The PC is ordinarily used to address the program code bytes from addresses, 0000h to FFFFh, program addresses higher than 0FFFh, which exceed the internal ROM capacity, will cause the 8051 to automatically fetch code bytes from external code memory. Codes bytes can also be fetched exclusively from an external memory, addresses 0000h to FFFFh, by connecting the external access pin (EA)', pin no. 31 on the DIP to ground. The PC doesn't care where the code is; the circuit designer decides whether the code is found totally in internal ROM, in external ROM or in a combination of internal and external ROM.

8.7 PIN CONFIGURATION

The 8051 is packaged in a 40-pin DIP. The Figure 8.7 shows the pin diagram of 8051. It is important to note that many pins of 8051 are used for more than one function. The alternative functions of pins are shown in bold letters.

Input/output Ports (I/O Ports): All 8051 microcontrollers have 4 I/O ports each comprising 8 bits which can be configured as inputs or outputs. Accordingly, in total of 32 input/output pins enabling the microcontroller to be connected to peripheral devices are available for use. Pin configuration, i.e., whether it is to be configured as an input (1) or an output (0), depends on its logic state. In order to configure a microcontroller pin as an input, it is necessary to apply logic zero (0) to appropriate I/O port bit. In this case, voltage level on appropriate pin will be 0. Similarly, in order to configure a microcontroller pin as an input, it is necessary to apply a logic one (1) to appropriate port. In this case, voltage level on appropriate pin will be 5 V (as is the case with any TTL input). This may seem confusing. It all becomes clear after studying simple electronic circuits connected to an I/O pin.

Port 0: The P0 port is characterized by two functions. If external memory is used then the lower address byte (addresses A0-A7) is applied on it. Otherwise, all bits of this port are configured as inputs/outputs.

The other function is expressed when it is configured as an output. Unlike other ports consisting of pins with built-in pull-up resistor connected by its end to 5 V power supply; pins of this port 0 have this resistor left out. If any pin of this port is configured as an input then it acts as if it "floats". Such an input has unlimited input resistance and in determined potential.

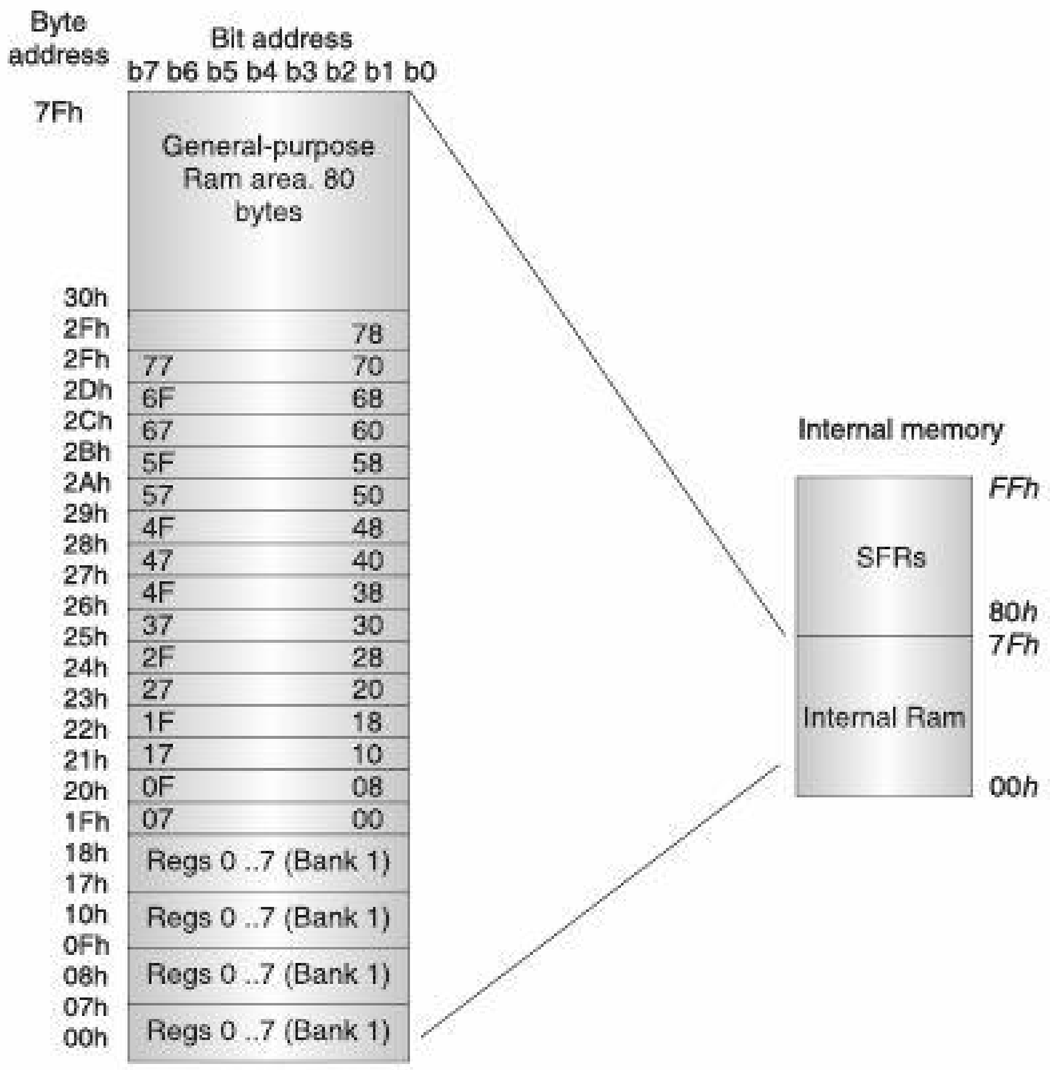


Fig. 8.7 Memory Map Organization.

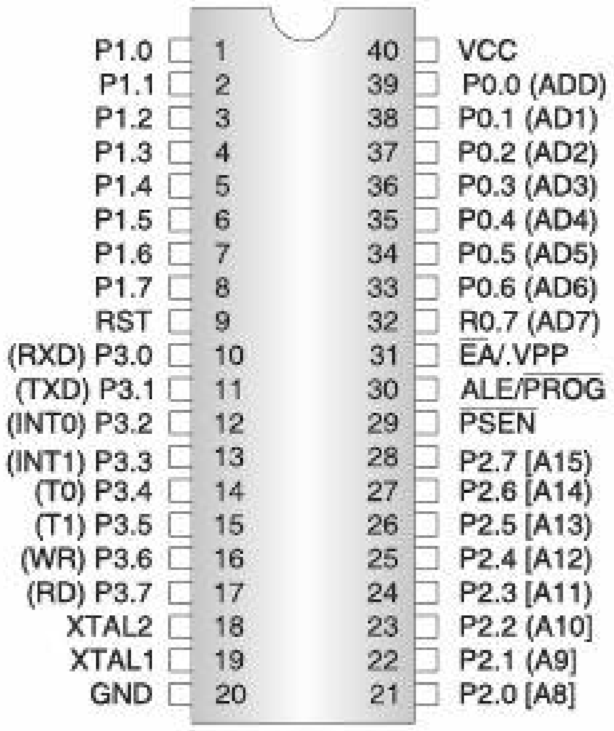


Fig. 8.8 Pin Configuration of 8051 Microcontroller.

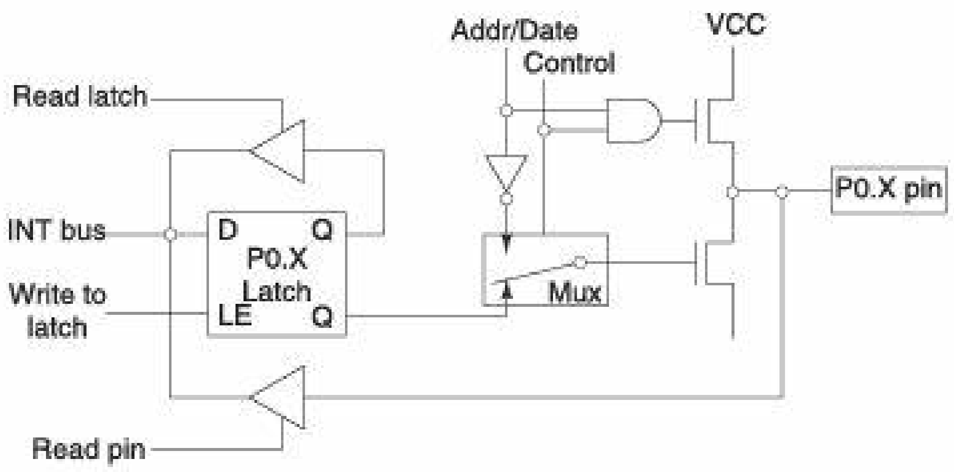


Fig. 8.9 Internal Diagram of Port 0.

When the pin is configured as an output, it acts as an "open drain". By applying logic 0 to a port bit, the appropriate pin will be connected to ground (0 V). By applying logic 1, the external output will keep on "floating". In order to apply logic 1 (5 V) on this output pin, it is necessary to built in an external pull-up resistor.

Port 1: P1 is a true I/O port, because it doesn't have any alternative functions as is the case with P0, but can be configured as general I/O only. It has a pull-up resistor built-in and is completely compatible with TTL circuits. Port 1 also receives the low-order address bytes during Flash programming and verification.

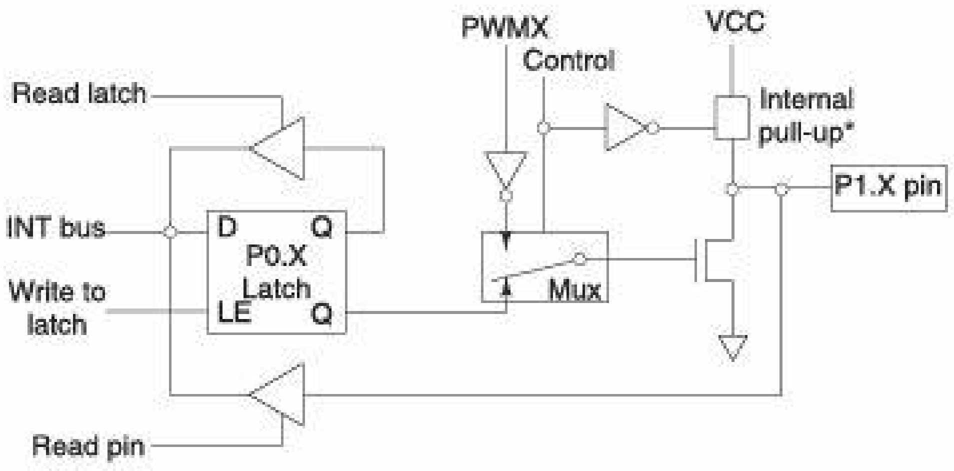


Fig. 8.10 Internal Diagram of Port 1.

Port 2: Port 2 is an 8-bit bidirectional I/O port with internal pull-ups. The Port 2 output buffers can sink/source four TTL inputs. When 1s are written to Port 2 pins, they are pulled high by the internal pull-ups and can be used as inputs. As inputs, Port 2 pins that are externally being pulled low will source current (IIL) because of the internal pull-ups. Port 2 emits the high-order address byte during fetches from external program memory and during accesses to external data memory.

Port 3: Port 3 is an 8-bit bidirectional I/O port with internal pull-ups. The Port 3 output buffers can sink/source four TTL inputs. When 1s are written to Port 3 pins, they are pulled high by

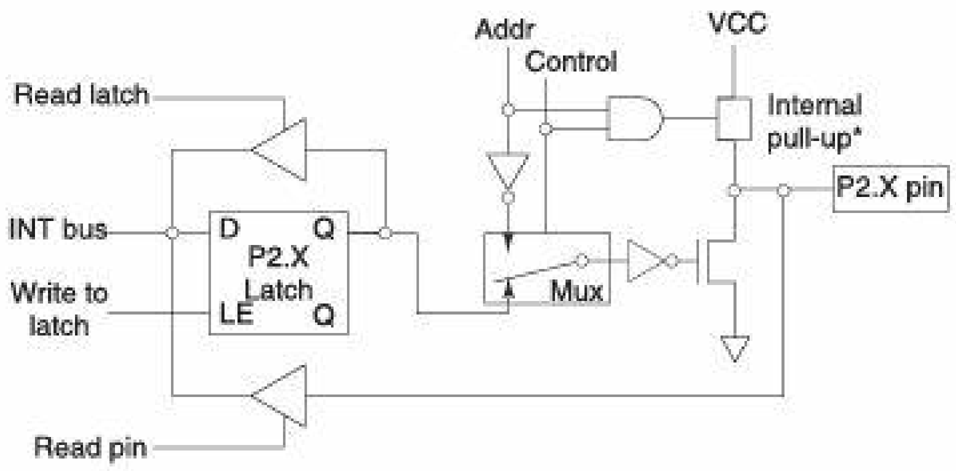


Fig. 8.11 Internal Diagram of Port 2.

the internal pull-ups and can be used as inputs. As inputs, Port 3 pins that are externally being pulled low will source current (IIL) because of the pull-ups. Port 3 also serves the functions of various special features of the AT89S8252, as shown in the following table. Port 3 also receives some control signals for Flash programming and verification.

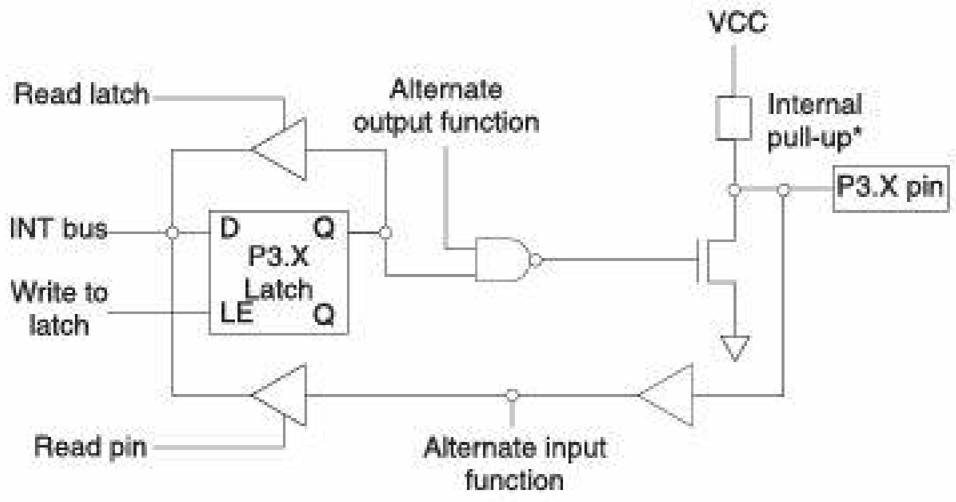


Fig. 8.12 Internal Diagram of Port 3.

Table 8.6 Alternate Function of Port 3

Port pin	Alternate Function
P3.0	RXD (Serial input port)
P3.1	TXD (Serial output port)
P3.2	INT0 (external interrupt 0)
P3.3	INT1 (external interrupt 1)
P3.4	T0 (timer 1 external input)
P3.5	T1 (timer 1 external input)
P3.6	WR (External data memory write strobe)
P3.7	RD (External data memory read strobe)

RST: Reset input. A high on this pin for two machine cycles while the oscillator is running resets the device.

ALE/PROG: Address Latch Enable is an output pulse for latching the low byte of the address during accesses to external memory. This pin is also the program pulse input (PROG) during Flash programming. In normal operation, ALE is emitted at a constant rate of 1/6 the oscillator frequency and may be used for external timing or clocking purposes. Note, however, that one ALE pulse is skipped during each access to external data memory. If desired, ALE operation can be disabled by setting bit 0 of SFR location 8EH. With the bit set, ALE is active only during a MOVX or MOVC instruction. Otherwise, the pin is weakly pulled high. Setting the ALE-disable bit has no effect if the microcontroller is in external execution mode.

PSEN: Program Store Enable is the read strobe to external program memory. When the AT89S8252 is executing code from external program memory, PSEN is activated twice each machine cycle, except that two PSEN activations are skipped during each access to external data memory.

EA/VPP: External Access Enable. EA must be strapped to GND in order to enable the device to fetch code from external program memory locations starting at 0000H up to FFFFH. Note, however, that if lock bit 1 is programmed, EA will be internally latched on reset. EA should be strapped to VCC for internal program executions. This pin also receives the 12-volt programming enable voltage (VPP) during Flash programming when 12-volt programming is selected.

XTAL1: Input to the inverting oscillator amplifier and input to the internal clock operating circuit.

XTAL2: Output from the inverting oscillator amplifier.

8.8 TIMERS AND COUNTERS

Many microcontrollers require the application of counting of external events, such as the frequency of a pulse train, or the generation of the precise internal time delays between computer actions. Two 16-bit up counters, T0 and T1, are provided for the general use of the programmer. Each programmer may programme to count internal clock pulses, acting as a timer, or programme to count external pulses as a counter.

The 8051 comes equipped with two timers, both of which may be controlled, set, read, and configured individually. The 8051 timers have three general functions: 1) keeping time and/or calculating the amount of time between events; 2) counting the events themselves, or 3) generating baud rates for the serial port. How does a timer count? The answer to this question is very simple: A timer always counts up. It doesn't matter whether the timer is being used as a timer, a counter, or a baud rate generator: A timer is always incremented by the microcontroller. When a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 every machine cycle. As you will recall from the previous chapter, a single machine cycle consists of 12 crystal pulses. Thus, a running timer will be incremented:

$$11{,}059{,}000/12 = 921{,}583$$

921,583 times per second. Unlike instructions–some of which require 1 machine cycle, others 2, and others 4–the timers are consistent: They will always be incremented once per machine cycle. Thus, if a timer has counted from 0 to 50,000 you may calculate:

$$50{,}000/921{,}583 = .0542$$

.0542 seconds have passed. In plain English, about half of a tenth of a second, or one-twentieth of a second. We've discussed how a timer can be used for the obvious purpose of keeping track of time. However, the 8051 also allows us to use the timers to count events? Let's say you had a sensor placed across a road that would send a pulse every time a car passed over it. This could be used to determine the volume of traffic on the road. We could attach this sensor to one of the 8051's I/O lines and constantly monitor it, detecting when it pulsed high and then incrementing our counter when it went back to a low state.

8.8.1 Basic Registers of Timer

Both Timer 0 and Timer 1 are 16-bit wide. Since 8051 has an 8-bit architecture, each 16 bit timer is accessed as two separate registers of low byte and high byte.

- **Timer 0 Register:** The 16-bit register of Timer 0 is accessed as low byte and high byte. The low byte register is called TL0 (Timer 0 low byte) and the high byte register is referred to as TH0 (Timer 0 high byte). These registers can be accessed as any other registers like A, R0, R1, R2, etc.
- **Timer 1 Register:** Timer 1 is also 16-bits, and its 16-bit register is split into bytes, referred to as TL1 (Timer 1 low byte) and TH1 (Timer 1 high byte). These registers are accessible as of mode 0.
- **TMOD (Timer Mode Register):** Both timers (T0 and T1) use the same register, TMOD, to set the various timer operation modes. TMOD is an 8-bit register in which the lower 4 bits are set aside for Timer 0 and upper 4 bits for Timer 1. In each case lower 2 bytes are to set the timer mode and upper 2 bytes are used to specify the operation.

GATE	C/T	M1	M0	GATE	C/T	M1	M0

←————————TIMER 1————————→ ←————————TIMER 0————————→

- **GATE:** Gating control—When set timer/counter is enable only while the INTx pin is high and the TRx control pin is set. When cleared, the timer is enabled whenever TRx control bit is set.
- **C/T:** Cleared for the timer operation (input from internal system clock). Set for counter operation (input from Tx input pin).
- **M0, M1:** M0 and M1 are control bits and used for selection of operation of modes.

Table 8.7 Selection of Operating Modes

M1	*M0*	*Mode*	*Operating mode*
0	0	0	13-bit timer mode- 8-bit timer/counter THx with TLx as 5-bit prescaler.
0	1	1	16-bit timer mode- 16-bit timer/counter THx and TLx are cascaded, there is no prescaler.
1	0	2	8-bit auto reload- 8-bit auto reload timer/counter, THx holds a value which is to be reloaded TLx each time it overflows.
1	1	3	Split timer mode

1. Timer control register (TCON)

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

- **TF2 (bit 7):** Timer 2 overflow flag set by a Timer 2 overflows and must be cleared by software. TF2 will not be set when either RCLK = 1 or TCLK = 1.
- **EXF2 (bit 6):** Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software. EXF2 does not cause an interrupt in up/down counter mode (DCEN = 1).
- **RCLK (bit 5):** Receive clock enable. When set, causes the serial port to use Timer 2 overflow pulses for it's receive clock in serial port Modes 1 and 3. RCLK = 0 causes Timer 1 overflow to be used for the receive clock.
- **TCLK (bit 4):** Transmit clock enable. When set, causes the serial port to use Timer 2 overflow pulses for it's transmit clock in serial port Modes 1 and 3. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock.
- **EXEN2 (bit 3):** Timer 2 external enable. When set, allows a capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
- **TR2 (bit 2):** Start/Stop control for Timer 2. TR2 = 1 starts the timer.
- **C/T2 (bit 1):** Timer or counter select for Timer 2. C/T2 = 0 for timer function. C/ T2 = 1 for external event counter (falling edge triggered).
- **CP/RL2 (bit 0):** Capture/Reload select. CP/RL2 = 1 causes captures to occur on negative transitions at T2EX if EXEN2 = 1. CP/RL2 = 0 causes automatic reloads to occur when Timer 2 overflows or negative transitions occur at T2EX when EXEN2 = 1. When either RCLK or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.

8.9 TIMER COUNTER INTERRUPTS

The counters have been included on the chip to relieve the processor of timing and counting chores. When the program wishes to count a certain number of internal pulses or external events, a number is placed in the counter. The counter increment from the initial number to the maximum and then rolls over to zero on the final pulse and sets a timer flag. The flag may be used to interrupt the program.

8.10 TIMING

If a counter is programmed to be a timer, it will count the internal clock frequency of the 8051 oscillator divided by 12d. The resultant timer clock is gated to the timer by means of the circuit. In order for oscillator clock pulses to reach the timer, the C/ (T)' bit in the TMOD register must be set to 0 (timer operation). Bit TRX in the TCON register must be set to 1 (timer run) and the gate bit in the TMOD register must be 0, or external pin (INTX)' must be 1. In other words, the counter is configured as a timer, then the timer pulses are gated to the counter by the run bit and the gate bit or the external input bits (INTX)'.

8.10.1 Timer Mode of Operation

The timer may operate on any of the four modes that are determined by the mode bits, M0 and M1, and the TMOD register.

8.10.1.1 Timer Mode 0

This mode is a 13-bit timer. Setting timer X mode bits to 00b in the TMOD register results in using the TXH as an 8-bit counter and TLX as a 5-bit counter, the pulse input is divided by 32d in TL so that TH counts the original oscillator frequency reduced by a total 384d. The timer flag is set whenever THX go from FFh to 00h, or in 0.0164 seconds for a 6 MHz crystal if THX start at 00h.

8.10.1.2 Timer Mode 1

This mode is a 16-bit timer. Mode 1 is same to mode 0 except TLX is configured as a full 8-bit counter when mode bits are set to 01 b in TMOD. The timer flag would be set in 0.1311 seconds using a 6 MHz crystal.

The following are the characteristics and operations of mode1:

1. It is a 16-bit timer; therefore, it allows value of 0000 to FFFFH to be loaded into the timer's register TL and TH.
2. After TH and TL are loaded with a 16-bit initial value, the timer must be started. This is done by SETB TR0 for timer 0 and SETB TR1 for timer 1.
3. After the timer is started, it starts to count up. It counts up until it reaches its limit of FFFFH. When it rolls over from FFFFH to 0000, it sets high a flag bit called TF (timer flag).
   - Each timer has its own timer flag: TF0 for timer 0 and TF1 for timer 1.
   - This timer flag can be monitored.

   When this timer flag is raised, one option would be to stop the timer with the instructions CLR TR0 or CLR TR1, for timer 0 and timer 1, respectively.
4. After the timer reaches its limit and rolls over, in order to repeat the process. TH and TL must be reloaded with the original value, and TF must be reloaded to 0.

Steps to generate a time delay:

1. Load the TMOD value register showing either timer 0 or timer 1 is selected or which timer mode (0 or 1) is selected.
2. Load registers TL and TH with initial count value.
3. Start the timer.
4. Keep monitoring the timer flag (TF) with the JNB TFx, target instruction to see if it is raised. Get out of the loop when TF becomes high.
5. Stop the timer.
6. Clear the TF flag for the next round.
7. Go back to step 2 to load TH and TL again.

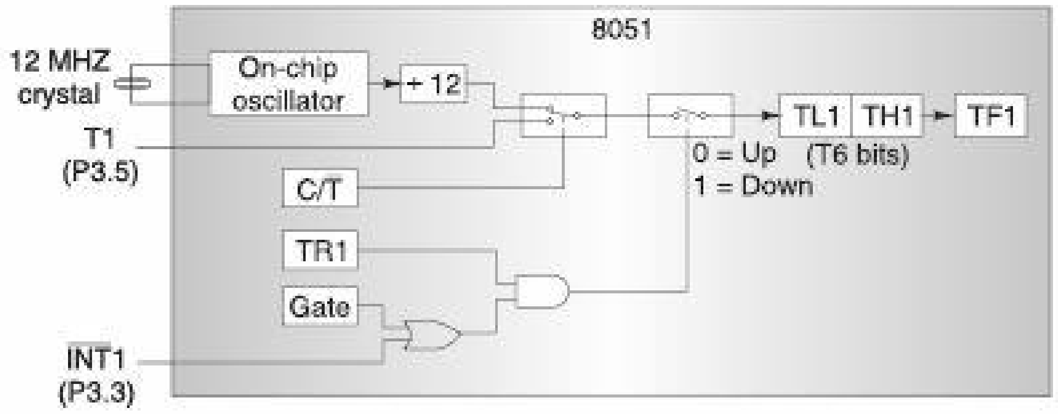


Fig. 8.13 Timer 1 Operating in Mode 1.

8.10.1.3 Timer Mode 2

This mode is an 8-bit timer. Setting the timer mode to 10b in TMOD configures the timer to use only the TLX counter as an 8-bit counter. THX are used to hold a value that is loaded into TLX every time TLX overflows from FFh to 00h. The timer flag is set when TLX flows. This mode exhibits an auto-reload feature: TLX will count up from the number in THX, overflow, and be initialized again with the contents of THX. For example, placing 9Ch in THX will result in a delay of exactly 0.0002 seconds before the overflow flag is set if a 6 MHz crystal is used.

The following are the characteristics and operation of mode 2:

1. It is an 8-bit timer, therefore, it allows only value of 00 to FFh to be loaded into the timer's register TH.
2. After TH is loaded with the 8-bit value, the 8051 gives a copy of it to TL. Then the timer must be started. This is done by the instruction SETB TR0 for timer 0 and SETB TR1 for timer 1.
3. After the timer is started, it starts to count up by incrementing the TL registers. I count up until it reaches its limit of FFH. When it rolls over from FFH to 00, it sets high the TF (timer flag).
4. When the TL registers rolls from FFH to 0 and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register.

Steps to generate a time delay

1. Load the TMOD value register indicating which timer is to be used, and the timer mode (mode 2) is selected.
2. Load the TH registers with the initial count value.
3. Start timer.
4. Keep monitoring the timer flag with the JNB TFx, target instruction to see whether it is raised.
5. Clear the TF flag.
6. Go back to step 4, since mode 2 is auto-reload mode.

8.10.1.4 Timer Mode 3

Timer 0 and 1 may be programmed to be in mode 0, 1 or 2 independently of a similar mode for the other time. This is not true for the mode 3; the timers do not operate independently if mode 3 is chosen for timer 0. Placing timer 1 in mode 3 causes it to stop counting; the control bit TR1 and the timer 1 flag 1 TF1 are then used by timer 0. Timer 0 in mode 3 becomes two completely separate 8-bit counters. TL0 is controlled by gate arrangement and sets timer flag TF0 whenever it flows from FFh to 00h. TH0 receives the timer clock under the control of T1 only and sets the TF1 flag when it overflows. Timer 1 may still be used in mode 0, 1 or 2 while timer 0 is in mode 3 with one important exception: No interrupts will be generated by timer 1 while timer 0 is using the TF1 overflow flag. Switching timer 1 to mode 3 will stop it.

8.11 COUNTING

The only difference between timing and counting is the source of the clock pulses to the counters. When used as a timer, the clock pulses are sourced from the oscillator through the divide-by-12-circuit. When used as counter pinT0 (P3.4) supplies pulses to counter 0 and pin T1 (P3.5) to counter 1. The C/ (T)' bit mode must be set to 1 to enable pulses from the TX pin to reach the control circuit. The input pulse on TX is sampled P2 of state 5 every machine cycle. A change on the input from high to low between samples will increment the counter. Each high and low state of the input pulse must thus be held constant for at least machine cycle to ensure reliable counting. Since this takes 24 pulses, the maximum input frequency that can be accurately counted is the oscillator frequency divided by 24.

The C/T bit in the TMOD registers decides the source of the clock for the timer. When C/T = 1, the timer is used as a counter and gets its pulses from outside the 8051.

The counter counts up as pulses are fed from pins 14 and 15, these pins are called T0 (timer 0 input) and T1 (timer 1 input).

8.12 SERIAL COMMUNICATION

8.12.1 Serial Data Input/Output

Computers must be able to communicate with other computers in modern multiprocessor distributed system. The 8051 has a serial data communication circuit that uses register SBUF to hold data. Register SCON controls data communication, register PCON controls data rates and pins RXD (P3.0) and TXD (P3.1) connect to the serial data network. The 8051 has two pins that are used specifically for transferring and receiving data serially. These two are RXD and TXD and are a part of the port 3 group (P3.0 and P3.1). Pin 11(P3.1) is assigned to TXD and pin 10(P3.0) is designated to RXD. SBUF is physically two registers. One is write only and is used to hold the data to be transmitted out of the 8051 via TXD. The other is read only and holds received data from external source via RXD. There are four programmable modes for serial data communication that are chosen by setting the SMX bits in SCON. Baud rates are determined by this mode.

8.12.2 Basic Registers for Serial Communication

1. **SBUF:** SBUF is an 8-bit register used solely for serial communication.
   - For a byte data to be transferred via the TxD line, it must be placed in the SBUF register.
   - SBUF holds the byte of data when it is received by 8051 RxD line.
2. **SCON:** SCON is an 8-bit register used to program the start bit, stop bit and data bits of data framing, among other things.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

   - **SM0, SM1(bit 7,6):** These are serial port mode control bits. Set/cleared by program to select mode.

Table 8.8 Serial Communication Mode Selection

SM0	*SM1*	*Mode*	*Description*
0	0	0	Shift register, baud = f/12
0	1	1	8-bit UART, baud = variable
1	0	2	9-bit UART,baud = f/32 or f/64
1	1	3	9-bit UART, baud = variable

   - **SM2 (bit 5):** Multiprocessor communications bit. Set/cleared by program to enable multiprocessor communications in modes 2 and 3. When set to 1 an interrupt is generated if bit 9 of the received data is 1, no interrupt is generated if bit 9 is 0. If set to 1 for mode 1, no interrupt will be generated unless a valid stop is received. Clear to 0 if mode 0 is in use.
   - **REN (bit 4):** (receive enable) It is a bit- addressable register. When it is high, it allows 8051 to receive data on RxD pin. If it is low, the receiver is disabled.
   - **TB8 (bit 3):** Transmitted bit 8. Set/cleared by program in modes 2 and 3.
   - **RB8 (bit 2):** Received bit 8. Bit 8 of received data in modes 2 and 3, stop bit in mode 1. Not used in mode 0.
   - **TI (bit 1):** (transmit enable) When 8051 finishes the transfer of 8-bit character, it raises TI flag to indicate that it is ready to transfer another byte. TI bit is raised at the beginning of the stop bit.
   - **RI (bit 0):** (receive enable) When 8051 receives data serially via RxD, it gets rid of the stop and start bits and places the byte in SBUF register. It raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost. RI is raised halfway through the stop bit.
3. **PCON:** PCON register is an 8-bit register. When 8051 is powered up, SMOD is zero. We can set it high by software and thereby double the baud rate.

SMOD	–	–	–	GF1	GF0	PD	IDL

- **SMOD (bit 7):** Serial baud rate modify bit. Set to 1 by program to double rate using timer 1 for modes 1, 2 and 3. Cleared by program to use timer 1 baud rate.
- **Bit (6-4):** Reserved.
- **GF1-GF0 (bit 3-2):** General-purpose user flag bit 0-1. Set/cleared by program.
- **PD (bit 1):** Power down bit. Set to 1 by program to enter power down configuration for CHMOS processors.
- **IDL (bit 0):** Idle mode bit. Set to 1 by program to enter idle mode configuration for CHMOS processors. PCON is not bit addressable.

8.13 SERIAL DATA INTERRUPTS

Serial data communication is relatively a slow process, occupying many milliseconds per data byte to accomplish. In order not to tie up valuable processor time, serial data flags are included in SCON to aid in efficient data transmission and reception. Note that data transmission is under the control of the program, but reception of data is unpredictable and at random times that are beyond the control of the program. The serial data flags in SCON, T1 and R1, are set whenever a data byte is transmitted or received. These flags are ORed together to produce an interrupt to the program. The program must read these flags to determine which caused the interrupt and clear the flag. It is the responsibility of the programmer to write routines that handle the serial data flags.

8.13.1 Data Transmission

Transmission of serial data bit starts at anytime data is written to SBUF. T1 is set to 1 when the data has been transmitted and signifies that SBUF is empty and that another data byte can be sent. If the program fails to wait for T1 flag and overwrites SBUF while a previous data byte is in the process of being transmitted, the results will be unpredictable.

The steps that 8051 goes through in transmitting a character via TxD:

1. The byte character to be transmitted is written into the SBUF register.
2. The start bit is transferred.
3. The 8-bit character is transferred on bit at a time.
4. The stop bit is transferred. (It is during the transfer of the stop bit that 8051 raises the TI flag, indicating that the last character was transmitted.)
5. By monitoring the TI flag, we make sure that we are not overloading the SBUF. (If we write another byte into the SBUF before TI is raised, the untransmitted portion of the previous byte will be lost).
6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by CLR TI in order for this new byte to be transferred.

8.13.2 Data Reception

Reception of serial data will begin if the received enable bit (REN) in SCON is set to 1 for all modes. Also for mode 0 only, R1 must be cleared to1 also. Receiver interrupt flag R1 is set after data has been received in all modes. Setting 1 is the only direct program control that limits the data of unexpected data; the requirement that R1 also be 0 for mode prevents the reception of new data until the program has deal with the old data and reset R1. Receptions can begin in mode 1, 2 and 3 if R1 is set when the serial stream of bits begins, R1 must have been reset by the program before the last bit is received. Incoming data is not transferred to SBUF until the last data bit has been received so that the previous transmission can be read from SBUF while new data is being received.

In receiving bit via its RxD pin, 8051 goes through the following steps:

1. It receives the start bit. (Indicating that the next bit is the first bit of the character byte it is about to receive.)
2. The 8-bit character is received one bit at time.
3. The stop bit is received. (When receiving the stop bit 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character.)
4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register. (We copy the SBUF contents to a safe place in some other register or memory before it is lost.)
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by CLR RI in order to allow next received character byte to be placed in SBUF.

8.14 SERIAL DATA TRANSMISSION MODES

The 8051 design includes four modes of serial data transmission that enable data communication to be done in various ways. Modes are selected by the programmer by setting the mode bits SM0 and SM1 in SCON. Baud rates are fixed for mode 0 and variable, using timer and the serial baud rate modify bit (SMOD) in PCON for modes 0, 1, 2 and 3.

8.14.1 Serial Data Mode 0- Shift Register Mode

Mode 0 is meant for data communication between the computers, but as high speed data collection methods using discrete logic to achieve high data rates. Setting bits SM0 and SM1 in SCON to 00b configures in SBUF to receive or transmit 8 data bits using pin RXD for both functions. Pin TXD is connected to internal shift frequency pulse source to supply shift to the external circuits. The shift frequency, or baud rate, is fixed at 1/12 of the oscillator frequency, the same rate used by the timer when in the timer configuration. The TXD shift clock is a square that is low for machine cycle states S3-S4-S5 and high for S6-S1-S2. When transmitting, data is shifted out of RXD; the data changes on falling edge of S6P2, or one clock pulse after the rising edge of the output TXD shift clock. Received data comes in on pin RXD and should be synchronized with the shift clock produced at TXD. Data is sampled on the falling edge on S5P2 and shifted in to SBUF on the rising edge of the shift clock.

For mode 0 the baud rate = fosc/12

8.14.2 Serial Data Mode 1- Standard UART

When SM0 and SM1 are set to 01b, SBUF becomes a 10-bit full duplex receiver/ transmitter that may receive and transmit data at the same time. Transmitted data is sent as a bit, eight data bit and stop bit. Interrupt flag T1 is set once all ten bits have been sent. Each bit interval is the inverse of the baud rate frequency and each bit maintained at high or low over that control. Received data is obtained in the same manner; reception is triggered by the falling edge of the start bit and continues if the stop bit is true (0 level) halfway through the start bit interval. This is an anti-noise measure; if the reception circuit is triggered by noise on the transmission line, the check for a low after half a bit interval should limit false data reception. Data bits are shifted into the receiver at the programmed baud rate and the data word would be loaded to SBUF if the following conditions are true; R1 must be 0, and mode bit SM2 is 0 or the stop bit is 1. R1 set to 0 implies that the program has read the previous data byte and is ready to receive the next; a normal stop bit will then complete the transfer of data to SBUF regardless of the state of SM2. SM2 set to 0 enables the reception of a byte with any stop bit state. SM2 set to 1 forces reception of only "good" stop bits, an anti-noise safeguard. Of the original ten bits, the start bit is discarded, the eight data bits go to SBUF and the stop bit is saved in it RB8 of SCON. R1 is set to 1, indicating a new byte has been received.

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{\text{OSCILLATOR FREQUENCY}}{12 \times [256 - \text{TH1}]}$$

8.14.3 Serial Data Mode 2- Multiprocessor Mode

Mode 2 is similar to mode 1 except 11 bits are transmitted; a start bit, nine bits and a stop bit. The ninth bit is gotten from TB8 in SCON during transmit and stored in bit RB8 of SCON when data is received. Both the start and stop bits are discarded. Data can be collected quickly from an extensive network of communicating microcontrollers if high baud rates are employed. Here R1 must be 0 before the last bit is received and SM2 must be 0 or the ninth data bit must be 1. Setting R1 based upon the state of SM2 in the receiving 8051 and the state of bit 9 in the transmitted message makes multiprocessing possible by enabling some receivers to be interrupted by certain messages, while other receivers ignore those messages. Only those 8051's that have SM2 set to 0 will be interrupted by receiving data that has the ninth data bit set to 0; those with SM2 set to 1 will not be interrupted by messages with data bits 9 at 0. All receivers will be interrupted by data words that have the ninth data bit set to 1, the state of SM2 will not block reception of such messages.

For mode 2 the baud rate $= 2^{\text{SMOD}} * (F_{\text{OSC}}/64)$

8.14.4 Serial Data Mode 3

Mode 3 is identical to mode 2 except that the baud rate is determined exactly as in mode 1, using Timer 1 to generate communication frequencies.

8.15 MULTIPROCESSOR COMMUNICATION

Modes 2 and 3 can be configured to allow communication between a master 8051 and a number of slave 8051s

The TXD pin of the master connects to the RXD pin of the slave 8051s. In modes 2 and 3, a 9th data bit is transmitted.

- This bit is received into the RB8 of the SCON register.
- If the SM2 bit of the SCON is set, the RI flag will only be set when a serial frame is received with the 9th bit = '1'.
- This 9th bit can be set/cleared to specify an address/data frame.
- On initialization all slaves set the SM2 bit in the SCON register. (Each slave has a unique 8-bit address stored in non-volatile memory.)
- Master 8051 sends an address frame 9th bit TB8 ='1'. (The address frame contains the 8-bit address of the destination slave.)
- All slaves will see the received address frame. (RI bit will be set because the 9th bit RB8 ='1'. Each slave checks the received address against it's own address.)
- The addressed slave will now clear it's SM2 bit in SCON to '0'.

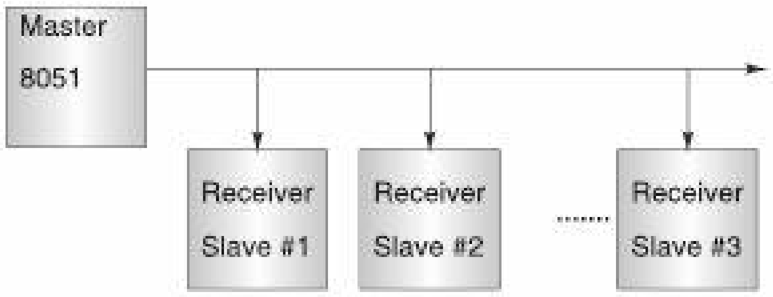


Fig. 8.14 Interconnection between Master and Slave.

- The master will now send some data frames with the 9th bit ='0'. (Only the addressed slave will see the data frames. No RI flag will be set in the other slaves because the RB8 bit is'0'. On completion of data reception the addressed slave will set the SM2 bit to await another address frame.)

8.16 INTERRUPTS

A computer program has only two ways to determine the conditions that exist in internal and external circuit. One method uses the software instructions that jump on the states of flags and port pins. The second respond to hardware signals, called interrupts, that force the program to call a subroutine. Software techniques use up processor time that could be devoted to other tasks; interrupts take processor time only when action by the program is needed. Most applications of microcontrollers involve responding to events quickly enough to control environment that generates the events (generally termed "real-time programming"). Interrupts are the only way in which real time programming can be done successfully. Interrupts may be generated by internal chip operations or provided by external sources. Any interrupt can cause the 8051 to perform a hardware call to an interrupt-handling subroutine that is located at a predetermined absolute address in program memory. Five interrupts are provided in the 8051. Three of these are generated automatically by internal operations: timer flag 0, timer flag 1, and the serial port interrupt (R1 or T1). Two interrupts are triggered by external signals provided by circuitry that is connected to pins (INT0)' and (INT1)' (port pins P3.2 and P3.3).

All interrupt functions are under the control of the program. The programmer is able to alter control bits in the interrupt enable register (IE), the interrupt priority register (IP), and the timer control register (TCON). The program can block all or any combination of the interrupts from acting on the program by suitable setting or clearing bits in these registers. All the interrupts has been handled by the interrupt subroutine, which is placed by the programmer at the interrupt location in program memory, the interrupted program must resume operation at the instruction where the interrupt takes place. Program resumption is done by storing the interrupted PC address on the stack in RAM before changing the PC to the interrupt address in ROM. The PC address will be restored from the stack after an RET1 instruction is executed at the end of the interrupt subroutine.

8.16.1 Timer Flag Interrupt

When a timer/counter overflows, the corresponding timer flag, TF0 or TF1, is set to 1. The flag is cleared to 0 when the resulting interrupt generates a program call to the appropriate timer subroutine in memory.

8.16.1.1 Interrupt Enable (IE) register

EA	–	ET2	ES	ET1	EX1	ET0	EX0

Bit 7	**EA**- Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
Bit 6	Reserved.
Bit 5	**ET2**- Timer 2 interrupt enable bit.
Bit 4	**ES**- Serial Port interrupt enable bit.
Bit 3	**ET1**- Timer 1 interrupt enable bit.
Bit 2	**EX1**- External interrupt 1 enable bit.
Bit 1	**ET0**- Timer 0 interrupt enable bit.
Bit 0	**EX0**- External interrupt 0 enable bit.

8.16.2 Types of Interrupts

There are total six hardware and software interrupts in 8051 as disussed below.

8.16.2.1 Serial Port Interrupt

If a data byte is received, an interrupt bit, R1, is set to 1 in SCON register. When a data byte has been transmitted an interrupt bit, T1, is set in SCON. These are ORed together to provide a single interrupt to the processor: the serial port interrupt. These bits are not cleared when the interrupt generated program call is made by the processor. The program that handles the serial data communication must reset T1 or R1 to 0 to enable the next data communication operation.

8.16.2.2 External Interrupts

Pins (INT0)' and (INT1)' are used by external circuitry. Inputs on the pins can set the interrupt flags IE0 and IE1 to 1 in the SCON register by two different methods. The IEX flags may be set when the (INTX)' pin signal reaches a low level, or the flags may be set when a high to low transition takes place on the (INTX)' pins. Bits IT0 and IT1 in TCON program the (INTX)' pins for low level interrupt when set to 0 and program the (INTX)' pins for transition interrupt when set to 1. Flags IEX will be reset when a transition generated interrupt is accepted by the processor and the interrupt subroutine is accessed.

8.16.2.3 Reset

A reset can be considered to be the ultimate interrupt because the program may not block the action of the voltage on the RST pin. This type of interrupt is often called "non-maskable," since no combination of bits in any register can stop, or mask the reset action.

8.16.3 Interrupt Enable/Disable

Bits in the EI register are set to 1 if the corresponding interrupt source is to be enabled and set to 0 to disable the interrupt source. Bit EA is a master, or "global," bit that can enable or disable all of the interrupts.

8.16.4 Interrupt Priority

Register IP determines if any interrupt is to have a high or low priority. Bits set to 1 give the accompanying interrupt a high priority while a 0 assigns a low priority. Interrupts with a high priority can interrupt another interrupt with a lower priority; the low priority interrupt continues after the higher is finished. If two interrupts with the same priority occur at the same time, then they have the following ranking:

1. IE0
2. TF0
3. IE 1
4. TF1
5. Serial = R1 or T1

The serial interrupt could be given the highest priority by setting PS it in IP to 1 and all others to 0.

8.17 ASSEMBLY LANGUAGE AND INSTRUCTION SET

Assembly language for 8051 is quite similar to 8085 except that it has instructions for MUL/DIV and other bit related operation. Let us first look at addressing mode of 8051.

8.17.1 Addressing Mode

An "addressing mode" refers to how you are addressing a given memory location. The addressing modes are as follows:

- Immediate addressing mode
- Direct addressing mode
- Register addressing mode
- Register indirect addressing mode
- Indexed addressing mode

Before moving ahead let us study some of the notation which will encounter throughout the topic.

Table 8.9 List of Symbols used in Programming

Symbol	*Function*
Rn direct	Register R7-R0 of the currently selected Register Bank.
	8-bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR [i.e., I/O port, control register, status register, etc. (128-255)].
@Ri	8-bit internal data RAM location (0-255) addressed indirectly through register R1or R0.
#data	8-bit constant included in instruction.
#data 16	16-bit constant included in instruction.
addr 16	16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64 Kbyte Program Memory address space.
addr 11	11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2 Kbyte page of program memory as the first byte of the following instruction.
relative	Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is –128 to + 127 bytes relative to first byte of the following instruction.
Bit	Direct Addressed bit in Internal Data RAM or Special Function Register.

1. Immediate Addressing

Immediate addressing is very fast but it is not very flexible. In this type of addressing, the value to be stored in memory immediately follows the operation code in memory.

```
MOV A, #38 H          ; load 38 H into A
MOV R3, #72           ; load 72 into R3
MOV B, #40 H          ; load 40 H into B
MOV DPTR, #4382 H     ; DPTR = 4382 H
MOV DPL, # 82 H       ; this is the same as above.
MOV DPH, #43 H
```

In assembly language, immediate operands are preceded by #. This addressing mode can be used to load the information into any registers, including 16-bit DPTR register. The DPTR register can also be accessed as two 8-bit registers where the high byte is DPH and low byte is DPL.

2. Direct Addressing

Direct Addressing specifies the operand by an 8-bit address field in the instruction. Only internal data RAM and SFRs can be directly addressed.

```
ADD A,55H             ; add A content and data at 55h memory location
```

3. Indirect Addressing

In indirect addressing, the instruction specifies a register which contains the address of the operand. In this, both internal and external RAM can be directly addressed.

```
MOV A,@R0            ; load the A with value from RAM which is pointed by R0.
```

4. Register Addressing

In this mode, the register banks, containing registers R0-R7, can be accessed by certain instructions which carry a 3-bit register specification within the opcode of the instruction. Those instructions which access the registers this way are code efficient, since this mode eliminates an address byte.

- Register to register transfer is not possible in 8051

```
MOV R0,R1            ; is not possible in 8051.
MOV A,R1             ; copy the content of R1 in A.
```

5. External Direct

Through external addressing, we can access external memory. Due to use of external memory, this type of addressing is called external addressing. In microcontrollers, we have only two commands that use External Direct addressing mode:

```
MOVX A, @DPTR
MOVX @DPTR, A
```

loaded with the address of external memory that you wish to read or write. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the accumulator. The second command will do the opposite: it will allow you to write the value of the accumulator to the external memory address pointed to by DPTR.

6. Relative Addressing

Relative addressing is used with certain jump instructions. Relative address (offset) is an 8-bit signed value (–128 to 127) which is added to the program counter to form the address of next instruction. Prior to addition, the program counter is incremented to the address following the jump (the new address is relative to the next instruction, not the address of the jump instruction). This detail is of no concern to the user since the jump destinations are usually specified as labels and the assembler. It determines the relative offset advantage of relative addressing: position independent codes

```
SJMP Label_X
```

7. Indexed Addressing

Indexed addressing uses a base register (either the program counter or data pointer) and an offset (the accumulator) in forming the effective address for a JMP or MOVC instruction

```
MOVC A, @A+DPTR
```

This instruction moves a byte of data from code memory to the accumulator. The address in code memory is found by adding the accumulator to the data pointer.

8.17.2 Instruction Set

8051 instructions have 8-bit opcode. Some instructions have one or two additional bytes for data or address. There are 139 1-byte instructions, 92 2-byte instructions, and 24 3-byte instruction in microcontrollers 8051 instructions are divided among five groups:

- Data transfer
- Arithmetic
- Logical
- Boolean variable
- Program branching

8.17.2.1 Data Transfer Instruction

To transfer the data there are three possibilities, which are mentioned below:

- Can transfer data within internal RAM. $\longrightarrow$ MOV
- Can transfer data within external RAM. $\longrightarrow$ MOVX
- Can read any data from ROM. $\longrightarrow$ MOVC

R0 and R1 are the only registers that can be used for pointers in register indirect addressing mode. Since R0 and R1 are 8 bits wide, their use is limited to access any information in the internal RAM. Whether accessing externally connected RAM or on-chip ROM, we need 16-bit pointer. In such case, the DPTR register is used.

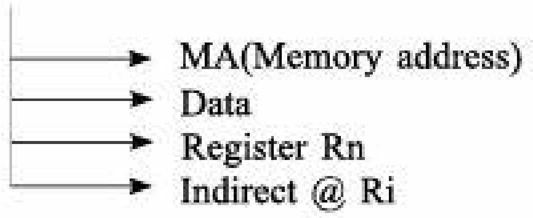
(i) Data within internal RAM.

The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

Syntax MOV <dest-byte>,<src-byte>

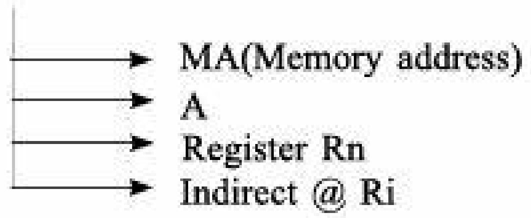
The valid possible transfer combination are listed below.

(a) MOV A, <source>



Example: MOV A, 20H, MOV A, #10H, MOV A, R0, MOV A, @R1

(b) MOV MA, <source>



Example: MOV 20H, A, MOV 20H, R0, MOV 20H, 10H, MOV 20H, @R0

(c) MOV R, <source>

→ MA(Memory address)
→ Imm.
→ A

Example: MOV R0, A, MOV R1, #12H, MOV R2, 22H

(d) MOV@ Ri, <source>

→ MA(Memory address)
→ Imm.
→ A

Example: MOV @Ri, A, MOV @Ri, 20H, MOV @Ri, #20H

(e) MOV DPTR, #16 Bit data

(ii) Data with in external RAM.

External RAM can either be read or write. All the possible combination are listed below.

```
MOVX A, @ Ri
MOVX A, @DPTR
MOVX @Ri, A
MOVX @DPTR, A
```

8.17.2.2 Branching Instruction

The branch instructions are very useful when one wishes either to skip some instruction and move to other part of the program or move to subroutine or interrupt service routine.

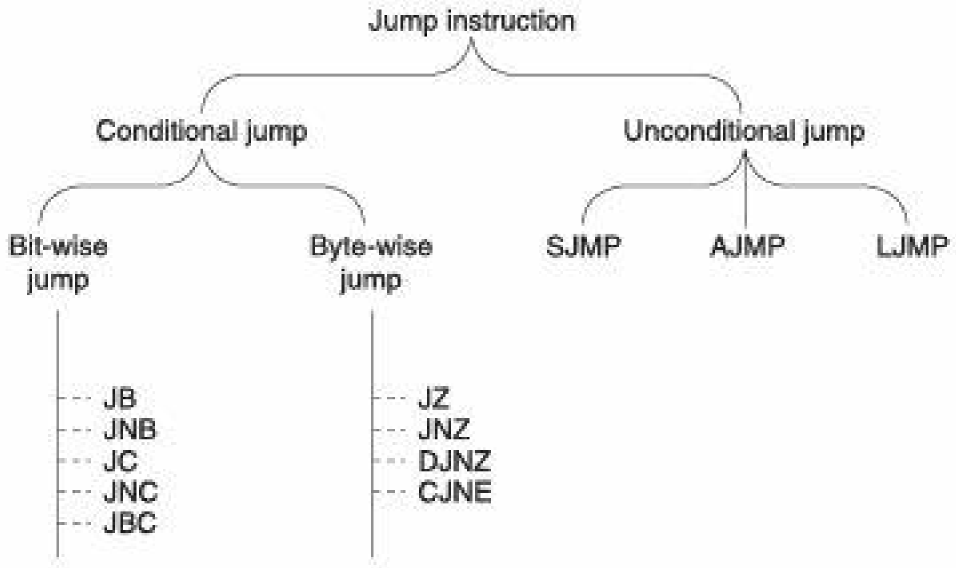


Fig. 8.15 Various Branch Instructions.

1. AJMP-(Absolute Jump)

We can jump anywhere within 2k memory using it, AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the

PC (after incrementing the PC twice), opcode bits 7 through 5, and the second byte of the instruction AJMP label

- Syntax: AJMP label
- 2-byte instruction

2. LJMP-(Long Jump)

LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64 K program memory address space. No flags are affected.

- Syntax LJMP < label>
- 3-byte instruction.

3. SJMP-(Short Jump)

The range of branching allowed is from 128 bytes preceding this instruction 127 bytes following it.

- Syntax SJMP < label>
- 2-byte instruction

4. JB-(Jump if Bit)

If the indicated bit is a one, JB jump to the address indicated; otherwise, it proceeds with the next instruction. The bit tested is not modified. No flags are affected.

- Syntax JB <Bit name>,< label>
- 3-byte instruction.
- Eg. JB p1.0, loop

5. JNB-(Jump if not Bit)

It will branch to new location if tested bit is not set or not equal to 1

- Syntax JNB <Bit name>, <label>
- 3-byte instruction.
- Eg. JNB p1.0, loop

6. JC-(Jump if Carry)

If the carry flag is set, JC branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

- Syntax JC <Bit name>,< label>
- 2-byte instruction.
- Eg. JC p1.0, loop

7. JBC-(Jump if Bit, Clear the Bit)

If the indicated bit is one, JBC branches to the address indicated; otherwise, it proceeds with the next instruction. No flags are affected.

- Syntax JBC< Bit name>, < label>
- 3-byte instruction.
- Eg. JBC p1.0, loop

8. JZ-(Jump if Accumulator Zero)

If all bits of the Accumulator are 0, JZ branches to the address indicated; otherwise, it proceeds with the next instruction. The Accumulator is not modified. No flags are affected.

- Syntax JZ <Label>
- 2-byte instruction.
- Eg. JZ Loop

9. DJNZ-(Decrement and Jump if Not Zero)

DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H underflows to 0FFH. No flags are affected.

- Syntax DJNZ <Rn>,<label>

  or

  DJNZ <MA>,<label>
- 2-byte instruction

10. CJNE-(Compare and Jump if Not Zero)

CJNE compares the magnitudes of the first two operands and branches if their values are not equal. The possible valid formats are written below.

- Syntax CJNE <A>,<# VALUE>, <LABEL>

  Or

  CJNE <Rn>,<# VALUE>, <LABEL>

  Or

  CJNE <@Ri>,<# VALUE>, <LABEL>

  Or

  CJNE <A>,<MA>, <LABEL>

8.17.2.3 Logic Instruction

The destination address of the operation can be the accumulator (register A), a general register, or a direct address. Status flags are not affected by these logical operations (unless PSW is directly manipulated). All the possible logic instructions with their format are listed below.

Table 8.10 Logical Instructions

Mnemonic	*Format*	*Description*
ANL	A,Rn	AND register to Accumulator
ANL	A,direct	AND direct byte to Accumulator

Contd...

Contd...

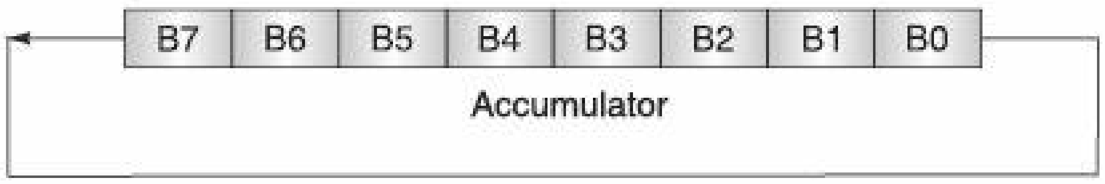
ANL	A,@Ri	AND indirect RAM to Accumulator
ANL	A,#data	AND immediate data to Accumulator
ANL	direct,A	AND Accumulator to direct byte
ANL	direct,#data	AND immediate data to direct byte
ORL	A,Rn	OR register to Accumulator
ORL	A,direct	OR direct byte to Accumulator
ORL	A,@Ri	OR indirect RAM to Accumulator
ORL	A,#data	OR immediate data to Accumulator
ORL	direct,A	OR Accumulator to direct byte
ORL	direct,#data	OR immediate data to direct byte
XRL	A,Rn	Exclusive-OR register to Accumulator
XRL	A,direct	Exclusive-OR direct byte to Accumulator
XRL	A,@Ri	Exclusive-OR indirect RAM to Accumulator
XRL	A,#data	Exclusive-OR immediate data to Accumulator
XRL	direct,A	Exclusive-OR Accumulator to direct byte
XRL	direct,#data	Exclusive-OR immediate data to direct byte
CLR	A	Clear Accumulator
CPL	A	Complement Accumulator
RL	A	Rotate Accumulator Left
RLC	A	Rotate Accumulator Left through the Carry
RR	A	Rotate Accumulator Right
RRC	A	Rotate Accumulator Right through the Carry
SWAP	A	Swap nibbles within the Accumulator

8.17.2.4 Rotate Instructions

The ability to rotate the A register (accumulator) data is useful to allow examination of individual bits. The options for such rotation are as follows:

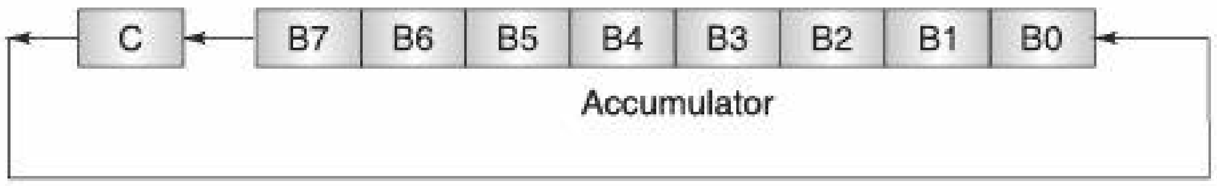
RL (Rotate accumulator left)

The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected. It is 1 byte instruction.



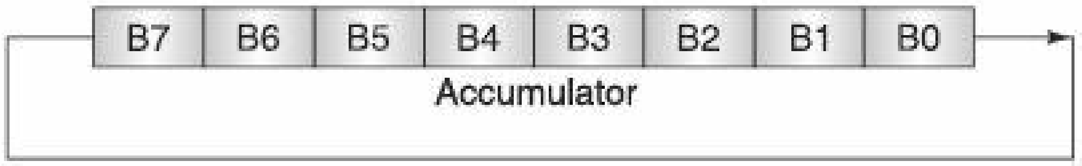
RLC (Rotate accumulator left through the carry flag)

The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.



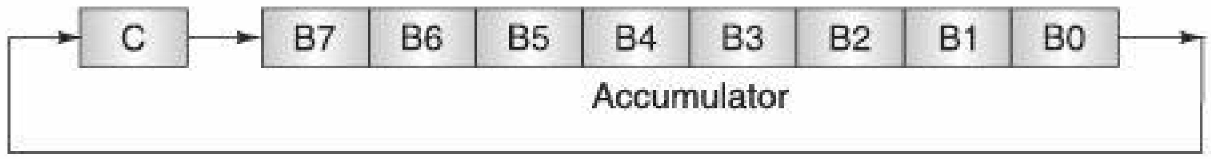
RR (Rotate accumulator right)

The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.



RRC (Rotate accumulator right through carry flag)

The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.



8.17.2.5 Arithmetic Instruction

These instructions are useful for manipulating data calculation. All arithmetic instructions are executed in one machine cycle except INC DPTR (two cycles) and MUL AB and DIV AB.

Table 8.11 Arithmetic Instructions

Mnemonic	*Format*	*Description*
ADD	A,Rn	Add register to Accumulator
ADD	A,direct	Add direct byte to Accumulator
ADD	A,@Ri	Add indirect RAM to Accumulator
ADD	A,#data	Add immediate data to Accumulator
ADDC	A,Rn	Add register to Accumulator with Carry
ADDC	A,direct	Add direct byte to Accumulator with Carry
ADDC	A,@Ri	Add indirect RAM to Accumulator with Carry
ADDC	A,#data	Add immediate data to Acc with Carry
SUBB	A,Rn	Subtract Register from Acc with borrow
SUBB	A,direct	Subtract direct byte from Acc with borrow
SUBB	A,@Ri	Subtract indirect RAM from Acc with borrow
SUBB	A,#data	Subtract immediate data from Acc with borrow
MUL	AB	Multiply A and B
DIV	AB	Divide A by B
DA	A	Decimal Adjust Accumulator

8.17.2.6 Incrementing and Decrementing

The simplest arithmetic operations involve adding or subtracting a binary 1 and a number. These simple operations become very powerful when coupled with the ability to repeat the operation that is, to "Increment" or "Decrement" until a desired result is reached. Register, Direct, and Indirect addresses may be Incremented or Decremented. No math flags (C, AC, and OV) are affected. The Table 8.12 lists the increment and decrement mnemonics.

Table 8.12 Increment/Decrement Instruction

Mnemonic	*Format*	*Description*
INC	A	Increment accumulator
INC	Rn	Increment register
INC	Direct	Increment direct byte
INC	@Ri	Increment direct RAM
DEC	A	Decrement accumulator
DEC	Rn	Decrement register
DEC	Direct	Decrement direct byte
DEC	@Ri	Decrement indirect RAM
INC	DPTR	Increment data pointer

8.17.2.7 Some Assembler Directives

The assembler directives are special instructions to the assembler program to define some specific operations but these directives are not part of the executable program.

Some of the most frequently assembler directives are listed as follows:

ORG	Originate, defines the starting address for the program in program (code) memory
EQU	Equate, assigns a numeric value to a symbol identifier so as to make the program more readable.
DB	Define a Byte, puts a byte (8-bit number) number constant at this memory location
DW	Define a Word, puts a word (16-bit number) number constant at this memory location
DBIT	Define a Bit, defines a bit constant, which is stored in the bit addressable section if the Internal RAM.
END	This is the last statement in the source file to advise the assembler to stop the assembly process.

1. A Program to Toggle all the Bits of P0, and P2 at every 1 microsecond Frequency = 11.0592 MHz

Answer:

```
Labels:                Mnemonic
ORG 0000 H
```

```
BACK:                         MOV A, #55H
MOV P0, A
MOV P1, A
MOV P2, A
ACALL DELAY
MOV A, #0AAH
MOV P0, A
MOV P1, A
MOV P2, A
ACALL DELAY
SJMP BACK
```

Subprogram for Delay

```
MOV R5, #11
MOV R4, #248
MOV R3, #255
DJNZ R3, HERE3
DJNZ R4, HERE2
DJNZ R5, HERE1
RET
END
```

2. A Program for Generating a Square Wave of 66% Duty Cycle on Bit 3 of Port 1.

Answer:

```
Labels                        Mnemonic
BACK:                         SETB P1.3
LCALL DELAY
LCALL DELAY
CLR P1.3
LCALL DELAY
SJMP BACK
```

Subprogram for Delay

```
MOV R5, #11
MOV R4, #248
MOV R3, #255
DJNZ R3, HERE3
DJNZ R4, HERE2
DJNZ R5, HERE1
RET
END
```

3. A Program to Check the Status of the Switch when it is Connected to Pin 1.7 and Make the Following Decision

(a) if SW = 0, send '0' to P2

(b) if SW = 1, send '1' to P2

Answer:

Labels	**Mnemonic**
SW	EQU 97H
MY DATA	EQU 0A0H
HERE	MOV C, SW
JC OVER	
MOV MY DATA, #'0'	
SJMP HERE	

4. A Program to Convert Binary into ASCII.

Answer:

Labels	**Mnemonic**
ADDRESS	EQU 40H
RESULT	EQU 50H
COUNT	EQU 3
ORG 0000H	
ACALL BIN_DEC	
ACALL DEC_BIN	
SJMP $	

A Subprogram for Binary to Decimal

```
MOV R0, #ADDRESS
MOV A, P1
MOV B, #10
DIV AB
MOV @R0, B
INC R0
MOV B, #10
DIV AB
MOV @R0, B
INC R0
MOV @R0, A
RET
```

A Subprogram for Decimal to Binary

```
MOV R0, #ADDRESS
MOV R1, #RESULT
```

```
    MOV R2, #3
    BACK: MOV A, @R0
    ORL A, #30H
    MOV @R1, A
    INC R0
    INC R1
    DJNZ R2, BACK
    RET
    END
```

5. A Program To exhibit the RAM direct addressing and bit addressing schemes of 8051 Microcontroller.

Answer:

Bit Addressing:

```
    SETB PSW.3
    MOV R0,#data1
    MOV A,#data2
    ADD A,R0
    MOV DPTR,#4500
    MOVX @DPTR,A
    HERE: SJMP HERE
```

Direct Addressing:

```
    MOV 30,#DATA1
    MOV A,#DATA2
    ADD A,30
    MOV DPTR,#4500
    MOVX @DPTR,A
    HERE: SJMP HERE
```

6. A Program to exchange the content of FFh and FF00h

Answer: Here one is internal memory location and other is memory external location. So first the content of external memory location FF00h is loaded in accumulator. Then the content of internal memory location FFh is saved first and then content of accumulator is transferred to FFh. Now saved content of FFh is loaded in accumulator and then it is transferred to FF00h.

```
    Mov dptr, #0FF00h; take the address in dptr
    Movx a, @dptr              ; get the content of 0050h in a
    Mov r0, 0FFh               ; save the content of 50h in r0
    Mov 0FFh, a                ; move a to 50h
    Mov a, r0                  ; get content of 50h in a
    Movx @dptr, a              ; move it to 0050h
```

7. A Program to store the higher nibble of r7 in to both nibbles of r6

Answer: First we shall get the upper nibble of r7 in r6. Then we swap nibbles of r7 and make OR operation with r6 so the upper and lower nibbles are duplicated.

```
Mov a, r7          ; get the content in accumulator
Anl a, #0F0h       ; mask lower bit
Mov r6, a          ; send it to r6
Swap a             ; xchange upper and lower nibbles of A.
Orl a, r6          ; OR operation
Mov r6, a          ; finally load content in r6
```

8. A Program Treat r6-r7 and r4-r5 as two 16-bit registers. Perform subtraction between them. Store the result in 20h (lower byte) and 21h (higher byte).

Answer: First we shall clear the carry. Then subtract the lower bytes afterward then subtract higher bytes.

```
Clr c              ; clear carry
Mov a, r4          ; get first lower byte
Subb a, r6         ; subtract it with other
Mov 20h, a         ; store the result
Mov a, r5          ; get the first higher byte
Subb a, r7         ; subtract from other
Mov 21h, a         ; store the higher byte
```

9. A Program divides the content of r0 by r1. Store the result in r2 (answer) and r3 (reminder). Then restore the original content of r0.

Answer: After getting answer to restore original content, we have to multiply answer with divider and then add reminder in that.

```
Mov a, r0          ; get the content of r0 and r1
Mov b, r1          ; in register A and B
Div ab             ; divide A by B
Mov r2, a          ; store result in r2
Mov r3, b          ; and reminder in r3
Mov b, r1          ; again get content of r1 in B
Mul ab             ; multiply it by answer
Add a, r3          ; add reminder in new answer
Mov r0, a          ; finally restore the content of r0
```

10. A Program to transfer the block of data from 20h to 30h to external location 1020h to 1030h.

Answer: Here we have to transfer 10 data bytes from internal to external RAM. So first, we need one counter. Then we need two pointers one for source second for destination.

```
Mov r7, #0Ah       ; initialize counter by 10d
Mov r0, #20h       ; get initial source location
Mov dptr, #1020h   ; get initial destination location
```

```
Nxt:      Mov a, @r0          ; get first content in accumulator
          Movx @dptr, a       ; move it to external location
          Inc r0              ; increment source location
          Inc dptr            ; increase destination location
          Djnz r7, nxt        ; decrease r7. if zero then over otherwise move next
```

11. A Program to find out how many equal bytes between two memory blocks 10h to 20h and 20h to 30h.

Answer: Here we shall compare each byte one by one from both the blocks. Increase the count every time when equal bytes are found.

```
          Mov r7, #0Ah        ; initialize counter by 10d
          Mov r0, #10h        ; get initial location of block1
          Mov r1, #20h        ; get initial location of block2
          Mov r6, #00h        ; equal byte counter. Starts from zero
Nxt:      Mov a, @r0          ; get content of block 1 in accumulator
          Mov b, a            ; move it to B
          Mov a, @r1          ; get content of block 2 in accumulator
          Cjne a, b, nomatch  ; compare both if equal
          Inc r6              ; increment the counter
Nomatch:  inc r0              ; otherwise go for second number
          Inc r1
          djnz r7, nxt        ; decrease r7. if zero then over otherwise move next
```

12. A Program to given block of 100h to 200h. Find out how many bytes from this block are greater than the number in r2 and less than number in r3. Store the count in r4.

Answer: In this program, we shall take each byte one by one from the given block. Now here two limits are given higher limit in r3 and lower limit in r2. So we check first higher limit and then lower limit if the byte is in between these limits then count will be incremented.

```
          Mov dptr, #0100h    ; get initial location
          Mov r7, #0FFh       ; counter
          Mov r4, #00h        ; number counter
          Mov 20h, r2         ; get the upper and lower limits in
          Mov 21h, r3         ; 20h and 21h
Nxt:      Movx a, @dptr       ; get the content in accumulator
          Cjne a, 21h, lower  ; check the upper limit first
          Sjmp out            ; if number is larger
Lower:    jnc out             ; jump out
          Cjne a, 20h, limit  ; check lower limit
          Sjmp out            ; if number is lower
```

```
Limit:    jc out                 ; jump out
          Inc r4                 ; if number within limit increment count
Out:      inc dptr               ; get next location
          Djnz r7, nxt           ; repeat until block completes
```

13. A program to continuously scan port P0. If data is other than FFh write a subroutine that will multiply it with 10d and send it to port P1.

Answer: Here we have to use polling method. We shall continuously pole port P0 if there is any data other than FFh. If there is data we shall call subroutine.

```
Again:    Mov p0, #0ffh          ; initialize port P0 as input port
Loop:     Mova, p0               ; get the data in accumulator
          Cjne a, #0FFh, dat     ; compare it with FFh
          Sjmp loop              ; if same keep looping
Dat:      acall multi            ; if different call subroutine
          Sjmp again             ; again start polling
          Multi
          Mov b,#10d             ; load 10d in register B
          Mul ab                 ; multiply it with received data
          Mov p1, a              ; send the result to P1
          Ret                    ; return to main program
```

8.18 INTERFACING WITH 8051

8.18.1 LED Interfacing with 8051

As its name implies it is a diode, which emits light when forward biased. Charge carrier recombination takes place when electrons from the N-side cross the junction and recombine with the holes on the P side. Electrons are in the higher conduction band on the N side whereas holes are in the lower valence band on the P side. During recombination, some of the energy is given up in the form of heat and light. In the case of semiconductor materials like Gallium arsenide (GaAs), Gallium phosphate (Gap) and Gallium arsenide phosphate (GaAsP) a greater percentage of energy is released during recombination and is given out in the form of light. LED emits no light when junction is reversed biased.

LED anode or cathode both can be connected to voltage supply. Here in this circuit if there is a output is reset at port then LED will glow. Here we are using port as output port.

If P0.0 = 0, first LED will glow for the time duration during which port pin is in reset state

If P0.1 = 0, then second LED will glow.

PROGRAM

To blink a LED connected on port P1.0 after every certain delay.

```
$mod51
ORG 0000H
```

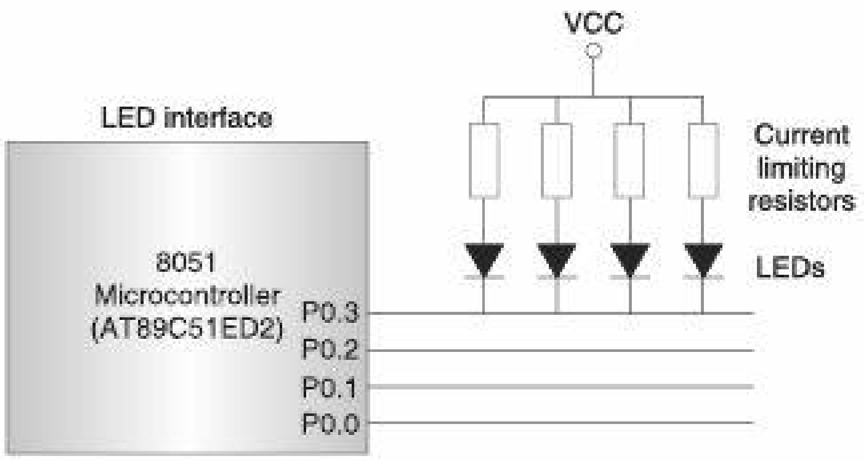


Fig. 8.16 LED Interfacing.

```
Again:                  SETB P1.0
ACALL delay
CLR P1.0
ACALL delay
AJMP Again
Delay:
                        MOV R0, #0FFH
LOOP:                   MOV R1,#0FFH
Again1:                 DJNZ R1,Again1
                        DJNZ R1,LOOP
                        RET
                        END.
```

8.18.2 LCD Interfacing with 8051

This is achieved by displaying their status on a small display module. LCD (Liquid Crystal Display) screen is such a display module and a 16x2 LCD module is very commonly used. These modules are replacing seven segments and other multi-segment LEDs for these purposes. The reasons being: LCDs are economical, easily programmable, have no limitation of displaying special and even custom characters (unlike in seven segments), animations, and so on. LCD can be easily interfaced with a microcontroller to display a message or status of a device. This topic explains the basics of a 16x2 LCD and how it can be interfaced with AT89C51 to display a character.

The LCD requires 3 control lines (RS, R/W & EN) & 8 (or 4) data lines. The number on data lines depends on the mode of operation. If operated in 8-bit mode then 8 data lines plus 3 control lines, i.e., total 11 lines are required. Here is a program for interfacing a LCD through 8051 in 8-bit mode.

The three control lines are ***RS*, *RW* & *Enable*.**

- **RS:** Register Select. When RS = 0 the command register is selected. When RS = 1 the data register is selected.

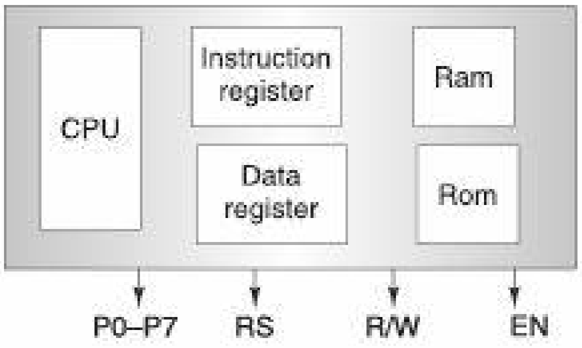


Fig. 8.17 Block Diagram of LCD.

- **RW:** Read or Write. When RW = 0 write to LCD. When RW = 1 read from LCD.
- **Enable:** Used to Latch data into the LCD. A HIGH TO LOW edge latches the data into the LCD.

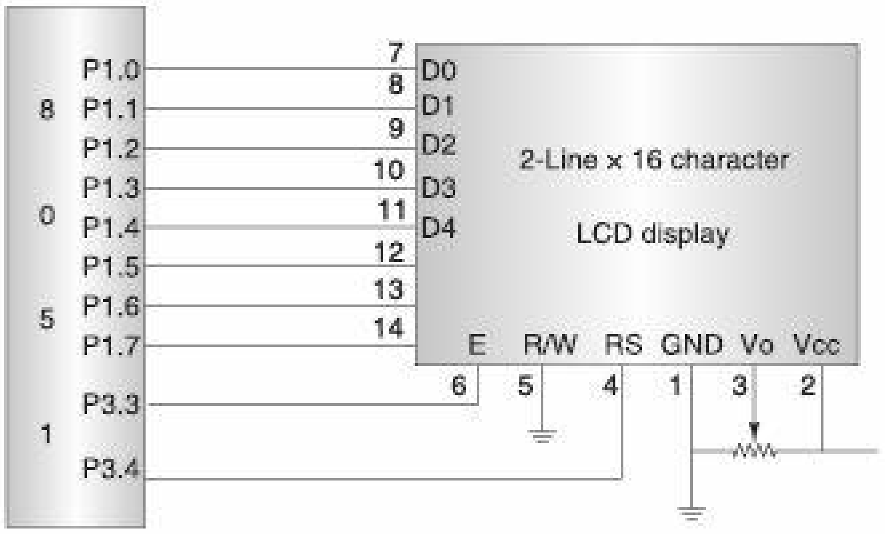


Fig. 8.18 LCD Interface with 8051.

Table 8.13 Pin Description

Pin no	*Symbol*	*Description*
1	GND	Details
2	Vcc	Ground
3	Vee	Supply Voltage +5 V
4	RS	Contrast adjustment
5	R/W	0-> Control input
6	EN	1-> Data input
7-14	DO-D7	Read/Write
15	VB1	Enable
16	VBO	Data

8.18.2.1 Algorithm to send data to LCD

1. Make R/W low
2. Make RS = 0; if data byte is command,
   RS = 1; if data byte is data (ASCII value)
3. Place data byte on data register
4. Pulse E (HIGH to LOW)
5. Repeat the steps to send another data byte

Table 8.14 LCD Command Codes

Hex code	*Command to LCD instruction register*
01	Clear screen display
02	Return home
04	Decrement cursor
06	Increment cursor
0E	Display on, cursor blinking
80	Force the cursor to the beginning of the 1st line
C0	Force cursor to the beginning of the 2nd line
38	Use 2 lines and 5x7 matrix
15	Shift entire display to left

A Assembly Language Code Interfacing LCD 16 × 2 in 4-bit mode. Port0 to higher nibble data pins of LCD Crystal at 3.579545 MHz to AT89C51

```
:P2.0 to RS pin
:P2.1 to Enable Pin
ORG 0000H
AJMP MAIN
ORG 0030H
MAIN:
MOV SP,#60H ;STACK POINTER
ACALL LCD_INIT ;Initialize lcd
MOV DPTR,#MESSAGE1
CALL LCD_STRING ;Display message on LCD
CALL NEXT_LINE ;Place cursor to;second Line
MOV DPTR,#MESSAGE2
CALL LCD_STRING ;Display message on LCD
HERE: AJMP HERE
LCD_INIT: ;initialize LCD in 4-bit mode
ANL P0,#0F0H
CALL LOOP
```

```
MOV DPTR,#LCDCODE
MOV A,#0H
MOV R6,#0H
MOV R7,#0H
CLR P2.0 ;RS COMMAND
NEXT: ;8-bit code is split into two 4-bit nibbles.
INC R6
MOVC A,@A+DPTR
MOV R7,A
ANL A,#0F0H
SWAP A
ANL P0,#0F0H
ORL P0,A
ACALL ENABLE; PULSE E sending first nibble
MOV A,R7
ANL A,#0FH
ANL P0,#0F0H
ORL P0,A
ACALL ENABLE; PULSE E sending second nibble
MOV A,R6
CJNE R6,#09H,NEXT
RET
LCD_STRING:
MOV P0,#00H
SETB P2.0 ;RS DATA
MOV A,#00H
MOV R6,#00H
NC: ;checks the end of message string
MOVC A,@A+DPTR
CJNE A,#2FH,NC1
RET
NC1:
LCALL LCD_WRITE
INC R6
MOV A,R6
AJMP NC
LCD_WRITE:
SETB P2.0 ;RS DATA
```

```
CALL LO
RET
NEXT_LINE:
MOV P0,#00H
CLR P2.0 ;RS COMMAND
MOV A,#0C0H
CALL LO
RET
LCD_CLEAR: ;This subroutine is used to clear LCD
CALL DELAYL
ANL P0,#00H
MOV A,#01H
CLR P2.0 ; RS command
LO: ;8-bit code is split into two 4-bit nibbles.
MOV R7,A
ANL A,#0F0H
SWAP A
ANL P0,#0F0H
ORL P0,A
CALL ENABLE
MOV A,R7
ANL A,#0FH
ANL P0,#0F0H
ORL P0,A
CALL ENABLE
RET
ENABLE: ;Give High-to-low pulse at enable pin
SETB P2.1
CALL DELAYL
CLR P2.1
CALL DELAYL
RET
:With respect to crystal frequency 3.579 MHz
DELAYL: ; 5 ms DELAY
SETB PSW.4 ; SELECT BANK 2
MOV R7,#25
HDH:
```

```
MOV R6,#60
DJNZ R6,$
DJNZ R7,HDH
CLR PSW.4 ;DEFAULT BANK
RET
LOOP: ;1 SEC DELAY
MOV R7,#100
LOOP1:
CALL DELAYL
CALL DELAYL
DJNZ R7,LOOP1
RET
:LCD INITIALIZING CODE (DO NOT DISTURB THIS)
LCDCODE:
DB 02H
DB 02H
DB 02H
DB 28H
DB 28H
DB 28H
DB 0CH
DB 06H
DB 01H
:DATA TO BE DISPLAYED
:Maximum message length = 16 characters.
:To notify end of message place '/' at the end.
MESSAGE1: DB "LCD INTERFACING /" ;Change Message1
MESSAGE2: DB "IT IS EASY /" ;Change Message2
END
```

8.18.3 Stepper Motor Control

A stepper motor is a widely used device that translates electrical pulses into mechanical movement. The stepper motor is used for position control in applications such as disk drivers, dot matrix printers, robotics, etc. Every stepper motor has a permanent magnet rotor (also called the shaft) surrounded by a stator.

Stepper motor control is a very popular application of microprocessor. There are two types of stepper motors:

(a) Permanent magnet (PM)
(b) Variable reluctance (VR)

Permanent Magnet (PM) Stepper Motor: It consists of two stator windings A and B, and a motor having two magnetic poles N and S.

When a voltage +V is applied to stator winding A, a magnetic field Fa is generated. The rotor positions itself such that its poles lock with the corresponding stator poles.

With winding A excited as before, winding B is now switched on to a voltage +V. This produces a magnetic field Fb in addition to Fa.

Variable Reluctance (VR) Stepper Motor: In Variable Reluctance, there are twelve teeth on the stator and eight on the rotor. The rotor does not carry either a permanent magnet or winding and is assembled from soft iron punching. The stator is also assembled from soft iron punching, and carries stator windings A, B and C.

When stator-winding A is excited, it creates patterns of N and S poles. The rotor then positions itself to minimize the reluctance of the magnetic circuit.

Step angle: It is angle through which motor shaft rotates in one step. Step angle is different for different motor. Selection of motor according to step angle depends on the application, simply if you require small increments in rotation choose motor having smaller step angle.

No. of steps require to rotate one complete rotation = 360 deg./step angle in deg.

Steps/second: The relation between RPM and steps per sec. is given by,

steps or impulses /sec. = (RPM * Steps /revolution)/60

The stepper motor interface uses four transistor pairs (SL100 and 2N3055) specification of the stepper motor used:

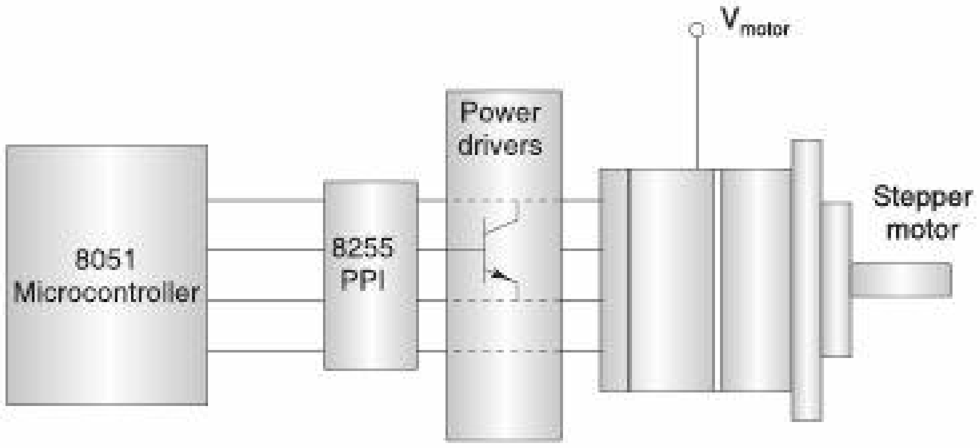


Fig. 8.19 Stepper Motor Interface.

The power requirement is +5V @ 1.2A

The step angle is 1.8* number of steps required is 200

Operation: In the Darlington pair configuration. Each Darlington pair is used to excite the particular winding of the motor connected to 4-pin connector (J4) (A B C D) on the interface. The inputs to these transistors are from **8255-PPI I/O** lines. Port A lower nibble (PA0, PA1, PA2, PA3) are the four lines brought out to the 26-pin FRC male connector (J1) on the interface. When the program is executed, the motor shaft rotates in steps at the speed depending upon the delay between successive steps, which is generated and controlled by the program. The direction of rotation can be controlled by the program.

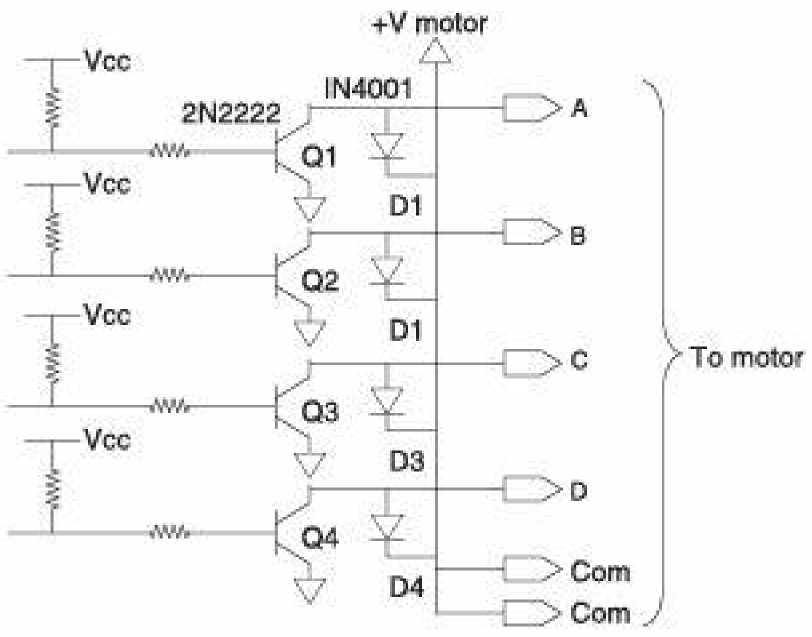


Fig. 8.20 Darlington Pair.

Table 8.15 Step Sequence

Clockwise				Anticlockwise			
A	*B*	*C*	*D*	*A*	*B*	*C*	*D*
1	0	0	1	1	0	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
1	0	1	0	1	0	0	0

A program of interfacing of stepper motor with 8051

```
Label                  Mnemonic
                       MOV DPTR, #0E903H
MOV A, #80H
MOVX @DPTR, A
MOV A, #088H
LOOP:                  MOV DPTR, #0E900H
MOVX @DPTR, A
LCALL DELAY
RR A
SJMP LOOP
DELAY:                 MOV 20H, #03H
L1: MOV 21H, #0FFH
DJNZ 21H, $
DJNZ 20H,L1
RET
END
```

8.18.4 Elevator Interface

This interface simulates the control and operation of an elevator. Four floors are assumed and for each floor a key and corresponding LED indicator are provided to serve as request buttons and request status indicators. A column of ten LEDs represents the elevator itself. Turning on successive LEDs, one at a time, can simulate the motion of the elevator. The delay between turning OFF one LED and turning ON the next LED can simulate the "speed" of the elevator. User can read the request status information through one port, reset the request indicators through another port and control the elevator (LED column) through another port.

Description of the Circuit

This interface has four keys, marked 0, 1, 2 and 3 representing the request buttons at the four floors. Presenting of a key causes a corresponding flip-flop to be set. The outputs of the four flip-flops can be read through port B (PB0, PB1, PB2 and PB3). Also, the status of these signals is reflected by a set of 4 LEDs. The flip-flop can be reset through port A(PA4, PA5, PA6, and PA7). A column of 10 LEDs, representing the elevator can be controlled through port A (PA0, PA1, PA2 and PA3).

These port lines are fed to the inputs of the decoder 7442 whose outputs are used to control the ON/OFF states of the LEDs, which simulate the motion of the elevator.

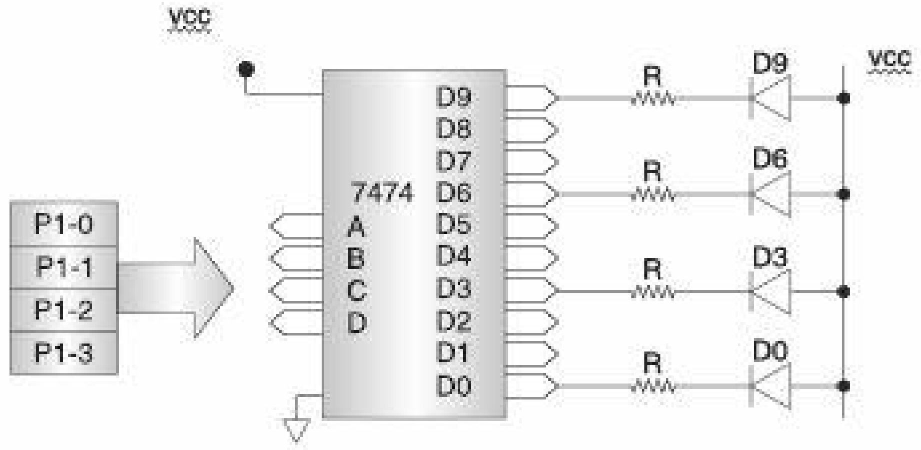


Fig. 8.21 Decoder Circuit.

PROGRAM

```
Elevator Interface
ORG 8000h
SEG EQU 0E9H
MOV 0A0H,#S; Initialize 8255 to mode0
MOV R0,#03H; PORTA as O/P
MOV A,#82H; PORTB as I/P
MOVX @R0,A; store the value in control word register
CLR A; clear the accumulator value
MOV R0,A; Elevator at the ground floor
```

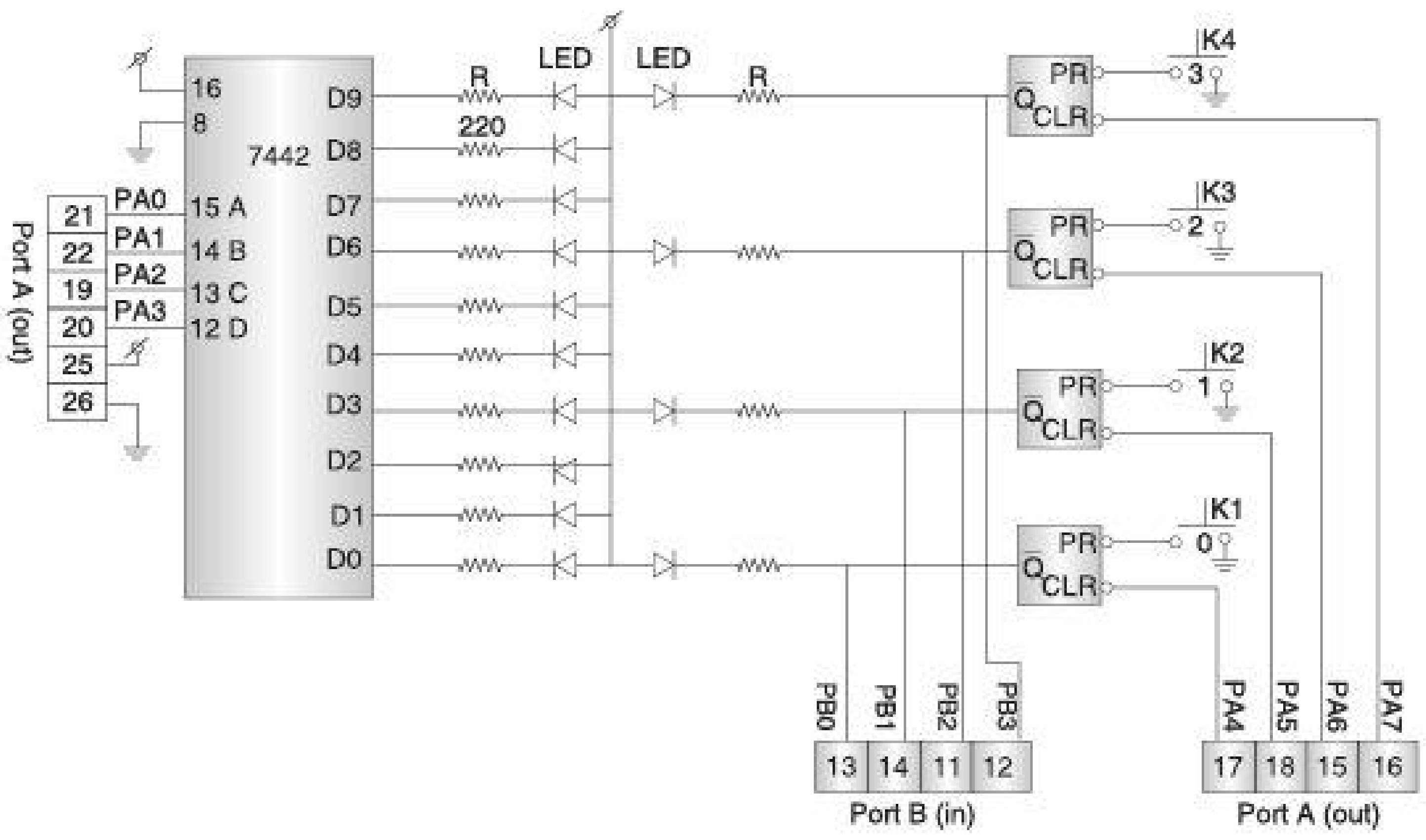


Fig. 8.22 Circuit Diagram.

```
MOV R1,A;
LOOP1: MOV A,R1
ORL A,#0F0H; Clear request(Only upper four Bits are
MOV R0,#00H; used)
MOVX @R0,A; move accumulator value to PORT A
MOV DPTR,#FLOOR; store the data in code memory
MOVX A,@DPTR; take the first data into accumulator
INC R0;
LOOP2: MOVX A,@R0; Get the status of the request
ORL A,#0F0H;
MOV R2,A;
INC A
JNZ LOOP2
LOOP3: MOV A,R2; NO, loop back determine the
RRC A; floor where service is required
MOV R2,A
JNC DECIDE
INC DPTR
SJMP LOOP3
```

```
DECIDE: LCALL DELAY
CLR A; Get the floor number
MOVC A,@A+DPTR
CJNE A,01H,L1
SJMP RESET
L1: JC DOWN; Elevator goes down
UP: INC R1; Elevator goes up
MOV A, R1; Direction indicator
ORL A, #0F0H
MOV R0, #00H; LED's turned ON
MOVX @R0, A
SJMP DECIDE
DOWN: DEC R1; Elevator down
MOV A,R1
ORL A, #0F0H; Direction indicator LED's turned ON
MOV R0, #00H
MOVX @R0, A
SJMP DECIDE
RESET: MOV A, #05H; Corresponding request indicator LED
turned OFF
ADD A, 82H
MOV 82H,A
CLR A
MOV R0, A
MOVC A, @A+DPTR
MOVX @R0, A
SJMP LOOP1
DELAY: PUSH 83H
PUSH 82H
MOV DPTR, #00H
DELAY1: INC DPTR
MOV A, 83H
ORL A, 82H
JNZ DELAY1
POP 82H
POP 83H
RET
```

```
FLOOR: DB 00H, 03H, 06H
DB 09H, 00H, 0E0H
DB 0D3H, 0B6H, 79H
END
```

8.18.5 Analog to Digital Converter

An analog quantity is one that has a continuous set of values over a given range, in contrast with discrete values for the digital case. Any measurable quantity is analog in nature, such as temperature, pressure, speed and time.

Analog to digital conversion is the process by which an analog quantity is converted into digital form. A/D conversion is necessary when measured quantities must be in digital form for processing in a computer, microcontroller or for display or storage.

1. Initialization of 8255

Port A	- Input
Port B	- Output
Port C lower	- Output
Port C upper	- Input

The control word for this configuration of ports is 98H.

The address for ports of 8255 are

Port A	- E800
Port B	- E801
Port C	- E802
Control word	-E803

2. Display routine: 019bh is used to display the digital data in the address field.

3. Preparation: The A/D converter used here is the ADC0808 by national semiconductor. This A/D converter is of 8 bits and has a microprocessor/microcontroller bus interface and single +5 V power. It has an 8 input analog multiplexer. The interface requires 6 lines in addition to the data bus. These are the 3 MUX input select address lines and 3 control lines. The 3 MUX address lines are connected to data bus bits 0 through 2 so that a nibble written to the A/D will select one of the eight inputs. There is a bus output enable line for reading the A/D and an end of convert line that signals the host that the conversion is complete. Lastly, there are 2 more lines that are tied together and connected as 1 to the host. This is the start/ALE line. Placing a nibble on the data bus and setting the start/ALE line to a one write the input MUX selection written to the A/D. When this line is taken back low, the conversion process starts. The EOC line is driven low. Within a few milliseconds, the conversion is complete and EOC line is driven high. The OE line is set high and the result is read from the A/D data bus. The OE line is set low, disabling the A/D output data bus. This completes one cycle of the A/D conversion process. There are two other lines that are tied, one each to the VCC and GND pins. These are the $+V_{ref}$ and $-V_{ref}$ pins. The A/D conversion is done based on the ratio of the input to the V_{ref} pins.

How does the ADC work?

The ADCs require clocking and can contain control logic including comparators and registers. The ADC0808 contains all of these and also has a multiplexer. In order to get it work, there are a total of seven control signals that must be sent. These are the address lines A, B and C. Address Latch Enable (ALE), clock, start and output enable (OE). There is also one control signal; it is the end of conversion (EOC) signal.

ALE [Address latch enable]

ALE is required to load the selected address lines into the ADC. Once loaded the multiplexer sends the appropriate channel to the converter on the chip. The ALE should be pulsed for at least 100 ns in order for the addresses to get loaded properly. As with all control signals it is required to have an input value of VCC–1.5 up to 15 V for a high and 1.5 V down to 0.3 V for a low. The following control signals are used to control the conversion.

Clock

The clock signal is required to cycle through the comparator stages to do the conversion. There are 8-clock cycle periods required in order to complete an entire conversion. This means, then an entire conversion takes at least 64 clock cycles. The maximum frequency of the clock is 1.2 MHz. The source impedance of the analog inputs affects the maximum clock frequency. It is recommended that the source resistance does not exceed 5 Kohms for operation at 1.2 MHz and 10 Kohms for operation at 640 kHz. When operating the ADC at 500 kHz and below the ALE signal and the start signal can be tied together.

Start

The purpose of the start signal is twofold. On the rising edge of the pulse the internal registers are cleared and on the falling edge of the pulse the conversion is initiated. Like the ALE pulse the minimum pulse width is 100 ns. The signal can be tied to the ALE signal when the clock frequency is below 500 kHz. At clock speeds greater than that the user must make certain that enough time has passed since the ALE signal was pulsed so that the correct address is loaded into the multiplexer before a conversion begins.

OE

The output enable signal causes the ADC to actually output the digital values on the output lines. The ADC stores the data in a tri-state output latch until the next conversion is started, but the data is only output when enabled.

EOC

The end of conversion is sent from the ADC. The signal goes low once a conversion is initiated by the start signal and remains low until a conversion is complete.

Interfacing of A/D converter ADC0808 (Program)

The connections are made as shown in Figure 8.23. The +5 V DC reference supply should be accurate. This is obtained from op-amp circuit. The I/O ports of intel 8255 has been used for interfacing ADC0808 to Intel 8051 microcontroller. 8255 indicates that the first unit of ports provided on the kit has been used. The ports are defined as follows:

Port A - Input

Port B - Output

Port C lower - Output

Port C upper - Input

The control word for this configuration of ports is 98H

The addresses for ports of 8255 are

Port A - E800

Port B - E801

Port C - E802

Control word register - E803

The output of ADC0808 is connected to Port A. The end of conversion pin is connected to pin PC7 of port C upper. The start of conversion pin and ALE pins are connected to pin PC3 of port C lower and the logic for multiplexer address C B A are connected to pins PC2, PC1, PC0 of port C lower. Multiplexer channel IN3 is selected for analog input voltage. ADC0808 is an 8-bit A/D converter with 8-channel multiplexer.

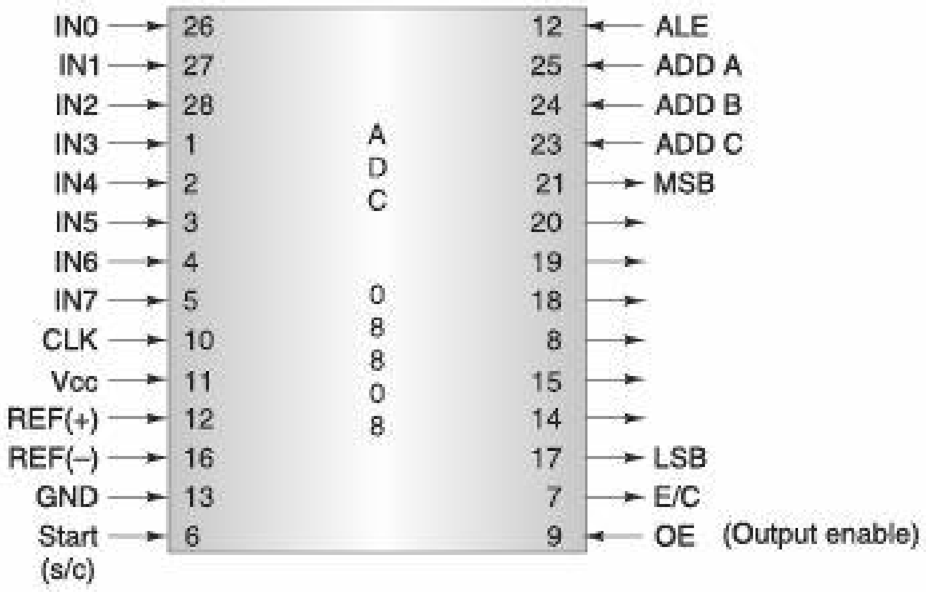


Fig. 8.23 Pin Diagram of ADC 0808.

Table 8.16 List of Addresses for Analog Channel

Analog	Address		
Channel	*C*	*B*	*A*
IN0	0	0	0
IN1	0	0	1
IN2	0	1	0

Contd...

Contd...

IN3	0	1	1
IN4	1	0	0
IN5	1	0	1
IN6	1	1	0
IN7	1	1	1

Analog to digital converter

An A/D converter is used to convert analog signals into digital quantity.

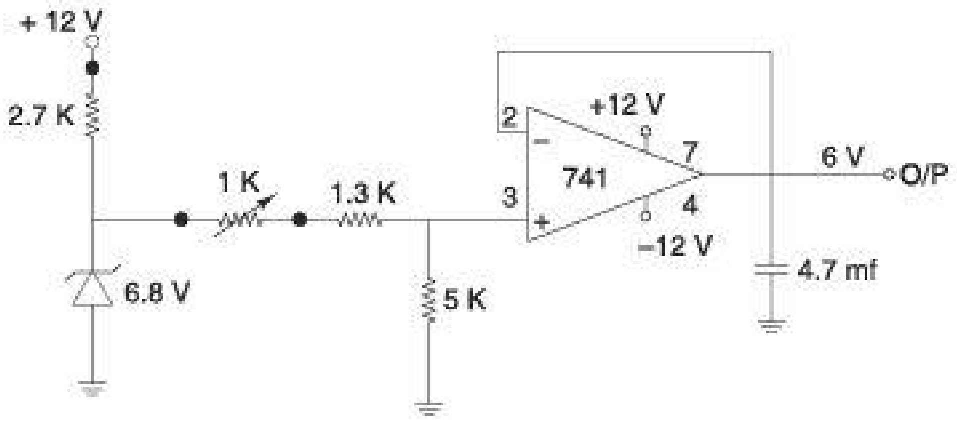


Fig. 8.24 ADC Using Op-amp.

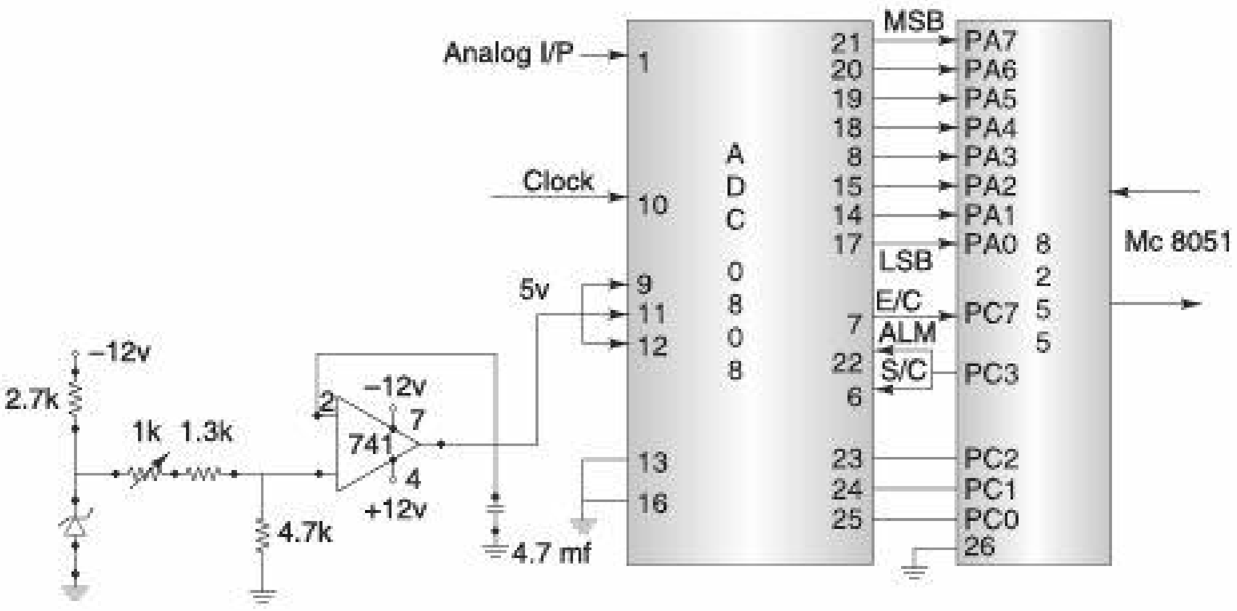


Fig. 8.25 Circuit Diagram.

PROGRAM

Mnemonic

```
        ORG 8000H
        MOV DPTR,#0E803H
        MOV A,#98H
        MOVX @DPTR,A
HERE:   MOV DPTR,#0E802H
        MOV A,#03H
        MOVX @DPTR,A
        MOV A,#0BH              ; Multiplexer Channel
        MOVX @DPTR,A
        MOV A,#03H
        MOVX @DPTR,A
        MOV B,#32H
LOOP:
        CLR C
READ:   MOVX A,@DPTR
        RLC A
        JNC READ
        MOV DPTR,#0E800H
        MOVX A,@DPTR
        MOV 60H,A
        LCALL 019BH             ; Display at Data Field.
        LJMP HERE
        END
```

8.18.6 Digital to Analog Converter

The DAC0800 series are monolithic 8-bit high-speed current O/P digital to analog converters. The DAC0800 features high compliance complementary current O/Ps to allow differential O/P voltages of 20 VP P with sample resistor loads.

The performance and characteristics of the device are essentially unchanged over the full ±4.5 V to ±18 V power supply range. Power dissipation is only 33 MW with ±5 V supplies and is independent of the logic input states.

Theoretical Calculation

The analog O/P voltage of the DAC is derived from the equation given below.

For 8-bit DAC

$$V0 = V_R/2^N[128A_7 + 64A_6 + 32A_5 + 16A_4 + 8A_3 + 4A_2 + 2A_1 + A_0]$$

where N = no. of bits, VR = reference voltage.

DAC 0800 is an 8-bit DAC and the output voltage variation is between – 5V and + 5V. The output voltage varies in steps of 10/256 = 0.04 (appx.). The digital data input and the corresponding output voltages are presented in Table 8.17.

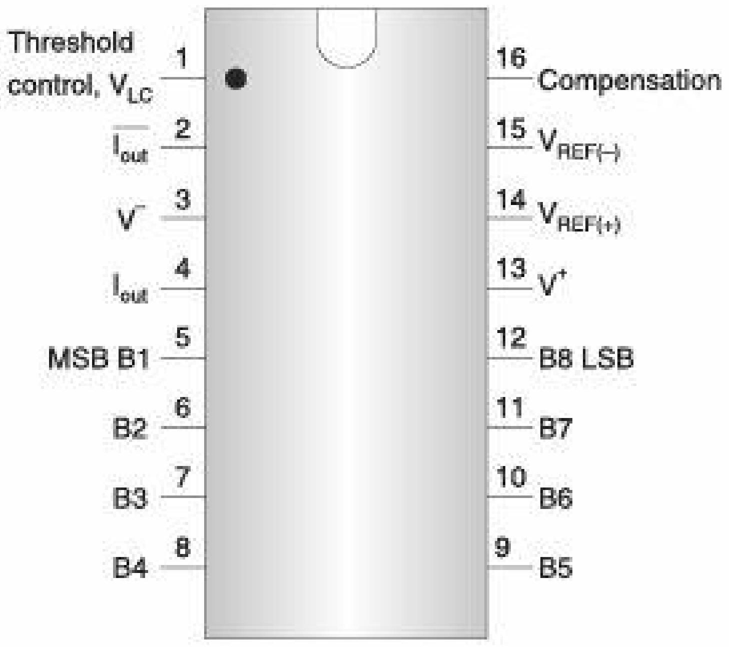


Fig. 8.26 Pin Diagram of 0800.

Table 8.17 List of I/P Data

Input data in Hex	*Ouput voltage*
00	5.00
01	4.96
02	4.92
...	.....
7F	0.00
....	.....
FD	4.92
FE	4.96
FF	5.00

Referring to Table 8.17, with 00 H as input to DAC, the analog output is –5 V. Similarly, with FF H as input, the output is +5 V. Outputting digital data 00 and FF at regular intervals, to DAC, results in different waveforms, namely, square, triangular, etc. Here the digital I/Ps are fed from microcontroller 8051 or from a switch. The DAC0800 is an 8-bit high-speed current O/P digital to analog converter which converts the digital I/Ps to analog voltages. The op-amp acts as a current to voltage converter. The readings practically got are shown below in the tabular column. When used as a multiplying DAC monotonic performance over a 40 to 1 reference current range is possible. The noise immune I/Ps of DAC0800 series will accept TTL levels with the logic threshold pin, VLC, grounded changing the VLC potential will allow direct interface to other logic families.

The DAC0800, DAC0802, DAC0800C and DAC0802C are a direct replacement for the DAC-08, DAC-08A, DAC-08C and DAC-08H respectively.

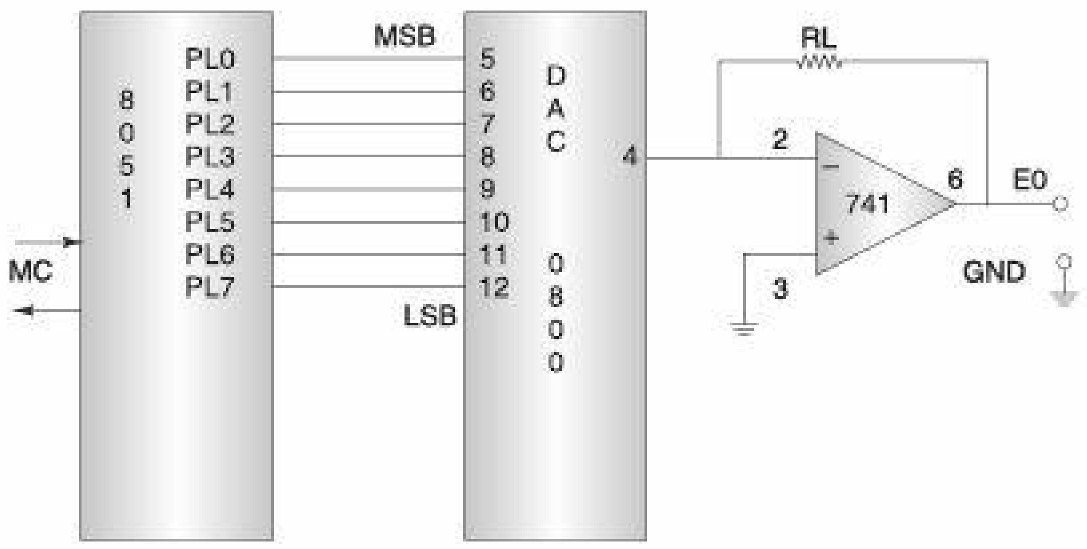


Fig. 8.27 DAC Interfacing With 8051.

PROGRAM

MNEMONIC COMMENT

```
    ORG 8000H
    BK: LCALL 02A2H          ; Call the Input Routine to Obtain Lower Nibble
    SWAP A                   ; Swap Lower Nibble
    MOV R7, A                ; Move the Contents of Accumulator to Register R7
    LCALL 02A2H              ; Call the Input Routine to Obtain Higher Nibble
    ORL A, R7                ; Or the Higher and Lower Nibble
    MOV 90H, A               ; move the contents of the accumulator to port1 of 8051
    MOV 60H, A               ; move the contents of accumulator to 60h location
    LCALL 019BH              ; Call the Display Routine
    LJMP BK
    END
```

PROGRAM TO GENERATE SQUARE WAVE

```
    MOV DPTR,#0E903H
    MOV A,#80H
    MOVX @DPTR,A
    MOV DPTR,#0E900H
    CONT: MOV A,#00H
    MOVX @DPTR,A
    LCALL DELAY
    MOV A,#0FFH
    MOVX @DPTR,A
    LCALL DELAY
    SJMP CONT
    END
```

PROGRAM TO GENERATE TRIANGULAR WAVE

```
MOV DPTR,#0E903H
MOV A,#80H
MOVX @DPTR,A
MOV DPTR,#0E900H
LOOP3: MOV A,#00H
LOOP: MOVX @DPTR,A
INC A
CJNE A,#0FFH,LOOP
LOOP2: DEC A
MOVX @DPTR,A
CJNE A,#00H,LOOP2
SJMP LOOP3
END
```

PROGRAM TO GENERATE SINE WAVE

```
ORG 8000H
MOV DPTR,#0E903H
MOV A,#80H
MOVX @DPTR,A
AGAIN: MOV DPTR,#TABLE
MOV R2,#24H
BACK:CLR A
MOVC A,@A+DPTR
PUSH 83H;
PUSH 82H;
MOV DPTR,#0E900H
MOVX @DPTR,A
POP 82H
POP 83H
INC DPTR
DJNZ R2,BACK
SJMP AGAIN
END
```

8.18.7 Interfacing of 8051 with External Memory

The system designer is not limited by the amount of internal ROM and RAM available on chip. Two external memory spaces are made available by the 16-bit PC and DPTR and different control pins for enabling external ROM and RAM chips. When a program is written in a high-level language, the program size exceeds 4 Kbytes and an external program is needed. External RAM, which is accessed by DPTR, may also be needed when 128 bytes of internal data storage is not sufficient. External RAM, up to 64 Kbytes, may also be added to any chip in 8051 family. The 8051 accesses external RAM when certain program instructions are executed. External ROM is accessed whenever (EA)' pin is connected to ground or when the PC contains an address higher than the last address in the internal 4 Kbytes ROM. During any memory access cycle, port 0 is time multiplexed, that is, it first provides the lower byte of the i6-bit memory address. The lower address byte from port 0 must be latched into an external register to save the byte. Address byte save is accomplished by the ALE clock pulse that provides the correct timing for the '373 type data latch. If the memory access for a byte of program code in the ROM, the (PSEN)' will go low to enable the ROM to place of program code on the data bus.

Interfacing is done by the following signals:

1. The (External Enable) pin is set to 1 when either internal ROM is also to be used or external memory chip is extending the internal ROM by providing new addresses. The (External Enable) pin is set to 0 when (i) internal ROM is not being used for program store and (ii) external memory chip all addresses from 0x0000 are used for program store.
2. AD_0-AD_7 bus signals are the inputs to a latch (74LS374). Latched output is used to connect A_0-A_7 pins of the ROM.

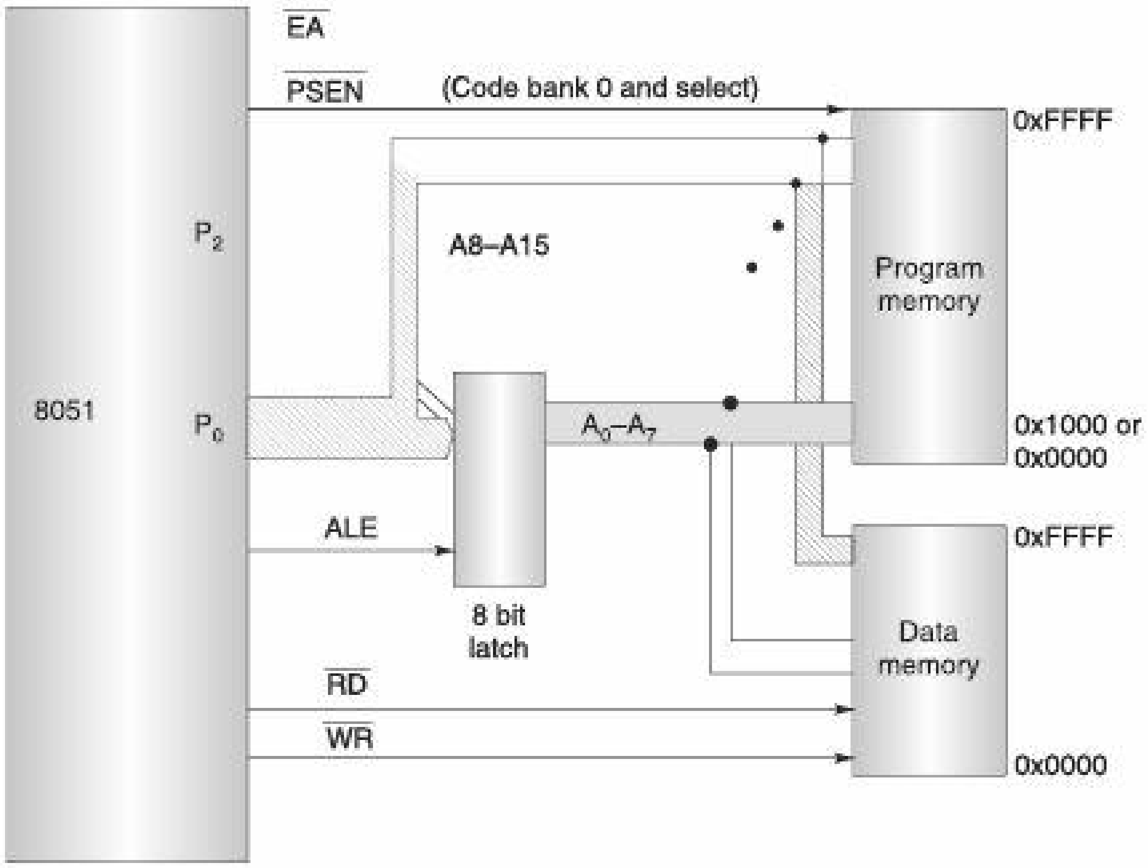


Fig. 8.28 Interfacing of External Memory with 8051.

3. ALE is connected to latch enable signal at latch.
4. Lower address bits from A0 to A7 are connected to ROM where m depends on capacity.
5. Higher address A8 to A15 bits from P2 are connected to a decoder and the decoder output connects the ROM.
6. Port P0 pins connect D0-D7 inputs of the ROM.
7. Port P0 pins connect D0-D7 inputs/outputs at 8-pins of RAM.
8. (P3.6) is connected to Out-Enable (read) Read pin of RAM and are connected to AND gate when RAM is used for control store (Program memory as well as data memory). Otherwise, is connected to pin of ROM and to pin of RAM.
9. Pin (P3.7) is connected to pin at the RAM. pin is connected to RAM. Pin also connects the program memory if flash or EEPROM needs to be written.

8.19 MICROCONTROLLER BASED APPLICATIONS

Metro Train Prototype

This microcontroller based application gives brief idea of how metro train, which is the major source of transport in metro cities. The train is equipped with the CPU which control the motion of the train. Basic component used in this application is

1. CPU
2. Stepper Motor
3. ULN 2003
4. LCD

The train is designed for five stations, named as Sirsa, Kherekan, Panjuana, Sahuwala Second, Panniwala Mota. The Stoppage time is of 3 sec and time between two consecutive stations is 6 sec. There is a LCD display for showing various messages in the train for passengers. There are indicators, which are used to show the train direction, i.e., Up path and Down path. Before stopping at station the train blows the buzzer. It also includes an emergency brake system due to which the train stops as soon as the brakes are applied and resumes journey when the emergency situation is over.

Message are displayed on the LCD Interface like as

WELCOME TO ALL	; when train starts its journey
CURRENT STATION	; when a station arrived
NEXT STATION	; displayed before arriving of next station

• Delay during the station is provided by the software ULN 2003

- Microcontroller pins lack sufficient current to drive the relay. They can provide a maximum of 1-2 mA current. While the stepper motor's coil needs around 10 mA to be energized. For this reason, we place a driver so that stepper motor can work efficiently.

Power supply

- Power supply to the this application is given by the employing bridge wave rectifier, and using 7805 IC.
- 7805 IC is a voltage regulator IC.
- Two led are used to show the forward and backward path.
- Lcd is interfaced to port 3.

Interfacing of 8051 with LCD, Stepper motor is previously defined.

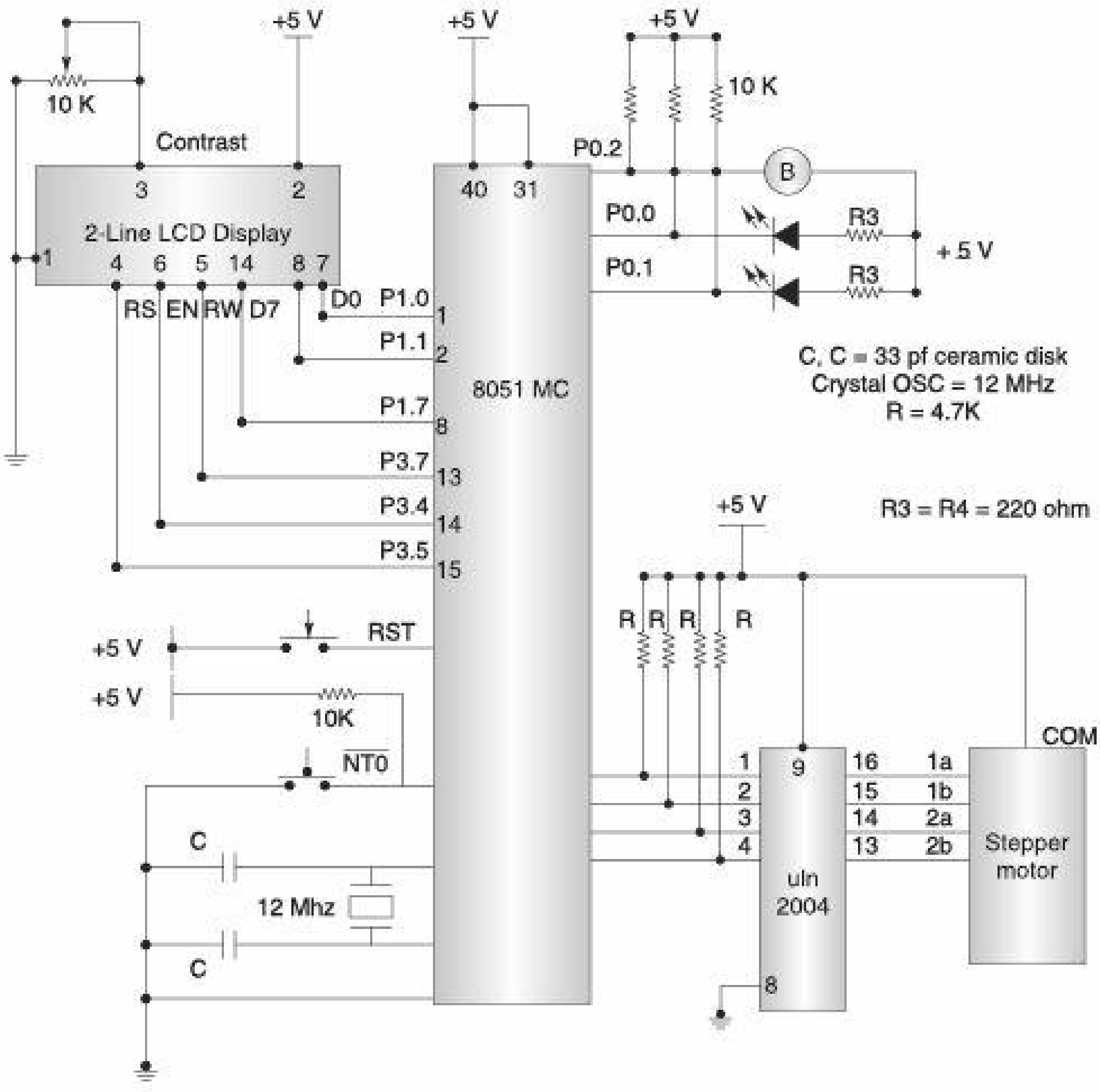


Fig. 8.29 Layout Diagram of Metro Train Program.

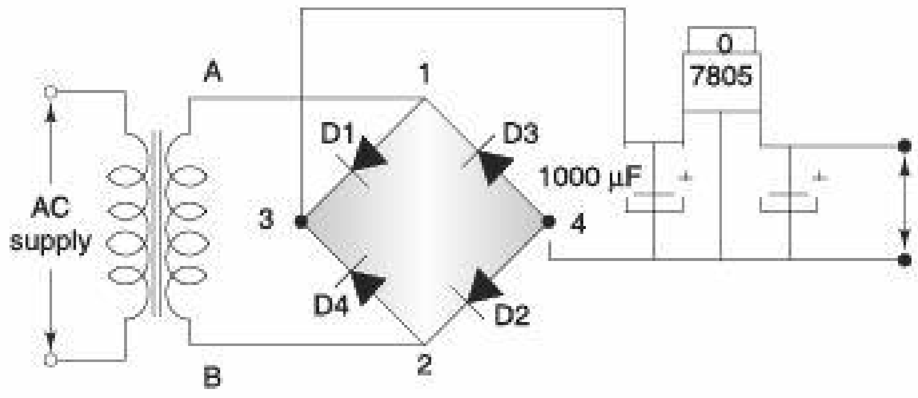


Fig. 8.30 Power Supply Used in Application.

Assembly language program for metro train

```
$MOD51
DATA1 EQU P1
BUSY EQU P1.7
RS EQU P3.5
RW EQU P3.4
EN EQU P3.3
BZR EQU P0.2
LEDF EQU P0.0
LEDR EQU P0.1
ORG 0000H
AJMP MAIN
ORG 0003H
TEST: MOV C,P3.2
JNC HALT
SETB BZR
RETI
HALT:
CLR BZR
AJMP TEST
MAIN:
SETB LEDR
MOV P2,#00H
SETB EA
SETB EX0;-------------------------------LCD STARTS---------------------------------------------
MOV DPTR,#SHOWI
ACALL INTI
ACALL DELAYS
```

```
MOV DPTR,#SHOWW
ACALL READ
ACALL DELAYS
CLR LEDF
ACALL DELAY1SEC
MOV A,#01H;-------------------CLEAR THE SCREEN------------------------------------------
-------------
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H;----------------------FORCE TO CURSOR BRING AT 2ND LINES FIFTH
POSITION--------------------
ACALL COMMAND
MOV DPTR,#SHOWS
ACALL READ
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWK
ACALL READ
ACALL DELAYS
ACALL STEPF
MOV A,#01H;------------------------TRAIN ARRIVES AT KHEREKAN-----------------------------
---------------
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWK
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWP
ACALL READ
ACALL STEPF
;----------------------------------------TRAIN ARRIVES AT PANJUANA--------------------------------
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWP
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWSA
ACALL READ
ACALL STEPF
```

```
;-------------------------------------TRAIN ARRIVES AT SAHUWALA--------------------------
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWSA
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWPA
ACALL READ
ACALL STEPF

;-------------------------TRAIN ARRIVES AT PANNIWALA MOTA--------------------------
MOV A,#01H;----------------------BACKWARD JOURNEY STARTS---------------------------
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWPA
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
```

```
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWSA
SETB LEDF
CLR LEDR
ACALL READ
ACALL STEPR
;---------------------BACK TO SAHUWALA-----------------------------------------
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWSA
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWP
ACALL READ
ACALL STEPR
;--------------------------------BACK TO PANJUANA---------------------------------------------
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWC
```

```
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWP
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWK
ACALL READ
ACALL STEPR
;------------------------------------BACK TO KHEREKAN-----------------------------------------
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWC
ACALL READ
MOV A,#0C0H
ACALL COMMAND
MOV DPTR,#SHOWK
ACALL READ
ACALL DELAY1SEC
ACALL DELAY1SEC
CLR BZR
ACALL DELAY1SEC
SETB BZR
MOV A,#01H
ACALL COMMAND
MOV DPTR,#SHOWN
ACALL READ
MOV A,#0C0H
```

```
ACALL COMMAND
MOV DPTR,#SHOWS
ACALL READ
ACALL STEPR;---------------------------BACK TO SIRSA-------------------------------------------
LJMP MAIN
;-------------------------COMMAND---------------------------------------------------------------------
COMMAND:
ACALL READY
MOV DATA1,A
CLR RS
CLR RW
SETB EN
ACALL DELAYS
CLR EN
RET
;---------------------------------------DISPLAY-------------------------------------------------------
DISPLAY:
ACALL READY
MOV DATA1,A
SETB RS
CLR RW
SETB EN
ACALL DELAYS
CLR EN
RET
;-----------------------------------------------------READY-------------------------------------------------
READY:
CLR RS
SETB RW
BACK: CLR EN
ACALL DELAYS
SETB EN
JB BUSY,BACK
RET
;-----------------------------------------------------INTI-------------------------------------------------
```

```
INTI:
CLR A
AGAIN:MOVC A,@A+DPTR
JZ LOOP2
ACALL COMMAND
CLR A
INC DPTR
SJMP AGAIN
LOOP2:
NOP
RET
;---------------------------------------------------READ--------------------------------------------
READ:
CLR A
AGAIN1:MOVC A,@A+DPTR
JZ LOOP3
ACALL DISPLAY
CLR A
INC DPTR
SJMP AGAIN1
LOOP3:
NOP
RET
;------------------------STEPPER MOTOR RUN FORWARD 6 SEC------------------------------
STEPF:
PUSH ACC
PUSH 00
PUSH 01
MOV A,#88H
MOV R0,#100
MOV TMOD,#01H
MOV TCON,#00H
LOOP4: MOV TH0,#15H
MOV TL0,#0A0H
SETB TR0
```

```
HERE: JNB TF0,AGAIN2
CLR TR0
CLR TF0
DJNZ R0,LOOP4
SJMP LOOP5
AGAIN2: MOV P2,A
RR A
ACALL DELAYS
SJMP HERE
LOOP5:
NOP
POP 01
POP 00
POP ACC
MOV P2,00H
RET
;------------------------STEPER MOTOR RUN BACKWARD 6 SEC------------------------------
STEPR:
PUSH ACC
PUSH 00
PUSH 01
MOV A,#88H
MOV R0,#100
MOV TMOD,#01H
MOV TCON,#00H
LOOP6: MOV TH0,#15H
MOV TL0,#0A0H
SETB TR0
HERE1: JNB TF0,AGAIN3
CLR TR0
CLR TF0
DJNZ R0,LOOP6
SJMP LOOP7
AGAIN3: MOV P2,A
RL A
ACALL DELAYS
SJMP HERE1
```

```
LOOP7:
NOP
POP 01
POP 00
POP ACC
MOV P2,00H
RET
;-----------------------------------DELAY FOR MOTOR SPEED-----------------------------------
--------------------
DELAYS:
PUSH 00
PUSH 01
MOV R0,#10H
LOOP8: MOV R1,#0AFH
AGAIN4: DJNZ R1,AGAIN4
DJNZ R0,LOOP8
POP 01
POP 00
RET
;------------------------------------DELAY FOR 1 SEC--------------------------------------
DELAY1SEC:
PUSH 00
PUSH 03
PUSH 04
MOV R0,#20
MOV TMOD,#01H
MOV TCON,#00H
LOOP61: MOV TH0,#3CH
MOV TL0,#0B0H
SETB TR0
HERE12: JNB TF0,HERE12
CLR TR0
CLR TF0
DJNZ R0,LOOP61
POP 04
POP 03
POP 00
```

```
RET
;---------------------------------------INITIALISATION IN ROM--------------------------------
ORG 295H
SHOWI: DB 38H,0EH,01H,06H,80H,0
SHOWW: DB 'WELCOME TO ALL',0
SHOWC: DB 'CURRENT STATION',0
SHOWN: DB 'NEXT STATION',0
SHOWS: DB 'SIRSA',0
SHOWK: DB 'KHEREKAN',0
SHOWP: DB 'PANJUANA',0
SHOWSA: DB 'SAHUWALA SECOND',0
SHOWPA: DB 'PANNIWALA MOTA',0
END
```

Things to Remember

◊ A microprocessor is the heart of a computer. On the other hand, microcontroller is designed to be all of that in one.

◊ Like the microprocessor, a microcontroller is a general-purpose device, but one which is meant to fetch data, perform limited calculations on that data, and control its environment based on those calculations.

◊ A microcontroller is a single-chip computer that is specifically manufactured for embedded computer control applications.

◊ The 8051 architecture consists of 8-bit CPU with registers A (the accumulator) and B, 16-bit program counter (PC) and data pointer (DPTR), 8-bit program status word (PSW), 8-bit stack pointer (SP), Internal ROM or EPROM (8751) of 0 (8031) to 4 K (8051).

◊ The 8051 contains 34 general-purpose, or working, registers. Two of these registers, A and B, comprise the mathematical core of the 8051 central processing unit (CPU). The other 32 are arranged as part of internal RAM in four banks, B0-B3, of eight registers each, named R0 to R7.

◊ The 8051 has four math flags that respond automatically to the outcomes of math operations and three general-purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired.

◊ The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly.

◊ The 8-bit stack pointer (SP) register is used by the 8051 to hold an internal RAM address that is called the "top of the stack".

◊ At this time the largest static RAMs available are 32 K in size; RAM can be expanded to 64 K by using two 32 K RAMs that are connected through address A14 of port 2.

◊ When SMO and SMI are set to 01b, SBUF becomes a 10-bit full-duplex receiver/transmitter that may receive and transmit data at the same time.

◊ A jump or call instruction can replace the contents of the program counter with a new program address number that causes program execution to begin at the code located at the new address.

Questions and Answers

1. What do you mean by a microcontroller? Compare 8051 family.

Answer: A device which contains the microprocessor with integrated peripherals like memory, serial ports, parallel ports, timer/counter, interrupt controller, data acquisition interfaces like ADC, DAC is called a microcontroller.

FEATURES	8051	8052	8031
ROM(on-chip program space in bytes)	4 K	8 K	0 K
RAM(bytes)	128	256	128
Timers	2	3	2
I/O Pins	32	32	32
Serial Port	1	1	1
Interrupt sources	6	8	6

2. Explain the 16-bit registers DPTR and SP of 8051?

Answer: DPTR: DPTR stands for data pointer. DPTR consists of a high byte (DPH) and a low byte (DPL). Its function is to hold a 16-bit address. It may be manipulated as a 16-bit data register or as two independent 8-bit registers. It serves as a base register in indirect jumps, lookup table instructions and external data transfer.

SP: SP stands for stack pointer. SP is a 8-bit wide register. It is incremented before the data is stored during PUSH and CALL instructions. The stack array can reside anywhere in on-chip RAM. The stack pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H.

3. Name the five interrupt sources of 8051?

Answer: The interrupts are:

	Vector address
• External interrupt 0:	IE0: 0003H
• Timer interrupt 0:	TF0: 000BH
• External interrupt 1:	IE1: 0013H
• Timer Interrupt 1:	TF1: 001BH
• Serial Interrupt	
Receive interrupt:	RI: 0023H
Transmit interrupt:	TI: 0023H

4. What is the function of accumulator in 8051. Also define the PSW, flags and pointers in 8051?

Answer: Accumulator (ACC) The accumulator register acts as an operand register, in case of some instructions. This may either be implicit or specified in the instruction. The ACC register has been allotted and address in the on-chip special function register bank.

B Register: The register is used to store one of the operands for multiply and divide instructions. In other instructions, it may just be used as a scratch pad. This register is considered a special function register.

Flags: Flags are 1-bit registers provided to store the results of certain program instructions. Other instructions can test the condition of the flags and make decisions based upon the flag states. In order that the flags may be conveniently addressed, they are grouped inside the program status word (PSW) and the power control (PCON) registers. The 8051 has four math flags that respond automatically to the outcomes of math operations and three general-purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired. The math flags include carry (C), auxiliary carry (AC), overflow (OV), and parity (P). User flags are named F0, GFO, and GF1; they are general-purpose flags that may be used by the programmer to record some event in the program. Note that all of the flags can be set and cleared by the programmer at will. The math flags, however, are also affected by math operations.

Program status word (PSW): This set of flags contains the status information and is considered one of the special registers. The PSW contains the math flags, user program flag F0, and the register select bits that identify which of the four general-purpose register banks is currently in use by the program.

Stack pointer (SP): The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly. The 8-bit stack pointer (SP) register is used by the 8051 to hold an internal RAM address that is called the "top of the stack." The address held in the SP register is the location in internal RAM where the last byte of data was stored by a stack operation. This 8-bit wide register is incremented before the data is stored onto the stack using push or call instructions. The register contains 8-bit stack top address. This stack may be defined anywhere in the on-chip 128 by the RAM. After reset, the SP register is initialized to 07. After each write to stack operation, the 8-bit contents of the operand are stored onto the stack after incrementing the SP register by one. Thus, if SP contains 07H, the forthcoming PUSH operation will store the data at address 08 H in the internal RAM. The SP content will be incremented to 08. The 8051 stack is not a top down data structure, like other Intel processors. This register has also been allotted an address in the special function register bank. When data is to be placed on the stack, the SP increments before storing data on the stack so that the stack grows up as data is stored. As data is retrieved from the stack, the byte is read from the stack, and then the SP decrements to point to the next available byte of stored data.

Data Pointer (DTPR): This 16-bit register contains a higher byte (DPH) and the lower byte (DPL) of a 16-bit external data RAM address. It is accessed as a 16-bit register or two 8-bit registers as specified above. It has been allotted two addresses in the special-function register bank for its two bytes DPH and DPL.

5. Explain the function of latches and drivers in 8051. Also discuss the timing and control unit?

Answer: Port 0 to 3 latches and drivers: These four latches and drivers pairs are allotted to each of the four on chip I/O ports. These latches have been allotted addresses in the special function register bank using the allotted address the user can communicate with these ports. These are identified as P0, P1, P2 and P3.

Serial data buffer: The serial data buffer internally contains two independent registers. One of them is a transmit buffer which is necessarily a parallel.

Timing and control unit: This unit derives all the necessary timing and control signals required for the internal operation of the circuit. It also derives control signal required for controlling the external system bus oscillator. This circuit generates the basic timing clock signal for the operation of the circuit using crystal oscillator.

6. Explain DJNZ instructions of intel 8051 microcontroller.

Answer:

(a) DJNZ Rn, rel Decrement the content of the register Rn and jump if not zero.

(b) DJNZ direct, rel

Decrement the content of direct 8-bit address and jump if not zero.

7. Explain the function of the pins PSEN and EA of 8051.

Answer: PSEN: PSEN stands for program store enable. In 8051 based system in which an external ROM holds the program code, this pin is connected to the OE pin of the ROM.

EA: EA stands for external access. When EA pin is connected to Vcc, program fetched to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. When the EA pin is grounded, all addresses fetched by program are directed to the external ROM/EPROM.

8. Explain the working of all the registers in 8051.

Answer: This register decodes the opcode of an instruction to be executed and gives information to the timing and control unit to generate necessary signals for the execution of the instruction.

EPROM and Program Address Register: These blocks provide an on chip EPROM and a mechanism to internally address it. Note that EPROM is not available in all 8051 versions.

RAM and RAM Address Register: This block provides internal 128 bytes of RAM and mechanism to address it internally.

ALU: The arithmetic and logic unit performs 8-bit arithmetic and logical operations over the operands held by the temporary registers TMPI and TMP2. Users cannot access these temporary registers.

SFR Register Bank: This is a set of special-function registers which can be addressed using their respective address which lie in the range 80 H to FFH. Finally, the interrupt, serial port and timer units control and perform their special functions under the control of the timing and control unit in serial out register. The other is called receive butter which is a serial in parallel out register. Loading a byte to the transmit buffer initiates serial transmission of that byte. The serial data buffer is identified as SBUF and is one of the special-function registers. If a byte is written to SBUF, it initiates serial transmission and if the SBUF is read, it reads received serial data.

Timer Register: These two 16-bit registers can be accessed as their lower and upper bytes. For example, TL0 represents the lower byte of the timing register 0, while TH0 represents higher bytes of the timing register 0. Similarly, TL1 and TH1 represents lower and higher bytes of timing register 1. All these registers can be accessed using the 4 addresses allotted to them which lie in the special-function registers. SFR address range, i.e. 80H to FFH.

Control Registers: The special-function registers IP, IE, TMOD, TCON, SCON and PCON contain control and status information for interrupt timer/counters and serial port. These registers have been allotted address in the SFR bank of 8051.

A AND B CPU REGISTERS: The 8051 contains 34 general-purpose and other registers. Two of these, registers A and B, comprise the mathematical core of the 8051 central processing unit (CPU). The other 32 are arranged as part of internal RAM in four banks, B0-B3, of eight registers each, named R0 to R7. The A (accumulator) register is the most versatile of the two CPU registers and is used for many operations, including addition, subtraction, integer multiplication and division, and Boolean bit manipulations. The A register is also used for all data transfers between the 8051 and any external memory. The B register is used with the A register for multiplication and division operations and has no other function other than as the location where data may be stored.

9. Explain the working of internal RAM and ROM.

Answer: **Internal RAM** is a general-purpose RAM having area above the bit area, from 30H to 7FH, addressable as bytes. Thirty-two bytes from address 00H to FFH that make up 32 working registers are organized as four banks of eight registers each. The four register banks are numbered 0 to 3 and are made up of eight registers named R0 to R7. Each register can be addressed by name (when its bank is selected) or by its RAM address. Thus, R0 of bank 3 is RO (if bank 3 is currently selected) or address 18H (whether bank 3 is selected or not). Bits RS0 and RS1 in the PSW determine which bank of registers is currently in use at any time when the program is running. Register banks not selected can be used as general purpose RAM. Bank 0 is selected upon reset. An addressable area of 16 bytes occupies RAM byte addresses 20H to 2FH, forming a total of 128 addressable bits. An addressable bit may be specified by its bit address of 00H to 7FH, or 8 bits may form any byte address from 20H to 2FH. Thus, for example, bit address 4FH is also bit 7 of byte address 29H. Addressable bits are useful when the program need only remember a binary event (switch on, light off, etc.).

INTERNAL ROM: The 8051 is organized so that data memory and program code memory can be in two entirely different physical memory entities. Each has the same address ranges. The structure of the internal RAM has been discussed earlier. A corresponding block of internal program code, contained in an internal ROM, occupies code address space 000H to 0FFFH. The PC is ordinarily used to address program code bytes from addresses 0000H to FFFFH. Program addresses higher than 0FFFH, which exceed the internal ROM capacity, will cause the

8051 to automatically fetch code bytes from external program memory. Code bytes can also be fetched exclusively from an external memory, addresses 0000H to FFFFH, by connecting the external access pin (EA pin 31 on the DIP) to ground. The PC does not care where the code is; the circuit designer decides whether the code is found totally in internal ROM, totally in external ROM, or in a combination of internal and external ROM.

10. Explain the contents of the accumulator after the execution of the following program segments:

```
MOV A,#3CH
MOV R4,#66H
ANL A,R4
```

Answer: A 3C
R4 66
A 24

11. State the function of RS1 and RS0 bits in the flag register of Intel 8051 microcontroller.

RS1, RS0 – Register bank select bits

RS1	*RS0*	*Register Bank Selected*
0	0	Register bank 0
0	1	Register bank 1
1	0	Register bank 2
1	1	Register bank 3

12. Write a program using 8051 assembly language to change the date 55H stored in the lower byte of the data pointer register to AAH using rotate instruction.

```
MOV DPL,#55H
MOV A, DPL
RL A
Label: SJMP Label
```

13. Write a short note on serial data input in 8051.

Answer: Computers must be able to communicate with other computers in modern multiprocessor distributed systems. One cost-effective way to communicate is to send and receive data bits serially. The 8051 has a serial data communication circuit that uses register SBUF to hold data. Register SCON controls data communication, register PCON controls data rates, and pins RXD (P3.0) and TXD (P3.1) connect to the serial data network. SBUF is physically two registers. One is writing only and is used to hold data to be transmitted out of the 8051 via TXD. The other is read only and holds received data from external sources via RXD. Both mutually exclusive registers use address 99H. There are four programmable modes for serial data communication that are chosen by setting the SMX bits in SCON. Baud rates are determined by the mode chosen.

14. How is data transmitted and received in microcontrollers? What are the different modes for data transmission in 8051?

Answer:

DATA TRANSMISSION: Transmission of serial data bits begins anytime data is written to SBUF. TI is set to a 1 when the data has been transmitted and signifies that SBUF is empty (for transmission purposes) and that another data byte can be sent. If the program fails to wait for the TI flag and overwrites SBUF while a previous data byte is in the process of being transmitted, the results will be unpredictable (a polite term for "garbage out").

DATA RECEPTION: Reception of serial data will begin if the receive enable bit (REN) in SCON is set to I for all modes. In addition, for mode 0 only, RI must be cleared to 0 also. Receiver interrupt flag RI is set after data has been received in all modes. Setting REN is the only direct program control that limits the reception of unexpected data; the requirement that RI also be 0 for mode 0 prevents the reception of new data until the program has dealt with the old data and reset RI. Reception can begin in modes 1, 2, and 3 if RI is set when the serial stream of bits begins. RI must have been reset by the program before the last bit is received or the incoming data will be lost. Incoming data is not transferred to SBUF until the last data bit has been received so that the previous transmission can be read from SBUF while new data is being received.

SERIAL DATA TRANSMISSION MODES: The 8051 designers have included four modes of serial data transmission that enable data communication to be done in a variety of ways and a multitude of baud rates. Modes are selected by the programmer by setting the mode bits SMO and SMI in SCON. Baud rates are fixed for mode 0 and variable, using timer I and the serial baud rate modify bit (SMOD) in PCON, for modes 1, 2, and 3.

15. Give the alternate functions for the port pins of port 3.

Answer:

WR	WR	WR	WR	WR	WR	WR	WR

RD – Read data control output.

WR – Write data control output.

T1 – Timer/Counter1 external input or test pin.

T0 – Timer/Counter0 external input or test pin.

INT1 – Interrupt 1 input pin.

INT 0 – Interrupt 0 input pin.

TXD – Transmit data pin for serial port in UART mode.

RXD – Receive data pin for serial port in UART mode.

16. Name the special-function registers available in 8051.

Answer:

(a) Accumulator

(b) B Register

(c) Program Status Word.

(d) Stack Pointer.
(e) Data Pointer.
(f) Port 0
(g) Port 1
(h) Port 2
(i) Port 3
(j) Interrupt priority control register.
(k) Interrupt enable control register.

17. Write a short note on EXTERNAL INTERRUPTS in Microcontrollers.

Answer: Pins INTO and INT 1 are used by external circuitry. Inputs on these pins can set the interrupt flags IE0 and IE1 in the TCON register to 1 by two different methods. The IEX flags may be set when the INTX pin signal reaches a low level, or the flags may be set when a high-to-low transition takes place on the INTX pin. Bits IT0 and IT1 in TCON program the INTX pins for low-level interrupt when set to 0 and program the INTX pins for transition interrupt when set to 1.

Flags IEX will be reset when a transition-generated interrupt is accepted by the processor and the interrupt subroutine is accessed. It is the responsibility of the system designer and programmer to reset any level-generated external interrupts when they are serviced by the program. The external circuit must remove the low level before an RETI is executed. Failure to remove the low will result in an immediate interrupt after RETI, from the same source.

18. What are the different addressing modes in microcontroller?

Answer: 8051 Addressing Modes

An "addressing mode" refers to how you are addressing a given memory location. In summary, the addressing modes are as follows, with an example of each: Immediate Addressing MOV A,#20h

Direct Addressing MOV A,30h

Indirect Addressing MOV A,@R0

External Direct MOVX A,@DPTR

Code Indirect MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility.

Immediate Addressing: Immediate addressing is so named because the value to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory. For example, the instruction: MOV A,#20h. This instruction uses Immediate Addressing because the Accumulator will be loaded with the value that immediately follows; in this case 20 (hexadecimal). Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

The source operand is a constant and the immediate data must be preceded by the pound sign, "£" can load information into any registers, including 16-bit DPTR register. The DPTR register can also be accessed as two 8-bit registers, the high byte DPH and low byte DPL.

```
MOV A, #38 H ; load 38 H into A
MOV R3, #72 ; load 72 into R3
MOV B, #40 H ; load 40 H into B
MOV DPTR, #4382 H ; DPTR = 4382 H
MOV DPL, # 82 H ; this is the same
MOV DPH, #43 H ; as above
```

Direct Addressing: Direct addressing is so named because the value to be stored in memory is obtained by directly retrieving it from another memory location. For example:

```
MOV A,30h
```

This instruction will read the data out of internal RAM address 30 (hexadecimal) and store it in the accumulator. Direct addressing is generally fast since, although the value to be loaded isn't included in the instruction, it is quickly accessible since it is stored in the 8051's Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address which may be variable. Also, it is important to note that when using direct addressing any instruction which refers to an address between 00h and 7Fh is referring to Internal Memory. Any instruction which refers to an address between 80h and FFh is referring to the SFR control registers that control the 8051 microcontroller itself.

Indirect Addressing: Indirect addressing is a powerful addressing mode which in many cases provide an exceptional level of flexibility. Indirect addressing is also the only way to access the extra 128 bytes of Internal RAM found on an 8052. Indirect addressing appears as follows:

```
MOV A,@R0
```

This instruction causes the 8051 to analyze the value of the R0 register. The 8051 will then load the accumulator with the value from Internal RAM which is found at the address indicated by R0.

External Direct: External Memory is accessed using a string of instructions which use what I call "External Direct" addressing. I call it this because it appears to be direct addressing, but it is used to access external memory rather than internal memory. There are only two commands that use External Direct addressing mode:

```
MOVX A,@DPTR
MOVX @DPTR,A
```

As you can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory that you want to read or write. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the accumulator. The second command will do the opposite: it will allow you to write the value of the accumulator to the external memory address pointed to by DPTR.

External Indirect: External memory can also be accessed using a form of indirect addressing which I call External Indirect addressing. This form of addressing is usually only used in relatively small projects that have a very small amount of external RAM. An example of this addressing mode is:

```
MOVX @R0,A.
```

Once again, the value of R0 is first read and the value of the accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh, the project would effectively be limited to 256 bytes of External RAM. There are relatively simple hardware/software tricks that can be implemented to access more than 256 bytes of memory using External Indirect addressing; however, it is usually easier to use External Direct addressing if your project has more than 256 bytes of External RAM.

External Memory is accessed using a string of instructions which use what I call "External Direct" addressing. I call it this because it appears to be direct addressing, but it is used to access external memory rather than internal memory. There are only two commands that use External Direct addressing mode:

```
MOVX A,@DPTR
MOVX @DPTR,A
```

As you can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory that you wish to read or write. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the accumulator. The second command will do the opposite: it will allow you to write the value of the accumulator to the external memory address pointed to by DPTR.

External Indirect: External memory can also be accessed using a form of indirect addressing which I call External Indirect addressing. This form of addressing is usually only used in relatively small projects that have a very small amount of external RAM. An example of this addressing mode is: MOVX @R0,A. Once again, the value of R0 is first read and the value of the accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh, the project would effectively be limited to 256 bytes of External RAM. There are relatively simple hardware/software tricks that can be implemented to access more than 256 bytes of memory using External Indirect addressing; however, it is usually easier to use External Direct addressing if your project has more than 256 bytes of External RAM.

19. How the RS-232C serial bus is interfaced to TTL logic device?

Answer: The RS-232C signal voltage levels are not compatible with TTL logic levels.

Hence, for interfacing TTL devices to RS- 232C serial bus, level converters are used.

The popularly used level converters are MC 1488 and MC 1489 or MAX 232.

20. What is HS0 of 8096?

Answer: HS0:

The High Speed Output unit (HSO) is used to trigger events at specific times with minimal CUP overhead. These events include: starting an A to D conversion, resetting Timer 2, setting 4 software flags, and switching up to 6 output lines.

21. Write a program to mask the 0th &7th bit using 8051.

Answer:

```
MOV A,#data
ANL A,#81
MOV DPTR,#4500
```

```
MOVX @DPTR,A
LOOP SJMP LOOP
```

22. Write a procedure for incrementing and decrementing in 8051.

Answer: The simplest arithmetic operations involve adding or subtracting a binary 1 and a number. These simple operations become very powerful when coupled with the ability to repeat the operation, that is, to "Increment" or "Decrement"—until the desired result is reached. Register, Direct, and Indirect addresses may be Incremented or Decremented. No math flags (C, AC, and OV) are affected. The following table lists the increment and decrement mnemonics.

Mnemonic	**Operation**
INC A	; Add a one to the A register
INC Rr	; Add a one to register Rr
INC add	; Add a one to the direct address
INC @Rp	; Add a one to the contents of the address in Rp
INC DPTR	; Add a one to the 16-bit DPTR
DEC A	; Subtract a one from register A
DEC Rr	; Subtract a one from register Rr
DEC add	; Subtract a one from the contents of the direct address
DEC @Rp	; Subtract a one from the contents of the address in register Rp

1. The 8051 microcontroller is of ___ pin package as a ______ processor. a) 30, 1byte b) 20, 1 byte c) 40, 8 bit d) 40, 8 byte
2. The SP is of ___ wide register. And this may be defined anywhere in the ______.
   (a) 8 byte, on-chip 128 byte RAM (b) 8-bit, on chip 256 byte RAM.
   (c) 16 bit, on-chip 128 byte ROM (d) 8-bit, on chip 128 byte RAM.
3. After reset, SP register is initialized to address________.
   (a) 8H (b) 9H (c) 7H (d) 6H
4. What is the address range of SFR Register bank?
   (a) 00H-77H (b) 40H-80H (c) 80H-7FH (d) 80H-FFH
5. Which pin of port 3 has an alternative function as write control signal for external data memory?
   (a) P3.8 (b) P3.3 (c) P3.6 (d) P3.1
6. What is the Address (SFR) for TCON, SCON, SBUF, PCON and PSW respectively?
   (a) 88H, 98H, 99H, 87H, 0D0H. (b) 98H, 99H, 87H, 88H, 0D0H
   (c) 0D0H, 87H, 88H, 99H, 98H (d) 87H, 88H, 0D0H, 98H, 99H
7. Match the following:
   1. TCON (i) contains status information
   2. SBUF (ii) timer/counter control register.
   3. TMOD (iii) idle bit, power down bit
   4. PSW (iv) serial data buffer for Tx and Rx.
   5. PCON (v) timer/counter modes of operation.
      (a) 1->ii, 2->iv, 3->v, 4->i, 5->iii. (b) 1->i, 2->v, 3->iv, 4->iii, 5->ii.
      (c) 1->v, 2->iii, 3->ii, 4->iv, 5->i. (d) 1->iii, 2->ii, 3->i, 4->v, 5->iv.

8. Which of the following is of bit operations?
   (i) SP (ii) P2 (iii) TMOD (iv) SBUF
   (v) IP
   (a) ii, v only (b) ii, iv, v only (c) i, v only (d) iii, ii only
9. Serial port interrupt is generated, if ____ bits are set
   (a) IE (b) RI, IE (c) IP, TI (d) RI, TI
10. In 8051 which interrupt has highest priority?
    (a) IE1 (b) TF0 (c) IE0 (d) TF1
11. Intel 8096 is of ___ bit microcontroller family called ______.
    (a) 8, MCS51 (b) 16, MCS51 (c) 8, MCS96 (d) 16, MCS96
12. 8096 has following features fill up the following,
    (i) ____ Register file,(ii) ____ I/O Ports (iii) ____ architecture.
    (a) 256 byte, five 8-bit, register to register
    (b) 256 byte, four 8-bit, register to register
    (c) 232 byte, five 8-bit, register to register
    (d) 232 byte, six 8-bit, register to register

Exercise

1. Explain the architecture and operational features of 8051.
2. Describe the memory and I/O addressing of 8051.
3. Describes the interrupt structure of 8051 microcontroller.
4. Explain the following instructions of 8051 with examples.
   CJNE1)
   MUL2) AB
   RRC3)A
   SW4)AP A.
5. Explain the RAM memory space allocation in 8051.
6. List the salient features of 8051 microcontroller.
7. Explain the different modes of operation of hardware timer in 8051 microcontroller indicating the register associated.
8. Explain the different modes of operation by serial part in 8051 in detail with its associated registers.
9. What is the significance of the transmit flag, TI, when it is cleared to 0? When set to 1?
10. The 8051 address bus is 16 bits wide and the data bus is 8-bits wide. What is the maximum size for the external CODE memory or DATA memory? Show your calculation.
11. What is the function of the PC register, i.e., the Program Counter?
12. Explain how the hardware RESET signal works and what memory location (reset vector) is used by the RESET scheme in the 8051 microcomputer.
13. There is no STOP instruction in the 8051 instruction. Describe a method for implementing program stop.

14. If the instruction cycle time in a 8051 microcomputer is 1 microsecond, suggest a simple programming method of delaying for, say, six microseconds.
15. Explain what the CJNE instruction does and show an example program line using CJNE instruction.
16. Does asynchronous communication between two microprocessors have to be done at standard baud rates? Name one reason why you might wish to use standard rates.
17. Write a test program that will "loop test" the serial port. The output of the serial port (TXD) is connected to the input (RXD), and the test program is run. Success is indicated by port I pin 1 going high.
18. Why is there no check for an initial timing value of 0000H in the program named "Hard time"?
19. Write a lookup table program, using the PC as the base that finds a one byte square root (to the nearest whole integer) of any number placed in A. For example, the square roots of 01 and 02 are both 01, while the roots of 03 and 04 are 02. Calculate the first four and last four table values.
20. Write a lookup table program, using the DPTR as the base that finds a two-byte square root of the number in A. The first byte is the integer value of the root, and the second byte is the fractional value. For example, the square root of 02 is 01,6AH. Calculate four first and last table values.
21. Write a lookup table program that converts the hex number in A (0-F) to its ASCII equivalent.
22. A PC based lookup table, which contains 256d values, is placed 50H bytes after the MOVC instruction that accesses it. Construct the table, showing where the byte associated with A = 00H is located. Find the largest number which can be placed in A to access the table.
23. Construct a lookup table program that converts the hex number in A into an equivalent BCD number in registers R4 (MSB) and R5 (LSB).
24. Outline a scheme for single-stepping the 8051 using a combination of hardware and software. (Hint: Use an INTX.)
25. While running the EPROM test, it is found that the program cannot jump from 2000H to 4000H successfully. Determine what address line(s) is faulty.
26. Calculate the error for the delay program "Softime" when values of 2d, 10d and 1000d milliseconds are passed in A and B. The program "Softime" has a bug. When A = 00h the delay becomes: (B + 1)d × 256d × delay. Find the bug and fix it without introducing a new bug.
27. Find the shortest and longest delays possible using "Softime" by changing only the equate value of the variable delay.
28. Give a general description of how you would test any time delay program. (Hint: use a port pin.)
29. In the discussion for the program named "Timer," the statement is made that an accurate 1 ms delay cannot be done due to the need for a count of 1333.33 using a 16 megahertz

clock. Find a way to generate an accurate 60-second delay using T0 for the basic delay and some registers to count the T0 overflows.

30. Calculate the shortest and longest delays possible using the program named "Timer" by changing the initial value of T0.
31. Determine whether the 8051 can be made to execute a single program instruction (single-stepped) using external circuitry (no software) only.
32. Which bits in which register(s) must be set to make INT0 level activated, and INT1 edge triggered?
33. Reverse Problem 15 and write a lookup table program that takes the BCD number in R4 (MSB) and R5 (LSB) and converts it into a hex number in A.
34. Verify the errors listed for the 16 megahertz crystal in the third comment after the program named "Sendchar."
35. Verify the error listed for the 11.059 megahertz crystal in the fourth comment after the program named "Sendchar."
36. Does asynchronous communication between two microprocessors have to be done at standard baud rates? Name one reason why you might wish to use standard rates.
37. Write a test program that will "loop test" the serial port. The output of the serial port (TXD) is connected to the input (RXD), and the test program is run. Success is indicated by port I pin 1 going high.
38. What is the significance of the transmit flag, TI, when it is cleared to 0? When set to 1?
39. The 8051 address bus is 16 bits wide and the data is 8-bits wide. What is the maximum size for the external CODE memory or DATA memory? Show your calculation.
40. What is the function of the PC register, i.e., the Program Counter?
41. Explain how the hardware RESET signal works and what memory location (reset vector) is used by the RESET scheme in the 8051 microcomputer.
42. There is no STOP instruction in the 8051 instruction. Describe a method for implementing program stop.
43. If the instruction cycle time in a 8051 microcomputer is 1 microsecond, suggest a simple programming method of delaying for, say, six microseconds.
44. Explain what the CJNE instruction does and show an example program line using CJNE instruction.

Chapter 9

Introduction to ARM and PIC Microcontrollers

- Introduction
- ARM Instruction Set
- ARM 7 Programming Model
- Stacks
- Exception and Interrupt Modes
- The ARM I/O System
- The PIC Microcontroller Family
- The AVR Microcontroller
- AVR Family Architecture
- AT90S8515 Microcontroller
- Program and Data Addressing Modes
- Instruction Set

9.1 INTRODUCTION

The ARM processor is a Reduced Instruction Set Computer (RISC). The RISC concept originated in processor research programs at Stanford and Berkeley universities around 1980. The ARM was originally developed at Acorn Computers Limited of Cambridge, England, between 1983 and 1985 and ARM stood for Acorn RISC Machine. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. The principal features of the ARM architecture are presented here in overview form.

In 1990, ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology, since when the ARM has been licensed to many semiconductor manufacturers around the world. It has become established as a market-leader for low power and cost-sensitive embedded applications.

No processor is particularly useful without the support of hardware and software development tools. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modeling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

Later, after a judicious modification of the acronym expansion to Advanced RISC Machine, it lent its name to the company formed to broaden its market beyond Acorn's product range. Despite the change of name, the architecture still remains close to the original Acorn design.

9.1.1 ARM Architecture

At the time the first ARM chip was designed, the only examples of RISC architectures were the Berkeley RISC I and II and the Stanford MIPS (which stands for Microprocessor without Interlocking Pipeline Stages), although some earlier machines such as the Digital PDP-8, the Cray-1 and the IBM 801, which predated the RISC concept, shared many of the characteristics which later came to be associated with RISCs.

The ARM architecture incorporated a number of features from the Berkeley RISC design, but a number of other features were rejected. Those that were used were:

- a load-store architecture
- fixed-length 32-bit instructions
- 3-address instruction formats.
- Control over both the *Arithmetic Logic Unit* (ALU) and shifter in most data- processing instructions to maximize the use of an ALU and a shifter.
- Auto-increment and auto-decrement addressing modes to optimize program loops
- Uniform and fixed-length instruction fields, to simplify instruction decode.
- Load and Store Multiple instructions to maximize data throughput.

The principal feature of the ARM 7 microcontroller is that it is a register based load and-store architecture with a number of operating modes. While the ARM7 is a 32-bit microcontroller, it is also capable of running a 16-bit instruction set, known as "THUMB". This helps it to achieve a greater code density and enhanced power saving. While all of the register-to-register data processing instructions are single-cycle, other instructions such as data transfer instructions, are multi-cycle. To increase the performance of these instructions, the ARM 7 has a three-stage pipeline. Due to the inherent simplicity of the design and low gate count, ARM 7 is the industry leader in low-power processing on a watts per MIP basis. Finally, to assist the developer, the ARM core has a built-in JTAG debug port and on-chip "embedded ICE" that allows programs to be downloaded and fully debugged in-system.

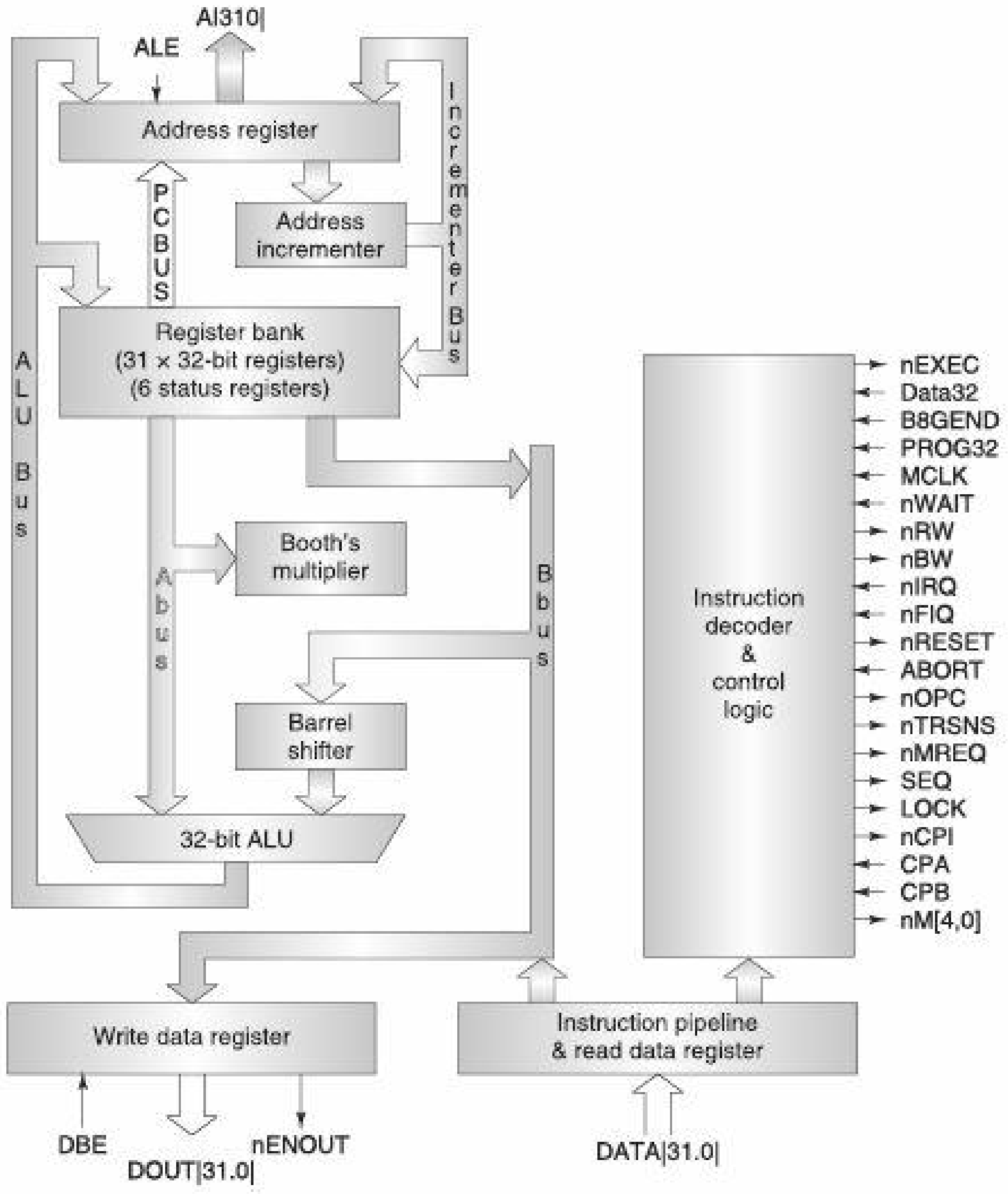


Fig. 9.1 Arm Architecture.

Instruction Decoder and Logic Control: The function of instruction decoder and logic control is to decode instructions and generate control signals to other parts of processor for execution of instructions.

Address Register: To hold a 32-bit address for address bus.

Address Increment: It is used to increment an address by four and place it in address register.

Register Bank: Register bank contains thirty-one 32-bit registers and six status registers.

Barrel Shifter: It is used for fast shift operation.

ALU: 32-bit ALU is used for Arithmetic and Logic Operation.

Write Data Register: The processor put the data in Write Data Register for write operation.

Read Data Register: When the processor reads from memory it places the result in this register.

In order to keep the ARM 7 both simple and cost-effective, the code and data regions are accessed via a single data bus. Thus, while the ARM 7 is capable of single-cycle execution of all data processing instructions, data transfer instructions may take several cycles since they will require at least two accesses onto the bus (one for the instruction one for the data). In order to improve performance, a three-stage pipeline is used that allows multiple instructions to be processed simultaneously.

The pipeline has three stages: FETCH, DECODE and EXECUTE. The hardware of each stage is designed to be independent so up to three instructions can be processed simultaneously. The pipeline is most effective in speeding up sequential code. However, a branch instruction will cause the pipeline to be flushed marring its performance. As we shall see later the ARM 7 designers had some clever ideas to solve this problem.

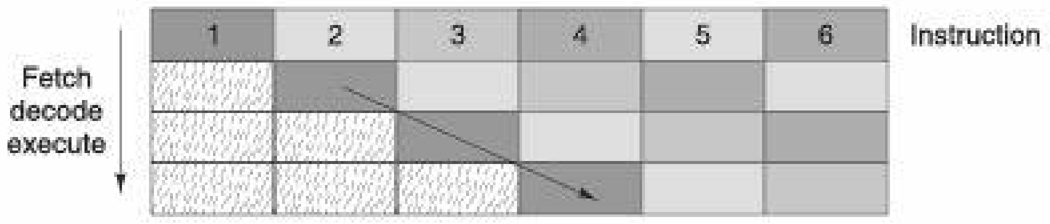


Fig. 9.2 ARM 3-Stage Pipeline.

9.2 ARM INSTRUCTION SET

The ARM instruction set can be divided into six broad classes of instruction:

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and store instructions
- Coprocessor instructions
- Exception-generating instructions.

Most data-processing instructions and one type of coprocessor instruction can update the four-condition code flags in the CPSR (Negative, Zero, Carry and Over flow) according to their result. Almost all ARM instructions contain a 4-bit *condition* field. One value of this field specifies that the instruction is executed unconditionally. Fourteen other values specify *conditional execution* of the instruction. If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing. The 14 available conditions allow:

- tests for equality and non-equality
- tests for <, <=, >, and >= inequalities, in both signed and unsigned arithmetic
- each condition code flag to be tested individually.

The sixteenth value of the condition field encodes alternative instructions. These do not allow conditional execution. Before ARMv5 these instructions were UNPREDICTABLE.

Table 9.1 Conditional Suffix

Suffix	*Description*	*Flag Tested*
EQ	Equal	Z = 1
NE	Not Equal	Z = 0
CS/HS	Unsigned higher or same	C = 1
CS/LO	Unsigned lower	C = 0
MI	Minus	N = 1
PL	Positive or zero	N = 0
VS	Overflow	V = 1
VC	Not Overflow	V = 0
HI	Unsigned higher	C = 1 & Z = 0
LS	Unsigned lower same	C = 0 & Z = 1
GE	Greater or equal	N = V
LT	Less than	N¡ = V
GT	Greater than	Z = 0 & N =V
LE	Less than or equal	Z = 1 & N =¡V
AL	Always	

9.2.1 Branch Instructions

As well as allowing many data-processing or load instructions to change control flow by writing the PC, a standard Branch instruction is provided with a 24-bit signed word offset, allowing forward and backward branches of up to 32 MB. There is a Branch and Link (BL) option that also preserves the address of the instruction after the branch in R14, the LR. This provides a subroutine call which can be returned from by copying the LR into the PC.

Syntax:=

1. Branch : `B{<cond>} label`
2. Branch with Link : `BL{<cond>} sub_routine_label`

The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank, i.e., the address of the next instruction following the branch with link (allowing for the pipeline).

To return from subroutine, simply need to restore the PC from the LR:

- MOV pc, lr
- Again, pipeline has to refill before execution continues.

The "Branch" instruction does not affect LR.

9.2.2 Data-processing Instructions

The data-processing instructions perform calculations on the general-purpose registers. There are five types of data-processing instructions:

- Arithmetic/logic instructions
- Comparison instructions
- Single Instruction Multiple Data (SIMD) instructions
- Multiply instructions
- Miscellaneous Data Processing instructions

9.2.3 Arithmetic/Logic Instruction

The following arithmetic/logic instructions share a common instruction format. These perform an arithmetic or logical operation on up to two source operands, and write the result to a destination register. They can also optionally update the condition code flags, based on the result.

Of the two source operands:

- one is always a register
- the other has two basic forms:
  - — an immediate value
  - — a register value, optionally shifted.

If the operand is a shifted register, the shift amount can be either an immediate value or the value of another register. Five types of shift can be specified. Every arithmetic/logic instruction can therefore perform an arithmetic/logic operation and a shift operation. As a result, ARM does not have dedicated shift instructions.

Arithmetic operations

• ADD	: operand1 + operand2
• ADC	: operand1 + operand2 + carry
• SUB	: operand1 - operand2
• SBC	: operand1 - operand2 + carry -1
• RSB	: operand2 - operand1
• RSC	: operand2 - operand1 + carry – 1

Syntax: `<Operation>{<cond>}{S} Rd, Rn, Operand2`

Examples

- ADD r0, r1, r2
- SUBGT r3, r3, #1
- RSBLES r4, r5, #5

Logical operations

Syntax:	<Operation>{<cond>}{S} Rd, Rn, Operand2
• AND	: operand1 AND operand2
• EOR	: operand1 EOR operand2
• ORR	: operand1 OR operand2
• BIC	: operand1 AND NOT operand2 [i.e., bit clear]

• Logical Shift Right

Shifts right by the specified amount (divides by powers of two), e.g., LSR #5 = divide by 32

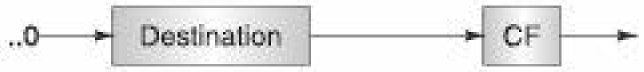


Fig. 9.3 Logical Shift Right.

• Arithmetic Shift Right

Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations.

e.g., ASR #5 = divide by 32

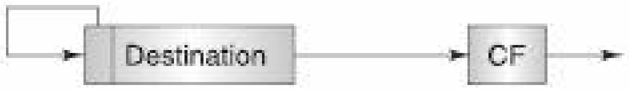


Fig. 9.4 Arithmetic Shift Right.

• Rotate Right (ROR)

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB. e.g. ROR #5

Note the last bit rotated is also used as the Carry Out.

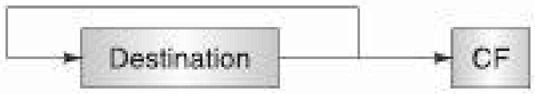


Fig. 9.5 Rotate Right.

• Rotate Right Extended (RRX)

This operation uses the CPSR C flag as a 33rd bit. Rotates right by 1 bit. Encoded as ROR #0.

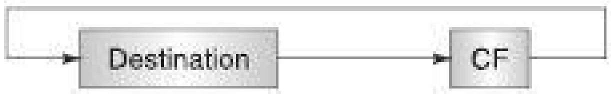


Fig. 9.6 Rotate Right Extended.

9.2.4 Comparison Instructions

The comparison instructions use the same instruction format as the arithmetic/logic instructions. These perform an arithmetic or logical operation on two source operands, but do not write the result to a register. They always update the condition flags, based on the result.

The source operands of comparison instructions take the same forms as those of arithmetic/logic instructions, including the ability to incorporate a shift operation.

The only effect of the comparisons is to

- UPDATE THE CONDITION FLAGS.

Operations are:

• CMP	: operand1 – operand2, but result not written
• CMN	: operand1 + operand2, but result not written
• TST	: operand1 AND operand2, but result not written
• TEQ	: operand1 EOR operand2, but result not written
Syntax:	<Operation>{<cond>} Rn, Operand2

Examples: CMP r0, r1, TSTEQ r2, #5

9.2.5 Multiplication Instructions

The Basic ARM provides two multiplication instructions.

Multiply

- MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs

  Multiply Accumulate - does addition for free
- MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn

  Restrictions on use:
- Rd and Rm cannot be the same register can be avoid by swapping Rm and Rs around. This works because multiplication is commutative.
- Cannot use PC These will be picked up by the assembler if overlooked. Operands can be considered signed or unsigned
- Upto user to interpret correctly.

9.2.6 Multiply-Long and Multiply-Accumulate Long Instructions

These are:

- MULL which gives RdHi,RdLo:=Rm*Rs
- MLAL which gives RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo

However, the full 64-bit of the result now matter (lower precision multiply instructions simply throws top 32-bits away)

- Need to specify whether operands are signed or unsigned

Therefore, syntax of new instructions are:

- UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs
- UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
- SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
- SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs

* **Not generated by the compiler.**

9.2.7 Status Register Transfer Instructions

The status register transfer instructions transfer the contents of the CPSR or an SPSR to or from a general-purpose register. Writing to the CPSR can:

- set the values of the condition code flags
- set the values of the interrupt enable bits
- set the processor mode and state
- alter the endianness of load and store operations.

PSR Transfer Instructions

MRS and MSR allow contents of CPSR/SPSR to be transferred from appropriate status register to a general-purpose register. All of status register, or just the flags, can be transferred.

Syntax:

- `MRS{<cond>} Rd,<psr> ; Rd = <psr>`
- `MSR{<cond>} <psr>,Rm ; <psr> = Rm`
- `MSR{<cond>} <psrf>,Rm ; <psrf> = Rm`

Where

- `<psr> = CPSR, CPSR_all, SPSR or SPSR_all`
- `<psrf>= CPSR_flg or SPSR_flg`

Also an immediate form

- MSR{<cond>} <psrf>,#Immediate
- This immediate must be a 32-bit immediate, of which the 4 most significant bits are written to the flag bits

9.2.8 Load and Store Instructions

The following load and store instructions are available:

- Load and Store Register
- Load and Store Multiple Registers
- Load and Store Register Exclusive

There are also swap and swap byte instructions, but their use is deprecated in ARMv6. It is recommended that all software migrates to using the load and store register exclusive instructions.

Load and store register

Load Register instructions can load a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from memory into a register or registers. Byte and halfword loads can be automatically zero-extended or sign-extended as they are loaded. Store Register instructions can store a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from a register or registers to memory. Load and Store Register instructions have three primary addressing modes, all of which use a *base register* and an *offset* specified by the instruction:

- In *offset addressing*, the memory address is formed by adding or subtracting an offset to or from the base register value.
- In *pre-indexed addressing*, the memory address is formed in the same way as for offset addressing. As a side effect, the memory address is also written back to the base register.
- In *post-indexed addressing*, the memory address is the base register value. As a side effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

In each case, the offset can be either an immediate or the value of an *index register*. Register-based offsets can also be scaled with shift operations. As the PC is a general-purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4 GB memory space.

Load and store multiple registers

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

- pre-increment
- post-increment
- pre-decrement
- post-decrement.

The base address is specified by a register value, which can be optionally updated after the transfer. As the subroutine return address and PC values are in general-purpose registers, very efficient subroutine entry and exit sequences can be constructed with LDM and STM:

- A single STM instruction at subroutine entry can push register contents and the return address onto the stack, updating the stack pointer in the process.
- A single LDM instruction at subroutine exit can restore register contents from the stack, load the PC with the return address, and update the stack pointer. LDM and STM instructions also allow very efficient code for block copies and similar data movement algorithms.

Single register data transfer

The basic load and store instructions are:

- Load and Store Word or Byte
  - LDR / STR / LDRB / STRB

    ARM Architecture Version 4 also adds support for half words and signed data.
  - LDRH / STRH
- Load Signed Byte or Halfword-load value and sign extend it to 32 bits.
  - LDRSB / LDRSH

All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.

e.g., LDREQB

Syntax: `<LDR|STR> {<cond>} {<size>} Rd, <address>`

Loading full 32-bit constants

Although the MOV/MVN mechanism will load a large range of constants into a register, sometimes this mechanism will not generate the required constant. Therefore, the assembler also provides a method which will load *ANY* 32-bit constant:

- LDR rd , = numeric constant

If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated. Otherwise, the assembler will produce an LDR instruction with a PC relative address to read the constant from a literal pool.

- `LDR r0,=0x42 ; generates MOV r0,#0x42`
- `LDR r0,=0x55555555 ;generate LDR r0,[pc,offset]`

As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.

9.2.9 Coprocessor Instructions

There are three types of coprocessor instructions:

(a) Coprocessor Data Processing: These start a coprocessor-specific internal operation. This instruction initiates a coprocessor operation. The operation is performed only on internal coprocessor state.

- For example, a Floating point multiplier, which multiplies the contents of two registers and stores the result in a third register

Syntax: `CDP{<cond>}<cp_num>,<opc_1>,CRd,CRn,CRm,{<opc_2}`

31 28	27 26 25 24	23 20	19 16	15 12	11 8	7 5	4	3 0
Cond	1 1 1 0	Opc_1	Crn	Crd	Cp_num	Opc_2	0	CRm

- **Cond**- Condional code specifier
- **Opc_1,opc_2**- Opcode
- **CRd**- Destination register
- **CRm**- Source register

(b) Coprocessor Register Transfers Instructions: These instructions allow a coprocessor value to be transferred to or from an ARM register, or a pair of ARM registers.

These two instructions move data between ARM registers and coprocessor registers.

- MRC : Move to Register from Coprocessor
- MCR : Move to Coprocessor from Register

An operation may also be performed on the data as it is transferred

- For example, a Floating Point Convert to Integer instruction can be implemented as a register transfer to ARM that also converts the data from floating point format to integer format.

Syntax: <MRC|MCR>{<cond>}<cpnum>,<opc1>,Rd,CRn,CRm,<op2>

31 28	27 26 25 24	23 20	19 16	15 12	11 8	7 5	4	3 0
Cond	1 1 1 0	Opc_1	Crn	Crd	Cp_num	Opc_2	0	CRm

- L- Transfer to/from coprocessor

Coprocessor Data transfer instructions

These transfer coprocessor data to or from memory. The address of the transfer is calculated by the ARM processor.

(c) Coprocessor Memory Transfers (1)

Load from memory to coprocessor registers.

Store to memory from coprocessor registers.

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CP_Num	Offset

- **P**- Pre/post Increment.
- **U**- Add/Subtract Offset.
- **N**- Transfer Length.
- **W**- Base Register Write Back.
- **L**-Load Store.
- **Off set**- Offset Address.

Coprocessor Memory Transfers (2)

Syntax of these is similar to word transfers between ARM and memory:

1. <LDC|STC>{<cond>}{<L>} <cp_num>,CRd,<address>

   PC relative offset generated if possible, else causes an error.

2. <LDC|STC>{<cond>}{<L>}<cpnum>,CRd,<[Rn,offset] {!}>

   Pre-indexed form, with optional writeback of the base register

3. <LDC|STC>{<cond>}{<L>}<cp_num>,CRd,<[Rn],offset>

Post-indexed form <L> when present causes a "long" transfer to be performed (N=1) else causes a "short" transfer to be performed (N=0). Effect of this is coprocessor dependant.

9.2.10 Exception-generating Instructions

Two types of instruction are designed to cause specific exceptions to occur.

(a) Software interrupt instructions (SWI): SWI instructions cause a software interrupt exception to occur. These are normally used to make calls to an operating system, to request an OS-defined service. The exception entry caused by a SWI instruction also changes to a privileged processor mode. This allows an unprivileged task to gain access to privileged functions, but only in ways permitted by the OS.

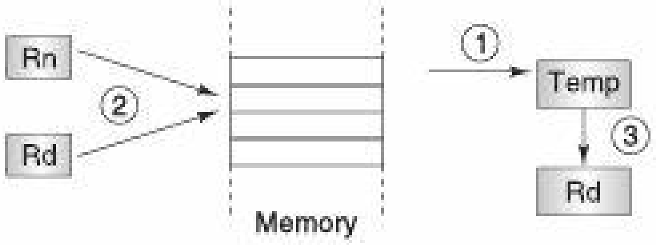
(b) **Software breakpoint instructions (BKPT):** BKPT instructions cause an abort exception to occur. If suitable debugger software is installed on the abort vector, an abort exception generated in this fashion is treated as a breakpoint. If debug hardware is present in the system, it can instead treat a BKPT instruction directly as a breakpoint, preventing the abort exception from occurring. In addition to the above, the following types of instruction cause an Undefined Instruction exception to occur:
- coprocessor instructions which are not recognized by any hardware coprocessor
- most instruction words that have not yet been allocated a meaning as an ARM instruction.

In each case, this exception is normally used either to generate a suitable error or to initiate software emulation of the instruction.

Swap and Swap Byte

Instruction

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory
- Syntax
  SWP {<cond>} {B} Rd , Rm , [Rn]



- Thus, to make actual swap make Rd = Rm

Quiz

1. Specify instructions which will implement the following:
```
(a) r0 = 16                         (b) r1 = r0 * 4
(c) r0 = r1 / 16 (r1 signed 2's comp.)    (d) r1 = r2 * 7
```
2. What will the following instructions do?
```
(a) ADDS r0, r1, r1, LSL #2 (b) RSB r2, r1, #0
```
3. What does the following instruction sequence do?
```
ADD r0, r1, r1, LSL #1
SUB r0, r0, r1, LSL #4
ADD r0, r0, r1, LSL #7
```

9.3 ARM 7 PROGRAMMING MODEL

The programmer's model of the ARM 7 consists of 15 user registers, as shown in Figure 9.7, with R15 being used as the Program Counter (PC). Since the ARM 7 is a load-and-store architecture, a user program must load data from memory into the CPU registers, process this data and then store the result back into memory. Unlike other processors no memory-to-memory instructions are available.

9.3.1 Registers and Flags

As stated above R15 is the Program Counter. R13 and R14 also have special functions; R13 is used as the stack pointer, though this has only been defined as a programming convention. Unusually, the ARM instruction set does not have PUSH and POP instructions so stack handling is done via a set of instructions that allow loading and storing of multiple registers in a single operation. Thus, it is possible to PUSH or POP the entire register set onto the stack in a single instruction. R14 has special significance and is called the "link register". When a call is made to a procedure, the return address is automatically placed into R14, rather than onto a stack, as might be expected. A return can then be implemented by moving the contents of R14 into R15, the PC. For multiple calling trees, the contents of R14 (the link register) must be placed onto the stack.

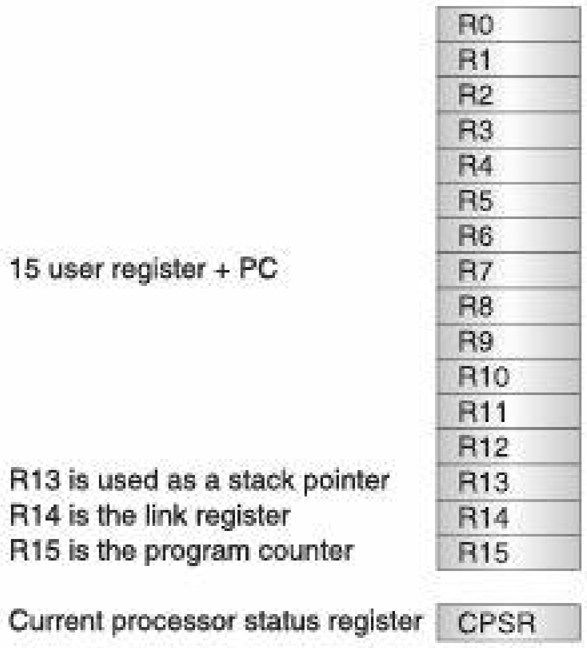

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
15 user register + PC
R13 is used as a stack pointer
R14 is the link register
R15 is the program counter
Current processor status register
CPSR

Fig. 9.7 User Mode Register Model.

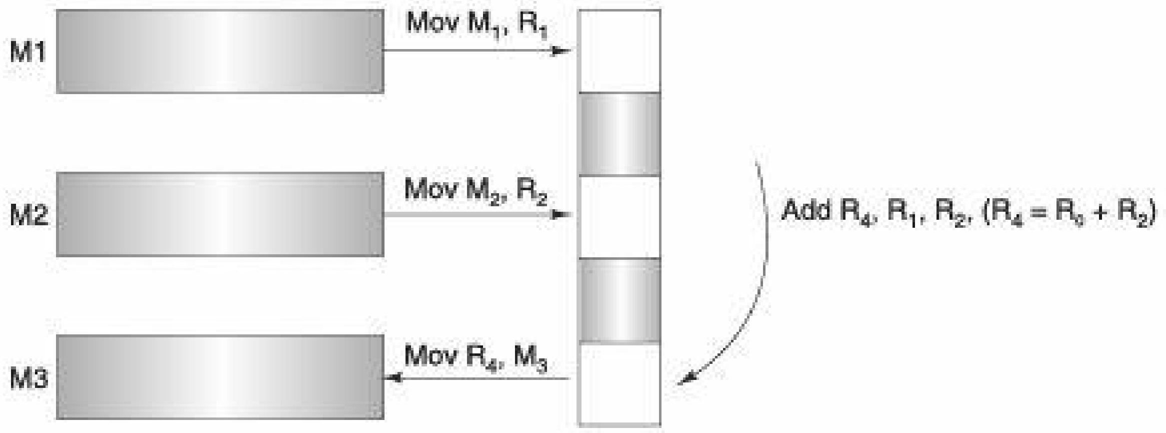


Fig. 9.8 Load and Store Architecture.

In addition to the 16 CPU registers, there is a current program status register (CPSR). This contains a set of condition code flags in the upper four bits that record the result of a previous instruction, as shown in Figure 9.9. In addition to the condition code flags, the CPSR contains a number of user-configurable bits that can be used to change the processor mode, enter Thumb processing and enable/disable interrupts.

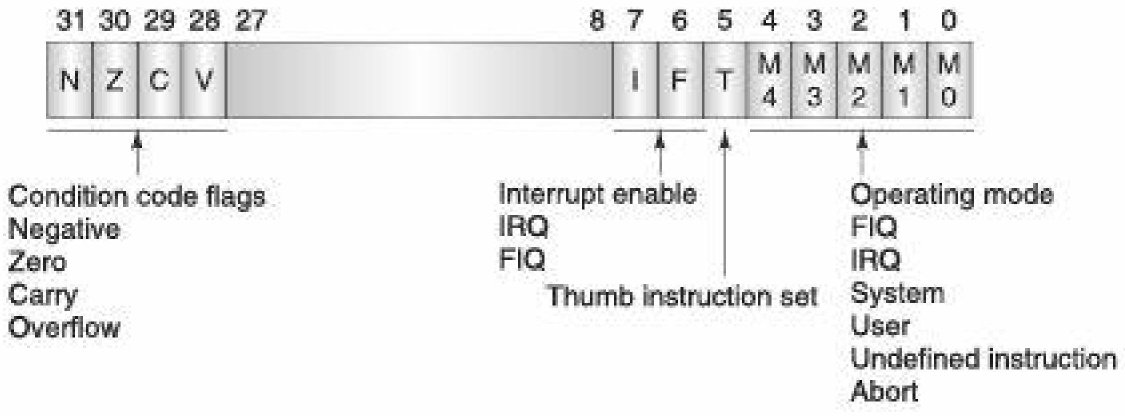


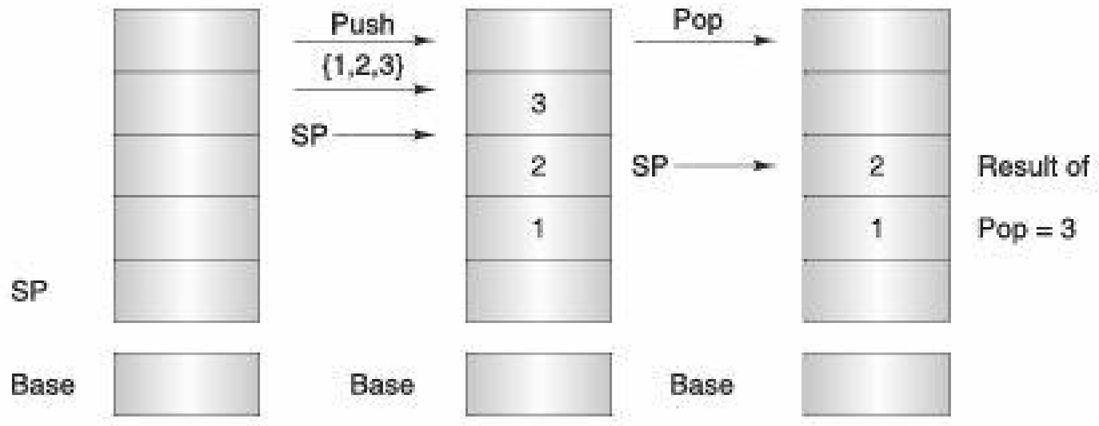
Fig. 9.9 Current Program Status Register and Flags.

9.4 STACKS

A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.

Two pointers define the current limits of the stack.

- A base pointer
  - used to point to the "bottom" of the stack (the first location).
- A stack pointer
  - used to point the current "top" of the stack.



Stack operation

Traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory. The value of the stack pointer can either:

- Point to the last occupied address (Full stack)
  - and so needs pre-decrementing (i.e., before the push)

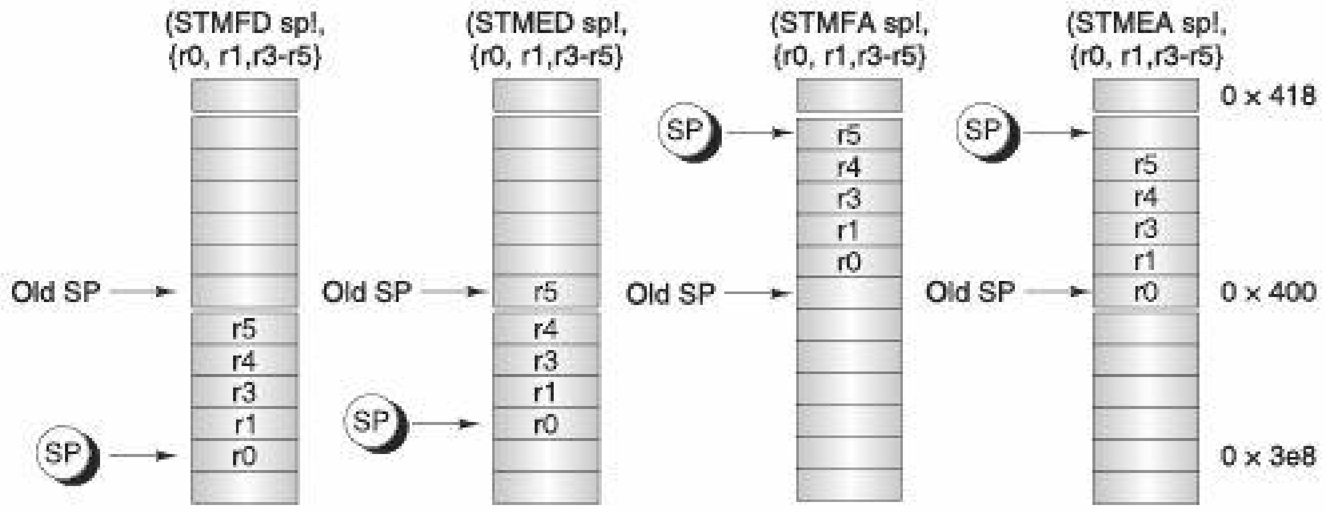
- Point to the next occupied address (Empty stack)
  - and so needs post-decrementing (i.e., after the push)

The stack type to be used is given by the postfix to the instruction:

- `STMFD / LDMFD : Full Descending stack`
- `STMFA / LDMFA : Full Ascending stack.`
- `STMED / LDMED : Empty Descending stack`
- `STMEA / LDMEA : Empty Ascending stack`

Note: ARM Compiler will always use a Full descending stack

STACK EXAMPLES


(STMFD sp!, {r0, r1,r3-r5}
(STMED sp!, {r0, r1,r3-r5}
(STMFA sp!, {r0, r1,r3-r5}
(STMEA sp!, {r0, r1,r3-r5}
Old SP
SP
r5
r4
r3
r1
r0
0 x 418
0 x 400
0 x 3e8

Stacks and subroutines

One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller:

```
STMFD sp!,{r0-r12, lr} ; stack all registers
LDMFD sp!,{r0-r12, pc} ; load all the registers
```

See the chapter on the ARM Procedure Call Standard in the SDT Reference Manual for further details of register usage within subroutines. If the pop instruction also had the 'S' bit set (using '^') then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

Direct functionality of Block Data Transfer

When LDM/STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is, i.e., specify whether to increment/decrement the base pointer, before or after the memory access.

In order to do this, LDM/STM support a further syntax in addition to the stack one:

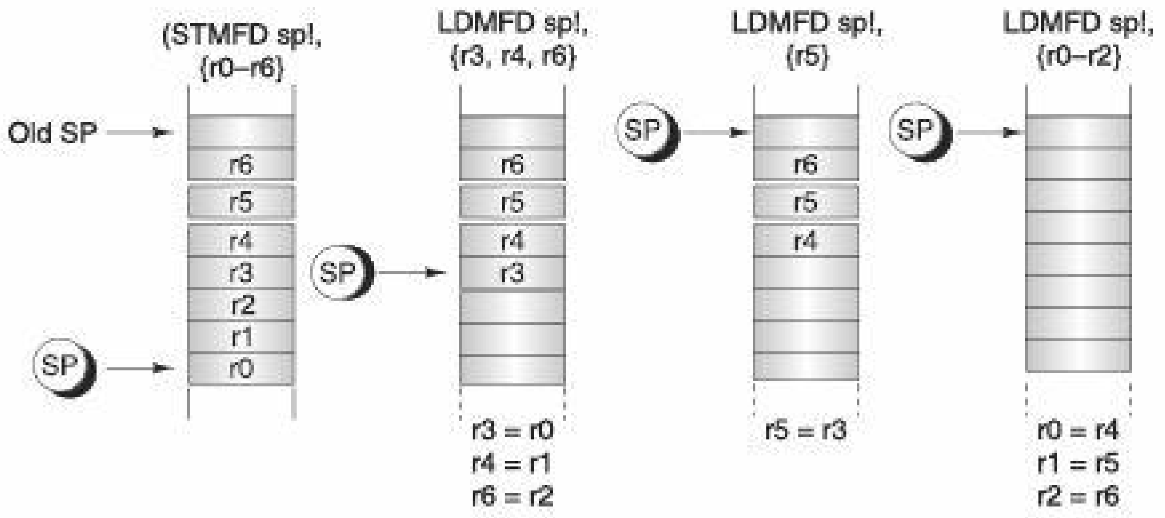
- STMIA / LDMIA : Increment After
- STMIB / LDMIB : Increment Before

- STMDA / LDMDA : Decrement After
- STMDB / LDMDB : Decrement Before

Quiz

Q1. The contents of registers r0 to r6 need to be swapped around thus:
- r0 moved into r3
- r1 moved into r4
- r2 moved into r6
- r3 moved into r5
- r4 moved into r0
- r5 moved into r1
- r6 moved into r2

Q.2. Write a segment of code that uses full descending stack operations to carry this out, and hence requires no use of any other registers for temporary storage.



Sample Space

9.5 EXCEPTION AND INTERRUPT MODES

The ARM 7 architecture has a total of six different operating modes, as shown below. These modes are protected or exception modes which have associated interrupt sources and their own register sets.

User: This mode is used to run the application code. Once in user mode, 9 the CPSR cannot be written to and modes can only be changed when an exception is generated.

FIQ: (Fast Interrupt Request) This supports high-speed interrupt handling. Generally, it is used for a single critical interrupt source in a system.

IRQ: (Interrupt Request) This supports all other interrupt sources in a system.

Supervisor: A "protected" mode for running system level code to access hardware or run OS calls. The ARM 7 enters this mode after reset.

Abort: If an instruction or data is fetched from an invalid memory region, an abort exception will be generated.

Undefined Instruction: If a FETCHED opcode is not an ARM instruction, an undefined instruction exception will be generated.

The User registers R0-R7 are common to all operating modes. However, FIQ mode has its own R8–R14 that replace the user registers when FIQ is entered. Similarly, each of the other modes have their own R13 and R14 so that each operating mode has its own unique Stack pointer and Link register. The CPSR is also common to all modes. However, in each of the exception modes, an additional register—the saved program status register (SPSR), is added. When the processor changes the current value of the CPSR stored in the SPSR, this can be restored on exiting the exception mode.

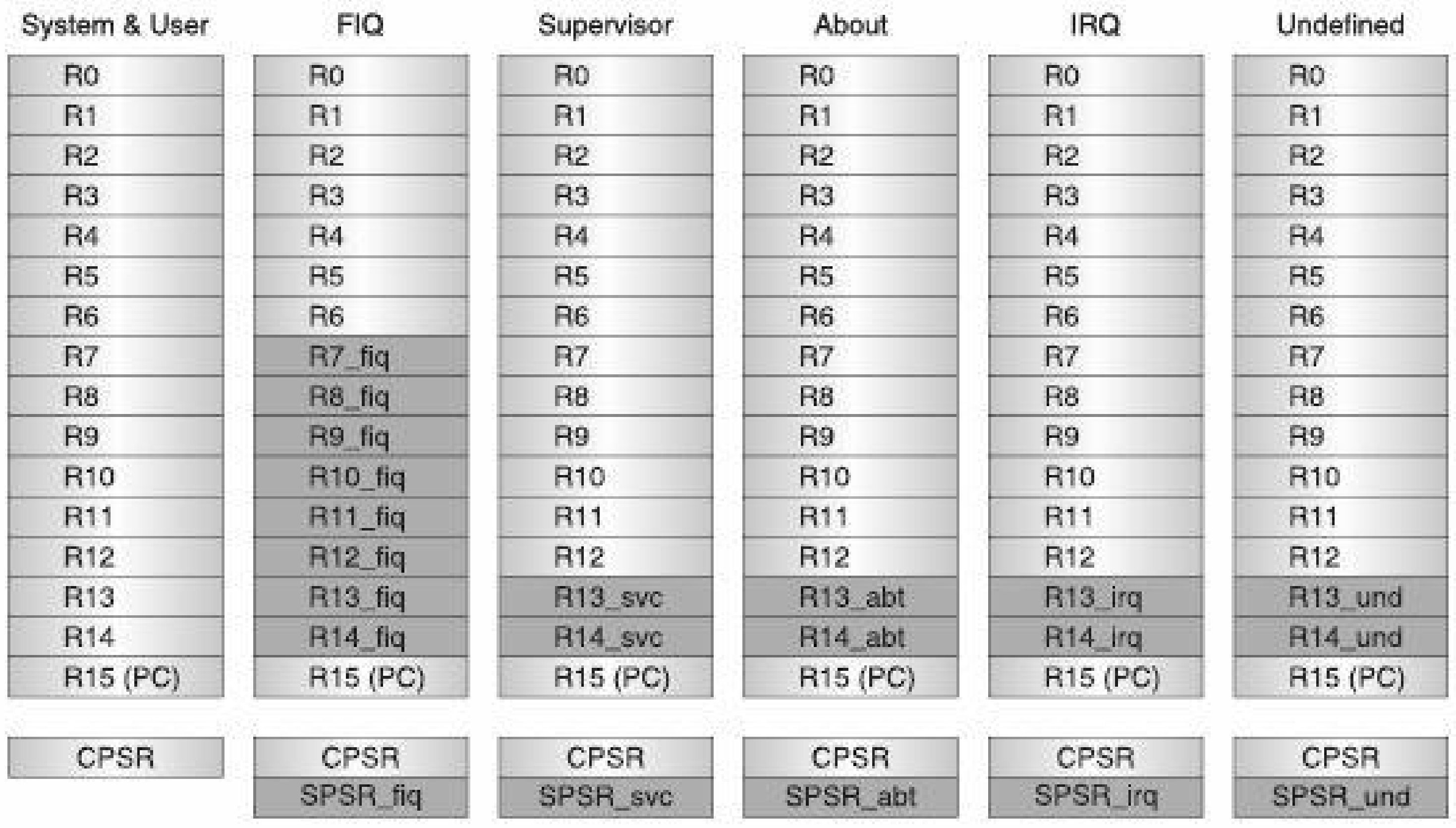


Fig. 9.10 Full Register Set for ARM 7.

Exceptions in the ARM processor can be split into three distinct types.

(i) Exceptions caused by executing an instruction, these include software interrupts, undefined Instruction exceptions and memory abort exceptions.

(ii) Exceptions caused as a side effect of an instruction such as abort caused by trying to fetch data from an invalid memory region.

(iii) Exceptions unrelated to instruction execution, this includes reset, FIQ and IRQ Interrupts.

In each case entry into the exception mode uses the same mechanism. On generation of the exception, the processor switches to the privileged mode, the current value of the PC+4 is saved into the Link register (R14) of the privileged mode and the current value of CPSR is saved into the privileged mode's SPSR. The IRQ interrupts are also disabled and if the FIQ mode is entered, the FIQ interrupts are also disabled. Finally, the Program Counter is forced to the exception vector address and processing of the exception can start. Usually, the first action of the exception routine will be to push some or all of the user registers onto the stack.

9.5.1 Software Interrupts

The ARM instruction set has a software interrupt instruction. Execution of this instruction forces an exception as described above; the processor will enter supervisor mode and jump to the SWI vector at 0x00000008.

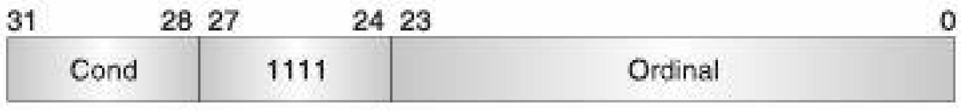


Fig. 9.11 Software Interrupt Instruction.

The bit field 0-23 of the SWI instruction as shown in Figure 9.11 is empty and can be used to hold an ordinal. On execution of an SWI instruction, this ordinal can be examined to determine which SWI procedure to run and gives over 16 million possible SWI functions. This can be used to provide a hardware abstraction layer. In order to access OS calls or SFR registers, the user code must make a SWI call. All these functions are then running in a supervisor mode, with a separate stack and link register.

As well as instructions to transfer data to and from memory and to CPU registers, the ARM 7 has instructions to save and load multiple registers. It is possible to load or save all 16 CPU registers or a selection of registers in a single instruction. Needless to say, this is extremely useful when entering or exiting a procedure.

The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions. The general way these are handled is the same in all cases:

1. The current state is saved by copying the PC into r14_exc and the CPSR into *SPSR_exc* (where *exc* stands for the exception type).
2. The processor operating mode is changed to the appropriate exception mode.
3. The PC is forced to a value between 0016 and 1C 16, the particular value depending on the type of exception.

The instruction at the location on the PC (is forced to the *vector address*) will usually contain a branch to the exception handler. The exception handler will use r13_*exc*, which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers. The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR automatically. This may involve some adjustment of the PC value saved in r14_*exc* to compensate for the state of the pipeline when the exception arose.

9.6 THE ARM I/O SYSTEM

The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support. The internal registers in these devices appear as addressable locations within the ARM's memory map and may be read and written using the same (load-store) instructions as any other memory locations.

Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt (IRQ) or the fast interrupt (FIQ) input. Both interrupt inputs are level-sensitive and maskable. Normally, most interrupt sources share the IRQ input, with just one or two time critical sources connected to the higher-priority FIQ input. Some systems may include direct memory access (DMA) hardware external to the processor to handle high-bandwidth I/O traffic.

PROGRAM → 1

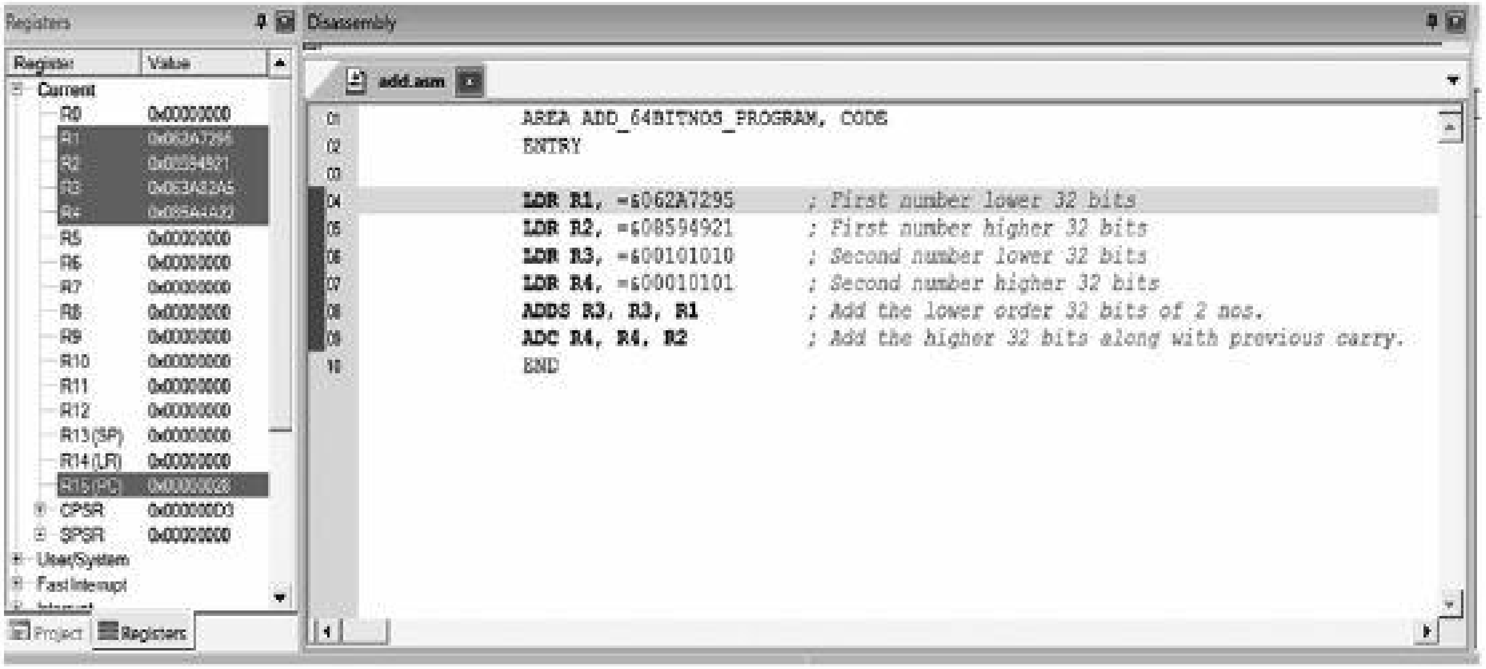
Program: Addition of Two 64-bit Numbers

****

Code:

```
AREA  ADD_64BITNOS_PROGRAM,  CODE
ENTRY
     LDR  R1,  =&062A7295   ; First  number  lower  32  bits
     LDR  R2,  =&08594921   ; First  number  higher  32  bits
     LDR  R3,  =&00101010   ; Second  number  lower  32  bits
     LDR  R4,  =&00010101   ; Second  number  higher  32  bits
     ADDS  R3,  R3,  R1     ; Add  the  lower  order  32  bits  of  2  nos.
     ADC  R4,  R4,  R2      ; Add the higher 32 bits along with previous carry.
END
```

RESULT



```
Register        Value
Current
  R0            0x00000000
  R1            0x062A7295
  R2            0x08594921
  R3            0x063A82A5
  R4            0x085A4A22
  R5            0x00000000
  R6            0x00000000
  R7            0x00000000
  R8            0x00000000
  R9            0x00000000
  R10           0x00000000
  R11           0x00000000
  R12           0x00000000
  R13 (SP)      0x00000000
  R14 (LR)      0x00000000
  R15 (PC)      0x00000028
  CPSR          0x000000D3
  SPSR          0x00000000
User/System
Fast Interrupt
```

```
AREA ADD_64BITNOS_PROGRAM, CODE
ENTRY

LDR R1, =&062A7295        ; First number lower 32 bits
LDR R2, =&08594921        ; First number higher 32 bits
LDR R3, =&00101010        ; Second number lower 32 bits
LDR R4, =&00010101        ; Second number higher 32 bits
ADDS R3, R3, R1           ; Add the lower order 32 bits of 2 nos.
ADC R4, R4, R2            ; Add the higher 32 bits along with previous carry.
END
```

PROGRAM → 2

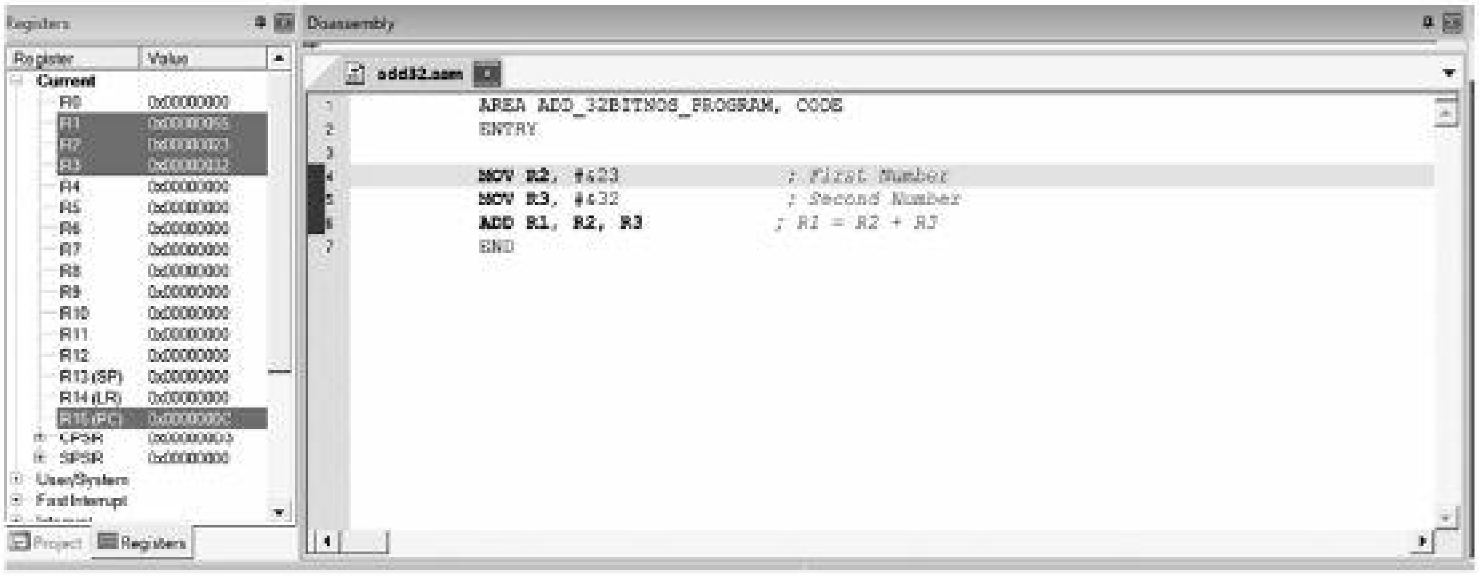
Program: Addition of Two 32-bit Numbers

****

Code:

```
AREA  ADD_32BITNOS_PROGRAM,  CODE
ENTRY

     MOV  R2,  #&23          ; First  Number
     MOV  R3,  #&32          ; Second  Number
     ADD  R1,  R2,  R3       ; R1  =  R2  +  R3
END
```

RESULT



```
Register     Value
Current
  R0         0x00000000
  R1         0x00000055
  R2         0x00000023
  R3         0x00000032
  R4         0x00000000
  R5         0x00000000
  R6         0x00000000
  R7         0x00000000
  R8         0x00000000
  R9         0x00000000
  R10        0x00000000
  R11        0x00000000
  R12        0x00000000
  R13 (SP)   0x00000000
  R14 (LR)   0x00000000
  R15 (PC)   0x0000000C
  CPSR       0x000000D3
  SPSR       0x00000000
User/System
Fast Interrupt
```

```
            AREA ADD_32BITNOS_PROGRAM, CODE
            ENTRY

            MOV R2, #&23                ; First Number
            MOV R3, #&32                ; Second Number
            ADD R1, R2, R3             ; R1 = R2 + R3
            END
```

PROGRAM → 3

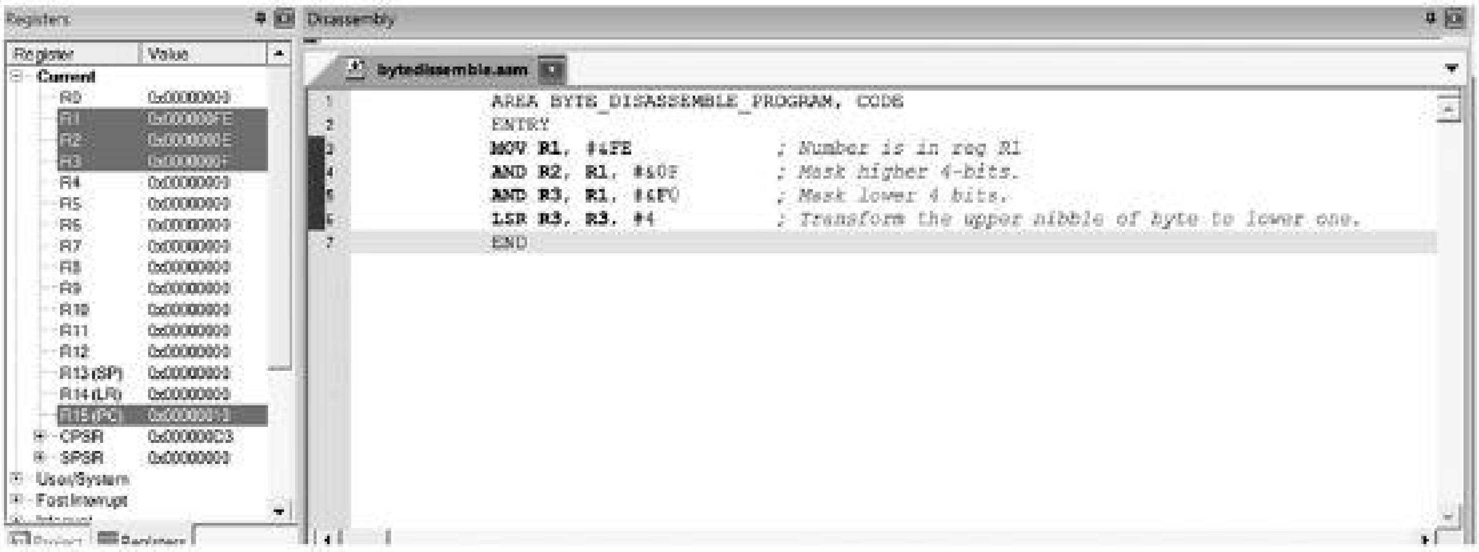
Program: To Disassemble a Byte

****

Code:

```
AREA  BYTE_DISASSEMBLE_PROGRAM,  CODE
ENTRY

     MOV  R1,  #&FE          ; Number  is  in  reg  R1
     AND  R2,  R1,  #&0F     ; Mask  higher  4  bits.
     AND  R3,  R1,  #&F0     ; Mask  lower  4  bits.
     LSR  R3,  R3,  #4       ; Transform the upper nibble of byte to lower one.
END
```

RESULT



Registers

```
Register      Value
Current
  R0          0x00000000
  R1          0x000000FE
  R2          0x0000000E
  R3          0x0000000F
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x00000010
  CPSR        0x000000D3
  SPSR        0x00000000
User/System
Fast Interrupt
```

Disassembly — bytedissemble.asm

```
        AREA BYTE_DISASSEMBLE_PROGRAM, CODE
        ENTRY
        MOV R1, #&FE            ; Number is in reg R1
        AND R2, R1, #&0F        ; Mask higher 4-bits.
        AND R3, R1, #&F0        ; Mask lower 4 bits.
        LSR R3, R3, #4          ; Transform the upper nibble of byte to lower one.
        END
```

PROGRAM → 4

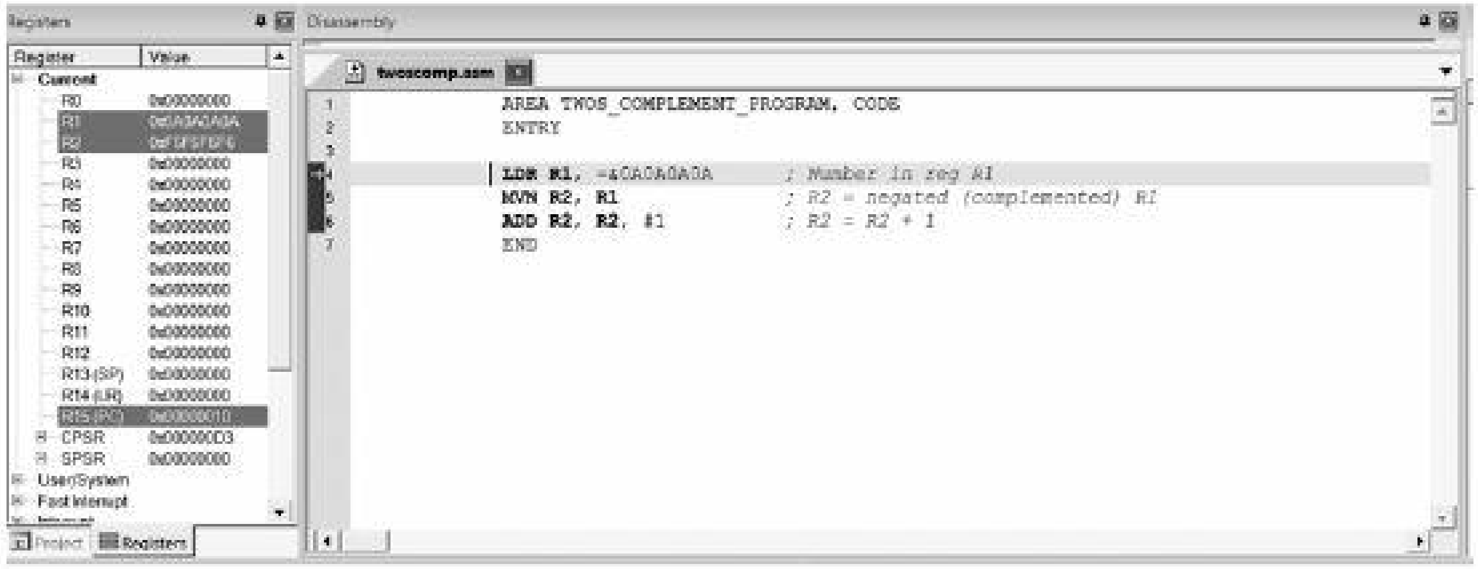
Program: To Compute 2's Complement of a Number

****

Code:

```
AREA  TWOS_COMPLEMENT_PROGRAM,  CODE
ENTRY

     MOV  R1,  #&FE         ; Number  is  in  reg  R1
     LDR  R1,  =&0A0A0A0A   ; Number  in  reg  R1
     MVN  R2,  R1           ; R2  =  negated  (complemented)  R1
     ADD  R2,  R2,  #1      ; R2  =  R2  +  1
END
```

RESULT



Registers

```
Register      Value
Current
  R0          0x00000000
  R1          0x0A0A0A0A
  R2          0xF5F5F5F6
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x00000010
  CPSR        0x000000D3
  SPSR        0x00000000
User/System
Fast Interrupt
```

Disassembly — twoscomp.asm

```
        AREA TWOS_COMPLEMENT_PROGRAM, CODE
        ENTRY

        LDR R1, =&0A0A0A0A      ; Number in reg R1
        MVN R2, R1              ; R2 = negated (complemented) R1
        ADD R2, R2, #1          ; R2 = R2 + 1
        END
```

PROGRAM → 5

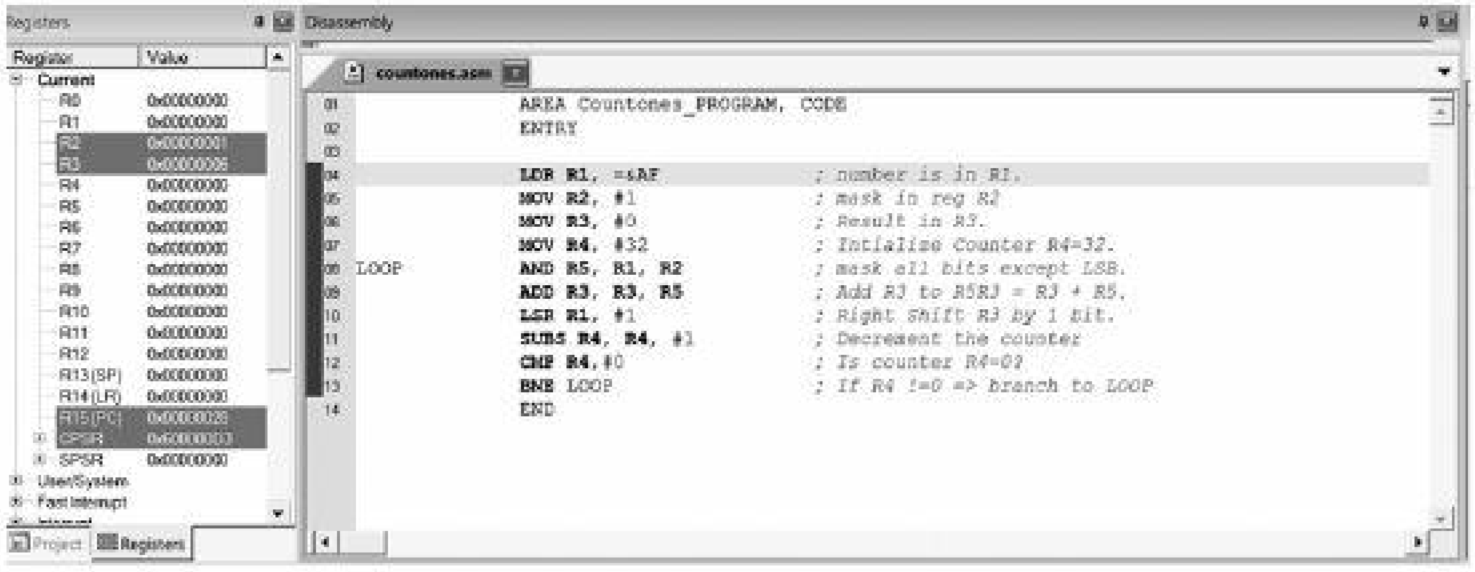
Program: To Calculate the Number of Ones in a Byte

****

Code:

```
AREA Countones_PROGRAM, CODE
ENTRY
        LDR R1, =&AF           ;number is in R1.
        MOV R2, #1             ;mask in reg R2.
        MOV R3, #0             ;Result in R3.
        MOV R4, #32            ;Intialize Counter R4=32.
LOOP    AND R5, R1, R2         ;mask all bits except LSB.
        ADD R3, R3, R5         ;Add R3 to R5R3 = R3 + R5.
        LSR R1, #1             ;Right Shift R3 by 1 bit.
        SUBS R4, R4, #1        ;Decrement the counter.
        CMP R4,#0              ;Is counter R4=0?
        BNE LOOP               ;If R4 !=0 => branch to LOOP
END
```

RESULT



```
Registers
Register      Value
Current
  R0          0x00000000
  R1          0x00000000
  R2          0x00000001
  R3          0x00000006
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13(SP)     0x00000000
  R14(LR)     0x00000000
  R15(PC)     0x00000028
  CPSR        0x60000003
  SPSR        0x00000000
User/System
Fast Interrupt
```

```
Disassembly
countones.asm

        AREA Countones_PROGRAM, CODE
        ENTRY

        LDR R1, =&AF           ; number is in R1.
        MOV R2, #1             ; mask in reg R2
        MOV R3, #0             ; Result in R3.
        MOV R4, #32            ; Intialize Counter R4=32.
LOOP    AND R5, R1, R2         ; mask all bits except LSB.
        ADD R3, R3, R5         ; Add R3 to R5R3 = R3 + R5.
        LSR R1, #1             ; Right Shift R3 by 1 bit.
        SUBS R4, R4, #1        ; Decrement the counter
        CMP R4,#0              ; Is counter R4=0?
        BNE LOOP               ; If R4 !=0 => branch to LOOP
        END
```

PROGRAM → 6

Program: To Divide Two Numbers

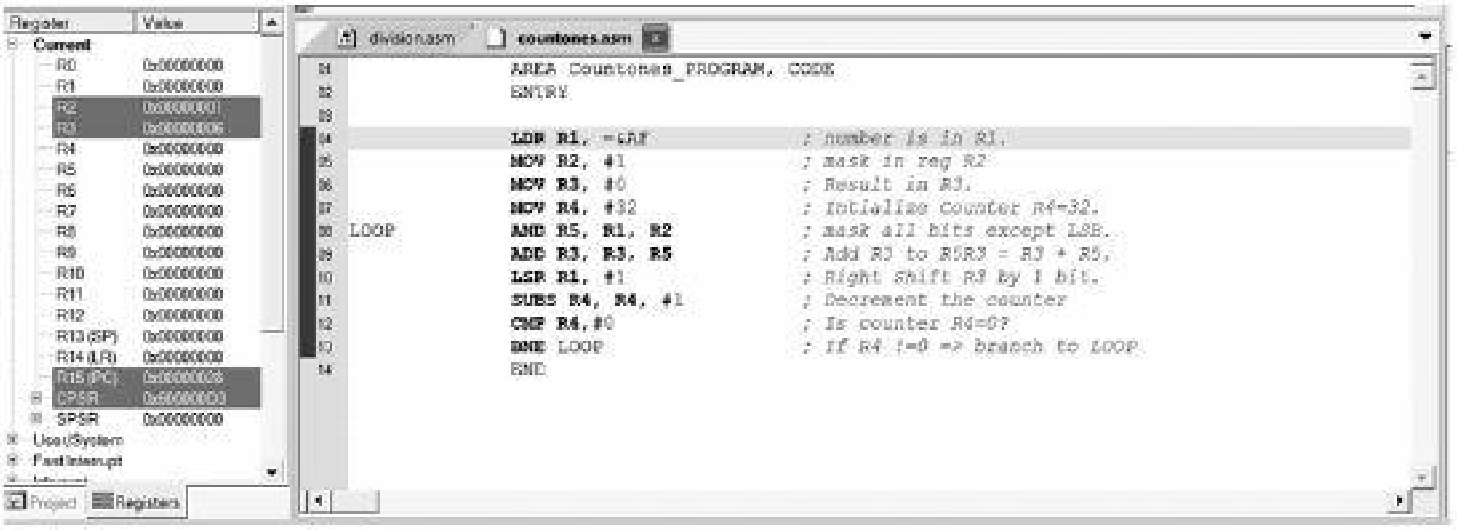
****

Code:

```
AREA MY_PROGRAM, CODE, READONLY
ENTRY
```

```
         MOV  R1,  #32          ; Dividend.
         MOV  R2,  #5           ; Divisor.
         MOV  R3,  #0           ; Quotient.
         MOV  R4,  #0           ; Remainder.
LOOP     SUB  R1,  R1,  R2      ; R1 = R1 - R2.
         ADD  R3,  R3,  #1      ; Increment quotient by 1.
         CMP  R1,  R2           ; Compare R1 and R2
         MOVLT  R4,  R1         ; If R1 < R2 => Remainder R4 = R1.
         BGE  LOOP              ; If R1 > R2 => branch back to LOOP.
END
```

RESULT



Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000006
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000028
CPSR	0x60000003
SPSR	0x00000000
User/System	
Fast Interrupt	

```
                 AREA Countones_PROGRAM, CODE
                 ENTRY

                 LDR R1, =&AF                   ; number is in R1.
                 MOV R2, #1                     ; mask in reg R2
                 MOV R3, #0                     ; Result in R3.
                 MOV R4, #32                    ; Intialize counter R4=32.
LOOP             AND R5, R1, R2                 ; mask all bits except LSB.
                 ADD R3, R3, R5                 ; Add R3 to R5R3 = R3 + R5.
                 LSR R1, #1                     ; Right Shift R3 by 1 bit.
                 SUBS R4, R4, #1                ; Decrement the counter
                 CMP R4,#0                      ; Is counter R4=0?
                 BNE LOOP                       ; If R4 !=0 => branch to LOOP
                 END
```

PROGRAM → 7

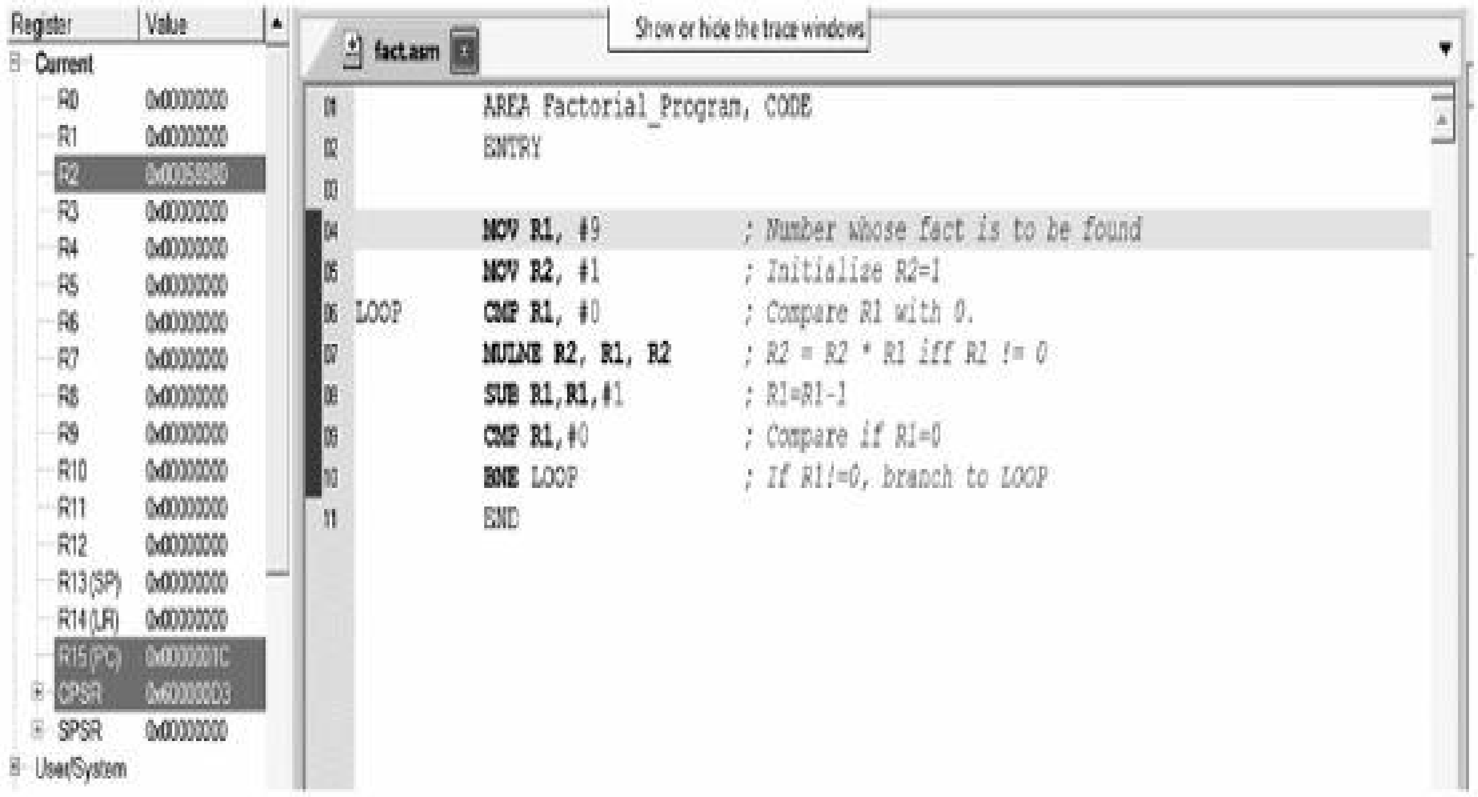
Program: To Compute the Factorial of a Number

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

Code:

```
AREA  Factorial_Program,  CODE
ENTRY

         MOV  R1,  #9           ; Number whose fact is to be found
         MOV  R2,  #1           ; Initialize R2 = 1.
LOOP     CMP  R1,  #0           ; Compare R1 with 0.
         MULNE  R2,  R1,  R2    ; R2 = R2 * R1 iff R1 != 0
         SUB  R1,R1,#1          ; R1=R1-1.
         CMP  R1,#0             ; Compare if R1=0.
         BNE  LOOP              ; If R1!=0, branch to LOOP.
END
```

RESULT



```
Register    Value
Current
  R0        0x00000000
  R1        0x00000000
  R2        0x00058980
  R3        0x00000000
  R4        0x00000000
  R5        0x00000000
  R6        0x00000000
  R7        0x00000000
  R8        0x00000000
  R9        0x00000000
  R10       0x00000000
  R11       0x00000000
  R12       0x00000000
  R13 (SP)  0x00000000
  R14 (LR)  0x00000000
  R15 (PC)  0x0000001C
  CPSR      0x600000D3
  SPSR      0x00000000
User/System
```

```
        AREA Factorial_Program, CODE
        ENTRY

        MOV R1, #9              ; Number whose fact is to be found
        MOV R2, #1              ; Initialize R2=1
LOOP    CMP R1, #0              ; Compare R1 with 0.
        MULNE R2, R1, R2        ; R2 = R2 * R1 iff R1 != 0
        SUB R1,R1,#1            ; R1=R1-1
        CMP R1,#0               ; Compare if R1=0
        BNE LOOP                ; If R1!=0, branch to LOOP
        END
```

PROGRAM → 8

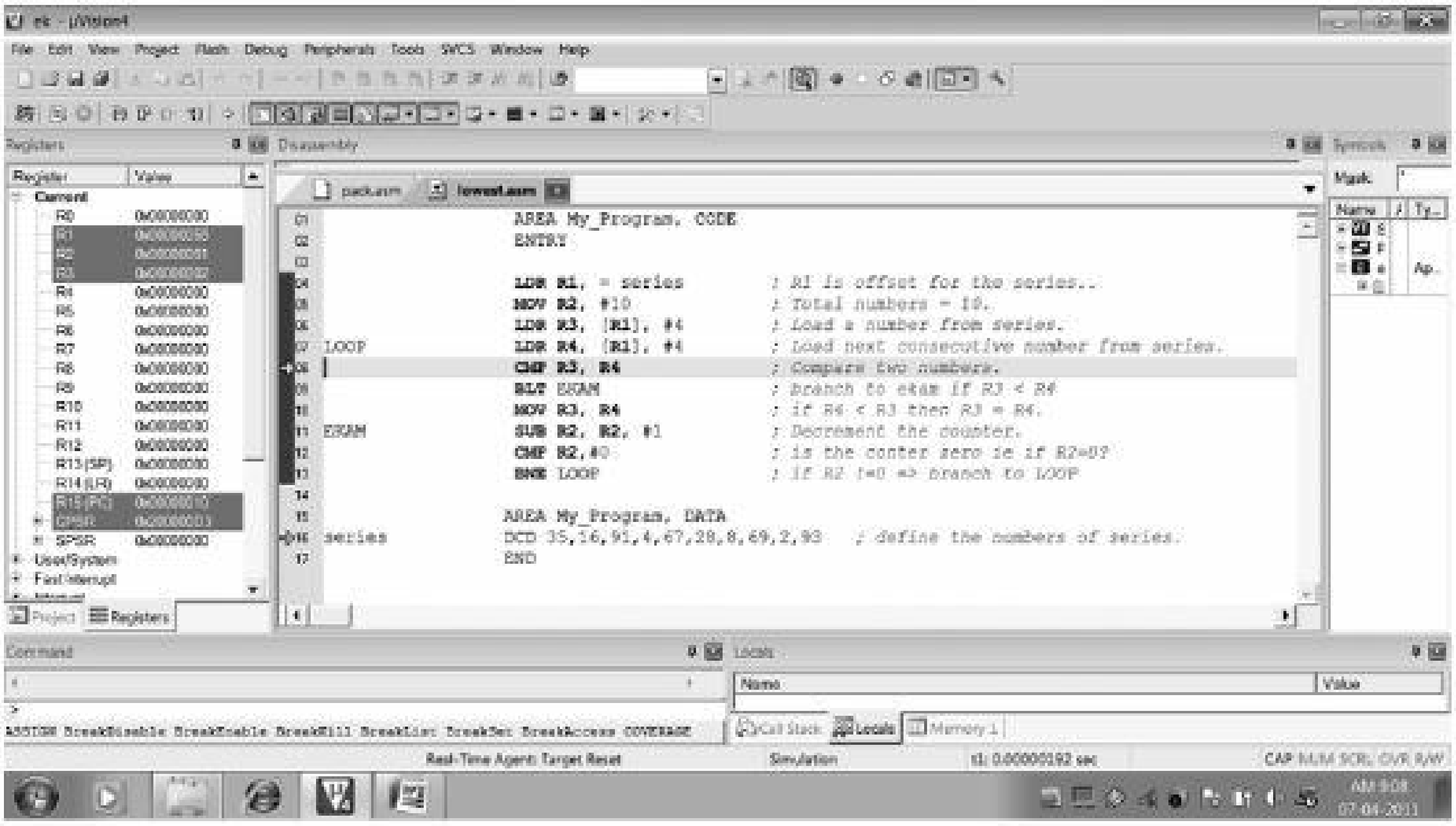
Program: To Find the Lowest Among the Series of a Numbers

****

Code:

```
AREA My_Program, CODE
ENTRY
        LDR R1, = series       ;R1 is offset for the series..
        MOV R2, #10            ;Total numbers = 10.
        LDR R3, [R1], #4       ;Load a number from series.
LOOP    LDR R4, [R1], #4       ;Load next consecutive number from series.
        CMP R3, R4             ;Compare two numbers.
        BLT EKAM               ;branch to EKAM if R3 < R4
        MOV R3, R4             ;if R4 < R3 then R3 = R4.
EKAM    SUB R2, R2, #1         ;Decrement the counter.
        CMP R2,#0              ;is the counter zero ie if R2=0?
        BNE LOOP               ;if R2 !=0 => branch to LOOP
AREA My_Program, DATA
series DCD 35,16,91,4,67,28,8,69,2,93 ; define the numbers of series.
END
```

RESULT



```
                AREA My_Program, CODE
                ENTRY

                LDR R1, = series        ; R1 is offset for the series..
                MOV R2, #10             ; Total numbers = 10.
                LDR R3, [R1], #4        ; Load a number from series.
LOOP            LDR R4, [R1], #4        ; Load next consecutive number from series.
                CMP R3, R4              ; Compare two numbers.
                BLT EXAM                ; branch to exam if R3 < R4
                MOV R3, R4              ; if R4 < R3 then R3 = R4.
EXAM            SUB R2, R2, #1          ; Decrement the counter.
                CMP R2,#0               ; is the conter zero ie if R2=0?
                BNE LOOP                ; if R2 !=0 => branch to LOOP

               AREA My_Program, DATA
series         DCD 35,16,91,4,67,28,8,69,2,93   ; define the numbers of series.
               END
```

PROGRAM → 9

Program: BCD Addition

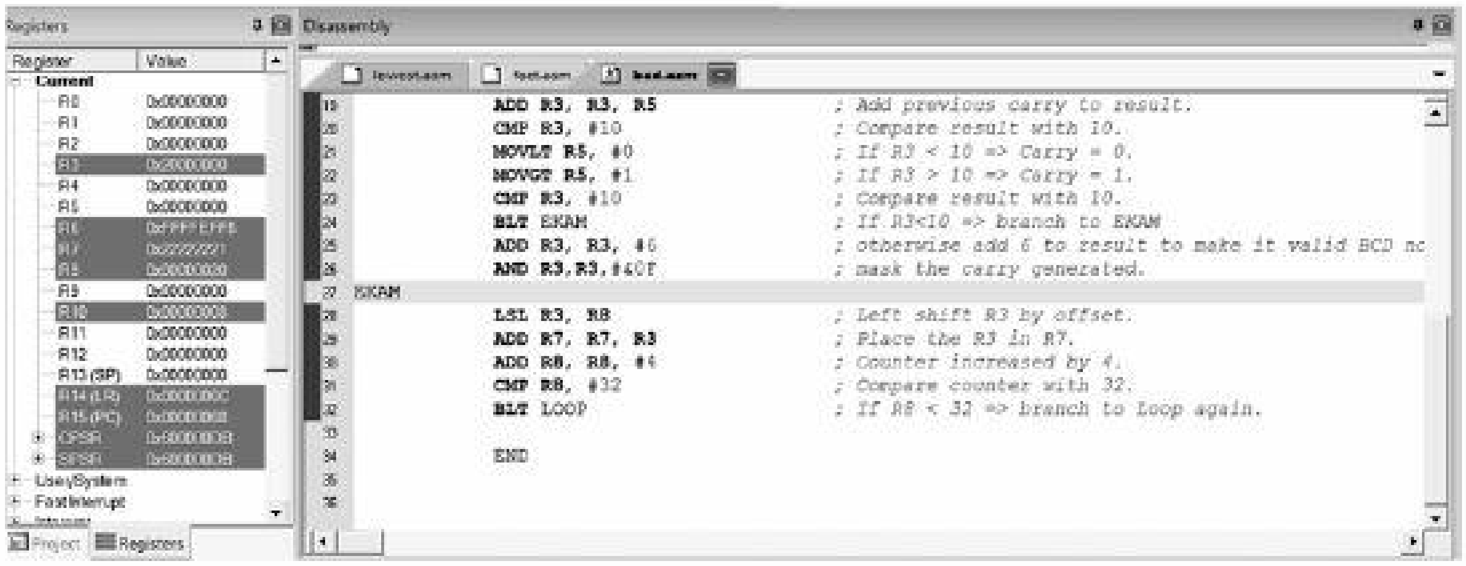
****

Code:

```
    AREA  BCD_ADDPROG,  CODE,  READONLY
    ENTRY
            MOV  R5,  #0            ;Initial  Carry  =  0
            MOV  R7,  #0            ;BCD  Result.
            MOV  R8,  #0            ;Counter  initialization.
            LDR  R1,  =&98765432    ;BCD  number  1.
            LDR  R2,  =&23456789    ;BCD  number  2.
    LOOP
            MOV  R3,  R1            ;Save  R1  in  R3.
            MOV  R4,  R2            ;Save  R2  in  R4.
            LSR  R1,  #4            ;Shift  right  R1  by  4  bits.
            LSR  R2,  #4            ;Shift  right  R2  by  4  bits.
            AND  R3,  R3,  #&0F     ;Mask.
            AND  R4,  R4,  #&0F     ;Mask.
```

```
        ADD R3, R3, R4          ;Add -> R3 = R3 + R4.
        ADD R3, R3, R5          ;Add previous carry to result.
        CMP R3, #10             ;Compare result with 10.
        MOVLT R5, #0            ;If R3 < 10 => Carry = 0.
        MOVGT R5, #1            ;If R3 > 10 => Carry = 1.
        CMP R3, #10             ;Compare result with 10.
        BLT EKAM                ;If R3<10 => branch to EKAM
        ADD R3, R3, #6          ;otherwise add 6 to result to make it
                                 valid BCD no.
        AND R3,R3,#&0F          ;mask the carry generated.
EKAM
        LSL R3, R8              ;Left shift R3 by offset.
        ADD R7, R7, R3          ;Place the R3 in R7.
        ADD R8, R8, #4          ;Counter increased by 4.
        CMP R8, #32             ;Compare counter with 32.
        BLT LOOP                ;If R8 < 32 => branch to Loop again.
END
```

RESULT



Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x20000000
R4	0x00000000
R5	0x00000000
R6	0xFFFFEFF8
R7	0x22222221
R8	0x00000020
R9	0x00000000
R10	0x00000008
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x0000000C
R15 (PC)	0x00000068
CPSR	0x800000D3
SPSR	0x800000D3
User/System	
Fast Interrupt	

```
              ADD R3, R3, R5                 ; Add previous carry to result.
              CMP R3, #10                    ; Compare result with 10.
              MOVLT R5, #0                   ; If R3 < 10 => Carry = 0.
              MOVGT R5, #1                   ; If R3 > 10 => Carry = 1.
              CMP R3, #10                    ; Compare result with 10.
              BLT EKAM                       ; If R3<10 => branch to EKAM
              ADD R3, R3, #6                 ; otherwise add 6 to result to make it valid BCD nc
              AND R3,R3,#&0F                 ; mask the carry generated.
EKAM
              LSL R3, R8                     ; Left shift R3 by offset.
              ADD R7, R7, R3                 ; Place the R3 in R7.
              ADD R8, R8, #4                 ; Counter increased by 4.
              CMP R8, #32                    ; Compare counter with 32.
              BLT LOOP                       ; If R8 < 32 => branch to Loop again.

              END
```

PROGRAM → 10

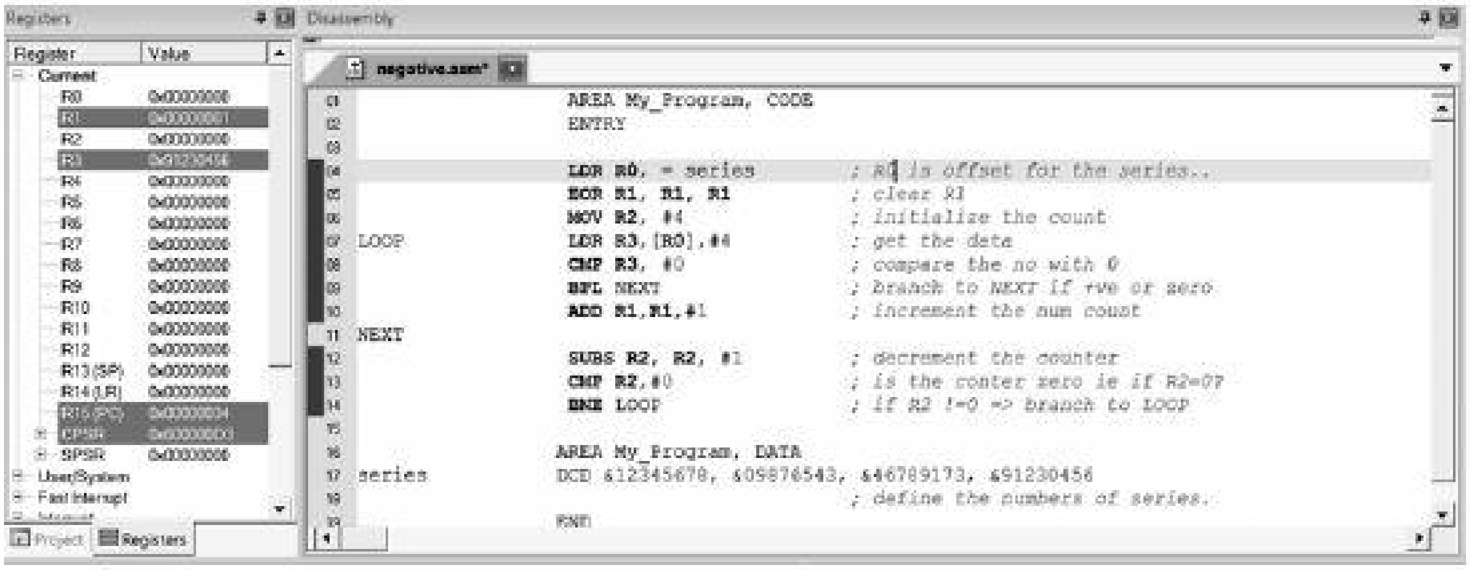
Program: To Scan Series of 32-bit Numbers and Count the Number of Negative Numbers in that

****

Code:

```
        AREA My_Program, CODE
        ENTRY
        LDR R0, = series       ;R0 is offset for the series.
        EOR R1, R1, R1         ;clear R1
        MOV R2, #4             ;initialize the count
LOOP    LDR R3,[R0],#4         ;get the data
        CMP R3, #0             ;compare the no with 0
        BPL NEXT               ;branch to NEXT if +ve or zero
        ADD R1,R1,#1           ;increment the num count
        NEXT
        SUBS R2, R2, #1        ;decrement the counter
        CMP R2,#0              ;is the conter zero ie if R2=0?
        BNE LOOP               ;if R2 !=0 => branch to LOOP
        AREA My_Program, DATA
series  DCD &12345678, &09876543, &46789173, &91230456
                                ; define the numbers of series.
        END
```

RESULT



Registers

Register	Value
Current	
R0	0x00000000
R1	0x00000001
R2	0x00000000
R3	0x91230456
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000034
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	

Disassembly — negative.asm*

```
                AREA My_Program, CODE
                ENTRY

                LDR R0, = series        ; R0 is offset for the series..
                EOR R1, R1, R1          ; clear R1
                MOV R2, #4              ; initialize the count
LOOP            LDR R3,[R0],#4          ; get the data
                CMP R3, #0              ; compare the no with 0
                BPL NEXT                ; branch to NEXT if +ve or zero
                ADD R1,R1,#1            ; increment the num count
NEXT
                SUBS R2, R2, #1         ; decrement the counter
                CMP R2,#0               ; is the conter zero ie if R2=0?
                BNE LOOP                ; if R2 !=0 => branch to LOOP

               AREA My_Program, DATA
series         DCD &12345678, &09876543, &46789173, &91230456
                                        ; define the numbers of series.
               END
```

PROGRAM → 11

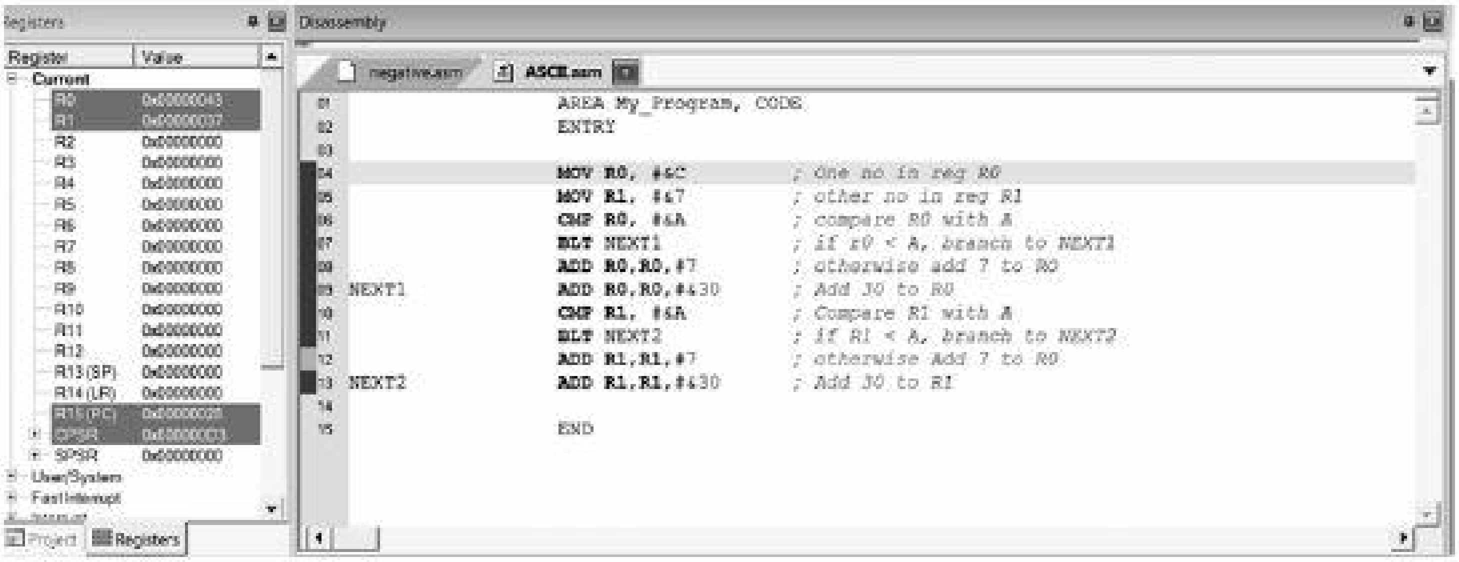
Program: Convert a Single HEX Digit to Its ASCII Equivalent

****

Code:

```
        AREA My_Program, CODE
        ENTRY
        MOV R0, #&C ; One no in reg R0
        MOV R1, #&7            ;other no in reg R1
        CMP R0,  #&A            ;compare R0 with A
        BLT NEXT1               ;if r0 < A, branch to NEXT1
        ADD R0,R0,#7            ;otherwise add 7 to R0
NEXT1   ADD R0,R0,#&30          ;Add 30 to R0
        CMP R1,  #&A            ;Compare R1 with A
        BLT NEXT2               ;if R1 < A, branch to NEXT2
        ADD R1,R1,#7            ;otherwise Add 7 to R0
NEXT2   ADD R1,R1,#&30          ;Add 30 to R1

        END
```

RESULT



```
Registers
Register        Value
Current
  R0            0x00000043
  R1            0x00000037
  R2            0x00000000
  R3            0x00000000
  R4            0x00000000
  R5            0x00000000
  R6            0x00000000
  R7            0x00000000
  R8            0x00000000
  R9            0x00000000
  R10           0x00000000
  R11           0x00000000
  R12           0x00000000
  R13 (SP)      0x00000000
  R14 (LR)      0x00000000
  R15 (PC)      0x00000028
  CPSR          0x000000D3
  SPSR          0x00000000
User/System
Fast Interrupt
```

```
Disassembly
negative.asm   ASCII.asm

                AREA My_Program, CODE
                ENTRY

                MOV R0, #&C             ; One no in reg R0
                MOV R1, #&7             ; other no in reg R1
                CMP R0, #&A             ; compare R0 with A
                BLT NEXT1               ; if r0 < A, branch to NEXT1
                ADD R0,R0,#7            ; otherwise add 7 to R0
NEXT1           ADD R0,R0,#&30          ; Add 30 to R0
                CMP R1, #&A             ; Compare R1 with A
                BLT NEXT2               ; if R1 < A, branch to NEXT2
                ADD R1,R1,#7            ; otherwise Add 7 to R0
NEXT2           ADD R1,R1,#&30          ; Add 30 to R1

                END
```

PROGRAM → 12

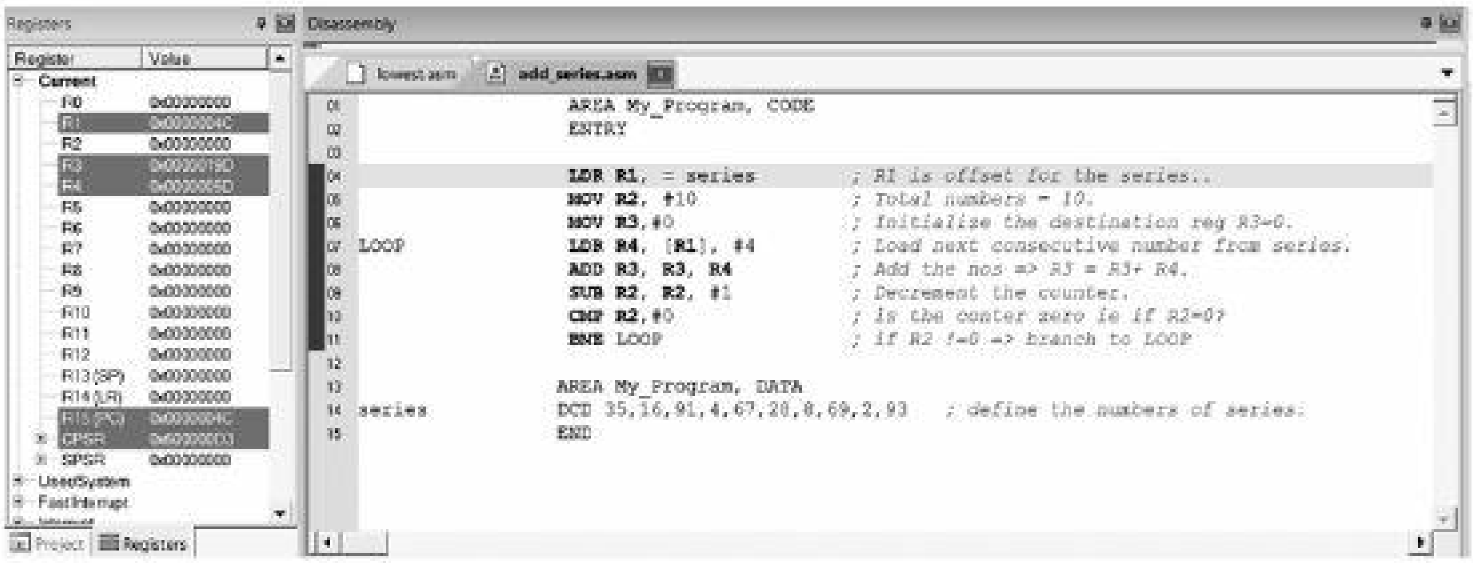
Program: To Add the Series of 16-bit Numbers

****

Code:

```
AREA My_Program, CODE
ENTRY
        LDR R1, = series        ;R1 is offset for the series.
        MOV R2, #10             ;Total numbers = 10.
        MOV R3,#0               ;Initialize the destination reg R3=0.
LOOP    LDR R4, [R1], #4        ;Load next consecutive number from series.
        ADD R3, R3, R4          ;Add the nos => R3 = R3 + R4.
        SUB R2, R2, #1          ;Decrement the counter.
        CMP R2,#0               ;is the counter zero ie if R2=0?
        BNE  LOOP                 ; if  R2  !=0  =>  branch  to  LOOP

        AREA  My_Program,  DATA
series  DCD  35,16,91,4,67,28,8,69,2,93  ;  define  the  numbers  of  series.
        END
```

RESULT



Register	Value
Current	
R0	0x00000000
R1	0x0000004C
R2	0x00000000
R3	0x0000019D
R4	0x0000005D
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x0000004C
CPSR	0x600000D3
SPSR	0x00000000
User/System	
Fast Interrupt	

```
                AREA My_Program, CODE
                ENTRY

                LDR R1, = series          ; R1 is offset for the series..
                MOV R2, #10               ; Total numbers = 10.
                MOV R3,#0                 ; Initialize the destination reg R3=0.
LOOP            LDR R4, [R1], #4          ; Load next consecutive number from series.
                ADD R3, R3, R4            ; Add the nos => R3 = R3+ R4.
                SUB R2, R2, #1            ; Decrement the counter.
                CMP R2,#0                 ; Is the conter zero ie if R2=0?
                BNE LOOP                  ; if R2 !=0 => branch to LOOP

               AREA My_Program, DATA
series         DCD 35,16,91,4,67,28,8,69,2,93   ; define the numbers of series.
               END
```

PROGRAM → 13

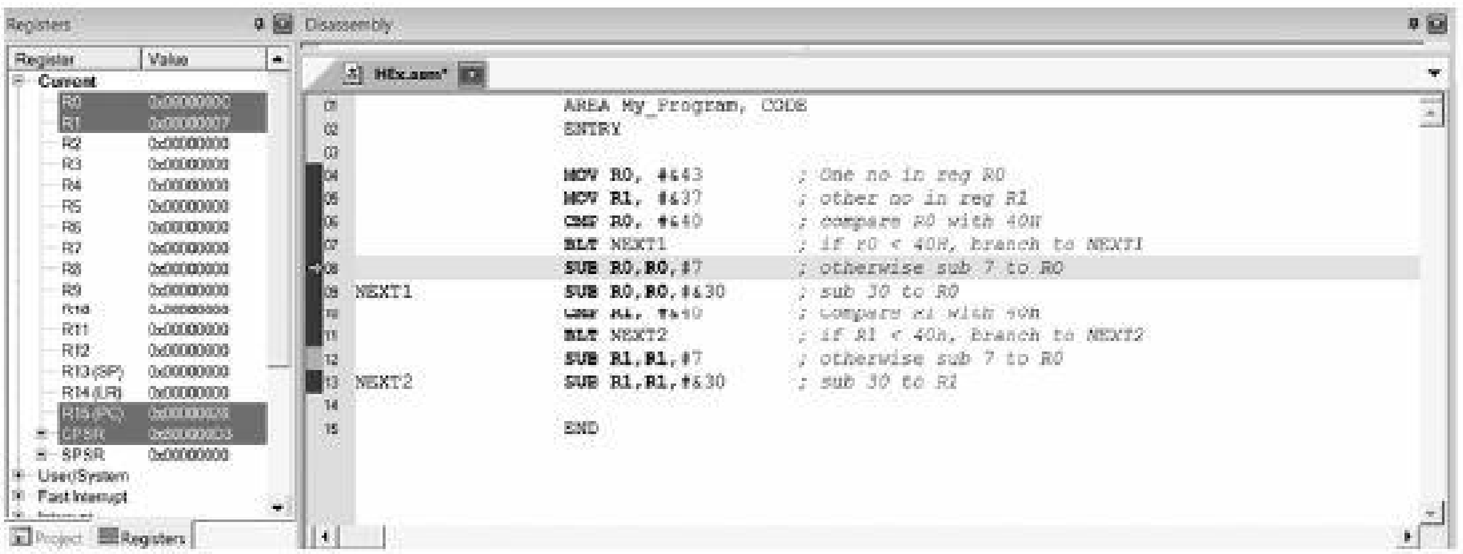
To Convert ASCII to its HEX Equivalent

****

Code:

```
AREA  My_Program,  CODE
ENTRY

        MOV  R0,  #&43          ;One  no  in  reg  R0
        MOV  R1,  #&37          ;other  no  in  reg  R1
        CMP  R0,  #&40          ;compare  R0  with  40H
        BLT  NEXT1              ;if  r0  <  40H,  branch  to  NEXT1
        SUB  R0,R0,#7           ;otherwise  sub  7  to  RO
NEXT1   SUB  R0,R0,#&30         ;sub  30  to  R0
        CMP  R1,  #&40          ;Compare  R1  with  40h
        BLT  NEXT2              ;if  R1  <  40h,  branch  to  NEXT2
        SUB  R1,R1,#7           ;otherwise  sub  7  to  R0
NEXT2   SUB  R1,R1,#&30         ;sub  30  to  R1

        END
```

RESULT



```
Register      Value
Current
  R0          [illegible]
  R1          0x00000007
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x00000028
  CPSR        0x000000D3
  SPSR        0x00000000
User/System
Fast Interrupt
```

```
                AREA My_Program, CODE
                ENTRY

                MOV R0, #&43            ; One no in reg R0
                MOV R1, #&37            ; other no in reg R1
                CMP R0, #&40            ; compare R0 with 40H
                BLT NEXT1               ; if r0 < 40H, branch to NEXT1
                SUB R0,R0,#7            ; otherwise sub 7 to R0
NEXT1           SUB R0,R0,#&30          ; sub 30 to R0
                CMP R1, #&40            ; Compare R1 with 40h
                BLT NEXT2               ; if R1 < 40h, branch to NEXT2
                SUB R1,R1,#7            ; otherwise sub 7 to R0
NEXT2           SUB R1,R1,#&30          ; sub 30 to R1

                END
```

PROGRAM → 14

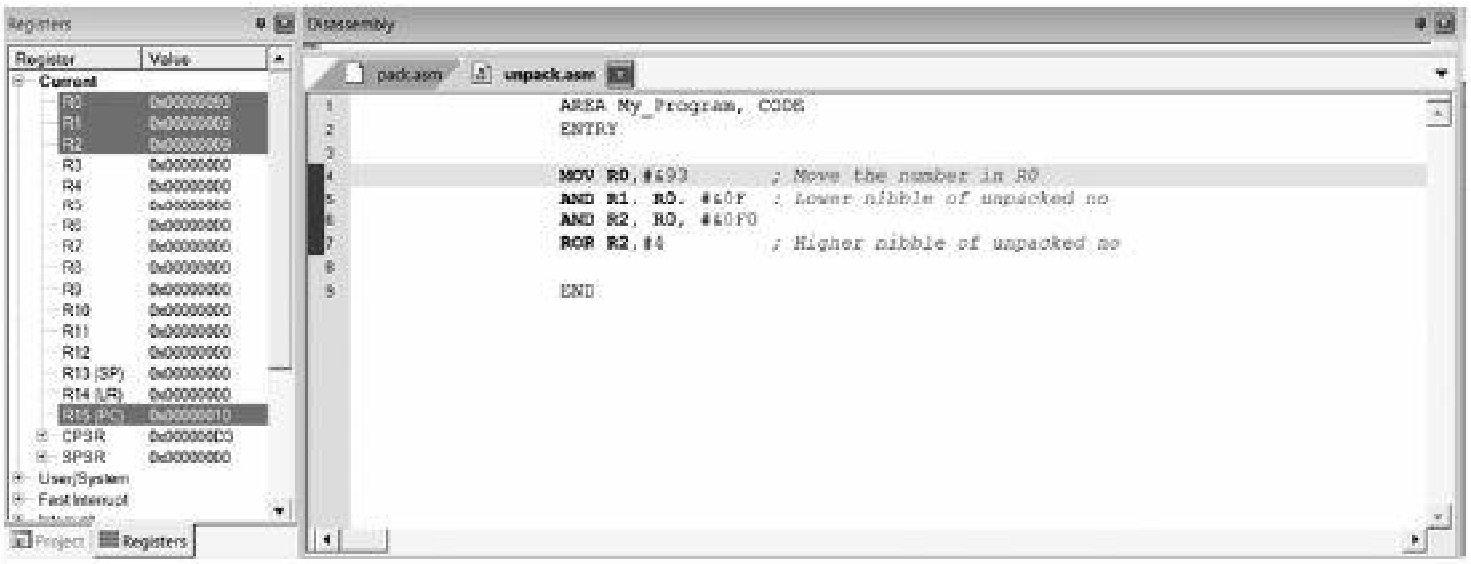
To Convert a Packed BCD Number to Unpacked Form

****

Code:

```
AREA My_Program, CODE
ENTRY

     MOV R0,#&93            ;Move the number in R0
     AND R1, R0, #&0F       ;Lower nibble of unpacked no
     AND R2, R0, #&0F0
     ROR R2,#4              ;Higher nibble of unpacked no

     END
```

RESULT



```
Register      Value
Current
  R0          [illegible]
  R1          [illegible]
  R2          [illegible]
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x00000010
  CPSR        0x000000D3
  SPSR        0x00000000
User/System
Fast Interrupt
```

```
                AREA My_Program, CODE
                ENTRY

                MOV R0,#&93         ; Move the number in R0
                AND R1, R0, #&0F    ; Lower nibble of unpacked no
                AND R2, R0, #&0F0
                ROR R2,#4           ; Higher nibble of unpacked no

                END
```

PROGRAM → 15

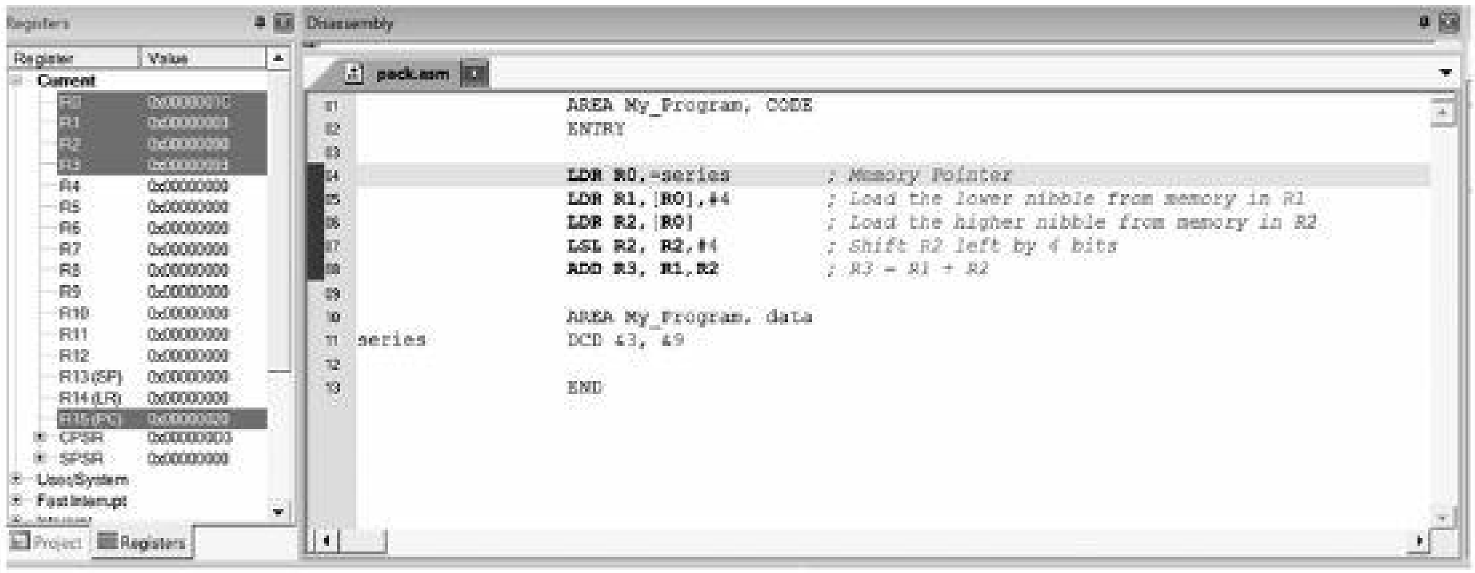
To Convert an Unpacked BCD No. to its Packed form (Little Endian)

****

Code:

```
    AREA  My_Program,  CODE
    ENTRY
           LDR  R0,=series          ;Memory  Pointer
           LDR  R1,[R0],#4          ;Load  the  lower  nibble  from  memory  in  R1
           LDR  R2,[R0]             ;Load  the  higher  nibble  from  memory  in  R2
           LSL  R2,  R2,#4          ;Shift  R2  left  by  4  bits
           ADD  R3,  R1,R2          ;R3  =  R1  +  R2
           AREA  My_Program,  data
    series DCD  &3,  &9
           END
```

RESULT



```
Register        Value
  Current
    R0          0x0000001C
    R1          0x00000003
    R2          0x00000090
    R3          0x00000093
    R4          0x00000000
    R5          0x00000000
    R6          0x00000000
    R7          0x00000000
    R8          0x00000000
    R9          0x00000000
    R10         0x00000000
    R11         0x00000000
    R12         0x00000000
    R13 (SP)    0x00000000
    R14 (LR)    0x00000000
    R15 (PC)    0x00000020
  + CPSR        0x000000D3
  + SPSR        0x00000000
+ User/System
+ Fast Interrupt
```

```
pack.asm
                AREA My_Program, CODE
                ENTRY

                LDR R0,=series          ; Memory Pointer
                LDR R1,[R0],#4          ; Load the lower nibble from memory in R1
                LDR R2,[R0]             ; Load the higher nibble from memory in R2
                LSL R2, R2,#4           ; Shift R2 left by 4 bits
                ADD R3, R1,R2           ; R3 = R1 + R2

                AREA My_Program, data
series          DCD &3, &9

                END
```

PROGRAM → 16

32-bit Division

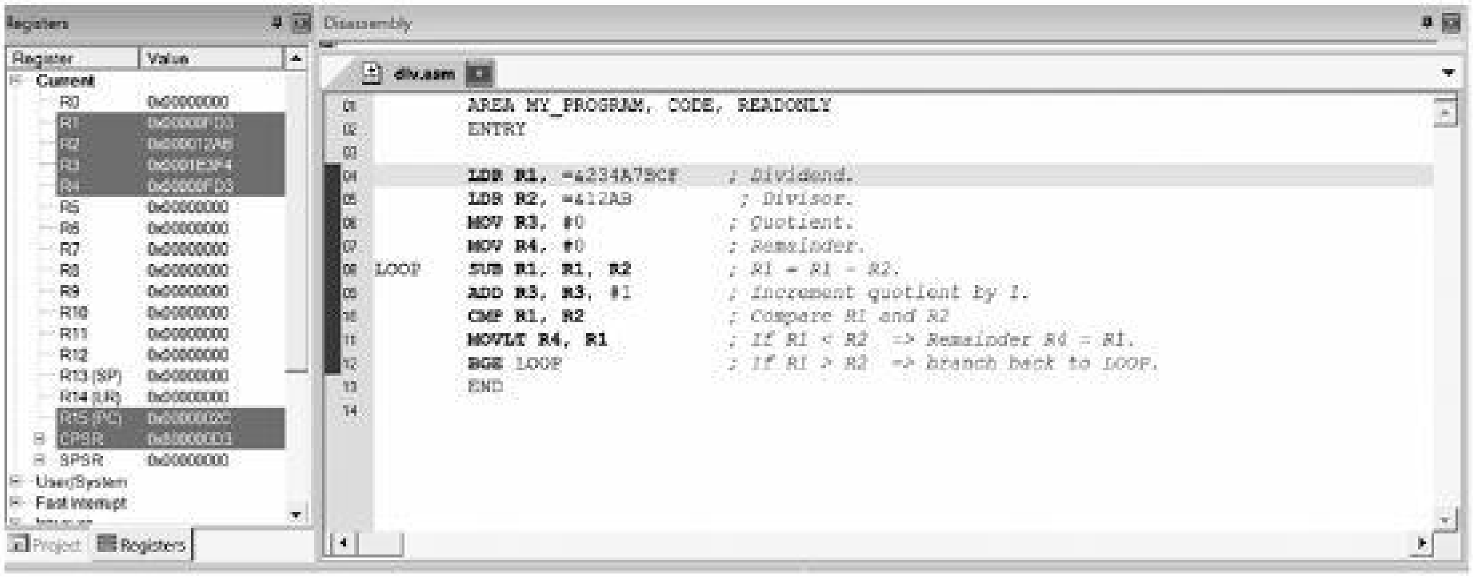
****

Code:

```
    AREA  MY_PROGRAM,  CODE,  READONLY
    ENTRY

           LDR  R1,  =&234A7BCF     ;Dividend.
           LDR  R2,  =&12AB         ;Divisor.
```

```
        MOV  R3,  #0           ;Quotient.
        MOV  R4,  #0           ;Remainder.
LOOP    SUB  R1,  R1,  R2      ;R1 = R1 - R2.
        ADD  R3,  R3,  #1      ;Increment quotient by 1.
        CMP  R1,  R2           ;Compare R1 and R2
        MOVLT  R4,  R1         ;If R1 < R2 => Remainder R4 = R1.
        BGE  LOOP              ;If R1 > R2 => branch back to LOOP.
        END
```

RESULT



Register	Value
Current	
R0	0x00000000
R1	0x00000FD3
R2	0x000012AB
R3	0x0001E3F4
R4	0x00000FD3
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x0000002C
CPSR	0x800000D3
SPSR	0x00000000
User/System	
Fast Interrupt	

```
div.asm
         AREA MY_PROGRAM, CODE, READONLY
         ENTRY

         LDR R1, =&234A78CF     ; Dividend.
         LDR R2, =&12AB          ; Divisor.
         MOV R3, #0             ; Quotient.
         MOV R4, #0             ; Remainder.
LOOP     SUB R1, R1, R2         ; R1 = R1 - R2.
         ADD R3, R3, #1         ; Increment quotient by 1.
         CMP R1, R2             ; Compare R1 and R2
         MOVLT R4, R1           ; If R1 < R2  => Remainder R4 = R1.
         BGE LOOP               ; If R1 > R2  => branch back to LOOP.
         END
```

9.7 THE PIC MICROCONTROLLER FAMILY

The PIC microcontroller is a family of microcontrollers manufactured by the Microchip Technology Inc. Currently, the PIC is one of the most popular microcontrollers used in education, and in commercial and industrial applications. The family consists of over 140 devices, ranging from simple 4-pin dual in-line devices with 0.5 K memory, to 80-pin complex devices with 32 K memory. Commonly, PIC stands for Peripheral Interface Controller. Even though the family consists of a large number of devices, all the devices have the same basic structure, offering the following fundamental features:

- reduced instruction set (RISC) with only 35 instructions;
- bidirectional digital I/O ports;
- RAM data memory;
- rewritable flash, or one-time programmable program memory;
- on-chip timer with pre-scalar;
- watchdog timer;
- power-on reset;
- external crystal operation;

- 25 mA current source/sink capability;
- power-saving sleep mode.

More complex devices offer the following additional features:

- analog input channels;
- analog comparators;
- serial USART;
- non-volatile EEPROM memory;
- additional on-chip timers;
- external and internal (timer) interrupts;
- PWM output;
- CAN bus interface;
- I2C bus interface;
- USB interface;
- LCD interface.

The PIC microcontroller product family currently consists of six groups:

- PIC10FXXX 12-bit program word;
- PIC12CXXX/PIC12FXXX 12/14-bit program memory;
- PIC16C5X 12-bit program word;
- PIC16CXXX/PIC16FXXX 14-bit program word;
- PIC17CXXX 16-bit program word;
- PIC18CXXX/PIC18FXXX 16-bit program word.

9.7.1 The 10FXXX Family

Table 9.2 gives a summary of the features of this family. PIC10F200 is a member of this family with the following features.

Table 9.2 Some PIC10FXXX Family Members

Microcontroller	Program Memory	Data Ram	I/O Pin	Comparator	A/D Convertor
10F200	256 × 12	16	4	0	0
10F202	512 × 12	24	4	0	0
10F204	256 × 12	16	4	0	0
10F206	512 × 12	24	4	1	0
10F220	256 × 12	16	4	0	3/8 bit

PIC10F200. This microcontroller is available in a 6-pin SOT-23 package, or in 8-pin PDIP package. The device has 33 instructions, 256 × 12 word flash program memory, 16 bytes of RAM data memory, four I/O ports, and one 8-bit timer. Clocking is from a precision 4 MHz internal oscillator. Other members of the family have larger memories and also an internal comparator.

9.7.2 The 12CXXX/PIC12FXXX Family

Table 9.3 gives a summary of the features of this family. The PIC12C508 is a member of this family with the following features.

Table 9.3 Some 12CXXX/PIC12FXXX Family Members

Microcontroller	Program Memory	Data Ram	I/O Pin	EEPROM	A/D Convertor
12c508	256 × 12	25	6	0	0
12c509	1024 × 12	41	6	0	0
12c671	1024 × 14	128	6	0	4
12F629	1024 × 14	64	6	128	0
12F675	256 × 12	64	6	128	4

PIC12C508. This is another low-cost microcontroller available in an 8-pin dual in-line package. The device has 512 × 12 word flash memory, 25 bytes of RAM data memory, six I/O ports, and one 8-bit timer. Operation is from a 4 MHz clock. Other members of the family have larger memories, higher speed, and A/D converters (e.g., 12C672).

The PIC12FXXX family has the same structure as the 12CXXX but with flash program memory and additional EEPROM data memory.

9.7.3 The 16C5X Family

Table 9.4 gives a summary of the features of this family. These devices have 14-, 18-, 20- and 28-pin packages. The PIC16C54 is a member of this family with the following features.

PIC16C54. This is an 18-pin microcontroller with 384 × 12 EPROM program memory. The device has 25 bytes of RAM, 12 I/O port pins, a timer and a watchdog timer. Other members of the family have larger memories and more I/O ports.

Table 9.4 Some 16C5X Family Members

Microcontroller	Program Memory	Data Ram	I/O	EEPROM	A/D Converter
16c54	512 × 12	25	12	0	0
16c56	1024 × 12	25	12	0	0
16c58	2048 × 14	73	12	0	0
16c505	1024 × 14	12	72	0	0

9.7.4 The 16CXXX Family

Table 9.5 gives a summary of the features of this family. These devices are similar to the 16CXX series, but they have 14 bits of program memory and some of them have A/D converters. The PIC16C554 is a member of this family with the following features.

Table 9.5 Some 16CXXX Family Members

Microcontroller	Program Memory	Data Ram	I/O Pin	EEPROM	A/D Converter
16c554	512 × 12	80	13	0	0
16c620	512 × 12	25	96	13	0
16c642	4096 × 14	176	22	0	0
16c716	2048 × 14	128	13	0	4
16c926	8192 × 14	336	52	0	5

PIC16C554. This is an 18-pin device with 512 × 14 program memory. The data memory is 80 bytes and 13 I/O port pins are provided. Other members of this family provide A/D converters (e.g., PIC16C76), USART capability (e.g., PIC16C67), larger memory, more I/O port pins and higher speed.

The PIC16FXXX family (e.g., PIC16F74) is upward compatible with the PIC16CXXX family. These devices are also 14 bits wide and have flash program memory and an internal 4 MHz clock oscillator as added features.

9.7.5 The 17CXXX Family

These are 16-bit microcontrollers. The program memory capacity ranges from 8192 × 16 (e.g., PIC17C42) to 16384 × 16 (e.g., PIC17C766). The devices also have larger RAM data memories, higher current sink capabilities and larger I/O port pins, e.g., the PIC17C766 provides 66 I/O port pins. Table 9.6 gives a summary of the features of this family.

Table 9.6 Some 17CXXX Family Members

Microcontroller	Program Memory	Data Ram	I/O Ram	EEPROM	A/D Convertor
17c42	2048 × 16	232	33	0	0
16c56	4096 × 16	454	33	0	0
17c762	8192 × 16	678	66	0	16
17c766	16384 × 14	902	66	0	16

9.7.6 The PIC18CXXX Family

These are high-speed 16-bit microcontrollers, with maximum clock frequency of 40 MHz. The devices in this family have large program and data memories, a large number of I/O pins, and A/D converters. They have an instruction set with 77 instructions, including multiplication. Table 9.7 gives a summary of the features of this family.

PIC18FXXX family members are upward compatible with PIC18CXXX. These microcontrollers in addition offer flash program memories and EEPROM data memories. Some members of the family provide up to 65 536 × 16 program memories and 3840 bytes of RAM memory.

Table 9.7 Some 18CXXX Family Members

Microcontroller	Program Memory	Data Ram	I/O	EEPROM	A/D Convertor
18c242	8192 × 16	512	2	0	5
18c452	16384 × 16	1536	34	0	8
18c658	16384 × 16	1536	52	0	12
18c858	16384 × 14	1536	68	0	16
18F242	8192 × 16	256	23	256	5

9.7.7 Minimum PIC Configuration

The minimum PIC configuration depends on the type of microcontroller used. Normally, the operation of a PIC microcontroller requires a power supply, reset circuit and an oscillator. The power supply is usually +5 V and, can be obtained from the mains supply by using a step-down transformer, a rectifier circuit and a power regulator chip, such as the LM78L05.

Although PIC microcontrollers have built-in power-on reset circuits, it is useful in many applications to have external reset circuits. When the microcontroller is reset, all of its special function registers are put into a known state and execution of the user program starts from address 0 of the program memory.

A reset is normally achieved by connecting a 4.7 K pull-up resistor from the master clear (MCLR) input to the supply voltage. Sometimes the voltage rises too slowly and the simple reset circuit may not work. In many applications it may be required to reset the microcontroller by pressing an external button.

PIC microcontrollers have built-in clock oscillator circuits. Additional components are needed to enable such clock oscillator circuits to function; some PIC microcontrollers have these built in, while others require external components. The internal oscillator can be operated in one of six modes as discussed below:

- External oscillator;
- LP – low-power crystal;
- XT – low-speed crystal/resonator;
- HS – high-speed crystal/resonator;
- RC – resistor and capacitor;
- No external components (only some PICs).

9.7.7.1 External Oscillator

An external oscillator can be connected to the OSC1 input. The oscillator should generate square wave pulses at the required frequency. The timing accuracy depends upon the accuracy of this external oscillator. When operated in this mode, the chip should be programmed for LP, XT or HS clock mode.

9.7.7.2 Crystal Operation

An external crystal should be used when very accurate timing is required. The crystal should be connected between the OSC1 and OSC2 inputs together with a pair of capacitors. For example, with a crystal of 4 MHz, two 22 pF capacitors can be used.

9.7.7.3 Resonator Operation

Resonators are usually available in the frequency range of about 4–8 MHz. Although resonators are not as accurate as crystals, they are usually accurate enough for most applications. Resonators have the advantages that they are low-cost and only one component is required compared to three components in the case of crystals (the crystal itself and two capacitors).

9.7.7.4 RC Operation

There are many low-cost applications where the timing accuracy is not important—flashing an LED every second, scanning a keyboard, reading the temperature every second, etc. In such applications the clock pulses can be generated by using an external resistor and a capacitor. The resistor and the capacitor should be connected to the OSC1 input of the microcontroller. The oscillator frequency depends upon the values of the resistor and the capacitor, the supply voltage and environmental factors, such as the temperature. For example, a 5 K resistor and a 20 pF capacitor can be used to generate a clock frequency of about 4 MHz.

9.7.7.5 Internal Clock

Some PIC microcontrollers (e.g., PIC12C672) have built-in clock generation circuitry and do not require any external components to generate the clock pulses. The built-in oscillator is usually 4 MHz and can be selected during the programming of the devices.

9.7.8 PIC16F84 Microcontroller

The PIC16F84 is one of the most popular PIC microcontrollers used in many commercial, industrial and hobby applications. This is an 18-pin device, which can operate at up to 20 MHz clock speed. It offers 1024 × 14 flash program memory, 68 bytes of RAM data memory, 64 bytes of EEPROM non-volatile data memory, 8-bit timer with pre-scaler, watchdog timer, 13 I/O pins, external and internal interrupt sources, and large current sink and source capability.

Figure 9.12 shows the pin configuration of the PIC16F84. The functions of various pins are as follows:

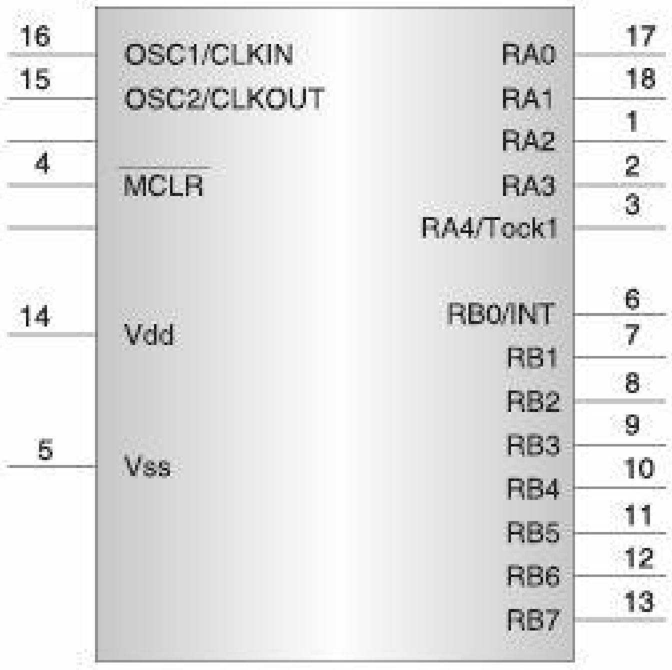


Fig. 9.12 PIC 16F84 pin Configuration.

RB0–RB7	Bidirectional port B pins
RA0–RA4	Bidirectional port A pins
Vdd	Supply voltage
Vss	Ground
OSC1	Crystal, resonator, or external clock input
OSC2	Crystal or resonator input
MCLR	Reset input
INT	External interrupt input (shared with RB0)
T0CK1	Optional timer clock input (shared with RA3)

Note that some pin names have a bar on them – for example, MCLR. This means that the pin will be active when the applied signal is at logic low (logic 0 in this case).

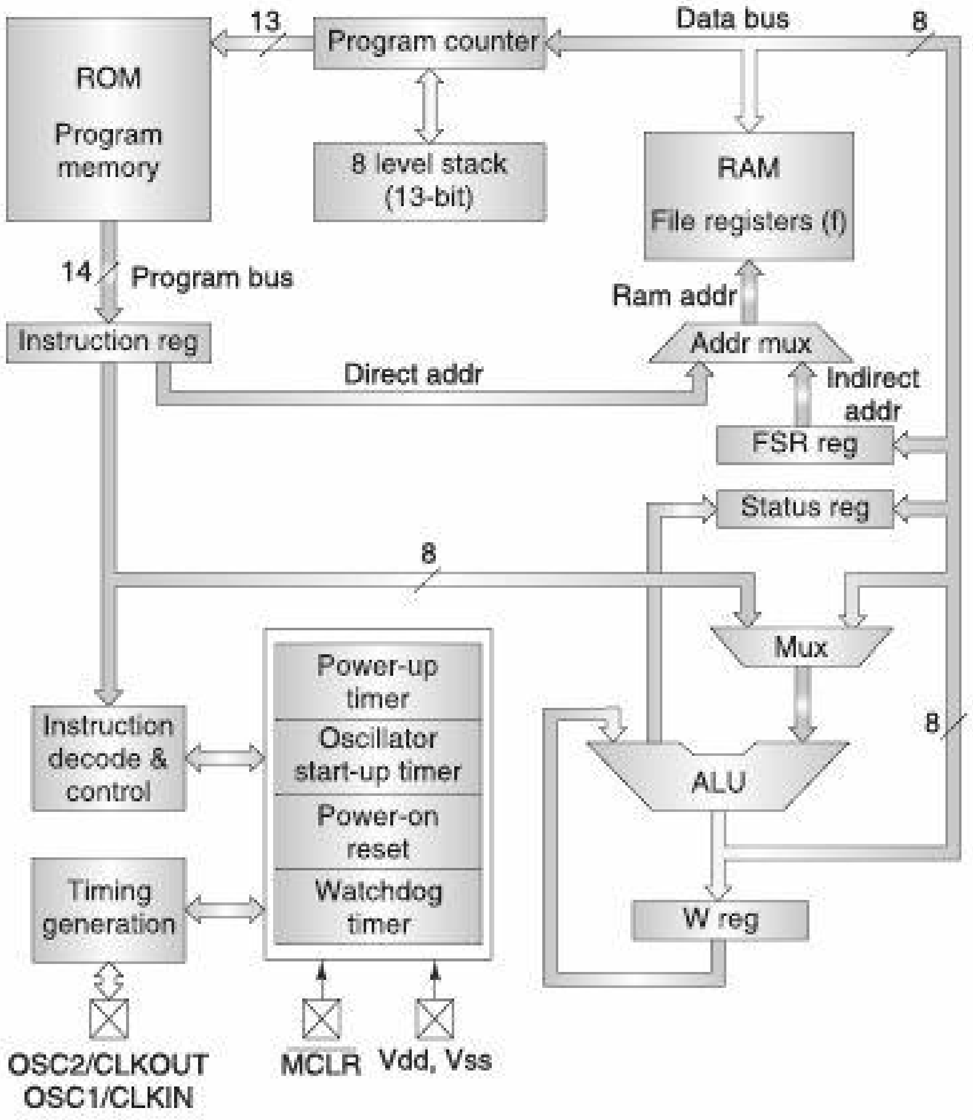


Fig. 9.13 Detailed Block Outline of PIC 16F84 Microcontroller.

The PIC16F84 provides four external or internal interrupt sources:

- External interrupt on INT pin;
- State change interrupt on the four higher bits of port B (RB4–RB7);
- EEPROM memory data write complete interrupt.

9.7.8.1 Central Processing Unit

Central processing unit (CPU) is the brain of a microcontroller as shown in Figure 9.14. That part is responsible for finding and obtaining the right instruction which needs to be executed, for decoding that instruction, and finally for its execution. Central processing unit connects all parts of the microcontroller into one whole. Surely, its most important function is to decode program instructions. When the programmer writes a program, instructions have a clear form like MOVLW 0 × 20.

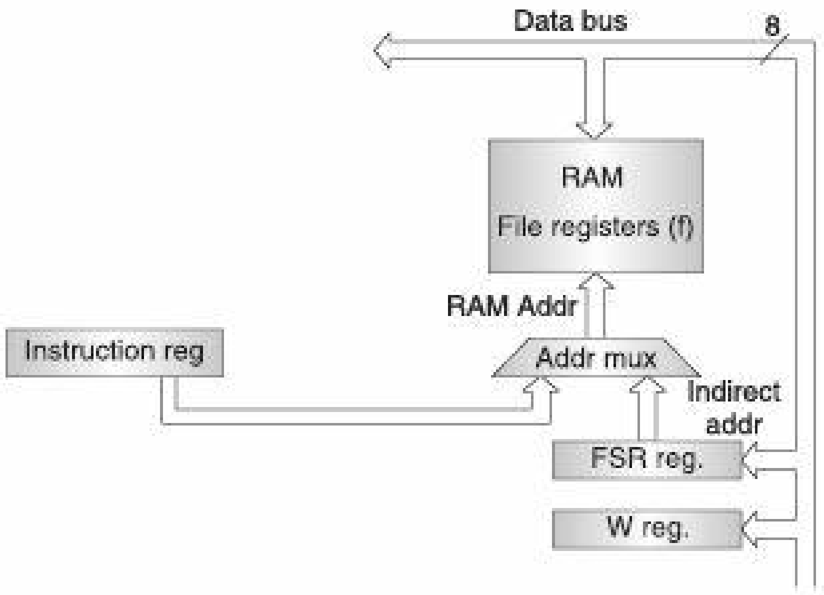


Fig. 9.14 Outline of the Central Processing Unit – CPU.

However, in order for a microcontroller to understand that, this 'letter' form of an instruction must be translated into a series of zeros and ones which is called an 'opcode'. This transition from a letter to binary form is done by translators such as assembler translator (also known as an assembler). Instruction thus derived from program memory must be decoded by a central processing unit.

We can then select from the table of all the instructions a set of actions which execute a needed assignment defined in that instruction. As instructions may within themselves contain assignments which require different transfers of data from one memory into another, from memory onto ports, or some other calculations, CPU must be connected with all parts of the microcontroller. This is made possible through a data bus and an address bus.

9.7.8.2 Arithmetic Logic Unit (ALU)

Arithmetic logic unit as shown in Figure 9.15 is responsible for performing operations of adding, subtracting, moving (left or right within a register) and logic operations. Moving data inside a register is also known as 'shifting'. PIC16F84 contains an 8-bit arithmetic logic unit and 8-bit work registers. In instructions with two operands, ordinarily one operand is in work register (W register), and the other is one of the registers or a constant. By operand we mean the contents on which some operation is being done, and a register is any one of the GPR or

SFR registers. GPR is short for 'General-Purposes Registers', and SFR for 'Special Function Registers'. In instructions with one operand, an operand is either W register or one of the registers. As an addition in doing operations in arithmetic and logic, ALU controls status bits (bits found in STATUS register). Execution of some instructions affects status bits, which depends on the result itself. Depending on which instruction is being executed, ALU can affect values of Carry (C), Digit Carry (DC), and Zero (Z) bits in STATUS register.

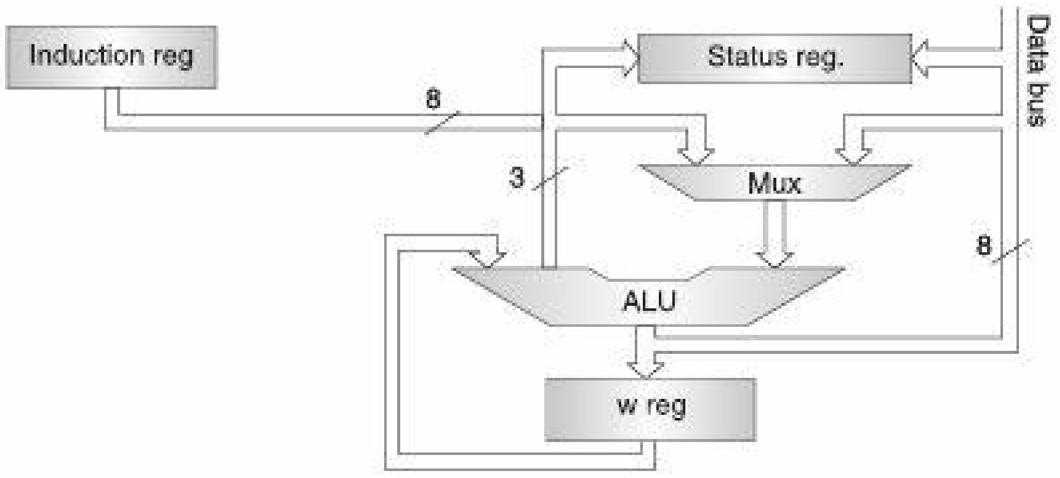


Fig. 9.15 Arithmetic-logic Unit and its Working.

9.7.8.3 STATUS Register

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	PR0	$\overline{\text{IO}}$	$\overline{\text{PD}}$	Z	DC	C
bit 7							bit 0

Legend:	
R = Readable bit	W = Writable bit
U = Unimplemented bit, read as '0'	–n = Value at POR reset

bit 0 C (Carry) Transfer

Bit affected by operations of addition, subtraction and shifting. This bit is set when a smaller value is being subtracted from a larger one, and is reset when a larger one is subtracted from a smaller one.

1 = transfer occurred from the highest resulting bit

0 = transfer did not occur

C bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

bit 1 DC (Digit Carry) DC Transfer

Bit affected by operations of addition and subtraction. Unlike C bit, this bit represents transfer from the fourth resulting place. It is set when smaller value is subtracted from a larger one, and is reset when a larger one is subtracted from a smaller one.

1 = transfer occurred on the fourth bit according to the order of the result

0 = transfer did not occur

DC bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

bit 2 Z (Zero bit) Indication of a zero result

This bit is set when the result of an executed arithmetic logic operation is zero.

1 = result equals zero

0 = result does not equal zero

bit 3 PD (Power-down bit)

Bit which is set whenever supply is brought to a microcontroller as it starts running, after each regular reset and after execution of instruction CLRWDT. Instruction SLEEP resets it when the microcontroller falls into low spending/usage regime. Its repeated setting is possible via reset or by turning the supply on, or off. Setting can be triggered also by a signal on RB0/INT pin, change on RB port, completion of writing in internal DATA EEPROM, and by a watchdog, too.

1 = after supply has been turned on

0 = executing SLEEP instruction

bit 4 TO Time-out ; Watchdog overflow.

Bit is set after turning on the supply and execution of CLRWDT and SLEEP instructions. Bit is reset when watchdog gets to the end signaling that something is not right.

1 = overflow did not occur

0 = overflow did occur

bit 5:6 RP1:RP0 (Register Bank Select bits)

Introduction to ARM and PIC Microcontrollers **669** These two bits are upper part of the address for direct addressing. Since instructions which address the memory directly have only seven bits, they need one more bit in order to address all 256 bytes which is how many bytes PIC16F84 has. RP1 bit is not used, but is left for some future expansions of this microcontroller.

01 = first bank

00 = zero bank

bit 7 IRP (Register Bank Select bit)

Bit whose role is to be an eighth bit for indirect addressing of internal RAM.

1 = bank 2 and 3

0 = bank 0 and 1 (from 00h to FFh)

STATUS register contains arithmetic status ALU (C, DC, Z), RESET status (TO, PD) and bits for selecting memory bank (IRP, RP1, RP0). Considering that selection of memory bank is controlled through this register, it has to be present in each bank. STATUS register can be a destination for any instruction, with any other register. If STATUS register is a destination for instructions which affect Z, DC or C bits, then writing to these three bits is not possible.

9.7.8.4 *OPTION Register*

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	PR0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C
bit 7							bit 0

Legend:	
R = Readable bit	W = Writable bit
U = Unimplemented bit, read as '0'	−n = Value at POR reset

Bits	TMR0	WDT
000	1.2	1.1
001	1.4	1.2
010	1.8	1.4
011	1.16	1.8
100	1.32	1.16
101	1.64	1.32
110	1.128	1.64
111	1.256	1.128

bit 3 PSA (Prescaler Assignment bit)

Bit which assigns prescaler between TMR0 and watchdog.

1 = prescaler is assigned to watchdog

0 = prescaler is assigned to a free timer TMR0

bit 4 T0SE (TMR0 Source Edge Select bit)

If it is allowed to trigger TMR0 by impulses from the pin RA4/T0CKI, this bit determines whether this will be to the falling or rising edge of a signal.

1 = falling edge

0 = rising edge

bit 5 TOCS (TMR0 Clock Source Select bit)

This pin enables free timer to increase its state either from internal oscillator on every 1/4 of oscillator clock, or through external impulses on RA4/T0CKI pin.

1 = external impulses

0 = 1/4 internal clock

bit 6 INTEDG (Interrupt Edge Select bit)

If interrupt is made possible this bit will determine the edge at which an interrupt will be activated on pin RB0/INT.

1 = rising edge

0 = falling edge

bit 7 RBPU (PORTB Pull-up Enable bit)

This bit turns on and off internal 'pull-up' resistors on port B.

1 = "pull-up" resistors turned off

0 = "pull-up" resistors turned on

9.7.8.5 Ports

Port refers to a group of pins on a microcontroller which can be accessed simultaneously, or on which we can set the desired combination of zeros and ones, or read from them an existing status. Physically, port is a register inside a microcontroller which is connected by wires to the pins of a microcontroller. Ports represent physical connection of Central Processing Unit with an outside world.

Microcontroller uses them in order to watch over or direct other components or devices. Due to functionality, some pins have twofold roles like PA4/TOCKI for instance, which is the fourth bit of port A and an external input for free counter. Selection of one of these two-pin functions is done in one of the configurationally registers. An illustration of this is the fifth bit T0CS in OPTION register. By selecting one of the functions the other one is disabled. All port pins can be defined as input or output, according to the needs of a device that's being developed. In order to define a pin as input or output pin, the right combination of zeros and ones must be written in TRIS register. If at the appropriate place in TRIS register a logical "1" is written, then that pin is an input pin, and if the opposite is true, it is an output pin. Every port has its proper TRIS register. Thus, port A has TRISA at address 85h, and port B has TRISB at address 86h.

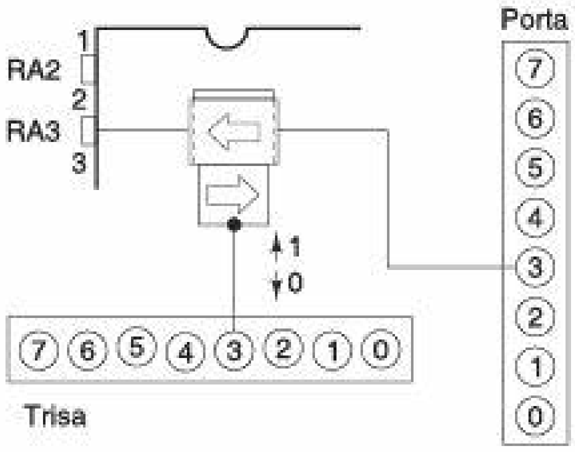


Fig. 9.16 Relationship between TRISA and Port A Register.

Port A

Port A as shown in Figure 9.16 has 5 pins joined to it. The corresponding register for data direction is TRISA at address 85h. Like with port B, setting a bit in TRISA register defines also the corresponding port pin as an input pin, and resetting a bit in TRISA register defines the corresponding port pin as an output pin. The fifth pin of port A has dual function. On that pin is also situated an external input for timer TMR0. One of these two options is chosen by setting or resetting the T0CS bit (TMR0 Clock Source Select bit). This pin enables the timer TMR0 to increase its status either from internal oscillator or via external impulses on RA4/T0CKI pin.

Port B

Port B has 8 pins joined to it. The appropriate register for direction of data is TRISB at address 86h. Setting a bit in TRISB register defines the corresponding port pin as an input pin, and resetting a bit in TRISB register defines the corresponding port pin as the output pin. Each pin on Port B has a weak internal pull-up resistor (resistor which defines a line to logic one) which can be activated by resetting the seventh bit RBPU in OPTION register. These 'pull-up' resistors are automatically being turned off when port pin is configured as an output. When a microcontroller is started, pull-up's are disabled.

Four pins Port B, RB7:RB4 can cause an interrupt which occurs when their status changes from logical one into logical zero and the other way around. Only pins configured as input can cause this interrupt to occur (if any RB7:RB4 pin is configured as an output, an interrupt won't be generated at the change of status.) This interrupt option along with internal pull-up resistors makes it easier to solve common problems we find in practice like for instance that of matrix keyboard. If rows on the keyboard are connected to these pins, each push on a key will then cause an interrupt. A microcontroller will determine which key is at hand while processing an interrupt It is not recommended to refer to port B at the same time that interrupt is being processed.

9.7.8.6 Memory Organization

PIC16F84 has two separate memory blocks, one for data and the other for program as shown in Figure 9.17. EEPROM memory and GPR registers in RAM memory make up a block for data, and FLASH memory makes up a program block.

Program Memory

Program memory has been realized in FLASH technology which makes it possible to program a microcontroller many times before it is installed into a device, and even after its installation if eventual changes in program or process parameters should occur. The size of program memory is 1024 locations with 14 bits width where locations zero and four are reserved for reset and interrupt vector.

Data Memory

Data memory consists of EEPROM and RAM memories. EEPROM memory consists of 64 eight-bit locations whose content is not lost during an interrupt in supply. EEPROM is not stored directly in memory space, but is accessed indirectly through EEADR and EEDATA registers. As EEPROM memory usually serves for storing important parameters (for example, of a given temperature in temperature regulators), there is a strict procedure for writing in EEPROM which must be followed in order to avoid accidental writing. RAM memory for data takes up space on a memory map from location 0x0C to 0x4F which comes to 68 locations. Locations of RAM memory are also called GPRs which is short for general-purpose registers. GPRs can be accessed regardless of which bank is selected at the moment.

9.7.8.7 SFR Registers

Registers which take up first 12 locations in banks 0 and 1 are registers of specialized function and have to do with working with certain blocks of the microcontroller. These are called Special Function Registers.

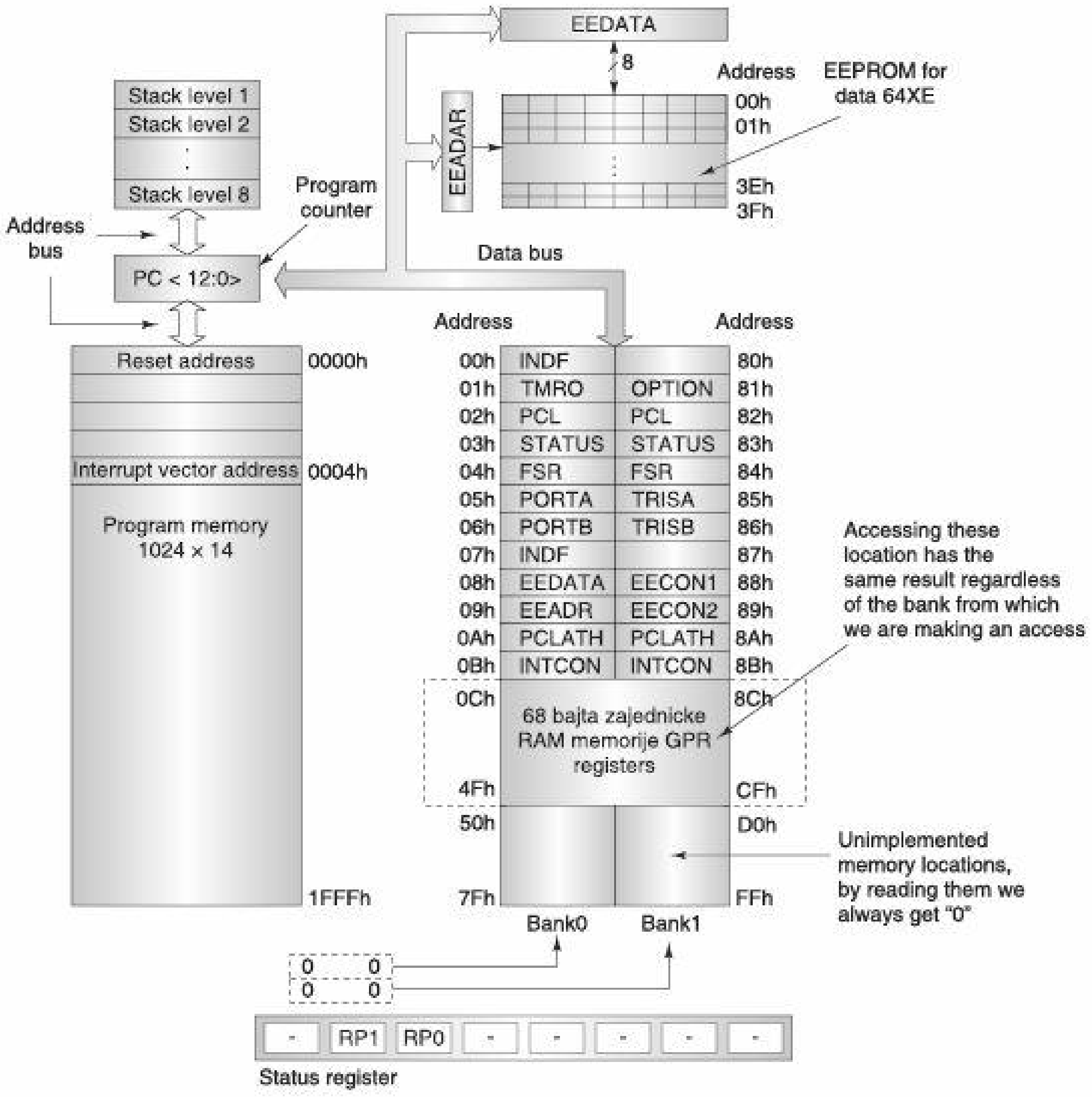


Fig. 9.17 Memory Organization of Microcontroller 16F84.

9.7.8.8 *Memory Banks*

Besides this 'linear' division to SFRs and GPRs, memory map is also divided in 'width' to two areas called 'banks'. Selecting one of the banks is done via RP0 and RP1 bits in STATUS register.

9.7.8.9 *Program Counter*

Program counter (PC) is a 13-bit register that contains the address of the instruction being executed. By its incrementing or change (e.g., in case of jumps) microcontroller executes program instructions one by one.

9.7.8.10 Stack

PIC16F84 has a 13-bit stack with 8 levels, or in other words, a group of 8 memory locations of 13 bits width with special function. Its basic role is to keep the value of program counter after a jump from the main program to an address of a subprogram being executed has occurred. In order for a program to know how to go back to the point where it started from, it has to return the value of a program counter from a stack. When moving from a program to a subprogram, program counter is being pushed onto a stack (example of this is CALL instruction). When executing instructions such as RETURN, RETLW or RETFIE, which are executed at the end of a subprogram, program counter is taken from a stack so that program could continue where it stopped before it was interrupted. These operations of placing on and taking off from a program counter stack are called PUSH and POP, and are named after instructions which exist on some bigger microcontrollers.

9.7.9 Interrupts

Interrupts are a mechanism of a microcontroller as shown in Figure 9.18 which makes it possible to respond to some events at the moment when they occur, regardless of what microcontroller is doing at the time. This is a very important part, because it provides connection between a microcontroller and the real world which surrounds us. Generally, each interrupt changes the flow of program execution, interrupts it and after executing an interrupt subprogram (interrupt routine) it continues from that same point on.

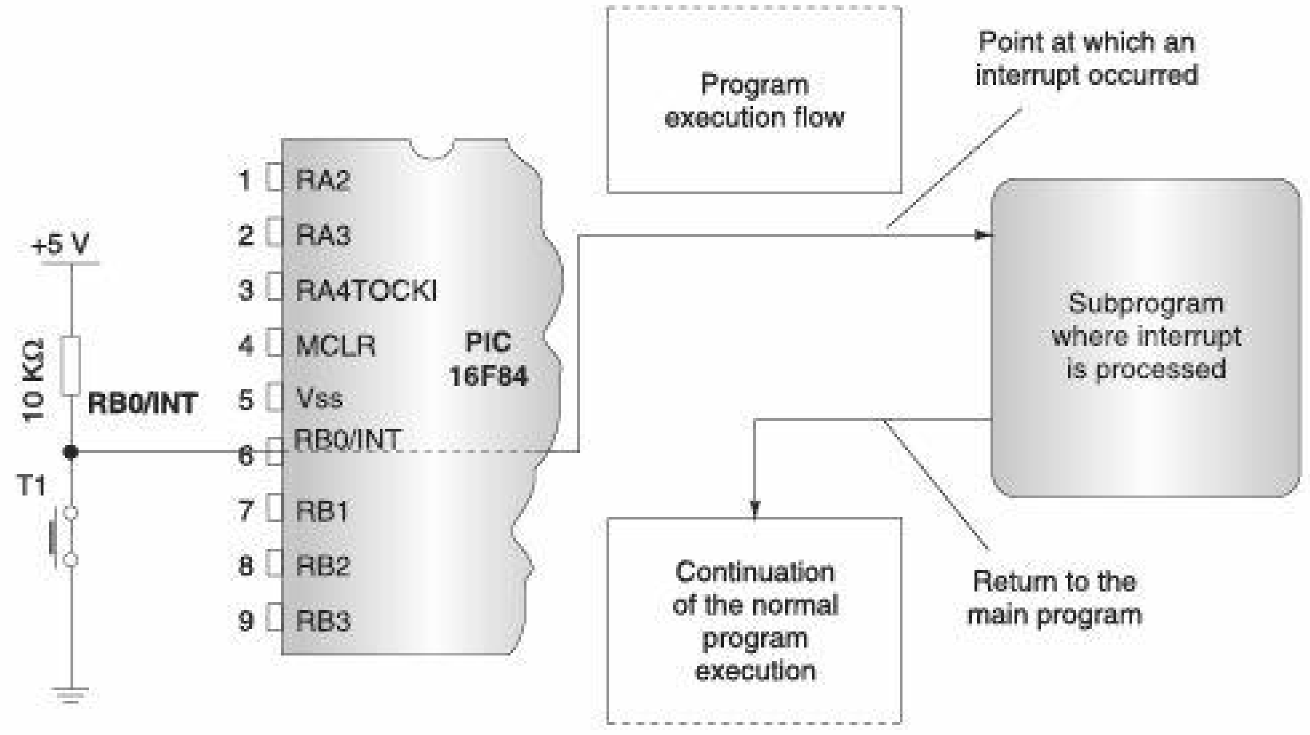

Program execution flow
Point at which an interrupt occurred
+5 V
10 KΩ
RB0/INT
T1
1 RA2
2 RA3
3 RA4TOCKI
4 MCLR
5 Vss
6 RB0/INT
7 RB1
8 RB2
9 RB3
PIC 16F84
Subprogram where interrupt is processed
Continuation of the normal program execution
Return to the main program

Fig. 9.18 One of the Possible Sources of an Interrupt and how it Affects the Main Program.

Control register of an interrupt is called INTCON and is found at 0Bh address. Its role is to allow or disallow interrupts, and in case they are not allowed, it registers specific interrupt requests through its own bits.

ITCON Register

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit
U = Unimplemented bit, read as '0'	–n = Value at POR reset

bit 0 RBIF (RB Port Change Flag bit) Bit which informs about changes on pins 4, 5, 6 and 7 of port B.

1 = at least one pin has changed its status

0 = no change occurred on any of the pins

bit 1 INTF (INT External Interrupt Flag bit) External interrupt occurred.

1 = interrupt occurred

0 = interrupt did not occur

If a rising or falling edge is detected on pin RB0/INT, (which is defined with bit INTEDG in OPTION register), bit INTF is set. Bit must be reset in interrupt subprogram in order to detect the next interrupt.

bit 2 T0IF (TMR0 Overflow Interrupt Flag bit) Overflow of counter TMR0.

1 = counter changed its status with FFh 00h

0 = overflow did not occur

Bit must be reset in program in order for an interrupt to be detected.

bit 3 RBIE (RB port change Interrupt Enable bit) Enables interrupts to occur at the change of status of pins 4, 5, 6, and 7 of port B.

1 = enables interrupts at the change of status

0 = interrupts disabled at the change of status

If RBIE and RBIF are simultaneously set, an interrupt will occur.

bit 4 INTE (INT External Interrupt Enable bit) Bit which enables external interrupt from pin RB0/INT.

1 = external interrupt enabled

0 = external interrupt disabled

If INTE and INTF are set simultaneously, an interrupt will occur.

bit 5 T0IE (TMR0 Overflow Interrupt Enable bit) Bit which enables interrupts during counter

1 = interrupt enabled

0 = interrupt disabled

If T0IE and T0IF are set simultaneously, interrupt will occur.

Bit 6 EEIE (EEPROM Write Complete Interrupt Enable bit) Bit, which enables an interrupt at the end of a writing routine to EEPROM.

1 = interrupt enabled

0 = interrupt disabled

If EEIE and EEIF (which is in EECON1 register) are set simultaneously, an interrupt will occur.

Bit 7 GIE (Global Interrupt Enable bit) Bit which allows or disallows all interrupts.

1 = all interrupts are enabled

0 = all interrupts are disabled

PIC16F84 has four interrupt sources:

1. Termination of writing data to EEPROM.
2. TMR0 interrupts caused by timer overflow.
3. Interrupt during alteration on RB4, RB5, RB6 and RB7 pins of port B.
4. External interrupts from RB0/INT pin of microcontroller.

Generally speaking, each interrupt source has two bits joined to it. One enables interrupts, and the other detects when interrupts occur. There is one common bit called GIE, which can be used to disallow or enable all interrupts simultaneously. This bit is very useful when writing a program because it allows for all interrupts to be disabled for a period of time, so that execution of some important part of a program would not be interrupted. When instruction, which resets GIE bit, is executed (GIE = 0, all interrupts disallowed), any interrupt that remained unsolved should be ignored.

Interrupts which remained unsolved and are ignored, are processed when GIE bit (GIE = 1, all interrupts allowed) is reset. When interrupt is answered, GIE bit is reset so that any additional interrupts would be disabled, return address is pushed onto stack and address 0004h is written in program counter—only after this does replying to an interrupt begin! After interrupt is processed, bit whose setting caused an interrupt must be reset, or interrupt routine will automatically be processed over again during a return to the main program.

Keeping the Contents of Important Registers

Only return value of program counter is stored on a stack during an interrupt (by return value of program counter, we mean the address of the instruction which was to be executed, but wasn't because interrupt occurred). Keeping only the value of program counter is often not enough. Some registers, which are already in use in the main program, can also be in use in interrupt routine. If they were not retained, main program would during a return from an interrupt routine get completely different values in those registers, which would cause an error in the program. One example for such a case is contents of the work register, W. If we suppose that main program was using work register, W, for some of its operations, and if it had stored in it some value that is important for the following instruction, then an interrupt which occurs before that instruction will change the value of work register (W) which will directly influence the main program. Procedure of recording important registers before going to an interrupt routine is called PUSH, while the procedure, which brings recorded values back, is called POP. PUSH and POP are instructions with some other microcontrollers (Intel), but are so widely accepted that a whole operation is named after them. PIC16F84 does not have instructions like PUSH and POP, and they have to be programmed.

Due to simplicity and frequent usage, these parts of the program can be made as macros. The concept of a Macro is explained in "Program assembly language". In the following example, contents of W and STATUS registers are stored in W_TEMP and STATUS_TEMP variables prior to interrupt routine. At the beginning of PUSH routine, we need to check presently selected bank because W_TEMP and STATUS_TEMP are found in bank 0. For exchange of data between these registers, SWAPF instruction is used instead of MOVF because it does not affect the status of STATUS register bits.

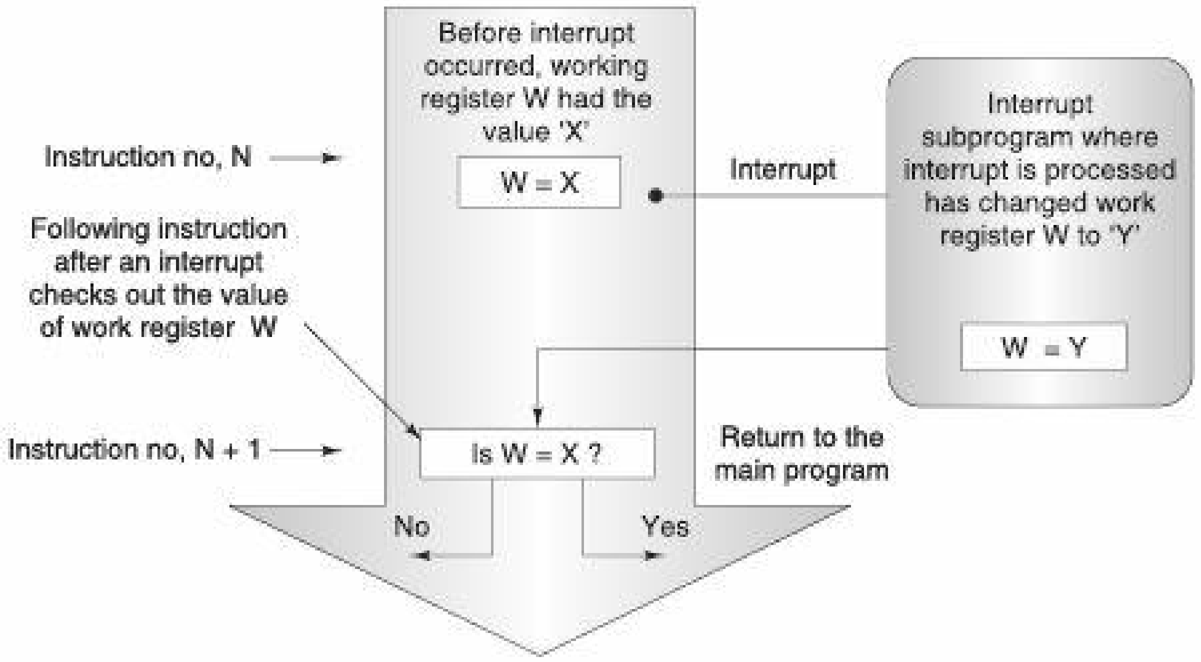

Before interrupt occurred, working register W had the value 'X'
Instruction no, N
W = X
Interrupt
Interrupt subprogram where interrupt is processed has changed work register W to 'Y'
Following instruction after an interrupt checks out the value of work register W
W = Y
Instruction no, N + 1
Is W = X ?
Return to the main program
No
Yes

Fig. 9.19 One of the Possible Cases of Errors if Saving is not done when going to a Subprogram of an Interrupt.

9.7.9.1 *External Interrupt on RB0/INT Pin of Microcontroller*

External interrupt on RB0/INT pin is triggered by rising signal edge (if bit INTEDG = 1 in OPTION<6> register), or falling edge (if INTEDG = 0). When correct signal appears on INT pin, INTF bit is set in INTCON register. INTF bit (INTCON<1>) must be reset in interrupt routine, so that interrupt wouldn't occur again while going back to the main program. This is an important part of the program, which a programmer must not forget, or the program will constantly go into interrupt routine. Interrupt can be turned off by resetting INTE control bit (INTCON<4>).

9.7.9.2 *Interrupt During a TMR0 Counter Overflow*

Overflow of TMR0 counter (with FFh on 00h) will set T0IF (INTCON<2>) bit. This is quite a significant interrupt because many real problems can be solved using this interrupt. One of the examples is time measurement. If we know how much time counter needs in order to complete one cycle from 00h to FFh, then a number of interrupts multiplied by that amount of time will yield the total of elapsed time. In interrupt routine some variable would be incremented in RAM memory, value of that variable multiplied by the amount of time the counter needs to count through a whole cycle, would yield total elapsed time. Interrupt can be turned on/off by setting/resetting T0IE (INTCON<5>) bit.

9.7.9.3 Interrupt During a Change on Pins 4, 5, 6 and 7 of Port B

Change of input signal on PORTB <7:4> sets RBIF (INTCON<0>) bit. Four pins RB7, RB6, RB5 and RB4 of port B, can trigger an interrupt which occurs when status on them changes from logic one to logic zero, or vice versa. For pins to be sensitive to this change, they must be defined as input. If any one of them is defined as output, interrupt will not be generated at the change of status. If they are defined as input, their current state is compared to the old value which was stored at the last reading from port B. Interrupt can be turned on/off by setting/resetting RBIE bit.

This interrupt is of practical nature only. Since writing to one EEPROM location takes about 10 ms (which is a long time in the notion of a microcontroller), it doesn't pay off to a microcontroller to wait for writing to end. Thus, interrupt mechanism is added which allows the microcontroller to continue executing the main program, while writing in EEPROM is being done in the background. When writing is completed, interrupt informs the microcontroller that writing has ended. EEIF bit, through, which this informing is done, is found in EECON1 register. Resetting the EEIE bit in INTCON register can disable occurrence of an interrupt.

9.7.9.4 Interrupt Initialization

In order to use an interrupt mechanism of a microcontroller, some preparatory tasks need to be performed. These procedures are in short called "initialization". By initialization, we define to what interrupts the microcontroller will respond, and which ones it will ignore. If we do not set the bit that allows a certain interrupt, program will not execute an interrupt subprogram.

Through this, we can obtain control over interrupt occurrence, which is very useful.

```
Clrf  INTCON             ; all  interrupt  disabled
Movlw B’  00010000       ; external  interrupt  only  is  enable
Bsf  INTCON,  GIE        ; occurrence  of  interrupt  allowed
```

The above example shows initialization of external interrupt on RB0 pin of a microcontroller. Where we see one being set, that means that interrupt is enabled. Occurrence of other interrupts is not allowed, and all interrupts together are disallowed until GIE bit is set to one.

9.7.10 EEPROM Data Memory

PIC16F84 has 64 bytes of EEPROM memory locations on addresses from 00h to 63h that can be written to or read from. The most important characteristic of this memory is that it does not lose its contents during supply. That practically means that what is written to it remains there even if the microcontroller is turned off. Data can be retained in EEPROM without supply for up to 40 years (as maker of PIC16F84 microcontroller claim), and up to 10000 cycles of writing can be executed.

In practice, EEPROM memory is used for storing important data or some process parameters. One such parameter is a given temperature, assigned when setting up a temperature regulator to some process. In case this data is not retained, it will be necessary to adjust a given temperature after each loss of supply. Since this is very impractical (and even dangerous), makers of microcontrollers have begun installing one smaller type of EEPROM memory.

EEPROM memory is contained in a special memory space and can be accessed through special registers.

These registers are:

- EEDATA at address 08h, which holds data that is read or that needs to be written.
- EEADR at address 09h, which contains an address of EEPROM location being accessed.
- EECON1 at address 88h, which contains control bits.
- EECON2 at address 89h. This register does not exist physically and serves to protect EEPROM from accidental writing.

EECON1 register at address 88h is a control register with five applied bits.

Bits 5, 6 and 7 are not used, and when read always are zero. Interpretation of EECON1 register bits follows.

EECON1 Register

bit 0 RD (Read Control bit)

Setting this bit initializes transfer of data from address defined in EEADR to EEDATA register. Since time is not as essential in reading data as in writing, data from EEDATA can already be used further in the next instruction.

1 = initializes reading

0 = does not initialize reading

U-0	U-0	U-0	R/W-1	R/W-1	R/W-x	R/W-0	R/W-x
–	–	–	EEIF (1)	WRERR	WREN	WR	RD
bit 7							bit 0

Legend:	
R = Readable bit	W = Writable bit
U = Unimplemented bit, read as '0'	–n = Value at POR reset

bit 1 WR (Write Control bit)

Setting of this bit initializes writing data from EEDATA register to the address on EEADR register.

1 = initializes writing

0 = does not initialize writing

bit 2 WREN (EEPROM Write Enable bit) Enables writing to EEPROM.

If this bit is not set, microcontroller will not allow writing to EEPROM.

1 = writing allowed

0 = writing disallowed

bit 3 WRERR (EEPROM Error Flag bit) Error during writing to EEPROM

This bit is set only in cases when writing to EEPROM was interrupted by a reset signal or by running out of time in watchdog timer (if it is activated).

1 = error occurred

0 = error did not occur

bit 4 EEIF (EEPROM Write Operation Interrupt Flag bit) Bit used to inform that writing data to EEPROM has ended.

When writing has terminated, this bit will be set automatically. Programmer must reset EEIF
bit in his program in order to detect new termination of writing.

1 = writing terminated

0 = writing not terminated yet, or has not started

9.7.10.1 Reading from EEPROM Memory

Setting the RD bit initializes transfer of data from address found in EEADR register to EEDATA register. As in reading data, we don't need so much time as in writing, data taken over from EEDATA register can already be used further in the next instruction.

9.7.10.2 Writing to EEPROM Memory

In order to write data to EEPROM location, the programmer must first write address to EEADR register and data to EEDATA register. Only then is it useful to set WR bit which sets the whole action in motion. WR bit will be reset, and EEIF bit set following a writing which may be used in processing interrupts. Values 55h and AAh are the first and the second key which make it impossible for accidental writing to EEPROM to occur. These two values are written to EECON2 which serves only that purpose, to receive these two values and thus prevent any accidental writing to EEPROM memory. Program lines marked as 1, 2, 3, and 4 must be executed in that order in even time intervals. Therefore, it is very important to turn off interrupts which could change the timing needed for executing instructions. After writing, interrupts can be enabled again in the end.

9.7.11 Instruction Set in PIC16Cxx Microcontroller Family

Complete set which encompasses 35 instructions is given in the following table. A reason for such a small number of instructions lies primarily in the fact that we are talking about a RISC microcontroller whose instructions are well optimized considering the speed of work, architectural simplicity and code compactness. The only drawback is that the programmer is expected to master "uncomfortable" technique of using a modest set of 35 instructions.

9.7.11.1 Data Transfer

Transfer of data in a microcontroller is done between work (W) register and an 'f' register that represents any location in internal RAM (regardless whether those are special or general-purpose registers).

First three instructions (look at the following table) provide for a constant being written in W register (MOVLW is short for MOVE Literal to W), and for data to be copied from W register onto RAM and data from RAM to be copied onto W register (or on the same RAM location, at which point only the status of Z flag changes). Instruction CLRF writes constant 0 in 'f' register, and CLRW writes constant 0 in register W. SWAPF instruction exchanges places of the 4-bit nibbles crosswise inside a register.

9.7.11.2 Arithmetic and Logic

Of all arithmetic operations, PIC like most microcontrollers supports only subtraction and addition. Flags C, DC and Z are set depending on a result of addition or subtraction, but with one exception: since subtraction is performed like addition of a negative value, C flag is inverse following a subtraction. In other words, it is set if operation is possible, and reset if larger number was subtracted from a smaller one.

Logic one of PIC has capability of performing operations AND, OR, EX-OR, negations (COMF) and rotation (RLF and RRF).

Instructions which rotate the register contents move bits inside a register through flag C by one space to the left (towards bit 7), or to the right (towards bit 0). Bit which "comes out" of a register is written in flag C, and status of that flag is written in a bit on the "opposite side" of the register.

9.7.11.3 Bit Operations

Instructions BCF and BSF do setting or resetting of one bit anywhere in the memory. Even though this seems like a simple operation, it is executed so that CPU first reads the whole byte, changes one bit in it and then writes in the entire byte at the same place.

9.7.11.4 Directing a Program Flow

Instructions GOTO, CALL and RETURN are executed the same way as on all other microcontrollers, only stack is independent of internal RAM and limited to eight levels.

'RETLW k' instruction is identical with RETURN instruction, except that before coming back from a subprogram a constant defined by instruction operand is written in W register. This instruction enables us to design easily the lookup tables (lists). Mostly we use them by determining data position on our table adding it to the address at which the table begins, and then we read data from that location (which is usually found in program memory).

Table can be formed as a subprogram which consists of a series of 'RETLW k' instructions, where 'k' constants are members of the table.

```
main        molov  2
            call  lookup
lookup      addwf  PCL,  f
            retlw  k
            retlw  k1
            retlw  k2
               .
               .
               .
            Retlw  kn
```

We write the position of a member of our table in W register, and using CALL instruction the position of a W register member to the starting address of our table, found in PCL register, and so we get the real data address in program memory. When returning from a subprogram we will have in W register the contents of an addressed table member. In a previous example, constant 'k2' will be in W register following a return from a subprogram.

RETFIE (RETurn From Interrupt-Interrupt Enable) is a return from interrupt routine and differs from a RETURN only in that it automatically sets GIE (Global Interrupt Enable) bit. Upon an interrupt, this bit is automatically reset. As interrupt begins, only the value of the program counter is put at the top of a stack. No automatic storing of register status is provided.

Conditional jumps are synthesized into two instructions: BTFSC and BTFSS. Depending on a bit status in 'f' register that is being tested, instructions skip or don't skip over the next program instruction.

9.7.11.5 Instruction Execution Period

All instructions are executed in one cycle except for conditional branch instructions if condition is true, or if the contents of program counter is changed by some instruction. In that case, execution requires two instruction cycles, and the second cycle is executed as NOP (No Operation). Four oscillator clocks make up one instruction cycle. If we are using an oscillator with 4 MHz frequency, the normal time for executing an instruction is 1 μs, and in case of conditional branching, execution period is 2 μs.

Word list:

F	any memory location in a microcontroller
W	register work
B	bit position in 'f' register
D	destination bit
Label	group of eight characters which marks the beginning of a part of the program
TO	top of stack
[ ]	option
< >	register bit field

PIC Programming Examples

Example 1: State the content of file register RAM location after the following program:

```
MOVLW        99H
MOVWF        85H
MOVLW        3FH
MOVWF        14H
MOVLW        63H
MOVWF        15H
MOVLW        12H
MOVWF        16H
```

Solution: After the execution of the MOVWF 12H filereg. RAM 12H has the 99H,

After the execution of the MOVWF 13H filereg. RAM 12H has the 85H,

After the execution of the MOVWF 14H filereg. RAM 12H has the 3FH,

After the execution of the MOVWF 15H filereg. RAM 12H has the 63H,

And, so on as shown in the chart

Address	Data
12h	99H
13H	85H
14H	14H
15H	3FH
16H	12H

Example 2: State the content of file register RAM locations 12H and WREG after the following program

```
MOVLW 0
MOVWF 12H
MOVLW 22H
ADDWF 12H, F
ADDWF 12H, F
ADDWF 12H, F
ADDWF 12H, F
```

Solution: The program clears both the WREG and RAM location 12H in the file register. Then it loads wreg with value 22h from then on, it adds the WREG register and location 12H of GP RAM has the value of 88H(4 × 22 H = 88H) and WREG = 22H.

Example 3: Rewrite the last example to place the sum in WREG as you add the file register locations and the WREG register.

```
MOVLW 0
MOVWF 12H
MOVLW 22H
ADDWF 12H, F
ADDWF 12H, F
ADDWF 12H, F
ADDWF 12H, F
```

Solution: The program adds WREG and location 12H and saves the result in WREG each time. At the end, location 12H has a value of 22H and WREG = 88H.

Example 4: Write a simple program to toggle the SFR of port B continuously forever.

Solution:

```
        MOVLW 55H           ; WREG = 55H
        MOVWF PORTB         ; move WREG to port B SFR(PB = 55H)
B1      COMF PORTB, F       ; Complement port B and place it in port B
        GOTO B1
```

9.8 THE AVR MICROCONTROLLER

AVR was developed in the year 1996 by Atmel Corporation. AVR derives its name from its developers and stands for Alf-Egil Bogen and Vegard Wollan. The AT90S8515 was the first microcontroller which was based on AVR architecture, however, the first microcontroller to hit the commercial market was AT90S1200 in the year 1997.

AVR microcontrollers are available in three categories:

- TinyAVR
- MegaAVR
- XmegaAVR

Table 9.8 AVR Series of Microcontrollers

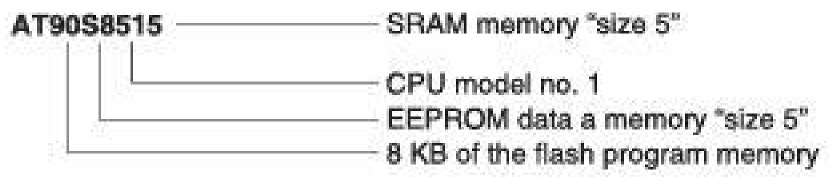
Series Name	Pins	Flash Memory	Special Feature
TinyAVR	6-32	0.5-8 KB	Small in size
MegaAVR	28-100	4-256 KB	Extended peripherals
XmegaAVR	44-100	16-384 KB	DMA , Event System included

The AVR family covers a huge range of devices. But the one special feature which differentiate it from other microcontroller family is that a program written for one type of AVR can be used in other type of AVR with a minimal changes.

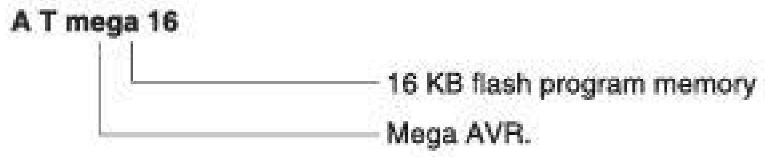
Table 9.9 Commercially Available AVR Microcontroller

Part Name	ROM	RAM	EEPROM	I/O pins	Timer	Interrupts	Voltage	Freq.	Packaging
ATmega8	8 KB	1 KB	512B	23	3	19	4.5-5.5 V	0-16 MHz	28
ATmega8L	8 KB	1 KB	512B	23	3	19	2.7-5.5 V	0-8 MHz	28
ATmega16	16 KB	1 KB	512B	32	3	21	4.5-5.5 V	0-16 MHz	40
ATmega16L	16 KB	1 KB	512B	32	3	21	2.7-5.5 V	0-8 MHz	40
ATmega32	32 KB	2 KB	1KB	32	3	21	4.5-5.5 V	0-16 MHz	40
ATmega32L	32 KB	2 KB	1KB	32	3	21	2.7-5.5 V	0-8 MHz	40
At90s8515	8 KB	512 KB	512KB	32	2	13	2.7-6.0V	0-4 MHZ	40

The name of the AVR model itself gives some information about its features. The following rule should be followed.


AT90S8515
SRAM memory "size 5"
CPU model no. 1
EEPROM data a memory "size 5"
8 KB of the flash program memory

0	1	2	3	4	5	6	7	8	9	A	B
0	32 byte	64 byte	128 byte	256 byte	512 byte	1 K	2 K	4 K	8 K	16 K	32 K



9.9 AVR FAMILY ARCHITECTURE

The AVR uses Harvard architecture. This entails separate data and program memory buses. The data memory data bus is an 8-bit bus and connects most of the peripheral components to the register file. The exact amount varies from processor to processor. The AT90S1200, the base level processor, has 1 Kbyte of program memory organized as 512-X-16 bits, while the Mega103 has 128 Kbytes of memory organized as 64 K-X-16 bits. A K here equals 1024 and not 1000. The program memory is accessed every clock cycle, and an instruction is loaded into the instruction register. The instruction register feeds the register file, selecting which of the registers will be used by the ALU for instruction execution. The instruction register output is also decoded by the instruction decoder to decide which control signals will be activated for completing the current instruction.

Data memory is divided into the following section:

1. A register file with 32 registers of 8-bit width. All processors of the AVR family have this register file. The working registers are mapped in as the first 32 memory addresses (0000_{16}–$001F_{16}$) and are followed by I/O registers.
2. 64 I/O registers of 8 bits each. All the processors do not have all the 64 registers. Some have more than others, depending on the number of peripheral components on the chip. The working registers are mapped in as registers (0020_{16}–$005F_{16}$).
3. Internal SRAM. Static Random Access Memory, this is the volatile memory of microcontroller, i.e., data is lost as soon as power is turned off. The amount of SRAM varies between 128 bytes and 4 Kbytes. The SRAM is used for stack as well as storing variables.
4. External SRAM. This is possible only on the larger processors of the AVR family. Those processors that have external data and memory access ports (such as the AT90S8515) can use any available external SRAM the user may decide to implement.
5. EEPROM. EEPROM is used to store the program dumped or burnt by the user on to the microcontroller. It can be easily erased electrically as a single unit. Flash memory is non-volatile, i.e., it retains the program even if the power is cut-off. The EEPROM is available on almost all AVR processors and is accessed in a separate memory map.

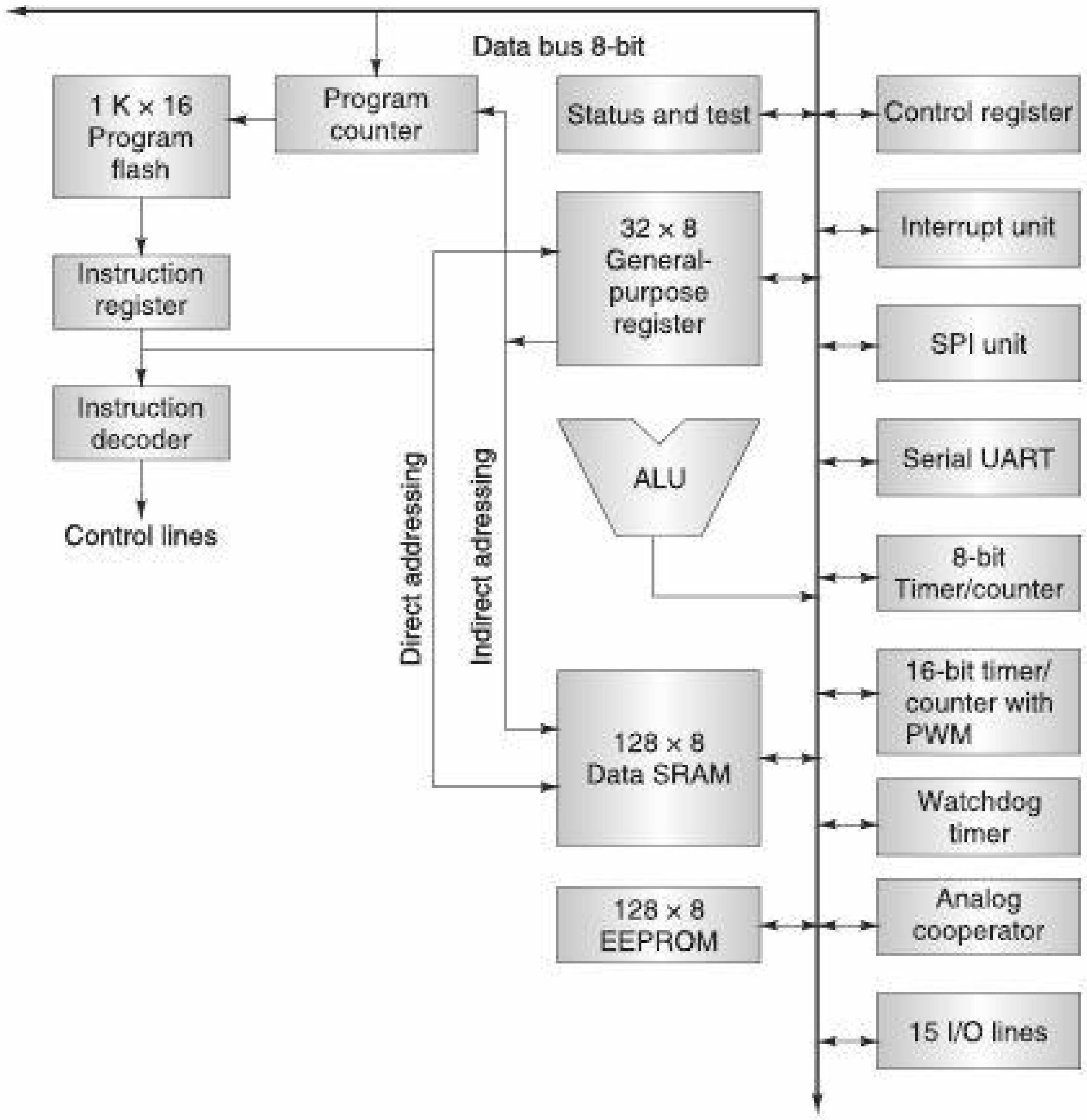


Fig. 9.20 AVR Processor Architecture.

Reading the EEPROM is faster than writing the EEROM. The EEPROM can be written to about 100,000 times.

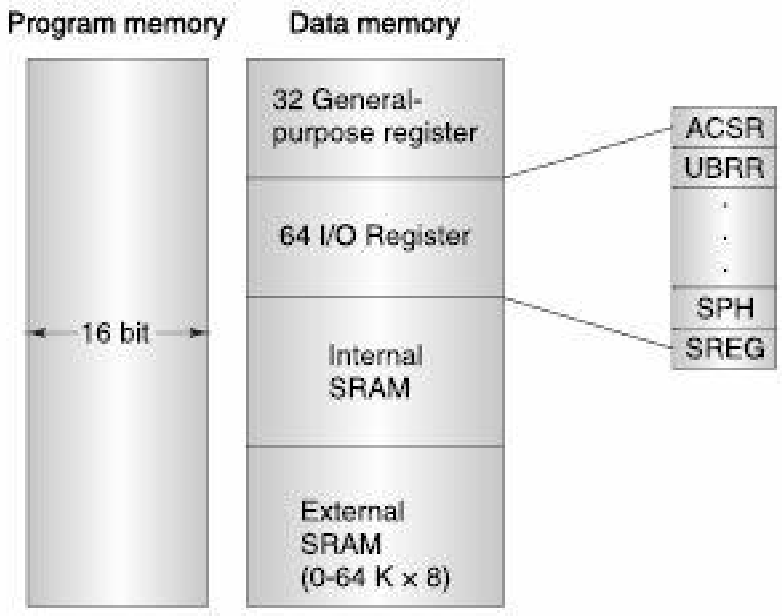


Fig. 9.21 AVR Processor Memory Map.

9.9.1 Arithmetic Logic Unit (ALU)

The arithmetic logic unit (ALU) mainly performs operations such as bit, arithmetic, and logic upon the contents of the registers and writes back the result into the register file into the designated register. These operations are performed in a single clock cycle. Each ALU operation affects the flags in the STATUS register, depending upon the instruction.

All the register operating instructions in the instruction set have direct and single-cycle access to all registers. The only exceptions are the five constant arithmetic and logic instructions SBCI, SUBI, CPI, ANDI and ORI between a constant and a register and the LDI instruction for load immediate constant data. These instructions apply to the second half of the registers in the register file (R16.....R31). The general SBC, SUB, CP, AND and OR and all other operations between two registers or on a single register apply to the entire register file.

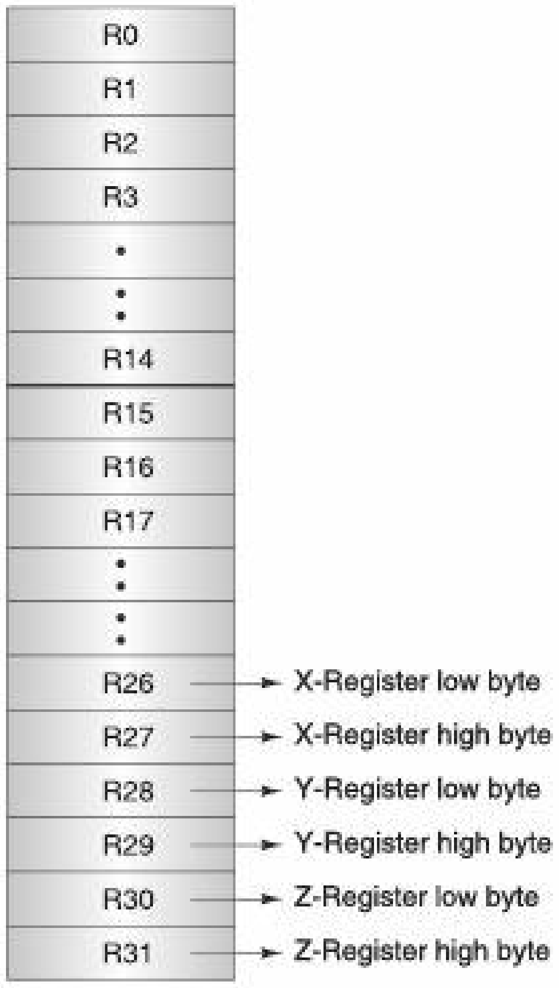
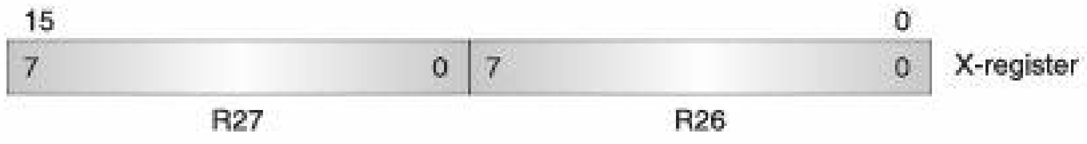
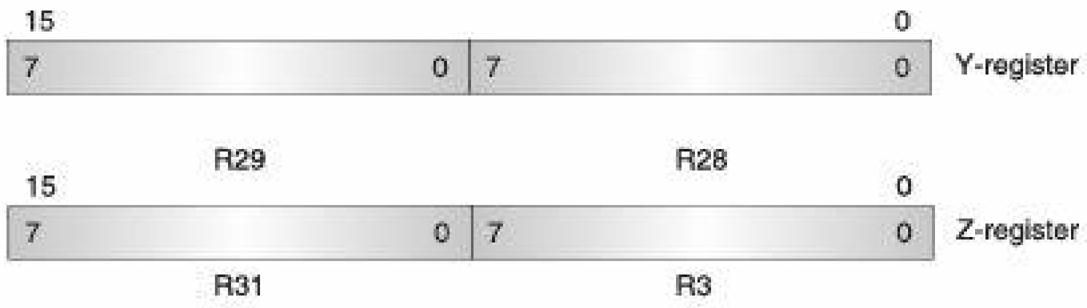


Fig. 9.22 AVR Register File.

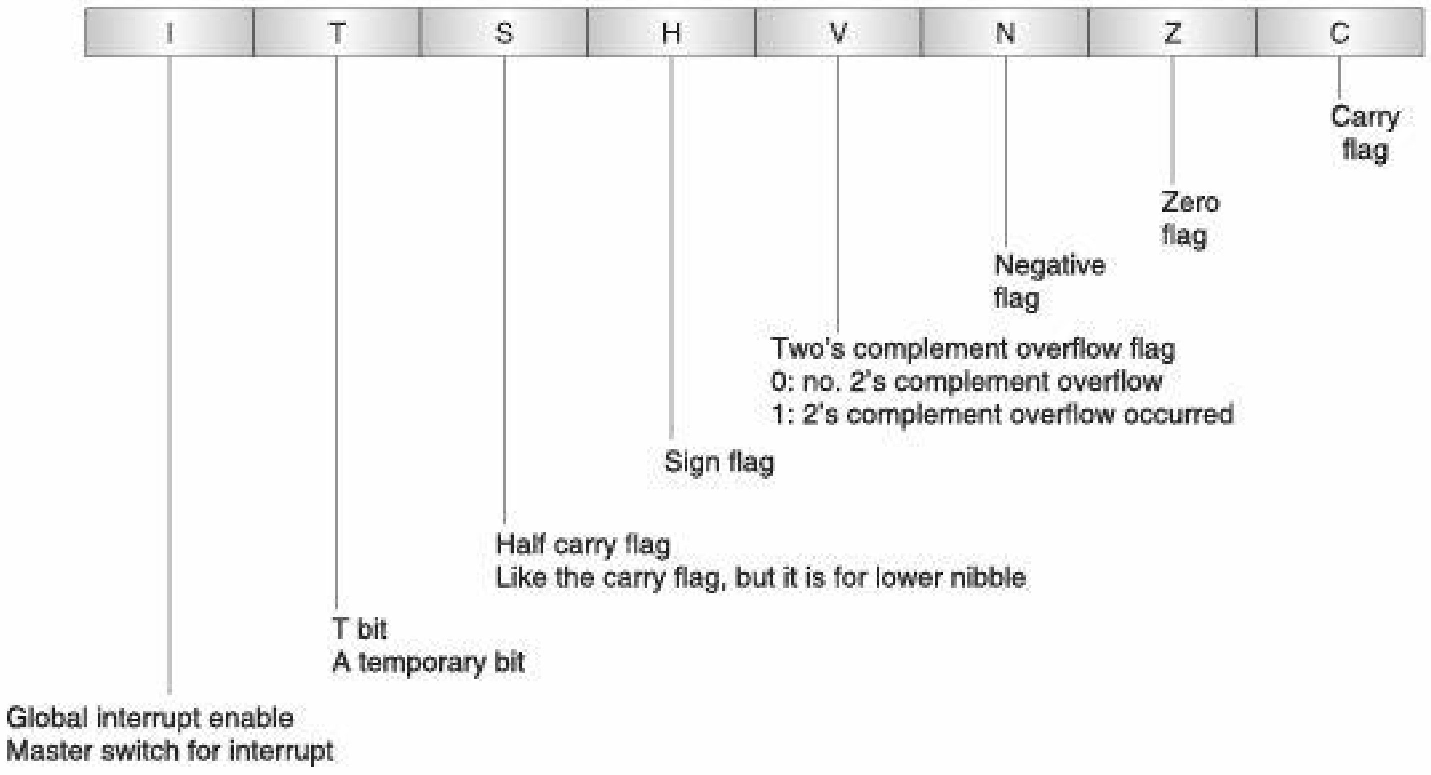
X,Y,Z-Register

Registers from R26 to R31 performs a special function These registers are address pointers for indirect addressing of the Data Space. The three indirect address registers X, Y, and Z are defined as





SREG-status register(3F)


I
T
S
H
V
N
Z
C
Carry flag
Zero flag
Negative flag
Two's complement overflow flag
0: no. 2's complement overflow
1: 2's complement overflow occurred
Sign flag
Half carry flag
Like the carry flag, but it is for lower nibble
T bit
A temporary bit
Global interrupt enable
Master switch for interrupt

The STATUS register is not stored by the machine during an interrupt operation. The instruction in an interrupt routine can modify the STATUS flag bits, and so the user program must store and retrieve the STATUS register during an interrupt.

SP-stack pointer ($3E)

This register is 1 byte wide for processors that have up to 256 bytes of SRAM and is 2 bytes wide (called SPH and SPL) for those processors that have more than 256 bytes of SRAM. This register is used to point to the area in SRAM that is the top of the stack. The stack is used to store return addresses by the processor during an interrupt and subroutine call. Since the SP is initialized to $00 (or $0000 for a 2-byte SP) at reset, the user program must initialize the SP appropriately, as the SRAM starting address is not $00.

SPH ($3E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8
SPL($3D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0

Fig. 9.23 Stack Pointer.

- SPH- Stack pointer higher bits.
- SPL- Stack pointer lower bits.

GIMSK-General interrupt mask register ($3B)

The GIMSK register is used to enable and disable individual external interrupts by setting and resetting the concerned bit respectively. However, the interrupt to be actually serviced, the I bit in the STATUS register (SREG), must also be set to "1."

7	6	5	4	3	2	1	0
INT 1	INT 0	–	–	–	–	–	

Fig. 9.24 GIMSK.

- INT1: External Interrupt Request 1 Enable
- INT0: External Interrupt Request 0 Enable

GIFR-general interrupt flag ($3A)

The bits in GIFR indicate if an interrupt has occurred. If an external interrupt occurs, the corresponding INT flag in GIFR is set to "1." If the interrupt gets serviced (which happens if the I bit and the corresponding INT bit in GIMSK register is "1"), then the flag is reset. The flag can also be reset by writing a logical "1" to it.

7	6	5	4	3	2	1	0
INTF 1	INTF 0	–	–	–	–	–	–

Fig. 9.25 GIFR.

- INT1: External Interrupt flag 1 Enable
- INT0: External Interrupt flag 0 Enable

TIMSK-timer/Counter interrupt mask register ($39)

7	6	5	4	3	2	1	0
TOIE 1	OCIE1A	OCIE1B	–	TICIE1	–	TOIE0	–

Fig. 9.26 TIMSK.

- **Bit 7 – TOIE1: Timer/Counter1 Overflow Interrupt Enable**
  When the TOIE1 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Overflow interrupt is enabled.
- **Bit 6 – OCE1A: Timer/Counter1 Output Compare A Match Interrupt Enable**
  When the OCIE1A bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Compare A Match interrupt is enabled.
- **Bit 5 – OCIE1B: Timer/Counter1 Output Compare B Match Interrupt Enable**
  When the OCIE1B bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Compare B Match interrupt is enabled.

- **Bit 3 – TICIE1: Timer/Counter1 Input Capture Interrupt Enable**

  When the TICIE1 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Input Capture Event interrupt is enabled.)
- **Bit 1 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

  When the TOIE0 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow interrupt is enabled.

TIFR-timer counter interrupt flag register ($38)

7	6	5	4	3	2	1	0
TOIE 1	OCIE1A	OCIE1B	–	TICIE1	–	TOIE0	–

Fig. 9.27 TIFR.

- **Bit 7 – TOV1: Timer/Counter1 Overflow Flag**

  The TOV1 is set (one) when an overflow occurs in Timer/Counter1. TOV1 is cleared by hardware when executing the corresponding interrupt handling vector.
- **Bit 6 – OCF1A: Output Compare Flag 1A**

  The OCF1A bit is set (one) when compare match occurs between the Timer/Counter1 and the data in OCR1A (Output Compare Register 1A). OCF1A is cleared by hardware when executing the corresponding interrupt handling vector.
- **Bit 5 – OCF1B: Output Compare Flag 1B**

  The OCF1B bit is set (one) when compare match occurs between the Timer/Counter1 and the data in OCR1B (Output Compare Register 1B). OCF1B is cleared by hardware when executing the corresponding interrupt handling vector.
- **Bit 3 – ICF1: Input Capture Flag 1**

  The ICF1 bit is set (one) to flag an input capture event, indicating that the Timer/Counter1 value has been transferred to the input capture register (ICR1). ICF1 is cleared by hardware when executing the corresponding interrupt handling vector.
- **Bit 1 – TOV: Timer/Counter0 Overflow Flag**

  The bit TOV0 is set (one) when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector.

MCUSR: MCU STATUS REGISTER ($35)

The MCU status register provides information about the source of reset.

7	6	5	4	3	2	1	0
SRE	SRW	SE	SM	ISC11	ISC10	ISC 01	ISC00

Fig. 9.28 MCUSR.

- **Bit 7: SRE. External SRAM Enable.**

  Setting this bit to "1" allows external SRAM access on processors that have the capability. PortA becomes AD0-7, PortC becomes A8-15, and WR* and RD* signals are activated on PortD as alternate pin functions. When this bit is "0," the ports function as normal ports and external SRAM access is disabled.

- **Bit 6: SRW. External SRAM Access Wait State Bit.**
  When this bit is "1," an extra wait state is inserted in the SRAM access cycle. Thus, the SRAM is accessed in 4 cycles. When this bit is "0," the SRAM is accessed in 3 cycles.
- **Bit 5: SE.**
  Sleep Enable. Setting this bit to "1" enables the processor to go in one of the sleep modes. After setting this bit to "1," the program must execute the SLEEP instruction.
- **Bit 4: SM.**
  Sleep Mode. A "1" in this bit puts the processor in idle mode. A "0" means power down mode.
- **Bit 3, 2: ISC11, ISC10. Interrupt sense control bit for INT1.**

ISC01, ISC00. Interrupt sense control bit for INT0.

The External Interrupt 1 is activated by the external pin INT1 if the SREG I-flag and the corresponding interrupt mask in the GIMSK are set. The level and edges on the external INT1 pin that activate the interrupt are defined in Table 9.10.

Table 9.10 Interrupt1 Sense Control

ISC11	*ISC10*	*Description*
0	0	Low level on INT1 generates interrupt
0	1	Reserved
1	0	Falling edge on INT1 generates interrupt
1	1	Rising edge on INT1 generates interrupt

TCNT0-timer counter 0($32)

This is the actual timer/counter register. A value loaded in this register is used as the starting value, and the timer increments this value at each of its clock signals if the counter/timer is enabled through the TCCR0 register.

7	6	5	4	3	2	1	0
MSB	–	–	–	–	–	–	LSB

Fig. 9.29 TCNT0.

TCCR0-timer/Counter0 Control Register ($33)

The Clock Select0 bits 2, 1 and 0 define the prescaling source of Timer/Counter0.

7	6	5	4	3	2	1	0
–	–	–	–	–	CS02	CSO1	CS00

Fig. 9.30 TCCR0.

Table 9.11 Clock 0 Prescale Set

CS02	CS01	CS00	Description
0	0	0	Stop, the Timer/Counter0 is stopped.
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	External Pin T0, falling edge
1	1	1	External Pin T0, rising edge

TCCR1A- Timer/Counter1 Control Register ($2F)

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	–	–	PWM11	PWM10

Fig. 9.31 TCCR1A.

Bits 7, 6: COM1A1, COM1A0: Compare Output Mode1, bits 1 and 0. The COM1A1 and COM1A0 control bits determine any output pin action following a compare match in Timer/Counter1. Output pin actions affect pin OC1-Output Compare pin1. This is an alternative function to the I/O port, and the corresponding direction control bit must be set to "1" to control an output pin. For devices with 2 compare functions, bits 5 and 4 of the control register have similar functions to bits 7 and 6.

Table 9.12 Compare 1 Mode Select

COM1X1	COM1X0	Description
0	0	Timer/Counter1 disconnected from output pin OC1X
0	1	Toggle the OC1X output line.
1	0	Clear the OC1X output line (to zero).
1	1	Set the OC1X output line (to one).

Table 9.13 PWM Mode Select

PWM11	PWM10	Description
0	0	PWM operation of Timer/Counter1 is disabled
0	1	Timer/Counter1 is an 8-bit PWM
1	0	Timer/Counter1 is a 9-bit PWM
1	1	Timer/Counter1 is a 10-bit PWM

EEAR-EEPROM address register ($1F)

The EEPROM address registers (EEARH and EEARL) specify the EEPROM address in the 512-byte EEPROM space for AT90S8515. The EEPROM data bytes are addressed linearly between 0 and 512.

EEARH($1F)	–	–	–	–	–	–	–	EEAR8
EEARH($1F)	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0

Fig. 9.32 EEAR.

EEDR-EEPROM data register ($1D)

For the EEPROM write operation, the EEDR register contains the data to be written to the EEPROM in the address given by the EEAR register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

7	6	5	4	3	2	1	0
MSB	–	–	–	–	–	–	LSB

Fig. 9.33 EEDR.

EECR-EEPROM control register ($1C)

7	6	5	4	3	2	1	0
–	–	–	–	–	EEMWE	EEWE	EERE

Fig. 9.34 EECR.

Bit 2: EEMWE:EEPROM Master Write Enable

Setting EEMWE to "1" and then setting EEWE to "1" only will write data in the EEDR register to the EEPROM. If EEMWE is set to "1," the hardware clears this bit to "0" after 4 clock cycles.

Bit 1: EEWE. EEPROM Write Enable

When set to "1" while EEMWE is also "1," the EEDR data is written to the EEPROM at the address specified by the EEPROM Address register. The EEWE bit remains "1" during the write cycle, which may take up to 2.5 ms at 5 V. After this time has elapsed, the EEWE is cleared by hardware to "0." The sequence for writing data to the EEPROM is as follows:

Wait till EEWE is cleared to "0."

Write EEPROM address to EEAR.

Write EEPROM data to EEDR.

Set EEMWE to "1" and within four clock cycles set EEWE to "1."

This will write the data in EEDR to the EEPROM location whose address is in EEAR.

Bit 0: EERE:EEPROM Read Enable

To read EEPROM data, load EEAR with the correct address, set EERE to "1," and then clear EERE to "0." This will get the data in EEDR. Before starting a read cycle, the program would poll the EEWE flag till EEWE is "0" to ensure that any write cycle is not in progress.

WATCH DOG TIMER ($21)

THE potential feature of AVR processor is watch dog timer, a 1 MHz internal timer will reset the AVR at regular interval in order to stop the AVR resetting. The watch dog timer must be cleared at regular intervals.

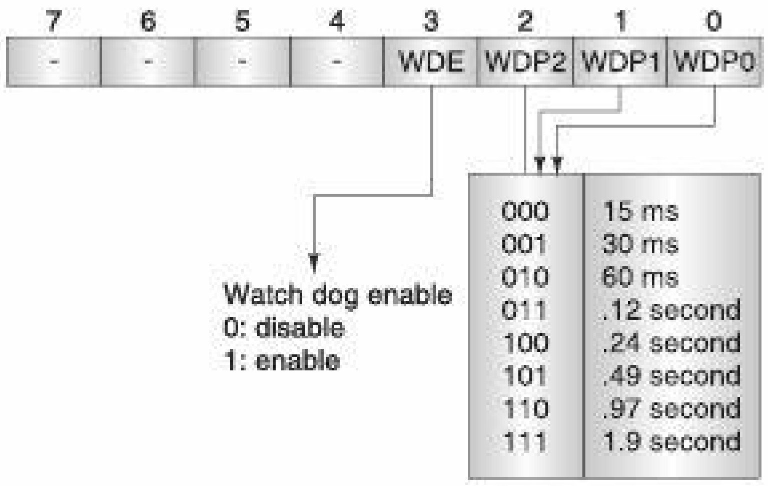


Fig. 9.35 Watch Dog Timer.

9.10 AT90S8515 MICROCONTROLLER

The AT90S8515 is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. It provides the following features: 2 Kbytes of In-System Programmable Flash, 15 general purpose I/O lines, 32 general purpose working registers, flexible timer/counters with compare modes, internal and external interrupts, a programmable serial UART, programmable Watchdog Timer with internal oscillator, an SPI serial port for Flash Memory downloading and two software selectable power saving modes.

Some of its features are given below.

- 118 Powerful Instructions, most of the instruction consumes one cycle only
- Up to 10 MIPS Throughput at 10 MHz
- Lock for Flash Program and EEPROM Data Security
- One 8-bit Timer/Counter and One 16-bit Timer/Counter
- Low-power Idle and Power Down Modes
- Low-power, High-speed CMOS Process Technology
- Programmable Watchdog Timer with On-chip Oscillator
- SPI Serial Interface for In-System Programming
- Low-power Idle and Power-down Modes
- External and Internal Interrupt Sources
- 15 Programmable I/O Lines
- On-chip Analog Comparator
- Full Duplex UART(universal asynchronous receiver transmitter)
- 128 bytes EEPROM,128 bytes SRAM
- 32 × 8 General-purpose register bank is provided to store the variables in inside the CPU rather than storing them in the memory
- Power On RESET circuit.
- On-chip programmable timer with separate prescalar. This is used for timing applications.

AT90S8515 is a 40 pin IC comes in three package PDIP. PLCC and QIP. In this four ports of 8 bits are provided, named as A, B, C and D

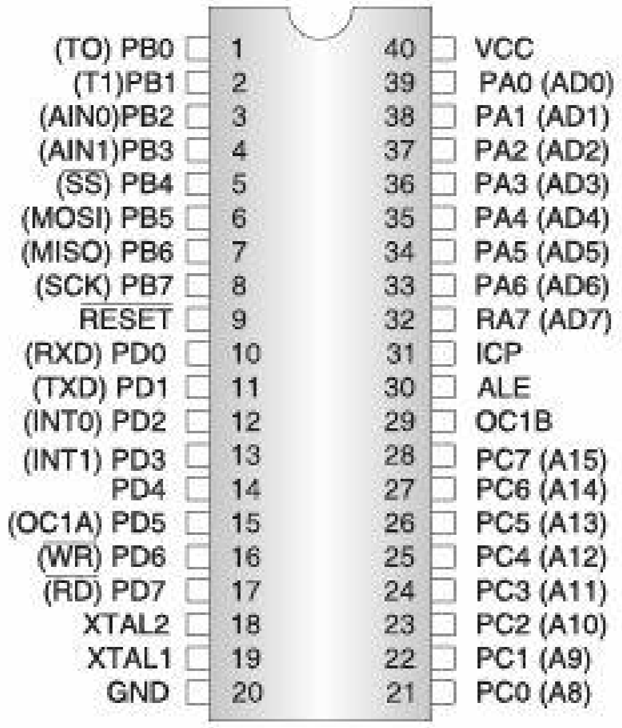


Fig. 9.36 AT90S8515 Pin Diagram.

Pin Descriptions

Port A (PA7..PA0): Port A is an 8-bit bidirectional I/O port. Port pins can provide internal pull-up resistors (selected for each bit). Port A serves as multiplexed address/data input/output when using external SRAM.

Port B (PB7..PB0): Port B is an 8-bit bidirectional I/O port with internal pull-up resistors. Each bit of port B also serve a special function.

Port C (PC7..PC0): Port C is an 8-bit bidirectional I/O port with internal pull-up resistors. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not active. Port C also serves as address output when using external SRAM.

Port D (PD7..PD0): Port D is an 8-bit bidirectional I/O port with internal pull-up resistors. Port D also serve some special functions.

RESET: The polarity of the RESET line was opposite (8051's having an active-high RESET, while the AVR has an active-low RESET). A low level on this pin for more than 50 ns will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a reset.

ICP: ICP is the input pin for the Timer/Counter1 Input Capture function.

OC1B: OC1B is the output pin for the Timer/Counter1 Output CompareB function.

ALE: ALE stands for address latch enable, used when the external memory is enabled. The

ALE strobe is used to latch the lower order address (8-bits) into an address latch during the first access cycle, and the AD0-7 pins are used for data during the second access cycle.

Crystal Oscillator: XTAL1 and XTAL2 are input and output, Either a quartz crystal or a ceramic resonator may be used. When using the MCU oscillator as a clock for an external device, an HC buffer should be connected.

VCC: Supply Voltage.

GND: Reference Ground.

9.11 PROGRAM AND DATA ADDRESSING MODES

This section describe the different addressing modes supported by the AVR microcontroller.

REGISTER DIRECT (SINGLE REGISTER)

The Register Direct instructions can operate on any of the 32 registers of the register file. It reads the contents of a register, operates on the contents of the register, and then stores the result of the operation back into the same register. For Example: COM R1,INC R2,DEC R7,TST R0.

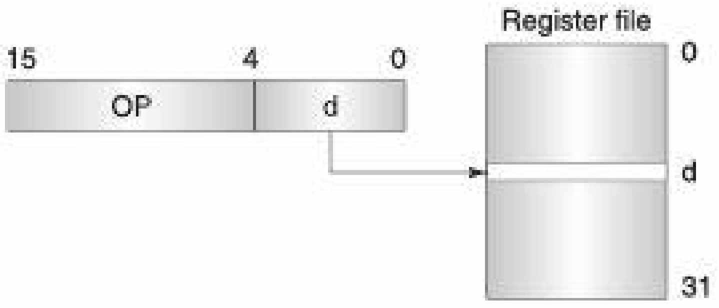


Fig. 9.37 Direct Single Register Access.

Here Rd - any register in register array, OP – operation code

REGISTER DIRECT (TWO REGISTERS)

In these types of instructions, two registers are involved. The two registers are named as the source register, Rs, and the destination register, Rd. The instruction reads the two registers and operates on their contents and stores the result back in the destination register.

For example: ADD Rd,Rs, SUB Rd,Rs

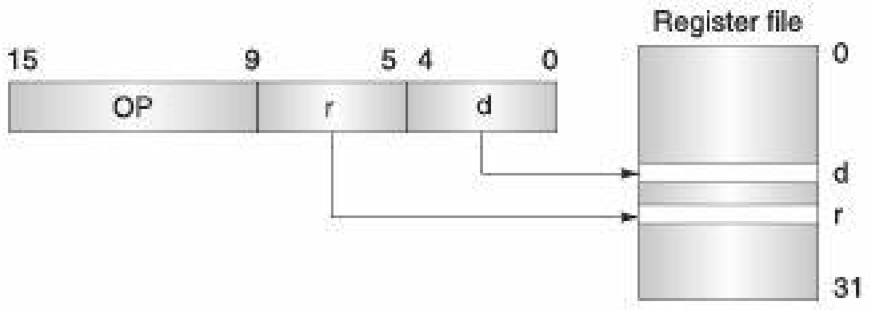


Fig. 9.38 Direct Double Register Access.

I/O DIRECT

These instructions are used to access the I/O space. The I/O registers can only be accessed using these instructions: In Rd, PORTADDRESS; Out PORTADDRESS, Rs. Rd, Rs can be any of the 32 registers from the register file, and the I/O registers can be any register from the entire range of \$00 to \$3F (a total of 64 I/O registers). Operand address is contained in six bits of the instruction word. n is the destination or source register address.

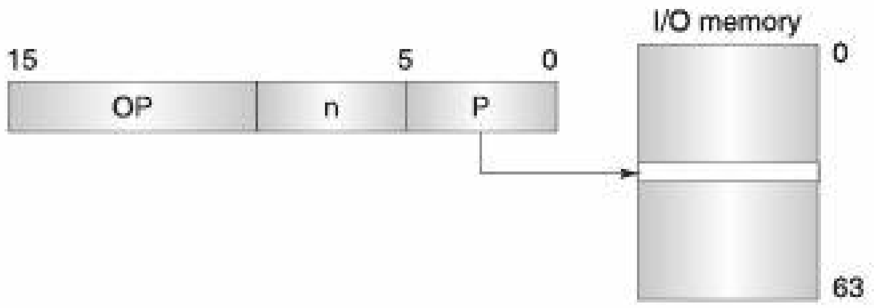


Fig. 9.39 Direct I/O Memory Access.

DATA DIRECT

These are two word instructions. One of the words is the address of the data memory space. So a maximum of 64 Kbyte data memory can be accessed using these types of instructions. The examples of these instructions are: LDS RD, K; K is a 16-bit address. STS K, Rs.

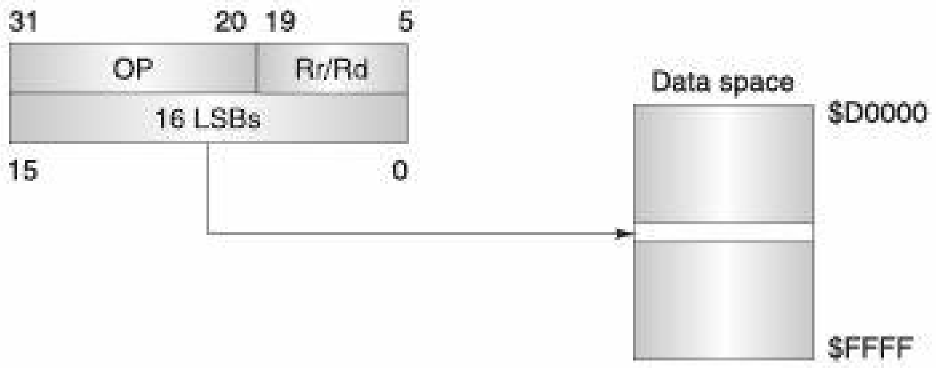


Fig. 9.40 Direct Data Memory Access.

DATA INDIRECT

These instructions are one word each, and a pointer register (X, Y, or Z) is used that has the base address of the data memory. To the base address in the pointer register, an offset can be added, as well as some increment/decrement operations on the pointer contents.

For example: LD Rd,X, LD Rd,x+

INDIRECT PROGRAM ADDRESSING

In these types of instructions, the Z register is used to point to the program memory. Up to 64 Kbytes of program memory can be accessed with the 16-bit Z register. Examples of these types of instructions are: IJMP and ICALL.

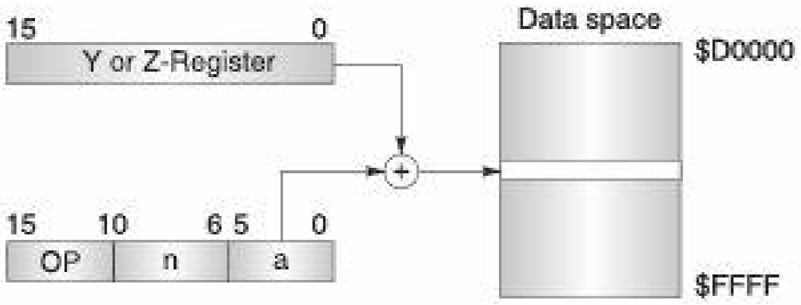


Fig. 9.41 Indirect Data Memory Access.

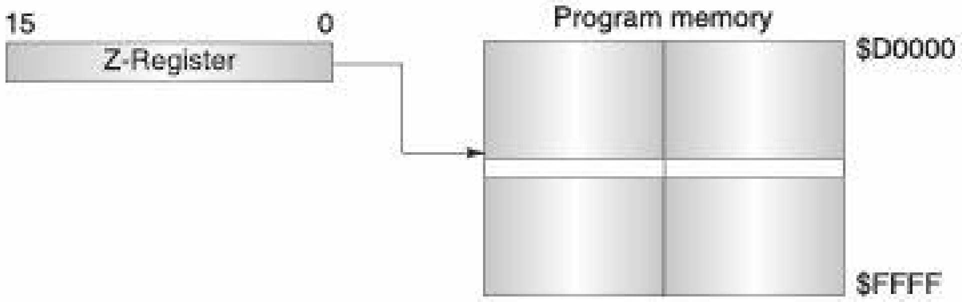


Fig. 9.42 Indirect Program Memory Instructions.

RELATIVE PROGRAM ADDRESSING

These instructions are of the type RJMP and RCALL, where an offset of 2K to the program counter is used. Figure 9.43 illustrates how relative program addressing instructions operate.

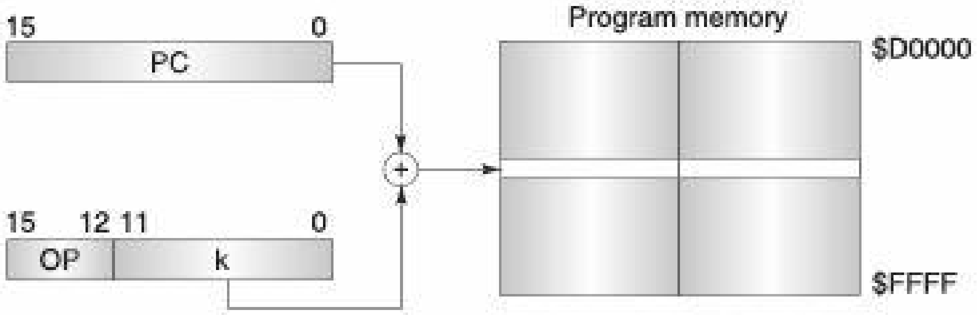


Fig. 9.43 Indirect Program Memory Instructions.

9.12 INSTRUCTION SET

Arithmetic and Logic Instructions

Instruction	Operation performed	Format	Cycle consumed
ADC	Add with Carry	ADC Rd, Rs Rd = Rd + Rs + C	1
ADIW	Add immediate constant	ADIW Rd, k	2
SUB	Subtract without Carry	SUB Rd, Rs Rd = Rd − Rs	1
SUBI	Subtract immediate without Carry	SUBI Rd, k Rd =Rd − k	1

Contd...

Contd...

SBCI	Subtract immediate with Carry	SBCI Rd, k Rd = Rd − k −C	1
SBC	Subtract with Carry	SBC Rd, Rs Rd = Rd − Rs −C	1
SBIW	Subtract immediate constant	SBIW Rd, k	2
AND	Logical AND	AND Rd, Rs	1
OR	Logical OR	OR Rd, Rs	1
EOR	Exclusive OR	EOR Rd, Rs	1
COM	One's complement	COM Rd	1
NEG	Two's complement	NEG Rd	1
SBR	Set bit	SBR Rdx, k	1
INC	Increment contents	INC Rd	1
DEC	Decrement contents	DEC Rd	1
TST	Test Rd for zero or minus	TST Rd	1
MUL	Unsigned multiplication	MUL Rd, Rs R1:R0 = Rd x Rs	2
MULS	Signed multiplication	MULS Rd, Rs	2
FMUL	Unsigned fractional multiplication	FMUL Rd, Rs R1:R0 = Rd xRs	2
FMULS	Signed fractional multiplication	FMULS Rd, Rs R1:R0 = Rd x Rs	2
CLR	Clear Register	CLR Rd	1
SER	Set Register	SER Rd	1

Program Control Instructions

Instruction	*Operation Performed*	*Format*	*Cycle consumed*
RJMP	Relative jump to a location in program memory	RJMP k K is _/_ 2K addresses	2
IJMP	Indirect jump to a location in program memory pointed by the Z register	IJMP	2
JMP	Jump to a location in program memory	JMP k	3
RCALL	Relative call to a subroutine	RCALL k	3/4
ICALL	Indirect call to a subroutine.	ICALL	3/4
CALL	Call to a subroutine	CALL k	4/5
RET	Subroutine return	RET	4/5
CPSE	Compare, Skip if Equal	CPSE Rd, Rs	1/2/3
CP	Compare	CP Rd, Rs Rd −Rs	1
SBRC	Skip if bit in register cleared	SBRC Rd, b	1/2/3
SBIC	Skip if bit in I/O register cleared	SBIC A, b If [A(b) =0]	1/2/3
BRBS	Branch if status flag set. conditional relative branch.	BRBS s, k	1/2

Data Transfer Instructions

Instruction	Operation performed	Format	Cycle consumed
MOV	Copy register	MOV Rd, Rs Rd = Rs	1
MOVW	Copy register pair	MOVW Rd, Rs Rd + 1: Rd = Rs + 1: Rs.	1
LDI	Load Immediate	LDI Rdx, k	1
LD	Load Indirect	LD Rd, X Rd = data memory (X) X is the pointer register pair	2
LDD	Load Indirect with displacement	LDD Rd, Y + q Y is the pointer register pair	2
STS	Store Immediate	STS Rs, k data memory (k) = Rs	2
ST	Store Indirect	ST X, Rs data memory (X) = Rs	2
STD	Store Indirect with displacement	STD Y + q, Rs	2
LPM	Load Program Memory	LPM R0 = Program Memory(Z)	3
IN	Input from input Port	IN Rd, A	1
OUT	Output to output Port	Output to output Port	1
PUSH	Push register on STACK	PUSH Rs	2
POP	Pop into register from STACK	Pop into register from STACK	2

Bit and Bit-test Instructions

Instruction	Operation performed	Format	Cycle consumed
LSL	Logical Shift Left	LSL Rd	1
LSR	Logical Shift Right	LSR Rd	1
ROL	Rotate Left though Carry	ROL Rd	1
ROR	Rotate Right through Carry	ROR Rd	1
ASR	Arithmetic Shift Right	ASR Rd	1
SWAP	Swap Nibbles	SWAP Rd	1
BSET	Flag Set	BSET s SREG(s) = 1	1
BCLR	Flag Reset	BCLR s SREG(s) = 0	1
SBI	Set bits in I/O register A	SBI A.s	1
CBI	Clear bits in I/O register A	CBI A, s	1
SEC	Set Carry flag	SEC	1
CLC	Clear Carry flag.	CLC	1
SEN	Set Negative flag	SEN	1

Contd...

Contd...

CLN	Clear Negative flag	CLN	1
SEI	Set Interrupt flag	SEI	1
CLI	Clear Interrupt flag	CLI	1

Things to Remember

◊ The ARM processor is a Reduced Instruction Set Computer (RISC) and ARM stood for Acorn RISC Machine.

◊ All ARM instructions are 32 bits wide and are aligned on 4-byte boundaries in memory.

◊ The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support.

◊ *PIC10F200r* is available in a 6-pin SOT-23 package, or in 8-pin PDIP package.

◊ *PIC12C508* is another low-cost microcontroller available in an 8-pin dual in-line package.

◊ *PIC16C54* is an 18-pin microcontroller with 384 × 12 EPROM program memory. 80 bytes and 13 I/O port pins are provided.

◊ PIC16F84 is an 18-pin device, which can operate at up to 20 MHz clock speed. It of EEPROM non-volatile data memory, 8-bit timer with watchdog timer, 13 I/O pins, external and internal interrupt sources, and large current sink and source capability.

◊ Physically, port is a register inside a microcontroller which is connected by wires to the pins of a microcontroller.

◊ PIC16F84 has two separate memory blocks, one for data and the other for program.

◊ EEPROM memory and GPR registers in RAM memory make up a block for data, and FLASH memory makes up a program block.

Exercise

Write programs that will accomplish the desired tasks listed below, using as few lines of code as possible. Comment on each line of code.

1. Address of a subroutine that handles a timer 1 interrupt.
2. Why a low-address byte latch for external memory is needed. How an I/O pin can be both an input and output?
3. Which port has no alternate functions? The maximum pulse rate that can be counted on pin T1 if the oscillator frequency is 6 megahertz.
4. Which bits in which registers must be set to give the serial data interrupt the highest priority?
5. The baud rate for the serial port in mode 0 for a 6 megahertz crystal. The largest possible time delay for a timer in mode 1 if a 6 megahertz crystal is used.
6. The setting of TH1, in timer mode 2, to generate a baud rate of 1200 if the serial port is in mode 1 and an 11.059 megahertz crystal is in use. Find the setting for both values of SMOD.
7. The address of the PCON special-function register. The time it will take a timer in mode 1 to overflow if initially set to 03AEh with a 6 megahertz crystal.

8. Which are bits in which registers must be set to 1 to have timer 0 count input pulses on pin T0 in timer mode 0?
9. When used in multiprocessing, which bit in which register is used by a transmitting 8051 to signal receiving 8051*$ that an interrupt should be generated?
10. The two conditions under which program opcodes are fetched from external, rather than internal, memory.
11. Place the number 3Bh in internal RAM locations 30h to 32h. Copy the data at internal RAM location Flh to R0 and R3.
12. Set the SP at the byte address just above the last working register address. Exchange the contents of the SP and the PSW.
13. Set the SP register to 07h and PUSH the SP register on the stack; predict what number is PUSHed to address 08h.
14. Exchange the contents of the B register and external RAM address 02CFh.
15. Rotate the bytes in registers R0 to R3; copy the data in R0 to Rl, Rl to R2, R2 to R3, and R3 to R0.
16. Exchange both low nibbles of registers R0 and Rl; put the low nibble of R0 in Rl, and the low nibble of Rl in R0.
17. Store the contents of register R3 at the internal RAM address contained in R2. (Be sure the address in R2 is legal.)
18. Store the contents of RAM location 20h at the address contained in RAM location 08h.
19. Invert the data on the port 0 pins and write the data to port I.
20. Swap the nibbles of R0 and Rl so that the low nibble of R0 swaps with the high nibble of Rl and the high nibble of R0 swaps with the low nibble of Rl.
21. Complement the lower nibble of RAM location 2Ah.
22. Make the low nibble of R5 the complement of the high nibble of R6.
23. Make the high nibble of R5 the complement of the low nibble of R6. Move bit 6 of R0 to bit 3 of port 3.
24. Move bit 4 of RAM location 30h to bit 2 of A.
25. XOR a number with whatever is in A so that the result is FFh.
26. Treat registers R0 and Rl as 16-bit registers, and rotate them one place to the left: bit 7 of R0 becomes bit 0 of Rl. bit 7 of Rl becomes bit 0 of R0, and so on.
27. Determine whether the 8051 can be made to execute a single program instruction (single-stepped) using external circuitry (no software) only.
28. Outline a scheme for single-stepping the 8051 using a combination of hardware and software. (Hint: Use an INTX.)
29. While running the EPROM test, it is found that the program cannot jump from 2000h to 4000h successfully. Determine what address line(s) is faulty.
30. Calculate the error for the delay program "Softime" when values of 2d. lOd and lOOOd milliseconds are passed in A and B.

31. Find the shortest and longest delays possible using "Softime" by changing only the equate value of the variable "delay."
32. In the discussion for the program named "Timer," the statement is made that an accurate 1 ms delay cannot be done due to the need for a count of 1333.33 using a 16 megahertz clock. Fin d a way to generate an accurate 60 second delay using T0 for the basic delay and some registers to count the T0 overflows.
33. Calculate the shortest and longest delays possible using the program named "Timer" by changing the initial value of T0.
34. Write a lookup table program, using the PC as the base, that finds a one byte square root (to the nearest whole integer) of any number placed in A. For example, the square roots of 01 and 02 are both 01, while the roots of 03 and 04 are 02. Calculate the first four and last four table values.
35. Write a lookup table, using the DPTR as the base, that finds a two-byte square root of the number in A. The first byte is the integer value of the root, and the second byte is the fractional value. For example, the square root of 02 is 01, 6Ah. Calculate four first and last table values.
36. A PC based lookup table, which contains 256d values, is placed 50h bytes after the MOVC instruction that accesses it. Construct the table, showing where the byte associated with A = OOh is located. Find the largest number which can be placed in A to access the table.
37. Construct a lookup table program that converts the hex number in A to an equivalent BCD number in registers R4 (MSB) and R5 (LSB).
38. Does asynchronous communication between two microprocessors have to be done at standard baud rates? Name one reason why you might wish to use standard rates.
39. Write a test program that will "loop test" the serial port. The output of the serial port (TXD) is connected to the input (RXD), and the test program is run. Success is indicated by port I pin 1 going high.
40. What is the significance of the transmit flag, TI, when it is cleared to 0? When set to 1?
41. Which features does ARM have in common with many other RISC architectures?
42. Which features of the ARM architecture are not shared by most other RISCs?
43. Which features of most other RISC architectures are not shared by the ARM?

Chapter 10
Embedded Systems

- Introduction
- Characteristics of Embedded Systems
- Embedded Hardware
- Embedded Software
- Real-Time Systems
- Application Areas

10.1 INTRODUCTION

The millions of computing systems are built every year that serve as personal computers, workstations, mainframes and servers. Computing systems are everywhere. The surprising part is that the billions of computing systems are built for a very different purpose every year, but in this one is embedded system (i.e., for a dedicated purpose). An embedded system is nearly any computing system other than a desktop, laptop, or mainframe computer used for a specific purpose.

Creating a precise definition of the embedded computing systems, or simply embedded systems, is not an easy task. It can be said that an embedded system is a combination of computer hardware, software and, perhaps additional mechanical parts, designed to perform a specific function. Some examples of embedded systems are microwave oven, VCR, and alarm clock, etc. These devices are used daily in every household. This is in direct contrast to the personal computer in the family room. It too comprises computer hardware and software and mechanical components. However, it is not designed to perform a specific function. Rather, it is able to do many different things. That is why the term general-purpose computer is used for personal computers many times. The general-purpose computer is used for different purposes by different users. One user may use it for a network file server and another may use it exclusively for playing games.

If an embedded system is designed well, the existence of the processor and software could be completely unnoticed by a user of the device. It can be seen in the case of microwave oven which is used everyday in the household but very few people realize that a processor and software are involved in the preparation of their lunch or dinner. In some cases, it would even be possible to build an equivalent device that does not contain the processor and software by replacing the combination with a custom integrated circuit that performs the same functions in hardware. However, replacing the combination with custom hardware has its own disadvantages. First and foremost is the loss in flexibility. It is much easier, and cheaper, to change a few lines of software than to redesign a piece of custom hardware.

10.2 CHARACTERISTICS OF EMBEDDED SYSTEMS

Embedded systems have several common characteristics:

- **Single-functioned:** Single functioned means that an embedded system executes only one program, repeatedly. Its task is to perform a specific function only over again and again. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.
- **Tightly constrained:** A design metric is a measure of an implementation's features, such as cost, size, performance, and power. All computing systems have constraints on design metrics, but those on embedded systems can be especially tight.
- **Reactive and real-time:** Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and

brake sensors. It must compute acceleration or deceleration amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

To illustrate the characteristics of the embedded system described above, an example of digital camera is considered. The digital camera system is shown in Figure 10.1. The Analog to Digital Converter, Digital to Analog Converter circuits convert analog images to digital and digital to analog, respectively. The JPEG codec compresses and decompresses (as the word codec is expanded) an image using the JPEG2 compression standard. This JPEG codec enables compact storage in the limited memory of the camera.

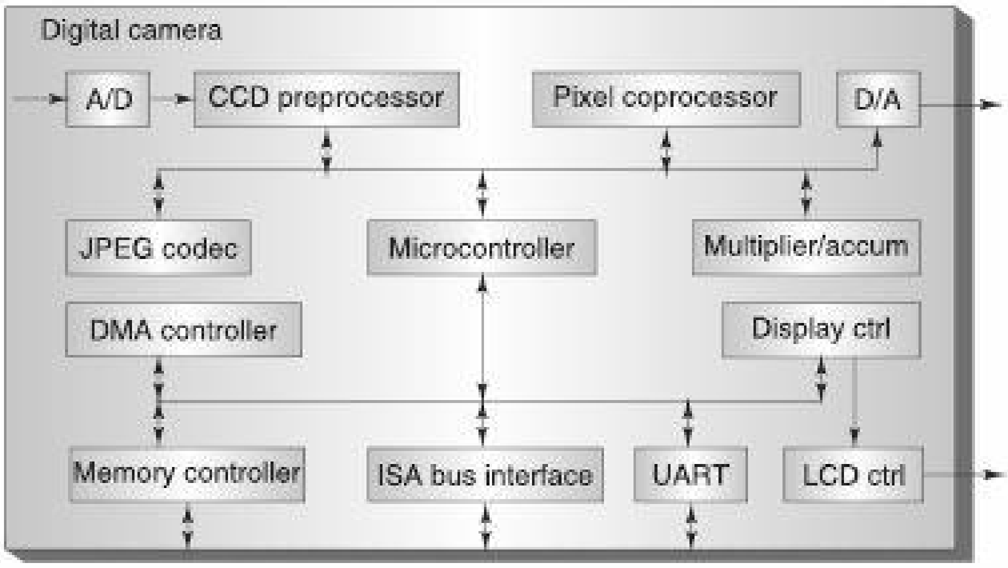


Fig. 10.1 Functional Block Diagram of a Digital Camera as an Embedded System Example.

The two processors used are CCD pre-processor and Pixel coprocessor. The CCD preprocessor is a charge-coupled device preprocessor. The Pixel coprocessor aids in rapidly displaying images. The memory controller controls access to a memory chip also found in the camera, while the DMA controller enables direct memory access without requiring the use of the microcontroller. The UART enables communication with a PC's serial port for uploading video frames, while the ISA bus interface enables a faster connection with a PC's ISA bus. The LCD ctrl and Display ctrl circuits control the display of images on the camera's liquid-crystal display device. A Multiplier/Accum circuit assists with certain digital signal processing. At the heart of the system is a microcontroller, which is a processor that controls the activities of all other circuits. We can think of each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks. Some of the characteristics are demonstrated by this example.

First, it performs a single function repeatedly. The system always acts as a digital camera, Where in it captures, compresses and stores frames, decompresses and displays frames, and uploads frames. Second, it is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must have high performance so that it can process numerous images in milliseconds.

It must consume little power so that the camera's battery will last a long time. However, there is no reactive and real-time characteristic of this particular system as it only needs to respond to the pressing of buttons by a user, which even for an avid photographer is still quite slow with respect to processor speeds.

Embedded systems are usually programmed in high-level language that is compiled into an executable ("machine") code. These are loaded into Read Only Memory (ROM) and called "firmware", "microcode" or a "microkernel". The microprocessor is 8-bit or 16-bit. The bit size refers to the amount of memory accessed by the processor.

10.3 EMBEDDED HARDWARE

An embedded system has

(i) A microcontroller for processing of information and execution of programs.
(ii) A memory in the form of ROM/RAM for storing embedded software programs and data.
(iii) I/O interfaces for external interface.

Any additional requirement in an embedded system is on the equipment it is controlling.

Very often these systems have a standard serial port, a network interface, I/O interface, or hardware to interact with sensors and activators on the equipment.

10.4 EMBEDDED SOFTWARE

Embedded software is computer software, written to control machines or devices that are not typically thought of as computers. It is typically specialized for the particular hardware that it runs on and has time and memory constraints

C has become the language of choice for embedded programmers, because it allows the programmer to concentrate on algorithms and applications, rather than on the details of processor architecture making it the processor independent. However, many of its advantages apply equally to other high-level languages as well. Perhaps the greatest strength of C is that it gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. Compilers and cross-compilers are also available for almost every processor with C. Any source code written in C or C++ or Assembly language must be converted into an executable image that can be loaded onto a ROM chip. The process of converting the source code representation of your embedded software into an executable image involves three distinct steps, and the system or computer on which these processes are executed is called a host computer.

First, each of the source files that make an embedded application must be compiled or assembled into distinct object files. Second, all of the object files that result from the first step must be linked into a final object file called the relocatable program. Finally, the physical memory address must be assigned to the relocatable program. The result of the third step is a file that contains an executable image that is ported on the ROM chip. This ROM chip, along with the processor and other devices and interfaces, makes an embedded system run.

There are some very basic differences between conventional programming and embedded programming. First, each target platform is unique. Second, there is a difference in the development and debugging of applications.

10.5 REAL-TIME SYSTEMS

In ways virtually unimaginable just a few decades ago, embedded systems are reshaping the way people live, work, and play. Embedded systems come in an endless variety of types, each exhibiting unique characteristics. For example, most vehicles driven today embed intelligent computer chips that perform value-added tasks, which make the vehicles easier, cleaner, and more fun to drive. Telephone systems rely on multiple integrated hardware and software systems to connect people around the world. Even private homes are being filled with intelligent appliances and integrated systems built around with embedded systems, which facilitate and enhance our everyday life.

Often referred to as *pervasive* or *ubiquitous* computers, embedded systems represent a class of dedicated computer systems designed for specific purposes. Many of these embedded systems are reliable and predictable. The devices that embed them are convenient, user-friendly, and dependable.

One special class of embedded systems is distinguished from the rest by its requirement to respond to external events in real time. This category is classified as the *real-time embedded system*.

As an introduction to embedded systems and real-time embedded systems, this chapter focuses on:

- examples of embedded systems,
- defining embedded systems,
- defining embedded systems with real-time behavior, and
- current trends in embedded systems.

Real-time operating system (RTOS) is key to many embedded systems today and, provides a software platform upon which to build applications. Not all embedded systems, however, are designed with an RTOS. Some embedded systems with relatively simple hardware or a small amount of software application code might not require an RTOS. Many embedded systems, however, with moderate-to-large software applications require some form of scheduling, and these systems require an RTOS.

10.5.1 Introduction

In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion, as shown in Figure 10.2. The response time is guaranteed. Were visit this definition after presenting some examples of real-time systems.

External events can have synchronous or asynchronous characteristics. Responding to external events includes recognizing when an event occurs, performing the required processing as a result of the event, and outputting the necessary results within a given time constraint.

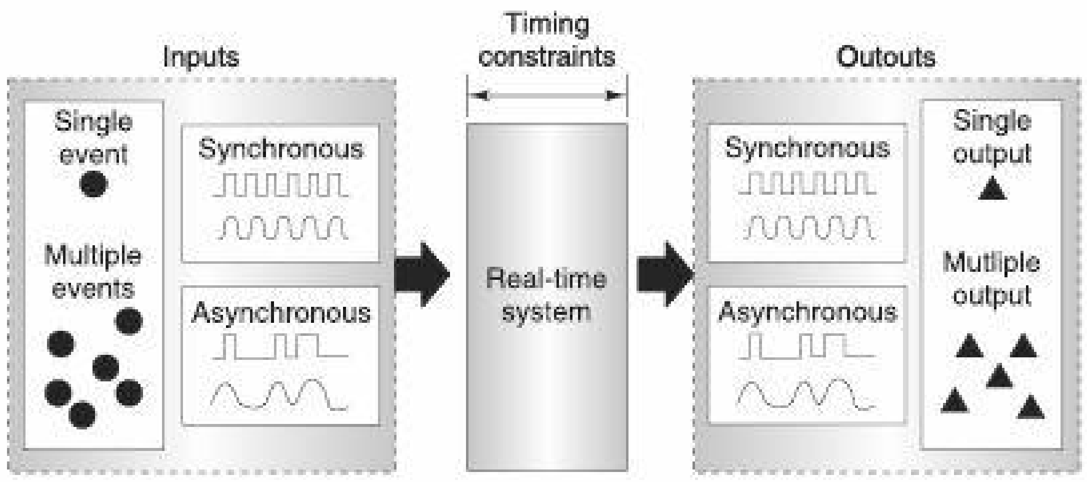


Fig. 10.2 A Simple View of Real-Time Systems.

Timing constraints include finish time, or both start time and finish time.

A good way to understand the relationship between real-time systems and embedded systems is to view them as two intersecting circles, as shown in Figure 10.3. It can be seen that not all embedded systems exhibit real-time behaviors nor are all real-time systems embedded. However, the two systems are not mutually exclusive, and the area in which they overlap creates the combination of systems known as *real-time embedded systems*.

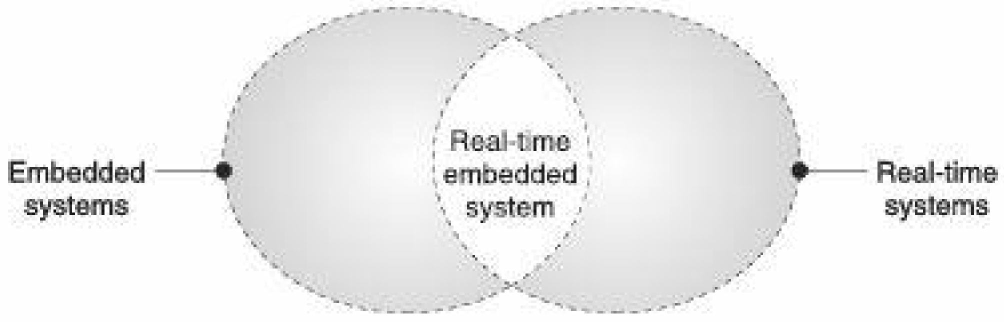


Fig. 10.3 Real-Time Embedded Systems.

Knowing this fact and because we have covered the various aspects of embedded systems in the previous sections, we can now focus our attention on real-time systems.

10.5.2 Real-Time Systems

The environment of the real-time system creates the external events. These events are received by one or more components of the real-time system. The response of the real-time system is then injected into its environment through one or more of its components. Decomposition of the real-time system, as shown in Figure 10.4 leads to the general structure of real-time systems.

The structure of a real-time system, as shown in Figure 10.4 is a controlling system and at least one controlled system. The controlling system interacts with the controlled system in various ways. First, the interaction can be *periodic*, in which communication is initiated from the controlling system to the controlled system. In this case, the communication is predictable and occurs at predefined intervals. Second, the interaction can be *aperiodic*, in which communication is initiated from the controlled system to the controlling system. In this case, the communication is unpredictable and is determined by the random occurrences of external events in the environment of the controlled system. Finally, the communication can be a combination of both types.

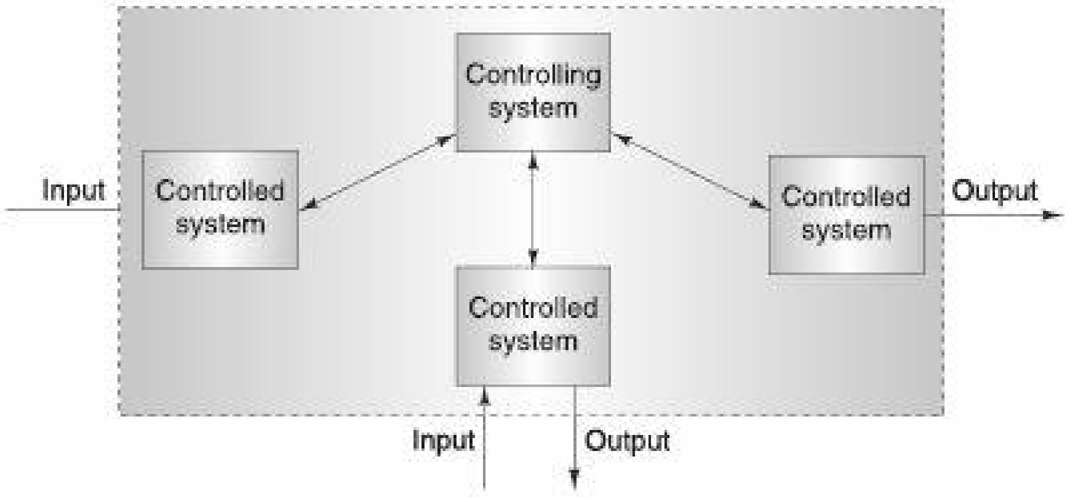


Fig. 10.4 Structure of Real-Time Systems.

The controlling system must process and respond to the events and information generated by the controlled system in a guaranteed time-frame. Imagine a real-time weapons defence system whose role is to protect a naval destroyer by shooting down incoming missiles. The idea is to shred an incoming missile into pieces with bullets before it reaches the ship. The weapons system comprises of a radar system, a command-and-decision

(C&D) system, and weapons firing control system. The controlling system is the C&D system, whereas the controlled systems are the radar system and the weapons firing control system.

- The radar system scans and searches for potential targets. Coordinates of a potential target are sent to the C&D system periodically with high frequency after the target is acquired.
- The C&D system must first determine the threat level by threat classification and evaluation, based on the target information provided by the radar system. If a threat is imminent, the C&D system must, at a minimum, calculate the speed and flight path or trajectory, as well as estimate the impact location. Because a missile tends to drift off its flight path with the degree of drift dependent on the precision of its guidance system, the C&D system calculates an area (a box) around the flight path.
- The C&D system then activates the weapons firing control system closest to the anticipated impact location and guides the weapons system to fire continuously within the moving area or box until the target is destroyed. The weapons firing control system comprises large-caliber, multi-barrel, high-muzzle velocity, high-power machine guns.

In this weapons defence system example, the communication between the radar system and the C&D system is aperiodic, because the occurrence of a potential target is unpredictable and the potential target can appear at any time. The communication between the C&D system and the weapons firing control system is, however, periodic because the C&D system feeds the firing coordinates into the weapons control system periodically (with an extremely high frequency) Initial firing coordinates are based on a pre-computed flight path but are updated in real-time according to the actual location of the incoming missile.

Consider another example of a real-time system—the cruise missile guidance system. A cruise missile flies at subsonic speed. It can travel at about 10 metres above water, 30 metres above flat ground, and 100 metres above mountain terrains. A modern cruise missile can hit a target within a 50-metre range. All these capabilities are due to the high-precision, realtime guidance system built into the nose of a cruise missile. In a simplified view, the guidance system comprises the radar system (both forward-looking and look-down radars), the navigation system, and the divert-and-altitude-control system. The navigation system contains digital maps covering the missile flight path. The forward-looking radar scans and maps out the approaching terrains. This information is fed to the navigation system in real time. The navigation system must then recalculate flight coordinates to avoid terrain obstacles. The new coordinates are immediately fed to the divert-and-altitude-control system to adjust the flight path. The look-down radar periodically scans the ground terrain along its flight path. The scanned data is compared with the estimated section of the pre-recorded maps. Corrective adjustments are made to the flight coordinates and sent to the divert-and-altitude-control system if data comparison indicates that the missile has drifted off the intended flight path.

In this example, the controlling system is the navigation system. The controlled systems are the radar system and the divert-and-altitude-control system. We can observe both periodic and aperiodic communications in this example. The communication between the radars and the navigation system is aperiodic. The communication between the navigation system and the diverand- altitude-control system is periodic.

Let us consider one more example of a real-time system—a DVD player. The DVD player must decode both the video and the audio streams from the disc simultaneously. While a movie is being played, the viewer can activate the on-screen display using a remote control. Onscreen display is a user menu that allows the user to change parameters, such as the audio output format and language options. The DVD player is the controlling system, and the remote control is the controlled system. In this case, the remote control is viewed as a sensor because it feeds events, such as pause and language selection, into the DVD player.

10.5.3 Characteristics of Real-Time Systems

The C&D system in the weapons defence system must calculate the anticipated flight path of the incoming missile quickly and guide the firing system to shoot the missile down before it reaches the destroyer. Assume T1 is the time the missile takes to reach the ship and is a function of the missile's distance and velocity. Assume T2 is the time the C&D system takes to activate the weapons firing control system and includes transmitting the firing coordinates plus the firing delay. The difference between T1 and T2 is how long the computation may take. The missile would reach its intended target if the C&D system took too long in computing the flight path. The missile would still reach its target if the computation produced by the C&D system was inaccurate. The navigation system in the cruise missile must respond to the changing terrain fast enough so that it can re-compute coordinates and guide the altitude control system to a new flight path. The missile might collide with a mountain if the navigation system cannot compute new flight coordinates fast enough, or if the new coordinates do not steer the missile out of the collision course.

Therefore, we can extract two essential characteristics of real-time systems from the examples given earlier. These characteristics are that real-time systems must produce correct computational results, called *logical or functional correctness*, and that these computations must conclude within a predefined period, called *timing correctness*.

Real-time systems are defined as those systems in which the overall correctness of the system depends on both the functional correctness and the timing correctness. The timing correctness is at least as important as the functional correctness. It is important to note that we said the timing correctness is at least as important as the functional correctness. In some real-time systems, functional correctness is sometimes sacrificed for timing correctness. We address this point shortly after we introduce the classifications of real-time systems.

Similar to embedded systems, real-time systems also have substantial knowledge of the environment of the controlled system and the applications running on it. This reason is one why many real-time systems are said to be deterministic, because in those real-time systems, the response time to a detected event is bounded. The action (or actions) taken in response to an event is known a priori. A deterministic real-time system implies that each component of the system must have a deterministic behavior that contributes to the overall determinism of the system. As can be seen, a deterministic real-time system can be less adaptable to the changing environment. The lack of adaptability can result in a less robust system. The levels of determinism and of robustness must be balanced. The method of balancing between the two is system- and application-specific.

10.5.4 Hard and Soft Real-Time Systems

In the previous section, we said that computation must complete before reaching a given deadline. In other words, real-time systems have timing constraints and are deadline-driven. Real-time systems can be classified, therefore, as either hard real-time systems or soft real-time systems. What differentiates hard real-time systems and soft real-time systems are the degree of tolerance of missed deadlines, usefulness of computed results after missed deadlines, and severity of the penalty incurred for failing to meet deadlines.

For hard real-time systems, the level of tolerance for a missed deadline is extremely small or zero tolerance. The computed results after the missed deadline may be useless for many of these systems. The penalty incurred for a missed deadline is catastrophe. For soft real-time systems, however, the level of tolerance is non-zero. The computed results after the missed deadline have a rate of depreciation. The usefulness of the results does not reach zero immediately passing the deadline, as in the case of many hard real-time systems. The physical impact of a missed deadline is non-catastrophic.

A *hard real-time system* is a real-time system that must meet its deadlines with a near-zero degree of flexibility. The deadlines must be met or catastrophes may occur. The cost of such catastrophe is extremely high and can involve human lives. The computation results obtained after the deadline have either a zero-level of usefulness or have a high rate of depreciation as time moves further from the missed deadline before the system produces a response.

A *soft real-time system* is a real-time system that must meet its deadlines but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. In a soft real-time system, a missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application. Penalty is an important aspect of hard real-time systems for several reasons.

- What is meant by 'must meet the deadline'?
- It means something catastrophic occurs if the deadline is not met. It is the penalty that sets the requirement.
- Missing the deadline means a system failure, and no recovery is possible other than a reset, so the deadline must be met. Is this a hard real-time system?

If a system failure means the system must be reset but no cost is associated with the failure, the deadline is not a hard deadline, and the system is not a hard real-time system. On the other hand, if cost is associated, either in human lives or financial penalty such as a $50 million lawsuit, the deadline is a hard deadline, and it is a hard real-time system. It is the penalty that makes this determination.

- What defines the deadline for a hard real-time system?
- It is the penalty. For a hard real-time system, the deadline is a deterministic value, and, for a soft real-time system, the value can be estimation.

One thing worth noting is that the length of the deadline does not make a real-time system hard or soft, but it is the requirement for meeting it within that time. The weapons defence and the missile guidance systems are hard real-time systems. Using the missile guidance system for an example, if the navigation system cannot compute the new coordinates in response to approaching mountain terrain before or at the deadline, not enough distance is left for the missile to change altitude. This system has zero tolerance for a missed deadline. The new coordinates obtained after the deadline are no longer useful because at subsonic speed the distance is too short for the altitude control system to navigate the missile into the new flight path in time. The penalty is a catastrophic event in which the missile collides with the mountain. Similarly, the weapons defence system is also a zero-tolerance system. The missed deadline results in the missile sinking the destroyer, and human lives potentially being lost.

Again, the penalty incurred is catastrophic. On the other hand, the DVD player is a soft real-time system. The DVD player decodes the video and the audio streams while responding to user commands in real time. The user might send a series of commands to the DVD player rapidly causing the decoder to miss its deadline or deadlines. The result or penalty is momentary but visible video distortion or audible audio distortion. The DVD player has a high level of tolerance because it continues to function. The decoded data obtained after the deadline is still useful.

Timing correctness is critical to most hard real-time systems. Therefore, hard real-time systems make every effort possible in predicting if a pending deadline might be missed. Returning to the weapons defence system, let us discuss how a hard real-time system takes corrective actions when it anticipates a deadline might be missed. In the weapons defence system example, the C&D system calculates a firing box around the projected missile flight path. The missile must be destroyed a certain distance away from the ship or the shrapnel can still cause damage. If the C&D system anticipates a missed deadline (for example, if by the time the precise firing coordinates are computed, the missile would have flown past the safe zone), the C&D system must take corrective action immediately. The C&D system enlarges the firing box and computes imprecise firing coordinates by methods of estimation instead of computing for precise values. The C&D system then activates additional weapons firing systems to compensate for this imprecision. The result is that additional guns are brought online to cover the larger firing box. The idea is that it is better to waste bullets than sink a destroyer.

This example shows why sometimes functional correctness might be sacrificed for timing correctness for many real-time systems. Because one or a few missed deadlines do not have a detrimental impact on the operations of soft real-time systems, a soft real-time system might not need to predict if a pending deadline might be missed. Instead, the soft real-time system can begin a recovery process after a missed deadline is detected.

For example, using the real-time DVD player, after a missed deadline is detected, the decoders in the DVD player use the computed results obtained after the deadline and use the data to make a decision on what future video frames and audio data must be discarded to resynchronize the two streams. In other words, the decoders find ways to catch up. So far, we have focused on meeting the deadline or the finish time of some work or job, e.g., a computation. At times, meeting the start time of the job is just as important. The lack of required resources for the job, such as CPU or memory, can prevent a job from starting and can lead to missing the job completion deadline. Ultimately this problem becomes a resource scheduling problem. The scheduling algorithms of a real-time system must schedule system resources so that jobs created in response to both periodic and aperiodic events can obtain the resources at the appropriate time. This process affords each job the ability to meet its specific timing constraints.

10.5.5 Defining an RTOS

A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. Good RTOSes, therefore, are scalable in order to meet different sets of requirements for different applications.

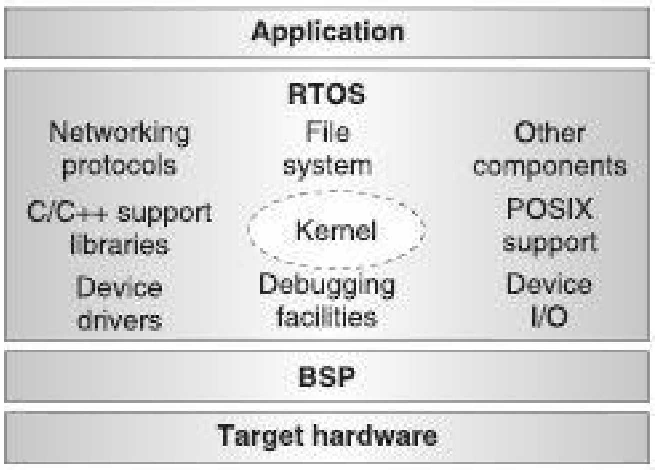


Fig. 10.5 High-level View of an RTOS, its Kernel, other Components in Embedded Systems.

Although many RTOSes can scale up or down to meet application requirements, this book focuses on the common element at the heart of all RTOSes the kernel. Most RTOS kernels contain the following components:

- **Scheduler** is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.
- **Objects** are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
- **Services** are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

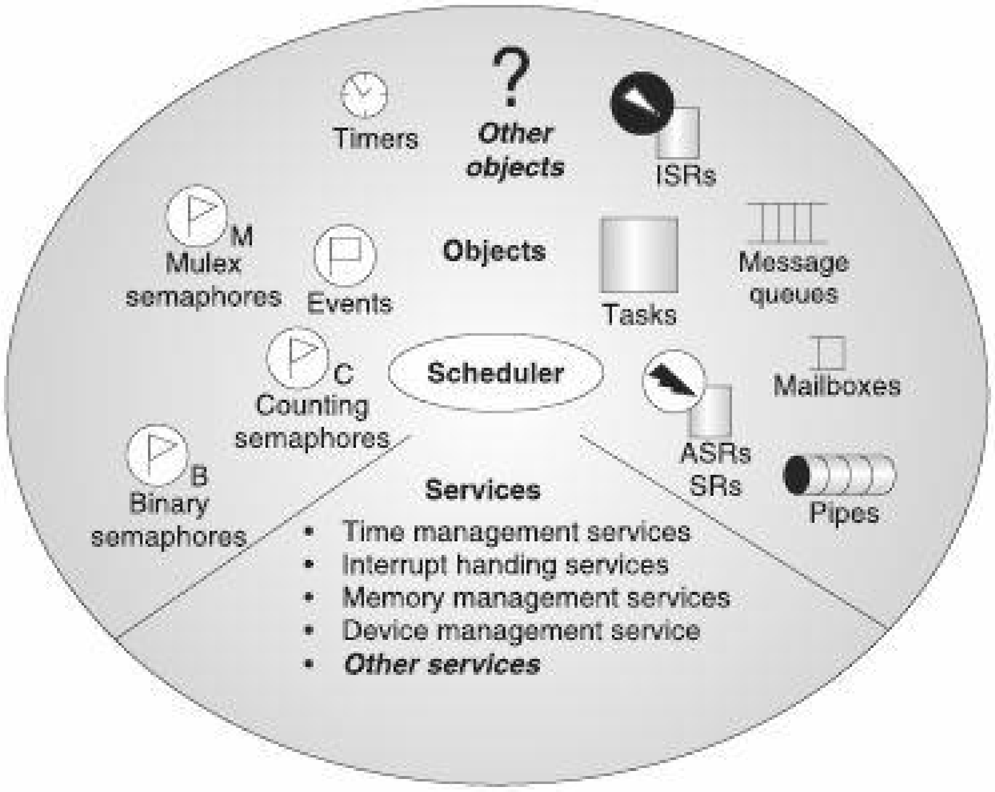


Fig. 10.6 Common Components in an RTOS Kernel that Including Objects, the Scheduler, and some Services.

This diagram is highly simplified; remember that not all RTOS kernels conform to this exact set of objects, scheduling algorithms, and services.

10.5.6 The Scheduler

The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. To understand how scheduling works, this section describes the following topics:

- schedulable entities,
- multitasking,
- context switching,

- dispatcher, and
- scheduling algorithms.

10.5.7 Schedulable Entities

A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. Tasks and processes are all examples of schedulable entities found in most kernels. A task is an independent thread of execution that contains a sequence of independently schedulable instructions. Some kernels provide another type of a schedulable object called a process. Processes are similar to tasks in that they can independently compete for CPU execution time.

Processes differ from tasks in that they provide better memory protection features, at the expense of performance and memory overhead. Despite these differences, for the sake of simplicity, this book uses task to mean either a task or a process.

So, how exactly does a scheduler handle multiple schedulable entities that need to run simultaneously? The answer is by multitasking. The multitasking discussions are carried out in the context of uniprocessor environments.

10.5.8 Multitasking

Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run. One such multitasking scenario is illustrated in Figure 10.7.

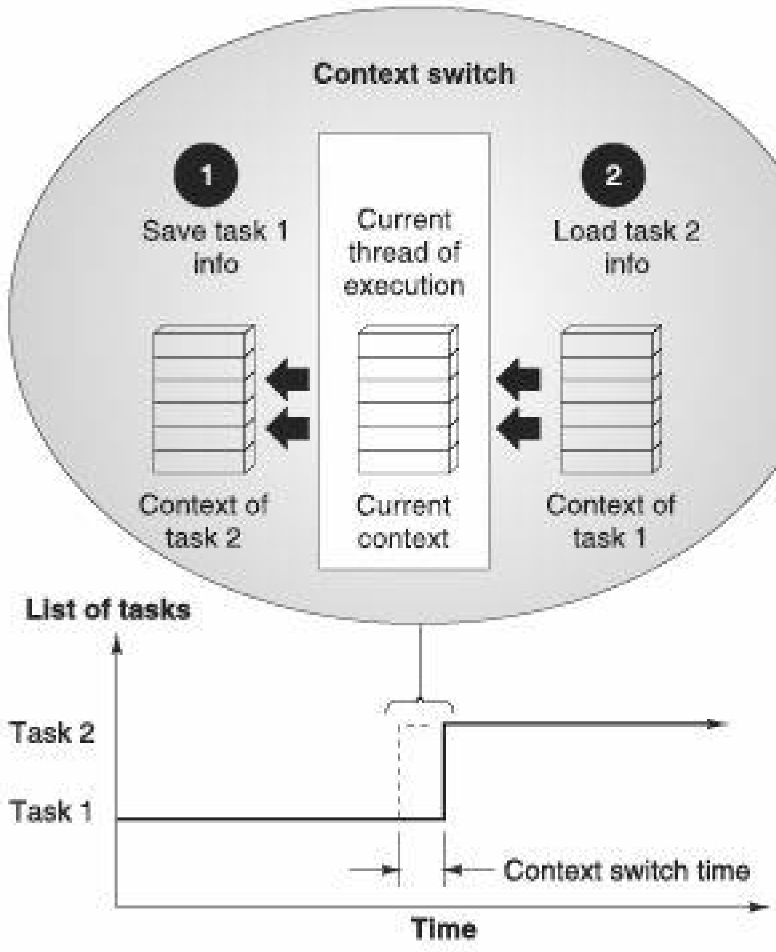


Fig. 10.7 Multitasking Using a Context Switch.

In this scenario, the kernel multitasks in such a way that many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm. An important point to note here is that the tasks follow the kernel's scheduling algorithm, while interrupt service routines (ISR) are triggered to run because of hardware interrupts and their established priorities.

As the number of tasks to schedule increases, so do CPU performance requirements. This fact is due to increased switching between the contexts of the different threads of execution.

10.5.9 The Context Switch

Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another. To better understand what happens during a context switch, let's examine further what a typical kernel does in this scenario. Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). TCBs are system data structures that the kernel uses to maintain task specific information. TCBs contain everything a kernel needs to know about a particular task. When a task is running, its context is highly dynamic. This dynamic context is maintained in the TCB. When the task is not running, its context is frozen within the TCB, to be restored the next time the task runs. A typical context switch scenario is illustrated in Figure 10.7. When the kernel's scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1. The kernel saves task 1's context information in its TCB.
2. It loads task 2's context information from its TCB, which becomes the current thread of execution.
3. The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

The time it takes for the scheduler to switch from one task to another is the context switch time. It is relatively insignificant compared to most operations that a task performs. If an application's design includes frequent context switching, however, the application can incur unnecessary performance overhead. Therefore, design applications in a way that does not involve excess context switching.

Every time an application makes a system call, the scheduler has an opportunity to determine if it needs to switch contexts. When the scheduler determines a context switch is necessary, it relies on an associated module, called the dispatcher, to make that switch happen.

10.5.10 The Dispatcher

The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. At anytime an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel. When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing the control to one of the tasks in the user's application. It will not necessarily be the same task that made the system call. It is the scheduling algorithms (to be discussed shortly) of the scheduler that determines which task executes next. It is the dispatcher that does the actual work of context switching and passing execution control. Depending on how the kernel is first entered, dispatching can happen differently. When a task makes system calls, the dispatcher is used to exit the kernel after every system call completes.

In this case, the dispatcher is used on a call-by-call basis so that it can coordinate task state transitions that any of the system calls might have caused. (One or more tasks may have become ready to run, for example.)

On the other hand, if an ISR makes system calls, the dispatcher is bypassed until the ISR fully completes its execution. This process is true even if some resources have been freed that would normally trigger a context switch between tasks. These context switches do not take place because the ISR must complete without being interrupted by tasks. After the ISR completes execution, the kernel exits through the dispatcher so that it can then dispatch the correct task.

10.5.11 Scheduling Algorithms

As mentioned earlier, the scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels support two common scheduling algorithms:

- Pre-emptive priority-based scheduling, and
- Round-robin scheduling.

The RTOS manufacturer typically predefines these algorithms; however, in some cases, developer scan create and define their own scheduling algorithms. Each algorithm is described next.

Pre-emptive priority-based scheduling

Of the two scheduling algorithms introduced here, most real-time kernels use pre-emptive priority-based scheduling by default. As shown in Figure 10.8 with this type of scheduling, the task that gets to run at any point is the task with the highest priority among all other tasks ready to run in the system.

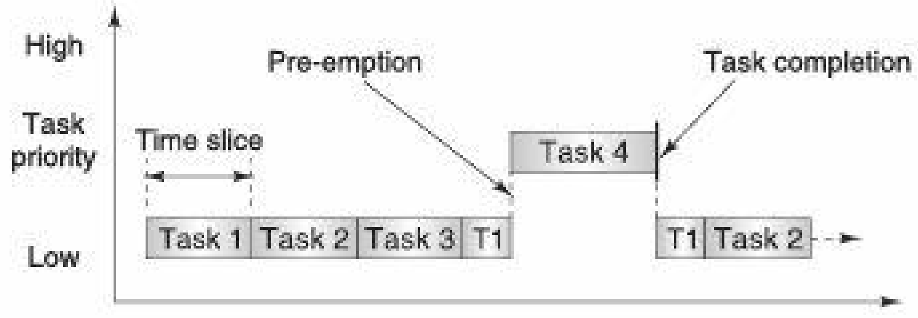


Fig. 10.8 Pre-emptive Priority-based Scheduling.

Real-time kernels generally support 256 priority levels, in which 0 is the highest and 255 the lowest. Some kernels appoint the priorities in reverse order, where 255 is the highest and 0 the lowest. Regardless, the concepts are basically the same. With a preemptive priority based scheduler, each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current task's context in its TCB the task and switches to the higher-priority task. As shown in Figure 10.8, task 1 is preempted by higher-priority task 2, which is then preempted by task 3. When task 3 completes, task 2 resumes; likewise, when task 2 completes, task 1 resumes.

Although tasks are assigned a priority when they are created, a task's priority can be changed dynamically using kernel-provided calls. The ability to change task priorities dynamically allows an embedded application the flexibility to adjust to external events as they occur, creating a true real-time, responsive system. Note, however, that misuse of this capability can lead to priority inversions, deadlock, and eventual system failure.

Round-robin scheduling

Round-robin scheduling provides each task an equal share of the CPU execution time. Pure round-robin scheduling cannot satisfy real-time system requirements because in real-time systems, tasks perform work of varying degrees of importance. Instead, pre-emptive, priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of the CPU for tasks of the same priority as shown in Figure 10.9.

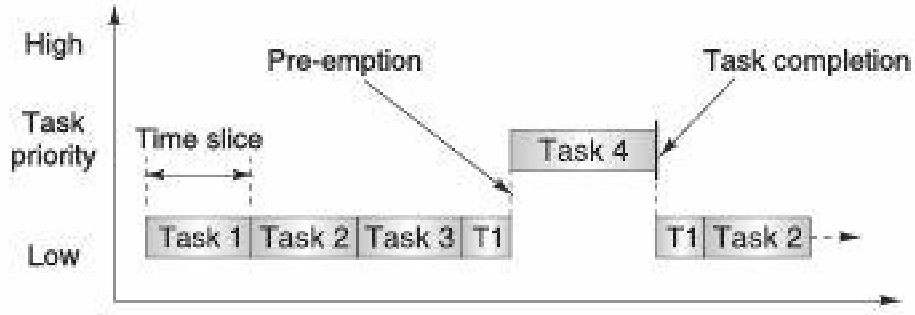


Fig. 10.9 Round-Robin and Pre-emptive Scheduling.

With time slicing, each task executes for a defined interval, or time slice, in an ongoing cycle, which is the round robin. A run-time counter tracks the time slice for each task, incrementing on every clock tick. When one task's time slice completes, the counter is cleared, and the task is placed at the end of the cycle. Newly added tasks of the same priority are placed at the end of the cycle, with their run-time counters initialized to 0.

If a task in a round-robin cycle is pre-empted by a higher-priority task, its run-time count is saved and then restored when the interrupted task is again eligible for execution. This idea is illustrated in Figure 8.40, in which task 1 is pre-empted by a higher-priority task 4 but resumes where it left off when task 4 completes.

10.5.12 Key Characteristics of an RTOS

An application's requirements define the requirements of its underlying RTOS. Some of the more common attributes are

- reliability,
- predictability,

- performance,
- compactness, and
- scalability.

These attributes are discussed next; however, the RTOS attribute an application needs depends on the type of application being built.

10.5.12.1 Reliability

Embedded systems must be reliable. Depending on the application, the system might need to operate for long periods without human intervention. Different degrees of reliability may be required. For example, a digital solar-powered calculator might reset itself if it does not get enough light, yet the calculator might still be considered acceptable. On the other hand, a telecom switch cannot reset during operation without incurring high associated costs for down time. The RTOSes in these applications require different degrees of reliability.

Although different degrees of reliability might be acceptable, in general, a reliable systemis one that is available (continues to provide service) and does not fail. A common way that developers categorize highly reliable systems is by quantifying their downtime per year, as shown in Table below. The percentages under the 'Number of 9s' column indicate the percent of the total time that a system must be available.

While RTOSes must be reliable, note that the RTOS by itself is not what is measured to determine system reliability. It is the combination of all system elements—including the hardware, BSP, RTOS, and application—that determines the reliability of a system.

Number of 9s	*Downtime per year*	*Typical application*
3 nines (99.9%)	9 hour	desktop
4 nines (99.99%)	5 hour	Enterprise server
5 nines (99.999%)	5 minutes	Carrier class server
6 nines (99.9999%)	31 seconds	Carrier switch equipment

Source: 'Providing Open Architecture High Availability Solutions,' Revision 1.0, Published by HA Forum, February 2001.

10.5.12.2 Predictability

Because many embedded systems are also real-time systems, meeting time requirements is key to ensuring proper operation. The RTOS used in this case needs to be predictable to a certain degree. The term deterministic describes RTOSes with predictable behavior, in which the completion of operating system calls occurs within known time-frames. Developers can write simple benchmark programs to validate the determinism of an RTOS. The result is based on timed responses to specific RTOS calls. In a good deterministic RTOS, the variance of the response times for each type of system call is very small.

10.5.12.3 Performance

This requirement dictates that an embedded system must perform fast enough to fulfill its timing requirements. Typically, the more deadlines to be met—and the shorter the time between them—the faster the system's CPU must be. Although underlying hardware can dictate a system's processing power, its software can also contribute to system performance. Typically, the processor's performance is expressed in million instructions per second (MIPS).

Throughput also measures the overall performance of a system, with hardware and software combined. One definition of throughput is the rate at which a system can generate output based on the inputs coming in. Throughput also means the amount of data transferred divided by the time taken to transfer it. Data transfer throughput is typically measured in multiples of bits per second (bps).

Sometimes developers measure RTOS performance on a call-by-call basis. Benchmarks are written by producing timestamps when a system call starts and when it completes. Although this step can be helpful in the analysis stages of design, true performance testing is achieved only when the system performance is measured as a whole.

10.5.12.4 Compactness

Application design constraints and cost constraints help determine how compact an embedded system can be. For example, a cell phone clearly must be small, portable, and low cost. These design requirements limit system memory, which in turn limits the size of the application and operating system. In such embedded systems, where hardware real estate is limited due to size and costs, the RTOS clearly must be small and efficient. In these cases, the RTOS memory footprint can be an important factor. To meet total system requirements, designers must understand both the static and dynamic memory consumption of the RTOS and the application that will run on it.

10.5.12.5 Scalability

Because RTOSes can be used in a wide variety of embedded systems, they must be able to scale up or down to meet application-specific requirements. Depending on how much functionality is required, an RTOS should be capable of adding or deleting modular components, including file systems and protocol stacks.

If an RTOS does not scale up well, development teams might have to buy or build the missing pieces. Suppose that a development team wants to use an RTOS for the design of a cellular phone project and a base station project. If an RTOS scales well, the same RTOS can be used in both projects, instead of two different RTOSes, which saves considerable time and money.

10.6 APPLICATION AREAS

Embedded software is present in almost every electronic device you use. There is embedded software inside your watch, cellular phone, automobile, thermostats, industrial control equipment, and scientific and medical equipment. Defence services use it to guide missiles and detect enemy aircraft. Thus, embedded systems cover such a broad range of products that generalization is difficult. Here are some broad categories:

Aerospace and defence electronics (ADE): Astronomical research, flight safety and flight management, fire control, robotics, vehicular control.

Broadcast and entertainment: Analog and digital sound products, audio control systems, DVD players, digital TV, set-top boxes.

Data communication: Analog modems, ATM broad band switches, cable modems.

Digital imaging: Digital still camera, digital video cameras, fax machines, printers, scanners.

Industrial measurement and control: Building environmental control systems, industrial sensors and test & measurement devices, traffic management systems.

Medical electronics: Cardiovascular devices, critical care systems, diagnostic devices, surgical devices.

Server I/O: Embedded servers, LAN devices, supercomputing, server management.

Mobile data infrastructures: Mobile data terminals, satellite terminals, wireless LANs, pagers, wireless phones.

Things to Remember

◊ An embedded system is built for a specific application. As such, the hardware and software components are highly integrated, and the development model is the hardware and software co-design model.

◊ Embedded systems are generally built using embedded processors.

◊ An embedded processor is a specialized processor, such as a DSP, that is cheaper to design and produce, can have built-in integrated devices, is limited in functionality, produces low heat, consumes low power, and does not necessarily have the fastest clock speed but meets the requirements of the specific applications for which it is designed.

◊ Real-time systems are characterized by the fact that timing correctness is just as important as functional or logical correctness.

◊ The severity of the penalty incurred for not satisfying timing constraints differentiates hard real-time systems from soft real-time systems.

◊ Real-time systems have a significant amount of application awareness similar to embedded systems.

◊ Real-time embedded systems are those embedded system with real-time behaviors. Embedded Systems

◊ RTOSes are best suited for real-time, application-specific embedded systems; GPOSes are typically used for general-purpose systems.

◊ RTOSes are programs that schedule execution in a timely manner, manage system resources, and provide a consistent foundation for developing application code.

◊ Kernels are the core module of every RTOS and typically contain kernel objects, services, and scheduler.

◊ Kernels can deploy different algorithms for task scheduling. The most common two algorithms are pre-emptive priority-based scheduling and round-robin scheduling.

◊ RTOSes for real-time embedded systems should be reliable, predictable, high performance, compact, and scalable.

Exercise

1. What is an embedded system?
2. What is the difference between software and hardware?
3. Give some characteristics of embedded systems?
4. List various design requirements of embedded systems?
5. Differentiate between general-purpose processors, single function processors and application specifiic processors?
6. What are the various applications of embedded systems?
7. What is the role of the infinite loop in embedded systems?
8. Why "const" is used in embedded systems?
9. What are real-time systems?
10. What is the function of volatile?
11. Write a program to set a bit.
12. Write a program to clear a bit.

Appendix **A**

The 8085 Instruction Set

ACI data (8b)

Description: Add Immediate 8-bit data to the accumulator with carry.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: CE

Flags: All flags are affected based upon the result of the addition.

ADC R

Description: Add register to accumulator with carry.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
8F	A
88	B
89	C
8A	D
8B	E
8C	H
8D	L

Flags: All flags are affected based upon the result of the addition.

ADC M

Description: Add contents of memory location pointed to by HL register pair to the accumulator with carry.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: 8E

Flags: All flags are affected based upon the result of the addition.

ADD R

Description: Add register to accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
87	A
80	B
81	C
82	D
83	E
84	H
85	L

Flags: All flags are affected based upon the result of the addition.

ADD M

Description: Add contents of memory location pointed to by HL to the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: 86

Flags: All flags are affected based upon the result of the addition.

ADI data (8b)

Description: Add the immediate 8-bit data to the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: C6

Flags: All flags are affected based upon the result of the addition.

ANA R

Description: The contents of the accumulator and the register are logically ANDed and the result is put in the accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
A7	A
A0	B
A1	C
A2	D
A3	E
A4	H
A5	L

Flags: S,Z and P are modified based upon the result of the operation. CY is reset, and AC is set.

ANA M

Description: The contents of the accumulator and the contents of the memory location pointed to by HL are logically ANDed, and result is put in the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: A6

Flags: S,Z and P are modified based upon the result of the operation. CY is reset, and AC is set.

ANI data (8b)

Description: The contents of accumulator and the 8-bit data are ANDed and the result is put in the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: E6

Flags: S,Z and P are modified based upon the result of the operation. CY is reset, and AC is set.

CALL address (16b)

Description: The program sequence is transferred to the address specified by 16-bit address. Before the program is transferred, the address of the instruction following the CALL instruction is pushed onto the stack.

Bytes/M-cycles/T-states: 3/5/18

Hex Codes: CD

Flags: No flags are affected.

CC address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if the CY flag is set. If CY=0, no transfer takes place. If the transfer takes place, the address of the instruction following the CC instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	if transfer is not taken
	3/5/18	if the transfer is taken

Hex Codes: DC

Flags: No flags are affected.

CNC address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if the CY flag is not set. If CY=1, no transfer takes place. If the transfer takes place, the address of the instruction following the CNC instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: D4

Flags: No flags are affected.

CP address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if positive, or if the S flag=0. If S=1, no transfer takes place. If the transfer takes place, the address of the instructon following the CP instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: F4

Flags: No flags are affected.

CM address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if minus, or if the S flag=1. If S=0, no transfer takes place. If the transfer takes place, the address of the instruction following the CM instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: FC

Flags: No flags are affected.

CPE address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if parity is even, or the P flag=1. If P=0, no transfer takes place. If the transfer takes place, the address of the instruction following the CPE instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: EC

Flags: No flags are affected.

CPO address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if parity is odd, or the P flag=0. If P=1, no transfer takes place. If the transfer takes place, the address of the instruction following the CPO instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: E4

Flags: No flags are affected.

CZ address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if zero, or if the Z flag=1. If Z=0, no transfer takes place. If the transfer takes place, the address of the instruction following the CZ instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: CC

Flags: No flags are affected.

CNZ address (16b)

Description: The program sequence is transferred to the address specified by the 16-bit address if not zero, or if the Z flag=0. If Z=1, no transfer takes place. If the transfer takes place, the address of the instruction following the CNZ instruction is pushed onto the stack.

Bytes/M-cycles/T-states:	3/2/9	(if transfer is not taken)
	3/5/18	(if transfer is taken)

Hex Codes: C4

Flags: No flags are affected.

CMA

Description: The contents of the accumulator are complemented.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 2F

Flags: No flags are affected.

CMC

Description: The carry flag is complimented.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 3F

Flags: The CY flag is complimented. No other flags are affected.

CMP R

Description: The contents of the register are compares to the contents of the accumulator. Both contents are unaffected, and the following flags are used to show the results of the compare:

If A<R	CY=1 and Z=0
If A=R	CY=0 and Z=1
If A>R	CY=0 and Z=0

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
BF	A
B8	B
B9	C
BA	D
BB	E

BC	H
BD	L

Flags: S, P and AC are also affected based upon the results of the operation, besides Z and CY.

CMP M

Description: The contents of the memory location pointed to by HL are compared to the contents of the accumulator. Both contents are unaffected, and the following flags are used to show the results of the compare:

If A<R	CY=1 and Z=0
If A=R	CY=0 and Z=1
If A>R	CY=0 and Z=0

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: BE

Flags: S,P and AC are also affected based upon the results of the operation, besides Z and CY.

CPI data (8b)

Description: The 8-bit data is compared with the contents of the accumulator. The contents of the accumulator are unaffected, and the following flags are used to show the results of the compare:

If A<R	CY=1 and Z=0
If A=R	CY=0 and Z=1
If A>R	CY=0 and Z=0

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: FE

Flags: S, P and AC are also affected based upon the results of the operation, besides Z and CY.

DAA

Description: The contents of the accumulator are converted from a binary value to two 4-bit Binary Coded Decimal (BCD) digits.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 27

Flags: S, Z, AC, P and CY are affected based upon the results of the operation.

DAD Rp

Description: The contents of register pair (Rp) are added to the contents of the register pair HL. The source register pair is unchanged, and the results are stored in HL.

Bytes/M-cycles/T-states: 1/3/10

Hex Codes:

	Register Pair
09	BC
19	DE

29	HL
39	SP

Flags: If the result is larger than 16 bits, the CY flag is set, otherwise no flags are affected.

DCR R.

Description: The contents of the register are decremented by one. The result is stored in the register.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
3D	A
05	B
0D	C
15	D
1D	E
25	H
2D	L

Flags: S, Z, AC and P are affected based upon the results of the operation. The CY flag is not affected.

DCR M

Description: The contents of the memory location pointed to by HL is decremented by one, and the results are stored in memory location.

Bytes/M-cycles/T-states: 1/3/10

Hex Codes: 35

Flags: S, Z, AC and P are affected based upon the results of the operation. The CY flag is not affected.

DCX Rp

Description: The contents of the register pair are decremented by 1. The result is stored in the as a 16-bit number.

Bytes/M-cycles/T-states: 1/1/6

Hex Codes:

	Register Pair
0B	BC
1B	DE
2B	HL
3B	SP

Flags: No flags are affected.

DI

Description: The Interrupt Enable flip-flop is reset, and the interrupts except the TRAP interrupt are disabled.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: F3

Flags: No flags are affected.

EI

Description: The Interrupt Enable flip-flop is set, and all of the interrupts are enabled.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: FB

Flags: No flags are affected.

HLT

Description: The MPU finishes executing the current instruction and halts any further execution. The MPU enters the Halt Acknowledge machine cycle, and Wait states are inserted in every clock period. It requires an interrupt or a reset to get the MPU out of the Halt state.

Bytes/M-cycles/T-states: One/two or more/five or more

Hex Codes: 76

Flags: No flags are affected.

IN port address (8b)

Description: The contents of the input port designated are read and loaded into the accumulator.

Bytes/M-cycles/T-states: 2/3/10

Hex Codes: DB

Flags: No flags are affected.

INR R

Description: The contents of the register are incremented by one and stored in the register.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
3C	A
04	B
0C	C
14	D
1C	E
24	H
2C	L

Flags: S, Z, P, AC are affected by the results of the operation. CY is not modified.

INR M.

Description: The contents of the memory location pointed to by HL are incremented by one and the result is put in memory location.

Bytes/M-cycles/T-states: 1/3/10

Hex Codes: 34

Flags: S, Z, P, AC are affected by the results of the operation. CY is not modified.

INX Rp

Description: The contents of the register pair are incremented by one and stored in the register pair. The instruction views the two registers as a 16-bit number.

Bytes/M-cycles/T-states: 1/1/6

Hex Codes:

	Register Pair
03	BC
13	DE
23	HL
33	SP

Flags: No flags are affected.

JMP address (16b)

Description: The program execution is transferred to the memory address specified.

Bytes/M-cycles/T-states: 3/3/10

Hex Codes: C3

Flags: No flags are affected.

JC address (16b)

Description: Program execution is transferred to the memory address specified if the carry flag is set, or CY=1. If CY=0, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: DA

Flags: No flags are affected.

JNC address (16b)

Description: Program execution is transferred to the memory address specified if the carry flag is not set, or CY=0. If CY=1, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: D2

Flags: No flags are affected.

JP address (16b)

Description: Program execution is transferred to the memory address specified if positive, or S=0. If S=1, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: F2

Flags: No flags are affected.

JM address (16b)

Description: Program execution is transferred to the memory address specified if minus, or S=1. If S=0, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: FA

Flags: No flags are affected.

JPE address (16b)

Description: Program execution is transferred to the memory address specified if parity is even, or P=1. If P=0, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: EA

Flags: No flags are affected.

JPO address (16b)

Description: Program execution is transferred to the memory address specified if parity is odd, or P=0. If P=1, no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: E2

Flags: No flags are affected.

JZ address (16b)

Description: Program execution is transferred to the memory address specified if zero , or Z=1.if Z=0,no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: CA

Flags: No flags are affected.

JNZ address (16b)

Description: Program execution is transferred to the memory address specified if zero , or Z=0.if Z=1,no transfer takes place.

Bytes/M-cycles/T-states:	3/2/7	if condition is not true
	3/3/10	if condition is true

Hex Codes: C2

Flags: No flags are affected.

LDA address (16b)

Description: The contents of the memory location specified are transferred to the accumulator.

Bytes/M-cycles/T-states: 3/4/13

Hex Codes: 3A

Flags: No flags are affected.

LDAX Rp

Description: The contents of the memory location pointed to by the register pair are loaded into the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes:

	Register pairs
0A	BC
1A	DE

Flags: No flags are affected.

LHLD address (16b)

Description: The contents of the memory location specified are loaded into register L and the contents of the next memory location are loaded into register H.

Bytes/M-cycles/T-states: 3/5/16

Hex Codes: 2A

Flags: No flags are affected.

LXI Rp, data (16b)

Description: The16-bit data is loaded into register pair.

Bytes/M-cycles/T-states: 3/3/10

Hex Codes:

	Register Pair
01	BC
11	DE
21	HL
31	SP

Flags: No flags are affected.

MOV Rd,Rs

Description: The contents of the source register Rs are transferred into the destination register Rd.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

Destination Register ← \ Source Register →	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	41	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

Flags: No flags are affected.

MOV M,Rs

Description: The contents of the source register Rs are transferred to the location pointed to by HL.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes:

	Source Register
77	A
70	B
71	C
72	D
73	E
74	H
75	L

Flags: No flags are affected.

MOV M,Rs

Description: The contents of the memory location pointed to by HL are transferred to the destination register.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes:

	Destination Register
7E	A
46	B
4E	C
56	D
5E	E
66	H
6E	L

Flags: No flags are affected.

MVI M, data (8b)

Description: The 8 bits of data are stored in the register.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes:

	Register
3E	A
06	B
0E	C
16	D
1E	E
26	H
2E	L

Flags: No flags are affected.

MVI M, data (8b)

Description: The 8 bits of data are stored in the memory location pointed to by HL.

Bytes/M-cycles/T-states: 2/3/10

Hex Codes: 36

Flags: No flags are affected.

NOP

Description: No operation is performed. The instruction is fetched and decoded, but no operation is executed.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 00

Flags: No flags are affected.

ORA R

Description: The contents of the accumulator are logically OR'd with the contents of the register. The result is stored in the accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
B7	A
B0	B
B1	C
B2	D
B3	E
B4	H
B5	L

Flags: Z, S and P are affected based upon the operation. AC and CY are reset.

ORA M

Description: The contents of the accumulator are logically OR'd with the contents of the memory location pointed to by HL.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: B6

Flags: Z, S and P are affected based upon the operation. AC and CY are reset.

ORI data(8b)

Description: The contents of the accumulator are logically OR'd with the 8 bits of data. The result is stored in the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: F6

Flags: Z, S and P are affected based upon the operation. AC and CY are reset.

OUT port address(8b)

Description: The contents of the accumulator are copied out to the output port specified.

Bytes/M-cycles/T-states: 2/3/10

Hex Codes: D3

Flags: No flags are affected.

PHCL

Description: The contents of register H and L are copied into the program counter. H is the high-order bits and L is the low-order bits.

Bytes/M-cycles/T-states: 1/1/6

Hex Codes: E9

Flags: No flags are affected.

POP Rp

Description: The contents of memory location (stack) pointed to by the stack pointer are copied to the low-order register of the register pair. {C, E, L and flags}. The stack pointer is then incremented and the contents of that memory location being pointed to are copied to the high-order register of the register pair.

Bytes/M-cycles/T-states: 1/3/10

Hex Codes:

	Register
C1	BC
D1	DE
E1	HL
F1	PSW

Flags: No flags are affected.

PUSH Rp

Description: The contents of the register pair are into the stack. The high-order register (B, D, H, A) is put on the stack first, then the contents of the low-order register (C, E, L, flags) are put onto the stack.

Bytes/M-cycles/T-states: 1/3/12

Hex Codes:

	Register pair
C5	B
D5	D
E5	H
F5	PSW

Flags: No flags are affected.

RAL

Description: The contents of accumulator are rotated left by one position through the carry flag.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 17

Flags: CY is modified according to bit D7. S, Z, P and AC are not affected.

RAR

Description: The contents of accumulator are rotated right by one position through the carry flag.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 1F

Flags: CY is modified according to bit D0. S, Z, P and AC are not affected.

RLC

Description: The contents of accumulator are rotated left by one position. Bit D7 is placed in both D0 and CY.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 07

Flags: CY is modified according to bit D7. S, Z, P and AC are not affected.

RRC

Description: The contents of accumulator are rotated right by one bit. Bit D0 is placed in both D7 and CY at same time.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 0F

Flags: CY is modified according to bit D0. S, Z, P and AC are not affected.

RET

Description: The program sequence is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the Program Counter and this is the address that program execution begins.

Bytes/M-cycles/T-states: 1/3/10

Hex Codes: C9

Flags: No flags are affected.

RC

Description: The program sequence is transferred from the subroutine to the calling program if the carry flag is set, or CY=1. If CY=0, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: D8

Flags: No flags affected.

RNC

Description: The program sequence is transferred from the subroutine to the calling program if the carry flag is not set, or CY=0. If CY=1, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: D0

Flags: No flags affected.

RP

Description: The program sequence is transferred from the subroutine to the calling program if positive, or S=0. If S=1, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: F0

Flags: No flags affected.

RM

Description: The program sequence is transferred from the subroutine to the calling program if minus, or S=1. If S=0, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: F8

Flags: No flags affected.

RPE

Description: The program sequence is transferred from the subroutine to the calling program if parity is even, or P=1. If P=0, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: E8

Flags: No flags are affected.

RPO

Description: The program sequence is transferred from the subroutine to the calling program if parity is odd, or P=0. If P=1, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: E0

Flags: No flags are affected.

RZ

Description: The program sequence is transferred from the subroutine to the calling program if zero, or Z=1. If Z=0, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: C8

Flags: No flags are affected.

RNZ

Description: The program sequence is transferred from the subroutine to the calling program if not zero, or Z=0. If Z=1, no transfer takes place.

Bytes/M-cycles/T-states:	1/1/6	if condition is not true
	1/3/12	if condition is true

Hex Codes: C0

Flags: No flags are affected.

RIM

Description: The instruction is used to both read in the status of interrupts 7.5, 6.5 and 5.5 as well as to read in the serial input data bit. An 8-bit word is read in and stored in the accumulator. The layout of that word is shown in Fig. A1.

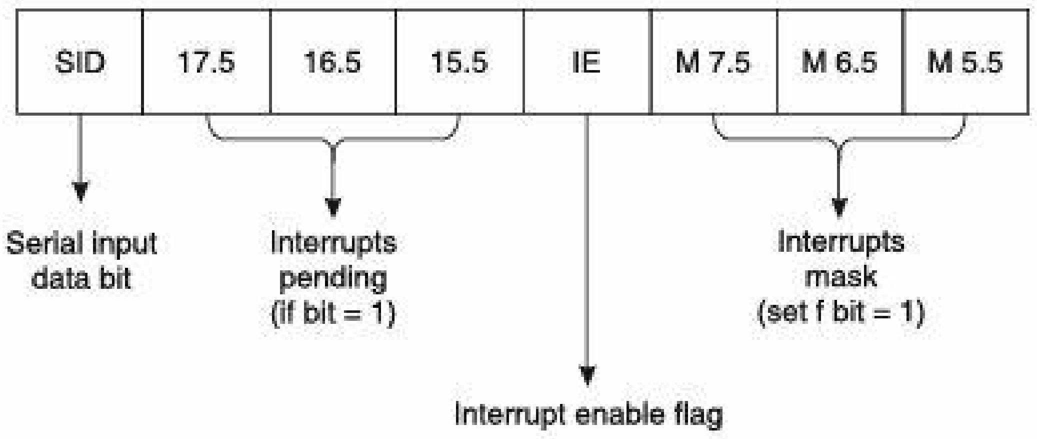


Fig. A1 The RIM Instructon Layout in the Accumulator

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 20

Flags: No flags are affected.

RST n (where n=0-7)

Description: This instruction operates like a call instruction that goes to one of the eight predetermined memory locations on page 0. Each instruction (RST 0-RST7) goes to a specific address listed text.

Instruction	Restart Address
RST0	0000
RST1	0008
RST2	0010
RST3	0018
RST4	0020
RST5	0028
RST6	0030
RST7	0038

Bytes/M-cycles/T-states: 1/3/12

Hex Codes:		
	C7	RST0
	CF	RST1
	D7	RST2
	DF	RST3
	E7	RST4
	EF	RST5
	F7	RST6
	FF	RST7

Flags: No flags are accepted.

SBB R

Description: The contents of a register and the borrow flag are subtracted from the contents of the accumulator and the results are stored in the accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
9F	A
98	B
99	C
9A	D
9B	E
9C	H
9D	L

Flags: All flags are affected based upon the results of the operation.

SBB M

Description: The contents of the memory location pointed to by HL and the borrow flag are subtracted from the contents of the accumulator. The results are then stored in the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: 9E

Flags: All flags are affected based upon the results of the operation.

SBI data (8b)

Description: The 8 bits of data and the borrow flag are subtracted from the accumulator and the results are stored in the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: DE

Flags: All flags are affected based upon the results of the operation.

SHLD address (16b)

Description: The contents of register L are stored at the memory location specified and contents of register H are stored at the next memory location by incrementing the operand by 1.

Bytes/M-cycles/T-states: 3/5/16

Hex Codes: 22

Flags: No flags are affected.

SIM

Description: This is an instruction that is used to set the interrupt masks as well as set the serial output data bit. The accumulator is laid out as shown in Fig. A2.

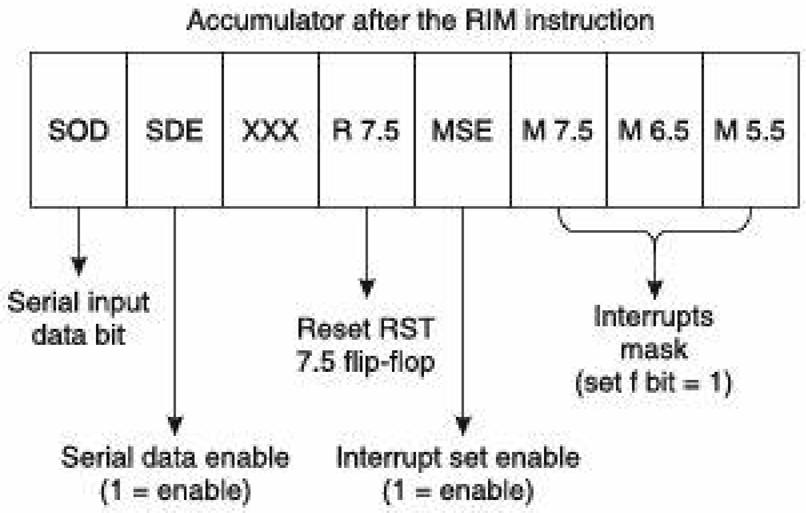

Accumulator after the RIM instruction
SOD
SDE
XXX
R 7.5
MSE
M 7.5
M 6.5
M 5.5
Serial input data bit
Reset RST 7.5 flip-flop
Interrupts mask (set f bit = 1)
Serial data enable (1 = enable)
Interrupt set enable (1 = enable)

Fig. A2 The SIM Instruction Accumulator Layout

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 30

Flags: No flags are affected.

SPHL

Description: The contents of register H and L are loaded into the Stack Pointer. H has the higher-order part of the address, while L has the low-order portion.

Bytes/M-cycles/T-states: 1/1/6

Hex Codes: F9

Flags: No flags are affected.

STA address (16b)

Description: The contents of the accumulator are stored at the memory location specified.

Bytes/M-cycles/T-states: 3/4/13

Hex Codes: 32

Flags: No flags are affected.

STAX Rp (only BC or DE)

Description: The contents of the accumulator are stored at the memory location pointed to by register pair. The contents of the accumulator are not affected.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes:

	Register Pair
02	BC
12	DE

Flags: No flags are affected.

STC

Description: The carry flag, CY, is set to 1.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: 37

Flags: Only the CY flag is affected.

SUB R

Description: The contents of the register are subtracted from the accumulator and the result is stored in the accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
97	A
90	B
91	C
92	D
93	E
94	H
95	L

Flags: All flags are affected by the result of the operation.

SUB M

Description: The contents of the memory location pointed to by HL are subtracted from the accumulator and the result is stored in the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: 96

Flags: All flags are affected by the result of the operation.

SUI data (8b)

Description: The 8-bit data is subtracted from the contents of the accumulator. The result is stored in the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: D6

Flags: All flags are affected by the result of the operation.

XCHG

Description: The contents of register H and register D are exchanged and the contents of register L and register E are exchanged.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes: EB

Flags: No flags are affected.

XRA R

Description: The contents of the register are Exclusive OR'd with the contents of the accumulator. The results are then stored in the accumulator.

Bytes/M-cycles/T-states: 1/1/4

Hex Codes:

	Register
AF	A
A8	B
A9	C
AA	D
AB	E
AC	H
AD	L

Flags: Z, S and P are affected based upon the operation. CY and AC are reset.

XRA M

Description: The contents of the memory location pointed to by HL are Exclusive OR'd with the contents of the accumulator. The results are stored in the accumulator.

Bytes/M-cycles/T-states: 1/2/7

Hex Codes: AE

Flags: Z, S and P are affected based upon the operation. CY and AC are reset.

XRI data (8b)

Description: The 8 bits of data are Exclusive OR'd with the contents of the accumulator.

Bytes/M-cycles/T-states: 2/2/7

Hex Codes: EE

Flags: Z, S and P are affected based upon the operation. CY and AC are reset.

XTHL

Description: The contents of the L register are exchanged with the stack location pointed to by the stack pointer. The contents of the H register are exchanged with the stack location pointed to by the stack pointer+1. The stack pointer remains unchanged.

Bytes/M-cycles/T-states: 1/5/6

Hex Codes: E3

Flags: No flags are affected.

Appendix **B**

The 8086 Instruction Set

Description	*Instruction code format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
MOV = Move				
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	Data	data if w = 1
Immediate to Register	1 0 1 1 w reg	Data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
=pop				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			

Description	*Instruction code format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
XCHG = Exchange				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Acc.	1 0 0 1 0 reg			
IN =Input from				

Contd...

Contd...

Fixed Port	1 1 1 0 0 1 0 w	Port		
Variable Port	1 1 1 0 1 1 0 w			
OUT = Output				
Fixed Port	1 1 1 0 0 1 1 w	Port		
Variable Port	1 1 1 0 1 1 1 w			
XLAT = Translate Byte to AL				
Implied	1 1 0 1 0 1 1 1			
LEA = Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS = Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES = Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF = Load AH with Flags	1 0 0 1 1 1 1 1			
SAHF = Store AH into Flags	1 0 0 1 1 1 1 0			
PUSHF = Push Flags	1 0 0 1 1 1 0 0			
POPF = Pop Flags	1 0 0 1 1 1 0 1			
ADD = Add				
Reg./Memory with Register to Either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	Data	data if s:w = 01

Description	*Instruction code format*			
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ADC = Add with Carry				
Immediate	1 0 0 0 0 0 s w	mod 0 1 0 r/m	Data	if s: w = 01
Immediate to Accumulator	0 0 0 1 0 1 0 w	Data	data if w = 1	
INC = Increment:				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA = ASCII Adjust for Add	0 0 1 1 0 1 1 1			
DAA = Decimal Adjust for Add	0 0 1 0 0 1 1 1			
SUB = Subtract				
Reg./Memory and Register to Either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	Data	data if s w = 01
Immediate from Accumulator	0 0 1 0 1 1 0 w	Data	data if w = 1	
SSB = Subtract with Borrow				
Reg./Memory and Register to Either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	Data	data if s w = 01
Immediate from Accumulator	0 0 0 1 1 1 w	Data	data if w = 1	

Description	*Instruction code format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP = Compare				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	Data	data if s:w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	0 0 1 1 1 1 1 1			
DAS = Decimal Adjust for Subtract	0 0 1 0 1 1 1 1			
MUL = Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL = Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM = ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV = Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV = Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD = ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW = Convert Byte to Word	1 0 0 1 1 0 0 0			
NOT = Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		

Description	*Instruction Code Format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
SHR = Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR = Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL = Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR = Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL = Rotate Left Through Carry Flag	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR = Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
= Logical AND Reg./Memory and Register to Either Immediate to Register/Memory Immediate to Accumulator	 0 0 1 0 0 0 d w 1 0 0 0 0 0 0 w 0 0 1 0 0 1 0 w	 mod reg r/m mod 1 0 0 r/m Data	 Data data if w = 1	 data if w = 1
TEST = Logical AND And Function to Flags, No Result Register/Memory and Register Immediate Data and Register/Memory Immediate Data and Accumulator	 1 0 0 0 0 1 0 w 1 1 1 1 0 1 1 w 1 0 1 0 1 0 0 w	 mod reg r/m mod 0 0 0 r/m Data	 Data data if w = 1	 data if w = 1
OR = Logical OR Reg./Memory and Register to Either Immediate to Register/Memory Immediate to Accumulator	 0 0 0 0 1 0 d w 1 0 0 0 0 0 0 w 0 0 0 0 1 1 0 w	 mod reg r/m mod 0 0 1 r/m Data	 Data data if w = 1	 data if w = 1

Contd...

Contd...

XOR = Logical XOR Reg./Memory and Register to Either Immediate to Register/Memory Immediate to Accumulator	 0 0 1 1 0 0 d w 1 0 0 0 0 0 0 w 0 0 1 1 0 1 0 w	 mod reg r/m mod 1 1 0 r/m Data	 Data data if w = 1	 data if w = 1
REP = Repeat	1 1 1 1 0 0 1 z			
MOVS = Move Byte/Word string	1 0 1 0 0 1 0 w			

Description	*Instruction Code Format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
LODS = Load Byte/Wd to AL/AX	1 0 1 0 1 1 0 w			
STOS = Store Byte/Wd from AL/A	1 0 1 0 1 0 1 w			
= Call procedure Direct within Segment Indirect within Segment Direct Intersegment Indirect Intersegment	 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1	 disp-low mod 0 1 0 r/m offset-low mod 0 1 1 r/m	 disp-high offset-high 	
JMP = Unconditional Jump Direct within Segment Direct within Segment-Short Indirect within Segment Direct Intersegment Indirect Intersegment	 1 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1	 disp-low Disp mod 1 0 0 r/m offset-low mod 1 0 1 r/m	 disp-high offset-high 	
RET = Return from CALL Within Segment Within Seg Adding Immediate to SP Intersegment Intersegment Adding Immediate to SP	 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 0	 data-low data-low	 data-high data-high	
JE/JZ = Jump on Equal/Zero	0 1 1 1 0 1 0 0	Disp		
JL/JNGE = Jump on Less/Not Greater or equal	0 1 1 1 1 1 0 0	Disp		
JLE/JNG = Jump on Less or Equal/ not greater	0 1 1 1 1 1 1 0	Disp		
JB/JNAE = Jump on Below/Not Above or equal	0 1 1 1 0 0 1 0	Disp		
JBE/JNA = Jump on Below or Equal/ not above	0 1 1 1 0 1 1 0	Disp		
JP/JPE = Jump on Parity/Parity Even	0 1 1 1 1 0 1 0	Disp		
JO = Jump on Overflow	0 1 1 1 0 0 0 0	Disp		
JS = Jump on Sign	0 1 1 1 1 0 0 0	Disp		
JNE/JNZ = Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	Disp		

Description	*Instruction Code Format*			
	7 6 5 4 3 2 1 0	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*	*7 6 5 4 3 2 1 0*
JNB/JAE = Jump on Not Below/Above or equal	0 1 1 1 0 0 1 1	Disp		
JNBE/JA = Jump on Not Below or equal/ above	0 1 1 1 0 1 1 1	Disp		
JNP/JPO = Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	Disp		
JNO = Jump on Not Overflow	0 1 1 1 0 0 0 1	Disp		
JNS = Jump on Not Sign	0 1 1 1 1 0 0 1	Disp		
LOOP = Loop unconditional, count in CX	1 1 1 0 0 0 1 0	Disp		
LOOPZ/LOOPE = Loop While Zero/ Equal count in CX	1 1 1 0 0 0 0 1	Disp		
LOOPNZ/LOOPNE = Loop While Not Zero/Equal count in CX	1 1 1 0 0 0 0 0	Disp		
JCXZ = Jump on CX Zero	1 1 1 0 0 0 1 1	Disp		
INT = Interrupt Type Specified Type 3	 1 1 0 0 1 1 0 1 1 1 0 0 1 1 0 0	 Type		
INTO = Interrupt on Overflow	1 1 0 0 1 1 1 0			
IRET = Interrupt Return	1 1 0 0 1 1 1 1			
CLC = Clear Carry	1 1 1 1 1 0 0 0			
CMC = Complement Carry	1 1 1 1 0 1 0 1			
STC = Set Carry	1 1 1 1 1 0 0 1			
CLD = Clear Direction	1 1 1 1 1 1 0 0			
STD = Set Direction	1 1 1 1 1 1 0 1			
CLI = Clear Interrupt	1 1 1 1 1 0 1 0			
STI = Set Interrupt	1 1 1 1 1 0 1 1			
HLT = Halt	1 1 1 1 0 1 0 0			
WAIT = Wait	1 0 0 1 1 0 1 1			
ESC = Escape (to External Device)	1 1 0 1 1 x x x	mod x x x r/m		
LOCK = Bus Lock prefix	1 1 1 1 0 0 0 0			

*Note
AL =8-bit accumulator AX = 16-bit accumulator
CX = Count register DS = Data segment
ES = Extra segment
W = 0, 8-bit data will be processed
W = 1, 16-bit data will be processed
D = 0, register specified by the REG field is source operand.
D = 1, register specified by the REG field is destination operand.

Reg(3bit) Field	*Registers name*	
	W=0	*W =1*
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	H

Mode	*Explanation*
00	Memory mode, no displacement
01	Memory mode, 8-bit displacement
10	Memory mode, 16-bit displacement
11	Register mode, no displacement

Effective Address Calculation in 8086					
R/M	*MOD 00*	*MOD 01*	*MOD 10*	*MOD 11*	
				W=0	*W=1*
000	(BX)+(SI) DS	(BX)+(SI)+D8 DS	(BX)+(SI)+D16 DS	AL	AX
001	(BX)+(DI) DS	(BX)+(DI)+D8 DS	(BX)+(DI)+D16 DS	CL	CX
010	(BP)+(SI) SS	(BP)+(SI) +D8 SS	(BP)+(SI) +D16 SS	DL	DX
011	(BP)+(DI) SS	(BP)+(DI)+D8 SS	(BP)+(DI)+D16 SS	BL	BX
100	(SI) DS	(SI) +D8 DS	(SI) +D16 DS	AH	SP
101	(DI) DS	(DI) +D8 DS	(DI) +D16 DS	CH	BP
110	D16 DS	(BP)+D8 DS	(BP) +D16 DS	DH	SI
111	(BX) DS	(BX) +D8 DS	(BX) +D16 DS	BH	DI

Appendix C

8051 Instruction Set

INSTRUCTIONS	NO. OF BYTES	HEX CODES	OPERATION	CYCLES	DESCRIPTION
NOP	1	00			No Operation
AJMP code addr	2	01	(PC)← (PC) + 2 (PC10-0) ← page address	2	Absolute Jump
LJMP code addr	3	02	(PC)← addr15-0	2	Long Jump
RR A	1	03	(An)← (An + 1) n = 0 – 6 (A7)← (A0)	1	Rotate Accumulator Right
INC A	1	04	(A)← (A) + 1	1	Increment Accumulator
INC direct	2	05	(direct)←(direct) + 1	1	Increment direct byte
INC @R0	1	06	((Ri))← ((Ri)) + 1	1	Increment direct RAM
INC @R1	1	07		1	
INC R0	1	08	(Rn)← (Rn) + 1	1	Increment register
INC R1	1	09		1	
INC R2	1	0A		1	
INC R3	1	0B		1	
INC R4	1	0C		1	
INC R5	1	0D		1	
INC R6	1	0E		1	
INC R7	1	0F		1	
JBC bit addr, code addr	3	10	(PC)←(PC) + 3 IF (bit) = 1 THEN (bit)← 0 (PC)←(PC) +code	2	Jump if direct Bit is set & clear bit

Contd...

Contd...

ACALL code addr	2	11	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
LCALL code addr	3	12	(PC)← (PC) + 3 (SP)← (SP) + 1 ((SP))←(PC7-0) (SP)←(SP) + 1 ((SP))←(PC15-8)	2	Long Subroutine Call
RRC A	1	13	(An)←(An + 1) n = 0 – 6 (A7)←(C)	1	Rotate Accumulator right through the Carry
DEC direct	2	15	(direct)←(direct) – 1	1	Decrement direct byte
DEC @R0	1	16	((Ri))←((Ri)) - 1	1	Decrement indirect RAM
DEC @R1	1	17		1	
DEC R0	1	18	(Rn)←(Rn) - 1	1	Decrement Register
DEC R1	1	19		1	
DEC R2	1	1A		1	
DEC R3	1	1B		1	
DEC R4	1	1C		1	
DEC R5	1	1D		1	
DEC R6	1	1E		1	
DEC R7	1	1F		1	
JB bit addr, code addr	3	2	(PC)←(PC) + 3 IF (bit) = 1 THEN (PC)←(PC) + code	2	Jump if direct Bit is set
AJMP code addr	2	21	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
RET	1	22	(PC15-8)←((SP)) (SP)←(SP) - 1 (PC7-0)←((SP)) (SP)←(SP) - 1	2	Return from Subroutine
RL A	1	23	(An + 1)←(An) n = 0 – 6 (A0) ←(A7)	1	Rotate Accumulator Left
ADD A,#data	2	24	(A)←(A) + #data	1	Add immediate data to accumulator
ADD A, data addr	2	25	(A)←(A)+(direct)	1	Add direct byte to Accumulator

Contd...

Contd...

ADD A,@R0	1	26	(A)← (A) + ((Ri))	1	Add indirect RAM to Accumulator
ADD A,@R1	1	27		1	
ADD A,R0	1	28	(A)← (A) + (Rn)	1	Add register to Accumulator
ADD A,R1	1	29		1	
ADD A,R2	1	2A		1	
ADD A,R3	1	2B		1	
ADD A,R4	1	2C		1	
ADD A, R5	1	2D		1	
ADD A,R6	1	2E		1	
ADD A,R7	1	2F		1	
JNB bit addr, code addr	3	30	(PC)←(PC) + 3 IF (bit) = 0 THEN (PC)←(PC) + code	2	Jump if direct Bit is Not set
ACALL code addr	2	31	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
RETI	1	32	(PC15-8)← ((SP)) (SP)← (SP) – 1 (PC7-0) ← ((SP)) (SP) ← (SP) - 1	2	Return from interrupt
RLC A	1	33	(An + 1)← (An) n = 0 - 6 (A0) ← (C) (C) ← (A7)	1	Rotate Accumulator Left through the Carry
ADDC A,#data	2	34	(A)←(A) + (C) + #data	1	Add immediate data to accumulator with carry
ADDC A, data addr	2	35	(A)←(A) + (C) + (direct)	1	Add direct byte to accumulator with carry
ADDC A,@R0	1	36	(A)← (A) + (C) + ((Ri))	1	Add indirect byte to accumulator with carry
ADDC A,@R1	1	37		1	
ADDC A,R0	1	38	(A)← (A) + (C) + (Rn)	1	Add register to accumulator with carry
ADDC A,R1	1	39		1	
ADDC A,R2	1	3A		1	
ADDC A,R3	1	3B		1	
ADDC A,R4	1	3C		1	
ADDC A,R5	1	3D		1	
ADDC A,R6	1	3E		1	
ADDC A,R7	1	3F		1	

Contd...

Contd...

JC code addr	2	40	(PC) ←(PC) + 2 IF (C) = 1 THEN (PC) ←(PC) + code	2	Jump if Carry is set
AJMP code addr	2	41	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
ORL direct, A	2	42	(direct)← (direct) ∨ (A)	1	OR Accumulator to direct byte
ORL direct, #data	3	43	(direct)← (direct) ∨ #data	1	OR immediate data to direct byte
ORL A,#data	2	44	(A)← (A) ∨ #data	1	OR immediate data to Accumulator
ORL A, direct	2	45	(A)← (A) ∨ (direct)	1	OR direct byte to Accumulator
ORL A,@R0	1	46	(A)← (A) ∨((Ri))	1	OR indirect RAM to Accumulator
ORL A, @R1	1	47		1	
ORL A,R0	1	48	(A)← (A) ∨ (Rn)	1	OR register to Accumulator
ORL A,R1	1	49		1	
ORL A,R2	1	4A		1	
ORL A,R3	1	4B		1	
ORL A,R4	1	4C		1	
ORL A,R5	1	4D		1	
ORL A,R6	1	4E		1	
ORL A,R7	1	4F		1	
JNC code addr	2	50	(PC)← (PC) + 2 IF (C) = 0 THEN (PC)←(PC) + code	2	Jump if Carry not set
ACALL code addr	2	51	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
ANL direct, A	2	52	(direct)← (direct) V (A)	1	AND Accumulator to direct byte
ANL direct, #data	3	53	(direct) ←(direct) ∧ #data	1	AND immediate data to direct byte
ANL A,# data	2	54	(A)← (A) ∧ #data	1	AND immediate data to Accumulator

Contd...

Contd...

ANL A, direct	2	55	(A) ←(A) ∧ (direct)	1	AND direct byte to Accumulator
ANL A, @R0	1	56	(A)← (A) ∧ ((Ri))	1	AND indirect RAM to Accumulator
ANL A, @R1	1	57		1	
ANL A, R0	1	58	(A)← (A) ∧ (Rn)	1	AND Register to Accumulator
ANL A, R1	1	59		1	
ANL A,R2	1	5A		1	
ANL A,R3	1	5B		1	
ANL A,R4	1	5C		1	
ANL A,R5	1	5D		1	
ANL A,R6	1	5E		1	
ANL A,R7	1	5F		1	
JZ code addr	2	60	(PC)← (PC) + 2 IF (A) = 0 THEN (PC)←(PC) + code	2	Jump if Accumulator is Zero
AJMP code addr	2	61	(PC)←(PC) + 2 (PC10-0)←page	2	Absolute Jump
XRL data addr, A	2	62	(direct)← (direct) V (A)	1	
XRL data addr, #data	3	63	(direct) ←(direct) V #data	1	
XRL A, #data	2	64	(A)← (A) V #data	1	
XRL A, direct	2	65	(A) ←(A) V (direct)	1	Exclusive-OR direct byte to Accumulator
XRL A, @R0	1	66	(A)← (A) V ((Ri))	1	Exclusive-OR indirect RAM to Accumulator
XRL A,@R1	1	67		1	
XRL A,R0	1	68	(A)← (A) V (Rn)	1	Exclusive-OR register to Accumulator
XRL A,R1	1	69		1	
XRL A,R2	1	6A		1	
XRL A,R3	1	6B		1	
XRL A,R4	1	6C		1	
XRL A,R5	1	6D		1	
XRL A,R6	1	6E		1	
XRL A,R7	1	6F		1	
JNZ code addr	2	70	(PC)← (PC) + 2 IF (A) ≠ 0 THEN (PC)← (PC)+code	2	Jump if Accumulator is Not Zero

Contd...

Contd...

ACALL code addr	2	71	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
ORL C, bit addr	2	72	(C)← (C) ∨ (bit)	2	OR direct bit to Carry
JMP @A+DPTR	1	73	(PC) ←(A) + (DPTR)	2	Jump indirect relative to the DPTR
MOV A, #data	2	74	(A)← #data	1	Move immediate data to Accumulator
MOV direct, #data	3	75	(direct)← #data	1	Move immediate data to direct byte
MOV @R0, #data	2	76	((Ri))← #data	1	Move immediate data to indirect RAM
MOV @R1, #data	2	77		1	
MOV R0, #data	2	78	(Rn)← #data	1	Move immediate data to register
MOV R1, #data	2	79		1	
MOV R2, #data	2	7A		1	
MOV R3, #data	2	7B	(Rn)← #data	1	Move immediate data to register
MOV R4, #data	2	7C		1	
MOV R5, #data	2	7D		1	
MOV R6, #data	2	7E		1	
MOV R7, #data	2	7F		1	
SJMP code addr	2	80	(PC)← (PC) + 2 (PC)← (PC) + code	2	Short Jump (relative addr)
AJMP code addr	2	81	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
ANL C, bit addr	2	82	(C) ← (C) ∧ (bit)	2	AND direct bit to CARRY
MOVC A,@A+PC	1	83	(A) ← ((A) + (PC))	2	Move Code byte relative to PC to Acc
DIV AB	1	84	(A)15-8 ← (A)/(B) (B)7-0	4	Divide A by B
MOV direct, direct	3	85	(direct) ← (direct)	2	Move direct byte to direct
MOV direct, @R0	2	86	(direct) ← ((Ri))	2	Move indirect RAM to direct byte
MOV direct, @R1	2	87		2	

Contd...

Contd...

MOV direct,R0	2	88	(direct) ← (Rn)	2	Move register to direct byte
MOV direct,R1	2	89		2	
MOV direct,R2	2	8A		2	
MOV direct,R3	2	8B		2	
MOV direct,R4	2	8C		2	
MOV direct,R5	2	8D		2	
MOV direct,R6	2	8E		2	
MOV direct,R7	2	8F		2	
MOV DPTR, #data	3	90	(DPTR)← #data15-0 DPH← DPL ← #data15-8 ← #data7-0	2	Load Data Pointer with a 16-bit constant
ACALL code addr	2	91	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
MOV bit addr, C	2	92	(bit) ←(C)	2	Move Carry to direct bit
MOVC A,@A+DPTR	1	93	(A)← ((A) + (DPTR))	2	Move Code byte relative to DPTR to Acc
SUBB A, #data	2	94	(A)← (A) - (C) - #data	1	Subtract immediate data from Acc with borrow
SUBB A, direct	2	95	(A) ←(A) - (C) - (direct)	1	Subtract direct byte from Acc with borrow
SUBB A,@R0	1	96	(A)← (A) - (C) -((Ri))	1	Subtract indirect RAM from ACC with borrow
SUBB A,@R1	1	97		1	
SUBB A,R0	1	98	(A)← (A)-(C)-(Rn)	1	Subtract Register from Acc with borrow
SUBB A,R1	1	99		1	
SUBB A,R2	1	9A		1	
SUBB A,R3	1	9B		1	
SUBB A,R4	1	9C		1	
SUBB A,R5	1	9D		1	
SUBB A,R6	1	9E		1	
SUBB A,R7	1	9F		1	
ORL C/,bit addr	2	A0	(C) ← (C) ∨ (bit)	2	OR complement of direct bit to Carry
AJMP code addr	2	A1	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
MOV C/,bit addr	2	A2	(C) ← (bit)	1	Move direct byte to carry

Contd...

Contd...

INC DPTR	1	A3	(DPTR)←(DPTR) + 1	2	Increment Data Pointer
MUL AB	1	A4	(A)7-0←(A) X (B) (B)15-8	4	Multiply A & B
reserved		A5			
MOV @R0,direct	2	A6	((Ri))← (direct)	2	Move direct byte to indirect RAM
MOV @R1direct	2	A7		2	
MOV R0,direct	2	A8	(Rn) ← (direct)	2	Move direct byte to register
MOV R1, direct	2	A9		2	
MOV R2,direct	2	AA		2	
MOV R3,direct	2	AB		2	
MOV R4 direct	2	AC		2	
MOV R5,direct	2	AD		2	
MOV R6,direct	2	AE		2	
MOV R7,direct	2	AF		2	
ANL C/,bit addr	2	B0	(C) ← (C) ∧ (bit)	2	
ACALL code addr	2	B1	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
CPL bit addr	2	B2		1	AND complement of direct bit to Carry
CPL C	1	B3		1	Complement Carry
CJNE A, #data, code addr	3	B4	(PC) ← (PC) + 3 IF (A) < > data THEN (PC) ← (PC) + relative offset IF (A) < data THEN (C) ← 1 ELSE (C) ← 0	2	Compare immediate to Acc and Jump if Not Equal
CJNE A, direct, code addr	3	B5	(PC) ← (PC) + 3 IF (A) < > (direct) THEN (PC) ← (PC) + relative offset IF (A) < (direct) THEN (C) ← 1 ELSE (C) ← 0	2	Compare direct byte to Acc and Jump if Not Equal

Contd...

Contd...

CJNE @R0, #data, code addr	3	B6	(PC) ← (PC) + 3
			IF ((Ri)) < > data
			THEN
			(PC) ← (PC) + relative offset
			IF ((Ri)) < data
			THEN
			(C) ← 1
			ELSE
			(C) ← 0
				2	Compare immediate to indirect and Jump if Not Equal

CJNE @R1, #data, code addr	3	B7		2	
AJMP code addr	2	C1	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
CLR bit addr	1	C2	(bit) ← 0	1	Clear direct bit
CLR C	1	C3	(C) ← 0	1	Clear Carry
SWAP A	2	C4	(A3-0) D (A7-4)	1	Swap nibbles within the Accumulator
XCH A, direct	1	C5	(A) D (direct)	1	Exchange direct byte with Accumulator
XCH A,@R0	1	C6	(A) D ((Ri))	1	Exchange indirect RAM with Acc.
XCH A,@R1	1	C7		1	
XCH A,R0	1	C8	(A) D ((Rn))	1	Exchange register with Accumulator
XCH A,R1	1	C9		1	
XCH A,R2	1	CA		1	
XCH A,R3	1	CB		1	
XCH A,R4	1	CC		1	
XCH A,R5	1	CD		1	
XCH A,R6	1	CE		1	
XCH A,R7	1	CF		1	
POP direct	2	D0	(direct) ← ((SP)) (SP) ← (SP) - 1	2	Pop direct byte from stack
ACALL code addr	2	D1	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
SETB bit addr	2	D2	(bit) ← 1	1	Clear direct bit
SETB C	1	D3	(C) ← 1	1	Set Carry

Contd...

Contd...

DAA	1	D4	IF [[(A3-0) > 9] ∨ [(AC) = 1]] THEN (A3-0) ← (A3-0)+6 AND IF [[(A7-4) > 9] ∨ [(C) = 1]] THEN (A7-4) ← (A7-4)+6	1	Decimal Adjust Accumulator
DJNZ direct, code addr	3	D5	(PC) ←(PC) + 2 (direct)←(direct)-1 IF (direct) > 0 or (direct) < 0 THEN (PC) ← (PC) +code		Decrement direct byte and Jump if Not Zero
XCHD A, @R0	1	D6	(A3-0) D ((Ri3-0))	1	Exchange low-order Digit indirect RAM with Acc
XCHD A, @R1	1	D7		1	
DJNZ R0,direct	2	D8	(PC) ← (PC) + 2 (Rn)← (Rn) – 1 IF (Rn) >0 or (Rn)<0 THEN (PC) ← (PC) + data	2	Decrement register and Jump if Not Zero
DJNZ R1,direct	2	D9		2	
DJNZ R2,direct	2	DA		2	
DJNZ R3,direct	2	DB		2	
DJNZ R4,direct	2	DC		2	
DJNZ R5,direct	2	DD		2	
DJNZ R6,direct	2	DE		2	
DJNZ R7,direct	2	DF		2	
MOVX A,@DPTR	1	E0	(A) ← ((DPTR))	2	Move Exernal RAM (16-bit addr) to Acc
AJMP code addr	2	E1	(PC)←(PC) + 2 (PC10-0)←page address	2	Absolute Jump
MOVX A, @R0	1	E2	(A) ← ((Ri))	2	Move External RAM (8-bit addr) to Acc
MOVX A, @R1	1	E3		2	
CLR A	1	E4	(A) ← 0	1	Clear Accumulator
MOV A, direct	2	E5	(A) ← (direct)	1	Move direct byte to Accumulator
MOV A, @R0	1	E6	(A) ← ((Ri))	1	Move indirect RAM to Accumulator
MOV A, @R1	1	E7		1	

Contd...

Contd...

MOV A, R0	1	E8	(A) ← (Rn)	1	Move register to Accumulator
MOV A, R1	1	E9		1	
MOV A, R2	1	EA		1	
MOV A, R3	1	EB		1	
MOV A, R4	1	EC		1	
MOV A, R5	1	ED		1	
MOV A, R6	1	EE		1	
MOV A, R7	1	EF		1	
MOVX @DPTR,A	1	F0	(DPTR) ← (A)	2	Move Acc to External RAM (16-bit addr)
ACALL code addr	2	F1	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP))← (PC7-0) (SP)← (SP) + 1 ((SP))← (PC15-8) (PC10-0)← page address	2	Absolute subroutine call
MOVX @R0, A	1	F2	((Ri)) ← (A)	2	Move Acc to External RAM (8-bit addr)
MOVX @R1, A	1	F3		2	
CPL A	1	F4		1	Complement Accumulator
MOV direct, A	2	F5	(direct)← (A)	1	Move Accumulator to direct byte
MOV @R0, A	1	F6	((Ri)) ← (A)	1	Move Accumulator t0 indirect RAM
MOV @R1, A	1	F7		1	
MOV R0, A	1	F8	(Rn)← (A)	1	Move Accumulator to register
MOV R1, A	1	F9		1	
MOV R2, A	1	FA		1	
MOV R3, A	1	FB		1	
MOV R4, A	1	FC		1	
MOV R5, A	1	FD		1	
MOV R6, A	1	FE		1	
MOV R7, A	1	FF		1	

Appendix D

C Programming for 8051 using KEIL IDE

The C source code is a high-level language, meaning that it is far from being at the base level of the machine language that can be executed by a processor. This machine language is basically just zero's and one's and is written in Hexadecimal format, that's why they are called HEX files.

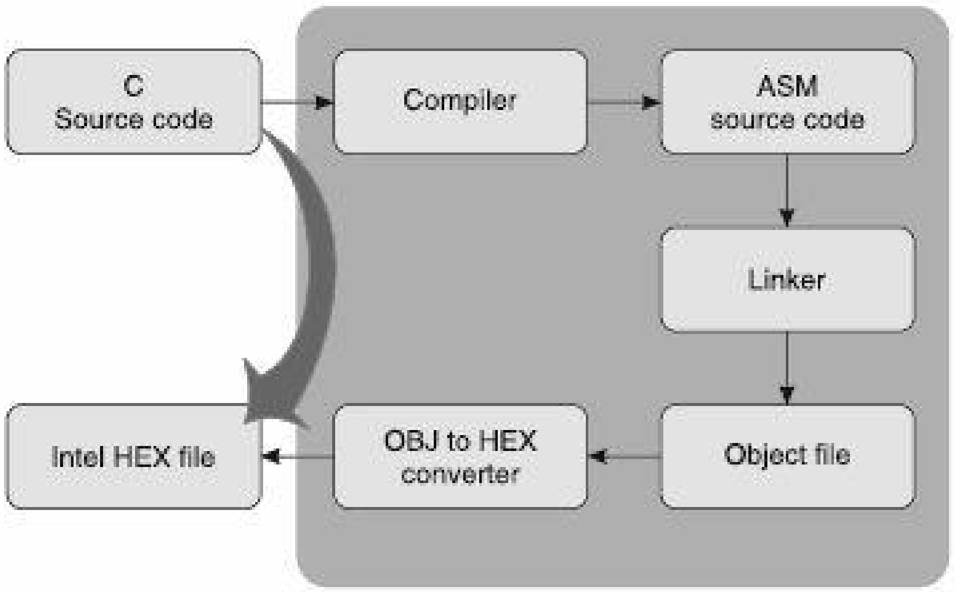


Fig. D.1

There are several types of HEX files. We are going to produce machine code in the INTEL HEX-80 format, since this is the output of the KEIL IDE that we are going to use. Figure D1 shows that to convert a C program to machine language, it takes several steps depending on the tool you are using, however, the main idea is to produce a HEX file at the end. This HEX file will be then used by the 'burner' to write every byte of data at the appropriate place in the EEPROM of the 89S52.

Using the KEIL environment

KEIL uVision is the name of a software dedicated to the development and testing of a family of microcontrollers based on 8051 technology, like the 89S52 which we are going to use along this tutorial. You can can download an evaluation version of KEIL at their website: http://www.keil.com/c51/. Most versions share merely the same interface, this tutorial uses KEIL C51 uVision 3 with the C51 compiler v8.05a.

To create a project, write and test the previous example source code, follow the following steps:

- Open KEIL and start a new project:

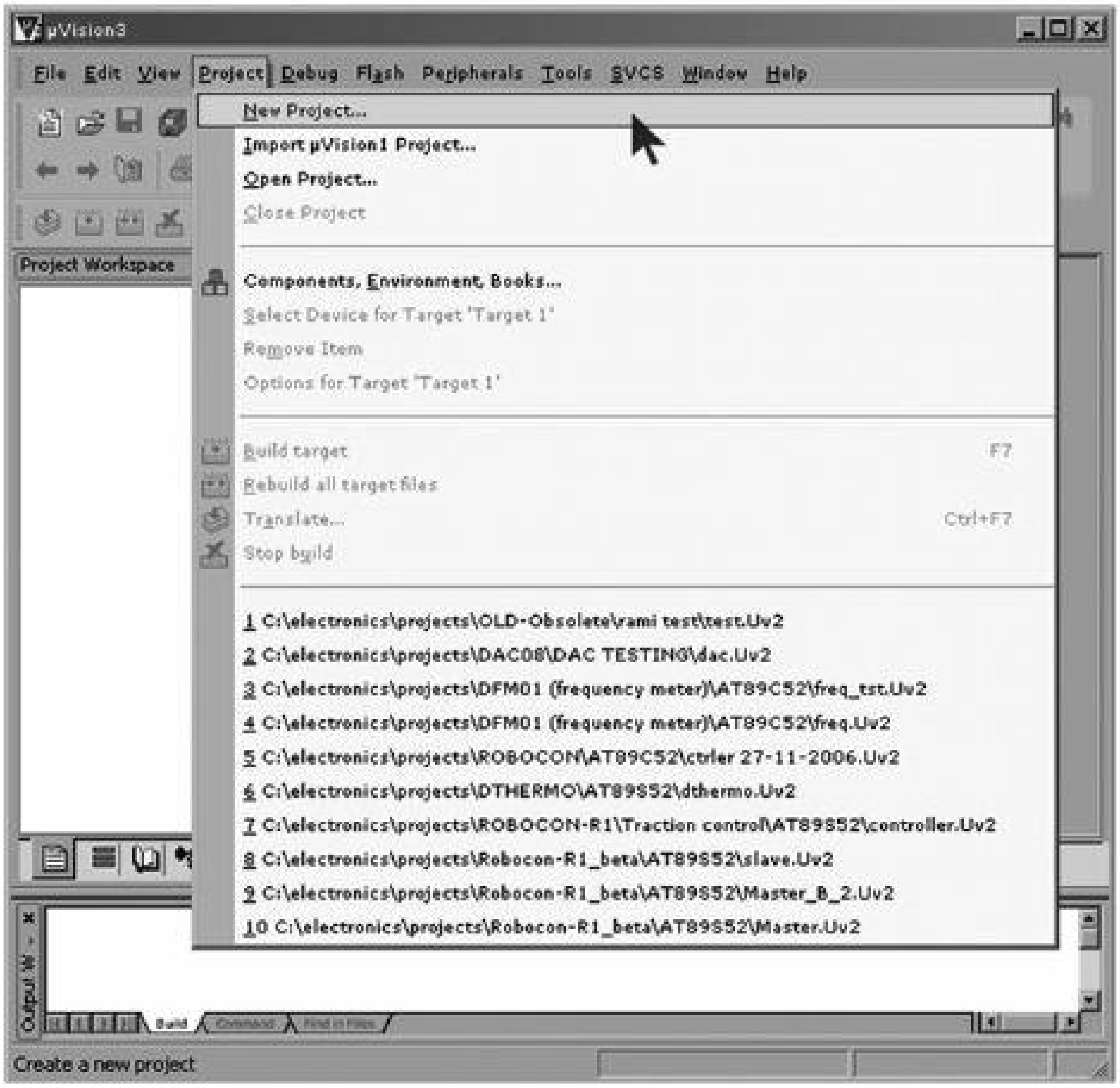
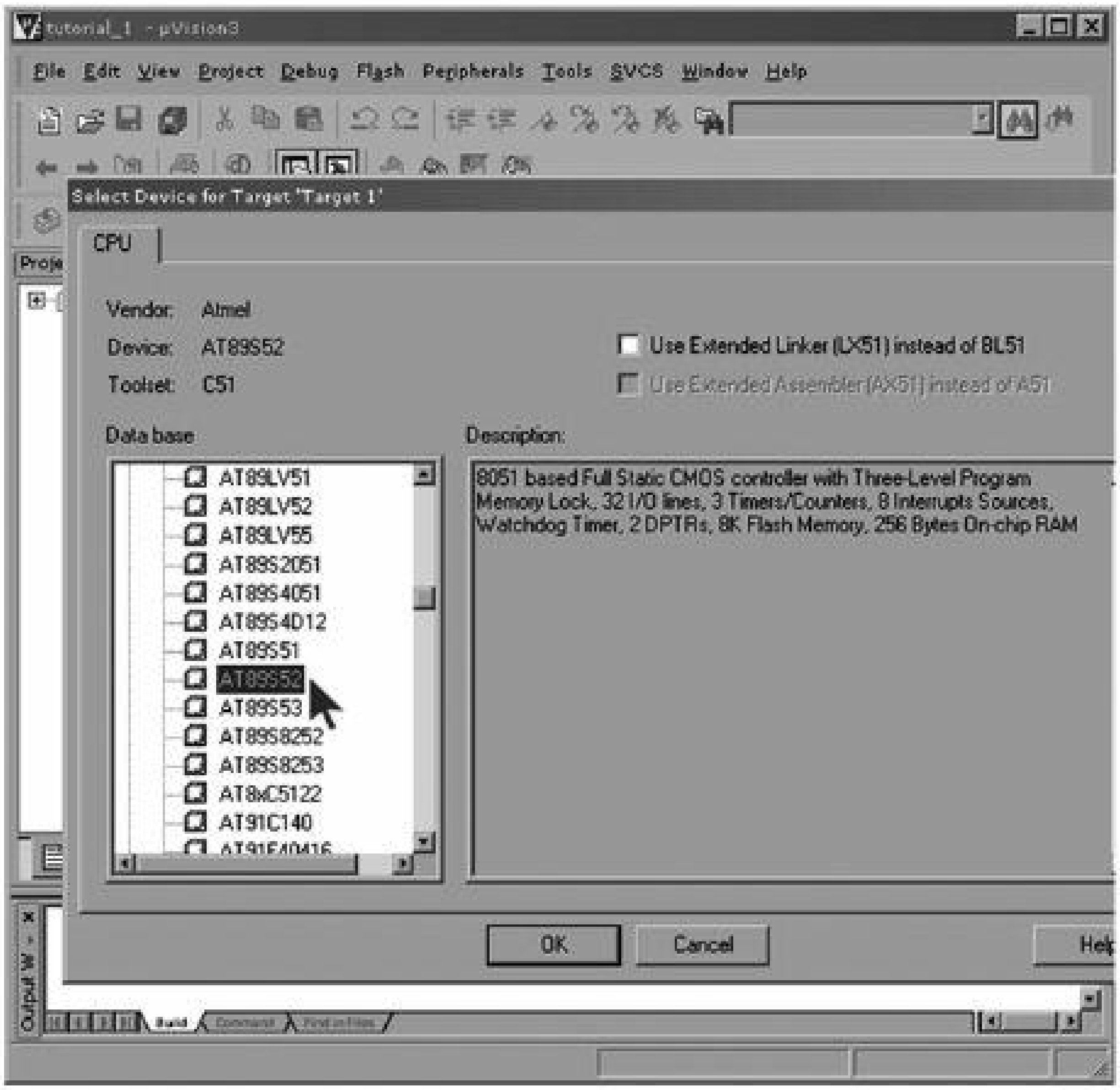

µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New Project...
Import µVision1 Project...
Open Project...
Close Project
Components, Environment, Books...
Select Device for Target 'Target 1'
Remove Item
Options for Target 'Target 1'
Build target F7
Rebuild all target files
Translate... Ctrl+F7
Stop build
1 C:\electronics\projects\OLD-Obsolete\vami test\test.Uv2
2 C:\electronics\projects\DAC08\DAC TESTING\dac.Uv2
3 C:\electronics\projects\DFM01 (frequency meter)\AT89C52\freq_tst.Uv2
4 C:\electronics\projects\DFM01 (frequency meter)\AT89C52\freq.Uv2
5 C:\electronics\projects\ROBOCON\AT89C52\ctrler 27-11-2006.Uv2
6 C:\electronics\projects\DTHERMO\AT89S52\dthermo.Uv2
7 C:\electronics\projects\ROBOCON-R1\Traction control\AT89S52\controller.Uv2
8 C:\electronics\projects\Robocon-R1_beta\AT89S52\slave.Uv2
9 C:\electronics\projects\Robocon-R1_beta\AT89S52\Master_B_2.Uv2
10 C:\electronics\projects\Robocon-R1_beta\AT89S52\Master.Uv2
Project Workspace
Output W
Build
Command
Find in Files
Create a new project

Fig. D2

- You will be prompted to choose a name for your new project, create a separate folder where all the files of your project will be stored, chose a name and click save. The following window will appear, where you will be asked to select a device for Target 'Target 1':


tutorial_1 - µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Select Device for Target 'Target 1'
CPU
Data base
Description:
OK
Cancel
Help
Build
Command
Find in Files
Output W...

Vendor: Atmel
Device: AT89S52
Toolset: C51

☐ Use Extended Linker (LX51) instead of BL51
☐ Use Extended Assembler (AX51) instead of A51

AT89LV51
AT89LV52
AT89LV55
AT89S2051
AT89S4051
AT89S4D12
AT89S51
AT89S52
AT89S53
AT89S8252
AT89S8253
AT8xC5122
AT91C140
AT91F40416

8051 based Full Static CMOS controller with Three-Level Program Memory Lock, 32 I/O lines, 3 Timers/Counters, 8 Interrupts Sources, Watchdog Timer, 2 DPTRs, 8K Flash Memory, 256 Bytes On-chip RAM

Fig. D3

- From the list at the left, seek for the brand name ***ATMEL***, then under ATMEL, select ***AT89S52***. You will notice that a brief description of the device appears on the right. Leave the two upper check boxes unchecked and click OK. The AT89S52 will be called your 'Target device', which is the final destination of your source code. You will be asked whether to '***copy standard 8051 startup code***' **click No**.
- Click File, New, and something similar to the following window should appear. The box named 'Text1' is where your code should be written later.
- Now you have to click 'File, Save as' and choose a file name for your source code ending with the letter '.c'. You can name is 'code.c' for example, and click save. Then you have to add this file to your project work space at the left as shown in the following screen shot:

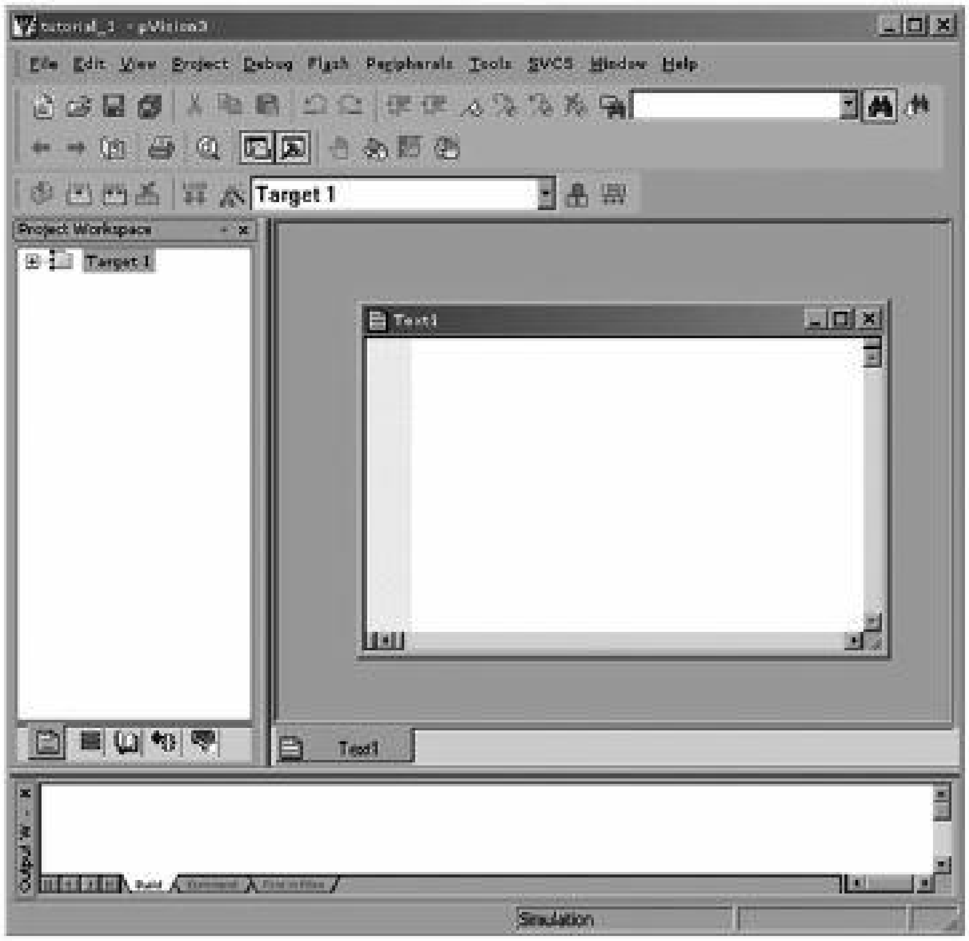

tutorial_1 - µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Text1
Text1
Build Command Find in Files
Simulation

Fig. D4

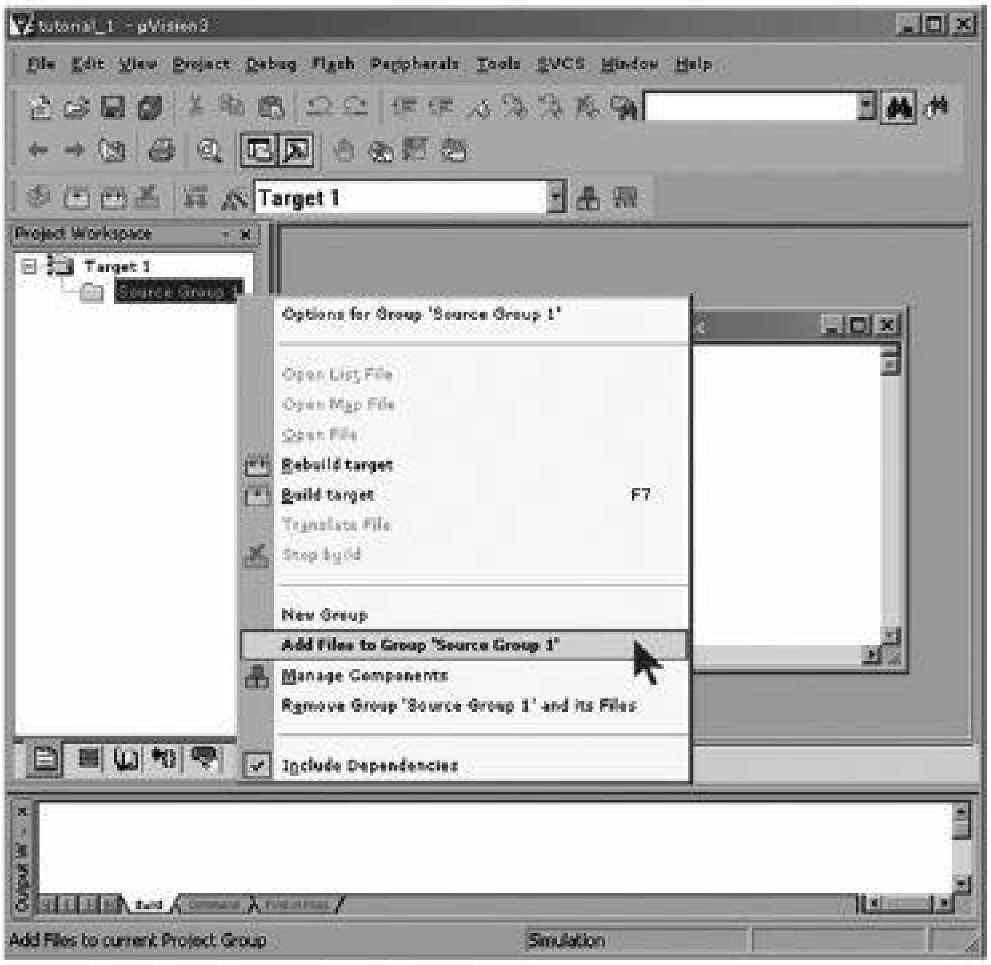
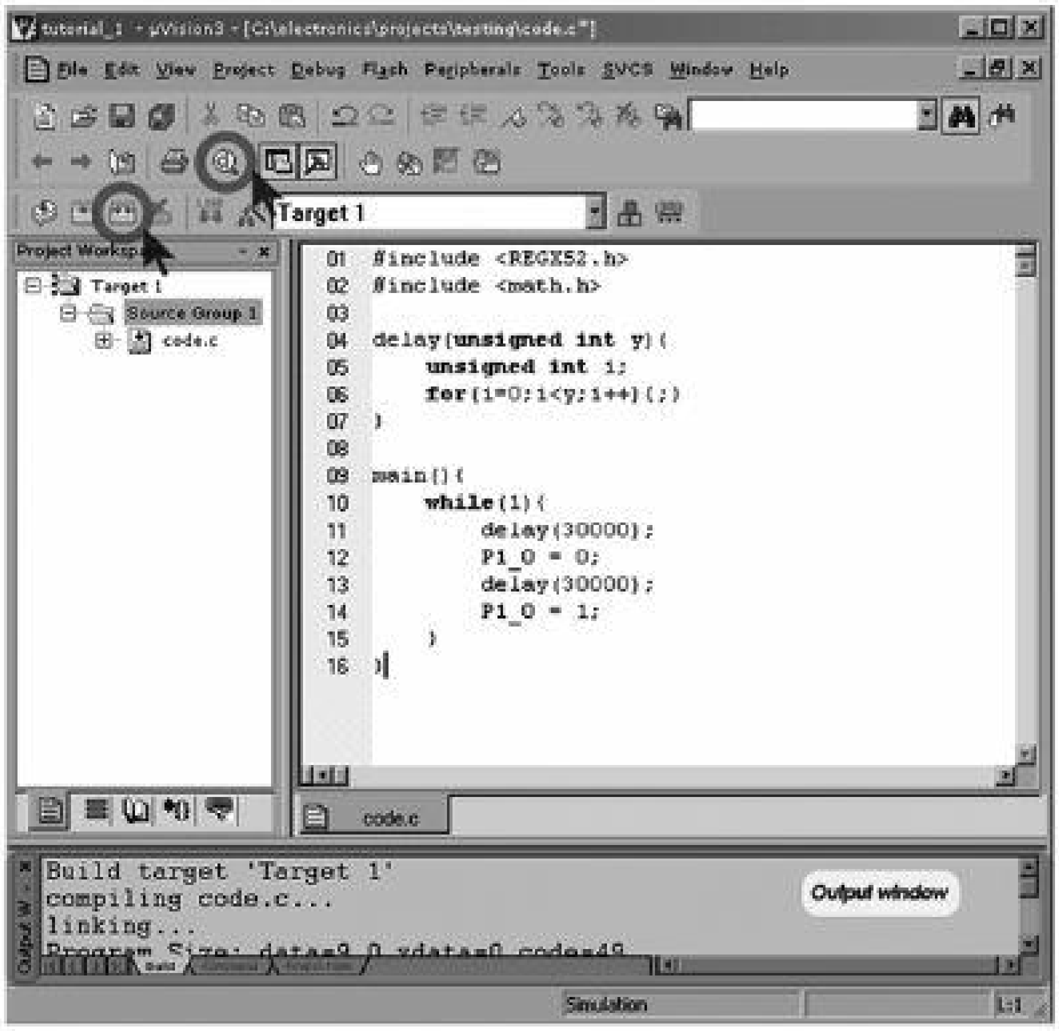

tutorial_1 - µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
Options for Group 'Source Group 1'
Open List File
Open Map File
Open File
Rebuild target
Build target F7
Translate File
Stop build
New Group
Add Files to Group 'Source Group 1'
Manage Components
Remove Group 'Source Group 1' and its Files
Include Dependencies
Add Files to current Project Group
Simulation

Fig. D5

- After right-clicking on '***source group 1***', click on '***Add files to group…***', then you will be prompted to browse the file to add to 'source group 1', chose the file that you just saved, eventually 'code.c' and add it to the source group. You will notice that the file is added to the project tree at the left.
- In some versions of this software you have to turn ON manually the option to generate HEX files. Make sure it is turned ON, by right-clicking on **target 1**, **Options for target 'target 1'**, then under the '**output**' tab, by checking the box '**generate HEX file**'. This step is very important as the HEX file is the compiled output of your project that is going to be transferred to the microcontroller.
- You can then start to write the source code in the window titled 'code.c' then before testing your source code, you have to compile your source code, and correct eventual syntax errors. In KEIL IDE, this step is called 'rebuild all targets' and has this icon: .


tutorial_1 - µVision3 - [C:\electronics\projects\testing\code.c*]
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
code.c
code.c
Output window
Simulation
L:1

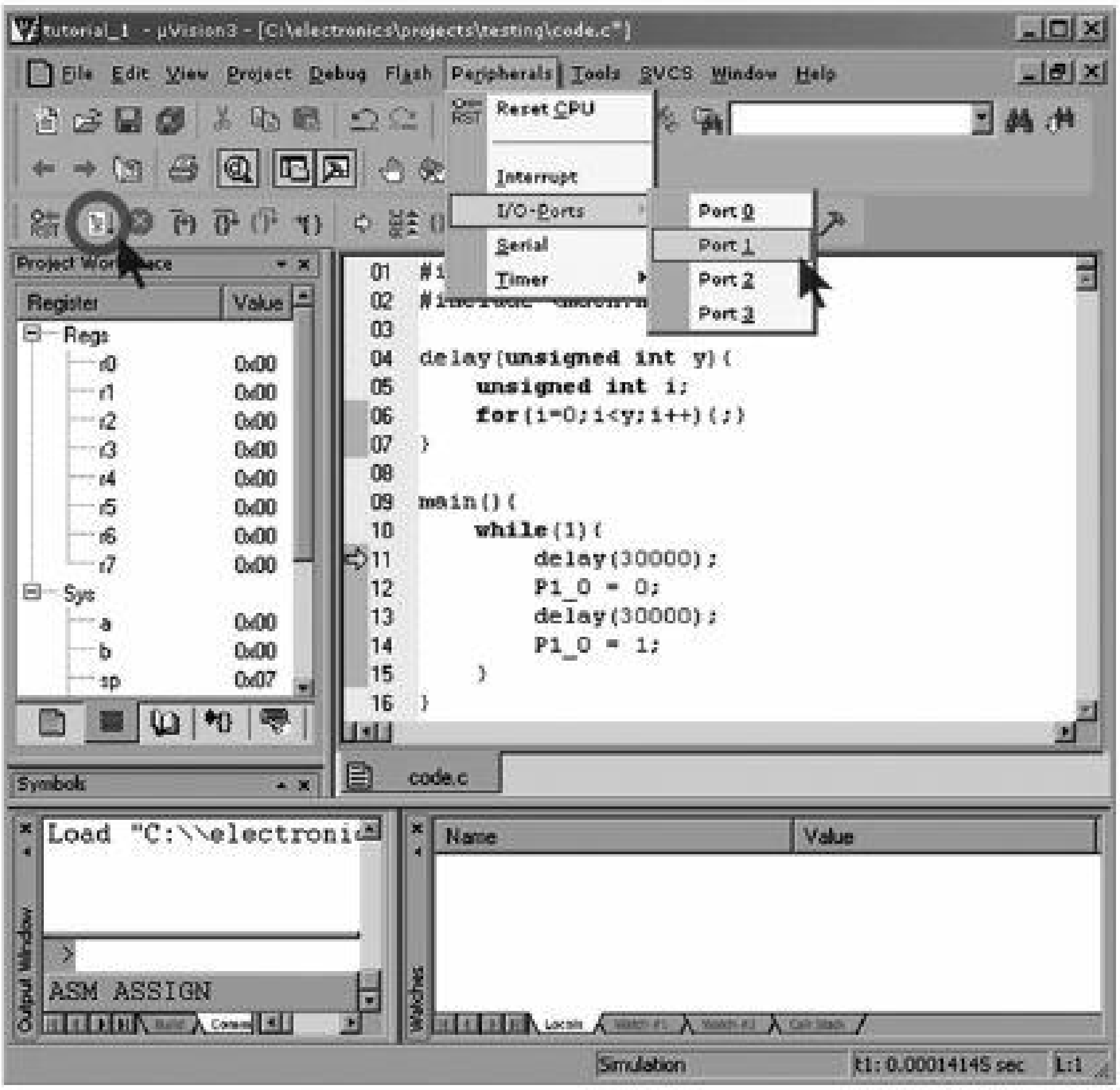
```
#include <REGX52.h>
#include <math.h>

delay(unsigned int y){
    unsigned int i;
    for(i=0;i<y;i++){;}
}

main(){
    while(1){
        delay(30000);
        P1_0 = 0;
        delay(30000);
        P1_0 = 1;
    }
}
```

```
Build target 'Target 1'
compiling code.c...
linking...
Program Size: data=9.0 xdata=0 code=49
```

Fig. D6

- You can use the output window to track eventual syntax errors, but also to check the FLASH memory occupied by the program (code = 49) as well as the registers occupied in the RAM (data = 9). If after rebuilding the targets, the 'output window' shows that there is **0 error**, then you are ready to test the performance of your code. In KEIL, like in most development environment, this step is called debugging, and has this icon: .

After clicking on the debug icon, you will notice that some part of the user interface will change, some new icons will appear, like the run icon circled in the following figure:



```
#i
#include
delay(unsigned int y){
    unsigned int i;
    for(i=0;i<y;i++){;}
}

main(){
    while(1){
        delay(30000);
        P1_0 = 0;
        delay(30000);
        P1_0 = 1;
    }
}
```

```
Load "C:\\electroni
>
ASM ASSIGN
```

Fig. D7

- You can click on the 'Run' icon and the execution of the program will start. In our example, you can see the behavior of the pin 0 or port one, but clicking on 'peripherals, I/O ports, Port 1'. You can always stop the execution of the program by clicking on the stop button ( ) and you can simulate a reset by clicking on the 'reset' button .
- You can also control the execution of the program using the following icons: which allows you to follow the execution step by step. Then, when you're finished with the debugging, you can always return to the programming interface by clicking again on the debug button ( ).
- There are many other features to discover in the KEIL IDE. You will easily discover them in the first couple of hours of practice, and the more important of them will be presented along the rest of this tutorial.

Variables and constants

Variables

One of the most basic concepts of programming is to handle variables. knowing the exact type and size of a variable is a very important issue for microcontroller programmers, because the RAM is usually limited is size. There are two main design considerations to be taken in account when choosing the variables types: the occupied space in ram and the processing speed. Logically, a variable that occupies a big number of registers in RAM will be more slowly processed than a small variable that fits on a single register.

For you to choose the right variable type for each one of your applications, you will have to refer to the following table:

Data Type	Bits	Bytes	Value Range
bit	1	–	0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	±1.175494E-38 to ±3.402823E+38

This table shows the number of bits and bytes occupied by each types of variables, noting that each byte will fit into a register. You will notice that most variables can be either 'signed' or unsigned 'unsigned', and the major difference between the two types is the range, but both will occupy the same exact space in memory.

The names of the variables shown in the table are the same that are going to be used in the program for variables declarations. Note that in C programming language, any variable have to be declared to be used. Declaring a variable, will attribute a specific location in the RAM or FLASH memory to that variable. The size of that location will depend on the type of the variable that have been declared.

To understand the difference between those types, consider the following example source code where we start by declaring three 'unsigned char' variables, and one 'signed char' and then perform some simple operations:

```
unsigned char a,b,c
signed char d;
a =100;
b =200;
c = a - b;
d = a - b;
```

In that program the values of 'c' will be equal to '155'! and not '-100' as you though, because the variable 'c' is an unsigned type, and when the value to be stored in a variable is bigger than the maximum value range of this variable, it overflows and rolls back to the other limit. Back to our example, the program is trying to store '-100' in 'c', but since 'c' is unsigned, its range of values is from '0 to 255' so, trying to store a value below zero, will cause the variable to overflow, and the compiler will subtract the '-100' from the other limit plus 1, from '255 + 1' giving '156'. We add 1 to the range because the overflow and roll back operation from 0 to 255 counts for the subtraction of one bit. On the other hand, the value of 'd' will be equal to '-100' as expected, because it is a 'signed' variable. Generally, we try to avoid storing value that are out of range, because sometimes, even if the compiler doesn't halt on that error, the results can be sometimes totally unexpected.

Note that in the C programming language, any code line is ended with a semicolon ';', except for the lines ending with brackets '{' '}'.

Like in any programming language, the concept of a variables 'array' can also be used for microcontrollers programming. An array is like a table or a group of variables of the same type, each one can be called by a specific number, for example an array can be declared this way:

```
char display[10];
This will create a group of 10 variables. Each one of them is accessible by its number, example:
display[0]=100;
display[3]=60;
display[1]= display[0]- display[3];
```

Where 'display[1]' will be equal to '40'. Note that 'display' contains 10 different variables, numbered from 0 to 9. In that previous example, according to the variable declaration, there is not such variable location as 'display[10]', and using it will cause an error in the compiler.

Constants

Sometimes, you want to store a very large amount of constant values, that wouldn't fit in the RAM or simply would take too much space. you can store this DATA in the FLASH memory reserved for the code, but it won't be editable, once the program is burned on your chip. The advantage of this technique is that it can be used to store a huge amount of variables, noting that the FLASH memory of the 89S52 is 8K bytes, 32 times bigger than the RAM memory. It is, however, your responsibility to distribute this memory between your program and your DATA.

To specify that a variable is to be stored in the FLASH memory, we use exactly the same variable types names but we add the prefix 'code' before it. Example:

```
codeunsignedchar message[500];
```

This line would cause this huge array to be stored in the FLASH memory. This can be interesting for displaying messages on an LCD screen.

To access the pins and the ports through programming, there are a number of predefined variables (defined in the header file, as you shall see later) that dramatically simplifies that task. There are four ports, Port 0 to Port 3, each one of them can be accessed using the ***char*** variables **P0**, **P1**, **P2** and **P3** respectively. In those char types variables, each one of the 8 bits represents a pin on the port. Additionally, you can access a single pin of a port using the bit type variables **PX_0** to **PX_7**, where X takes a value between 0 and 3, depending on the port being accessed. For example P1_3 is the pin number 3 of port 1.

You can also define your own names, using the '#define' directive. Note that this is a compiler directive, meaning that the compiler will use this directive to read and understand the code, but it is not a statement or command that can be translated to machine language. For example, you can define the following:

```
#define LED1 P1_0
```

With the definition above, the compiler will replace every occurrence of **LED1** by **P1_0**. This makes your code much more easier to read, especially when the new names you give make more sense.

You could also define a numeric constant value like this:

```
#define led_on_time 184
```

Then, each time you write led_on_time, it will be replaced by 184. Note that this is not a variable and accordingly, you cannot write something like:

```
led_on_time=100;//That's wrong, you cannot change a constant's value in code.
```

The utility of using defined constants, appears when you want to adjust some delays in your code, or some constant variables that are reused many times within the code: With a predefined constant, you only change it's value once, and it's applied to the whole code. that's for sure apart from the fact that a word **like led_on_time** is much more comprehensive than simply **'184'**!

Along this tutorial you will see how port names, and special function registers are used exactly as variables, to control input/output operations and other features of the microcontroller like timers, counters and interrupts.

Mathematical & logic operations

Now that you know how to declare variables, it is time to know how to handle them in your program using mathematical and logic operations.

Mathematical operations

The most basic concept about mathematical operations in programming languages, is the '=' operator which is used to store the content of the expression at its right, into the variable at its left. For example, the following code will store the value of 'b' into 'a' :

```
a = b;
```

And subsequently, the following expression is totally invalid:

```
5= b;
```

Since 5 is a constant, trying to store the content of 'b' in it will cause an error.

You can then perform all kinds of mathematical operations, using the operators '+','-','*' and '/'. You can also use brackets '( )' when needed. Example:

```
a =(5*b)+((a/b)*(a+b));
```

If you include 'math.h' header file, you will be able to use more advanced functions in your equations like Sin, Cos and Tan trigonometric functions, absolute values and logarithmic calculations like in the following example:

```
a =(c*cos(b))+sin(b);
```

To be able to successfully use those functions in your programs, you have to know the type of variables that those functions take as parameter and return as a result. For example a Cosine function takes an angle in radians whose value is a float number between -65535 and 65535 and it will return a float value as a result. You can usually know those data types from the 'math.h' file itself, for example, the cosine function, like all the others is declared in the top of the math header file, and you can read the line:

```
extern float cos (float val);
```

From this line you can deduce that the 'cos' function returns a float data type, and takes as a parameter a float too (the parameter is always between brackets). Using the same technique, you can easily know how to deal with the rest of the functions of the math header file. The following table shows a short description of those functions:

Function	*Description*
char cabs (char val);	Return an the absolute value of a *char* variable.
int abs (intval);	Return an the absolute value of a *int* variable.
long labs (long val);	Return an the absolute value of a *long* variable.
float fabs (float val);	Return an the absolute value of a *float* variable.
float sqrt (float val);	Returns the square root of a float variable.
float exp (float val);	Returns the value of the Euler number 'e' to the power of *val*
float log (float val);	Returns the natural logarithm of *val*
float log10 (float val);	Returns the common logarithm of *val*

Contd...

Contd...

float sin (float val);	A set of standard trigonometric functions. They all take angles measured in radians whose valuehave to be between -65535 and 65535.
float cos (float val);	
float tan (float val);	
float asin (float val);	
float acos (float val);	
float atan (float val);	
float sinh (float val);	
float cosh (float val);	
float tanh (float val);	
float atan2 (float y, float x);	This function calculates the arc tan of the ratio y / x, using the signs of both x and ytodetermine the quadrant of the angle and return a number ranging from -pi to pi.
float ceil (float val);	Calculates the smallest integer that is bigger than *val*. Example: ceil(4.3) = 5.
float floor (float val);	Calculates the largest integer that is smaller than *val*. Example: ceil(4.8) = 4.
float fmod (float x, float y);	Returns the remainder of x / y. For example: fmod(15.0,4.0) = 3.
float pow (float x, float y);	Returns x to the power y.

Logical operations

You can also perform logic operations with variables, like AND, OR and NOT operations, using the following operators:

Operator	*Description*
!	NOT (bit level) Example: P1_0 = !P1_0;
~	NOT (byte level) Example: P1 = ~P1;
&	AND
|	OR

Note that those logic operations are performed on the bit level of the registers. To understand the effect of such operation on registers, it's easier to look at the bits of a variable (which is composed of one or more register). For example, a NOT operation will invert all the bits of a register. Those logic operators can be used in many ways to merge different bits of different registers together.

For example, consider the variable 'P1', which is of type 'char', and hence stored in an 8-bit register. Actually P1 is an SFR, whose 8 bits represent the 8 I/O pins of Port 1. It is required in that example to clear the four lower bits of that register without changing the state of the four others which may be used by other equipment. This can be done using logical operators according to the following code:

```
P1 = P1 &0xF0;//Adding '0x' before a number indicates that it is a hexadecimal one
```

Here, the value of P1 is ANDed with the variable 0xF0, which in the binary base is '11110000'. Recalling the two following relations:

1 AND X = X
0 AND X = 0
(where 'X' can be any binary value)

You can deduce that the four higher bits of P1 will remain unchanged, while the four lower bits will be cleared to 0.

By the way, note that you could also perform the same operation using a decimal variable instead of a hexadecimal one, for example, the following code will have exactly the same effect than the previous one (because 240 = F0 in HEX):

```
P1 = P1 &240;
```

A similar type of operations that can be performed on a port, is to to set some of its bits to 1 without affecting the others. For example, to set the first and last bit of P1, without affecting the other, the following source code can be used:

```
P1 = P1 |0x81;
```

Here, P1 is ORed with the value 0×81, which is '10000001' in binary. Recalling the two following relations:

1 OR X = 1
0 OR X = X
(where 'X' can be any binary value)

You can deduce that the first and last pins of P1 will be turned on, without affecting the state of the other pins of port 1. Those are just a few examples of the manipulations that can be done to registers using logical operators. Logic operators can also be used to define very specific conditions, as you shall see in the next section.

The last types of logic operation studied is the shifting. It can be useful the shift the bit of a register the right or to the left in various situations. this can be done using the following two operators:

Operator	Description
>>	Shift to the right
<<	Shift to the left

The syntax is quite intuitive, for example:

```
P1 =0x01;// After that operation, in binary, P1 = 0000 0001
P1 =(P1 <<7)// After that operation, in binary P1 = 1000 0000
```

You can clearly notice that the content of P1 have been shifted 8 steps to the left.

Conditions and loops

In most programs, it is required at a certain time, to differentiate between different situations, to make decision according to specific input, or to direct the flow of the code depending on some criteria. All the above situations describe an indispensable aspect of programming: 'conditions'. In other words, this feature allows to execute a block of codes only under certain conditions, and otherwise execute another code block or continue with the flow of the program.

The most famous way to do that is to use the 'if' statement, according to the following syntax.

```
if(expression){
...
code to be executed
...
}
```

It is important to see how the code is organized in this part. The 'expression' is the condition that shall be valid for the 'code block' to be executed. The code block is all delimited by the two brackets '{' and '}'. In other words, all the code between those two brackets will be executed if and only if the expression is valid. The expression can be any combination of mathematical and logical expressions, as you can see in the following example:

```
if((P1 ==0)&(a <=128)){
...
code to be executed
...
}
```

Notice the use of the two equal signs (==) between two variables or constants. In C language, this means that you are asking whether P1 equals 0 or not. Writing this expression with only one equal sign, would cause the the compiler to store 0 in P1. This issue is a source of logical error for many beginners in C language, this error won't generate any alert from the compiler and is hard to identify in a big program, so pay attention, it can save you lot of debugging time. Otherwise it is clear that in that previous example, the code block is only executed if both the two expressions are true. Here is a list of all the operators you can use to write an expression describing a certain condition:

Operator	*Description*
==	Equal to
<, >	Smaller than, bigger than.
<=, >=	Smaller than or equal to, bigger than or equal to.
!=	Not equal to

The 'If' code block can get a little more sophisticated by introducing the 'else' and 'else if' statement. Observe the following example source code:

```
if(expression_1){
...
code block 1
...
}elseif(expression_2){
...
code block 2
...
}elseif(expression_3){
...
code block 3
...
}else{
...
code block 4
...
}
```

Here, there are four different code blocks, only one shall be executed if and only if the corresponding condition is true. The last code block will only be executed if none of the previous expressions is valid. Note that you can have as many 'else if' blocks as you need, each one with its corresponding condition, BUT you can only have one 'else' block, which is completely logical. However, you can choose not to have and 'else' block at all if you want.

There are some other alternatives to the 'if…else' code block, that can provide faster execution speeds, but also have some limitations and restrictions like the 'Select…case' code block. For now, it is enough to understand the 'if…else' code block, whose performance is quite fair and have a wide range of applications.

Another very important tool in the programming languages is the ***loop***. In C language like in many others, loops are usually restricted to certain number of loops like in the 'for' code block or restricted to a certain condition like the 'while' block.

Let's start with the 'for' code block, which is a highly controllable and configurable loop. consider the following source code:

```
for(i=0;i<10;i++){
P0 = i;
}
```

Here the code between the two brackets '{' '}' will be be executed a certain number of times, each time with the counting variable 'i' increasing by 1 according to the statement 'i++'. The code will keep looping as long as the condition 'i<10' is true. Usually, the counting value 'i' is reused in the body of the loop, which makes the particularity of this loop. The 'for' loop functioning can be recapitulated by the following syntax:

```
for(start;condition;step){
...
code block
...
}
```

Where ***start*** represents the start value assigned to the count value before the loop begins. The ***condition*** is the expression that is is to remain true for the loop to continue; as long as this condition is satisfied, the code will keep looping. Finally, ***step*** is the increase or decrease of the counting variable, it can be any statement that changes its value, whether by addition or subtraction.

The second type of loop that we are going to study is the 'while' loop. The syntax of this one is simpler than the previous one, as you can observe in the following example source code, that is equivalent to the previous method:

```
while(i <10){
P0 = i;
i = i +1;
}
```

Here there is only one parameter to be defined, which is the condition to keep this loop alive, which is 'i < 10' in our example. Then, it is the responsibility of the programmer to design the software carefully to provide an exit for that loop, or to make it an infinite loop. Both the techniques are commonly used in microcontroller programs, as you shall see later on along this tutorial.

Functions

Functions are ways of organizing your code, reducing its size, and increasing its overall performance, by grouping relatively small parts of the code to be reused many times in the same program. A new function can be created according to the following syntax:

```
Function_name(parameter_1, Parameter_2, Parameter_3){
...
function body
...
return value //optional
...
}
```

This is the general form of a function. The number of parameters of the function can be more than the three parameters of the examples above, as it can be zero, all depends on the type and use of the function. The function's body is usually a subprogram that implies the parameters to produce the required result. Some functions will also generate an output, like the cos() function, through the 'return' command, which will output the value next to it. Usually the 'return' command is used at the end of the function.

A very common use of functions without return value is to create delays in a software, consider the following function:

```
delay(unsignedint y){
unsignedint i;
for(i=0;i<y;i++){
;
}
}
```

In this last piece of code a function named 'delay' is created, with an unsigned integer 'y' as a parameter, and implying a locally defined unsigned int 'i'. The function will repeat a loop for a couple hundreds or thousand of times to generate precise delays in a program. A function like this can be called from anywhere in the program according to the following syntax:

```
delay(30000);
```

This line of code would cause the program to pause for approximately one second on a 12 MHz clock on a 8051 microcontroller.

A common example of a function with a return value, is a function that will calculate the angle in radians of a given angle in degrees, as all the trigonometric functions that are included by default take angles in radians. This function can be as the following:

```
deg_to_rad(floatdeg){
float rad;
rad=(deg*3.14)/180;
retrun rad;
}
```

This function named 'deg_to_rad' will take as a parameter an angle in degrees and output an angle in radians. It can be called in your program according to this syntax:

```
angle=deg_to_rad(102,18);
```

Where angle should be already defined as a float, and where will be stored the value returned by the function, which is the angle in radians equivalent to 102.18°.

Another important note about functions is the 'main' function. Any C program must contain a function named 'main' which is the place where the program's execution will start. More precisely, for microcontrollers, it were the execution will start after a reset operation, or when a microcontroller circuit is turned ON. The 'main' function has no parameters, and is written like this:

```
main(){
...
code of the main functions
...
}
```

Organization of a C program

All C programs have this common organization scheme, sometimes it's followed, sometimes it's not, however, it is imperative for this category of programming that this organization scheme be followed in order to be able to develop your applications successfully. Any application can be divided into the following parts, noting that is should be written in this order:

1. **Headers Includes and constants definitions:** In this part, header files (.h) are included into your source code. Those header files can be system headers to declare the name of SFRs, to define new constants, or to include mathematical functions like trigonometric functions, root square calculations or numbers approximations. Header files can also contain your own functions that would be shared by various programs.
2. **Variables declarations:** More precisely, this part is dedicated to 'Global Variables' declarations. Variables declared in this place can be used anywhere in the code. Usually in microcontroller programs, variables are declared as global variables instead of local variables, unless you are running short of RAM memory and want to save some space, so we use local variables, whose values will be lost each time you switch from a function to another. To summarize, global variables are easier to use and implement than local variables, but they consume more memory space.
3. **Functions' body:** Here you group all your functions. Those functions can be simple ones that can be called from another place in your program, as they can be called from an 'interrupt vector'. In other words, the subprograms to be executed when an interrupt occurs is also written in this place.
4. **Initialization:** The particularity of this part is that it is executed only once when the microcontroller was just subjected to a 'RESET' or when power is just switched ON, then the processor continues executing the rest of the program but never executes this part again. This particularity makes it the perfect place in a program to initialize the values of some constants, or to define the mode of operation of the timers, counters, interrupts, and other features of the microcontroller.
5. **Infinite loop:** An infinite loop in a microcontroller program is what is going to keep it alive, because a processor has to be allayed running for the system to function, exactly

like a has have to be always beating for a person to live. Usually, this part is the core of any program, and it's from here that all the other functions are called and executed.

Simple C program for 89S52

Example program for 8051

Program 1: Write an 8051 C program to send values 00 – FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char z;
for (z=0;z<=255;z++)
P1=z;
}
```

Program 2: Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mynum[]="012345ABCD";
unsigned char z;
for (z=0;z<=10;z++)
P1=mynum[z];
}
```

Program 3: Write an 8051 C program to toggle all the bits of P1 continuously.

Solution:

```
//Toggle P1 forever
#include <reg51.h>
void main(void)
{
for (;;)
{
```

```
p1=0x55;
p1=0xAA;
}
}
```

Program 4: Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
//Singed numbers
#include <reg51.h>
void main(void)
{
charmynum[]={+1,-1,+2,-2,+3,-3,+4,-4};
unsigned char z;
for (z=0;z<=8;z++)
P1=mynum[z];
}
```

Program 5: Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

Solution:

```
#include <reg51.h>
sbit MYBIT=P1^0;
void main(void)
{
unsignedint z;
for (z=0;z<=50000;z++)
{
MYBIT=0;
MYBIT=1;
}
}
```

Program 6: Write an 8051 C program to toggle bits of P1 continuously forever with some delay.

Solution:

```
//Toggle P1 forever with some delay in between
//"on" and "off"
```

```
#include <reg51.h>
void main(void)
{
unsignedint x;
for (;;) //repeat forever
{
p1=0x55;
for (x=0;x<40000;x++); //delay size //unknown
p1=0xAA;
for (x=0;x<40000;x++);
}
}
```

Program 7: Write an 8051 C program to toggle bits of P1 ports continuously with a 250 ms.

Solution:

```
#include <reg51.h>
voidMSDelay(unsigned int);
void main(void)
{
while (1) //repeat forever
{
p1=0x55;
MSDelay(250);
p1=0xAA;
MSDelay(250);
}
}
voidMSDelay(unsigned intitime)
{
unsignedinti,j;
for (i=0;i<itime;i++)
for (j=0;j<1275;j++);
}
```

Program 8: LEDs are connected to bits P1 and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

```
#include <reg51.h>
#defind LED P2;
void main(void)
{
P1=00; //clear P1
LED=0; //clear P2
for (;;) //repeat forever
{
P1++; //increment P1
LED++; //increment P2
}
}
```

Index