Microprocessors and Interfacing

8086, 8051, 8096, and advanced processors

N. Senthil Kumar
Professor
Department of Electrical and Electronics Engineering
Mepco Schlenk Engineering College
Sivakasi, Tamil Nadu

M. Saravanan
Professor
Department of Electrical and Electronics Engineering
Thiagarajar College of Engineering
Madurai, Tamil Nadu

S. Jeevananthan
Professor
Department of Electrical and Electronics Engineering
Pondicherry Engineering College
Puducherry

S.K. Shah
Professor and Head
Department of Electrical Engineering
MS University of Baroda
Vadodara, Gujarat

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

First published in 2012

ISBN-13: 978-0-19-807906-4
ISBN-10: 0-19-807906-0

Typeset in Times New Roman
by Trinity Designers & Typesetters, Chennai
Printed in India by Tara Art Printers (P) Ltd, Noida

Brief Contents

Detailed Contents

1

Microprocessors—Evolution and Introduction to 8085

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Importance of microprocessors
- Origin and evolution of microprocessors
- Classification of microprocessors and memories
- Common input and output devices for computers
- Bus structures used in computers and technology improvements
- 8085 microprocessor architecture and instruction set

1.1 INTRODUCTION

The microprocessor is an electronic chip that functions as the central processing unit (CPU) of a computer. In other words, the microprocessor is the heart of any computer system. Microprocessor-based systems with limited resources are called microcomputers. Today, microprocessors can be found in almost all consumer electronic devices such as computer printers, washing machines, microwave ovens, mobile phones, fax machines, and photocopiers and in advanced applications such as radars, satellites, and flights. Any middle-class household will have about a dozen microprocessors in different forms inside various appliances. The recent developments in the electronics industry and the large-scale integration of devices have led to rapid cost reduction and increased application of microprocessors and their derivatives.

Typically, basic microprocessor chips have arithmetic and logic functional units along with the associated control logic to process the instruction execution. Almost all microprocessors use the basic concept of *stored-program execution*. Programs or instructions to be executed by the microprocessor are stored sequentially in memory locations. The microprocessor, or the processor in general, fetches the instructions one after another and executes them in its arithmetic and logic unit. So all microprocessors have a built-in memory access and management part as well as some amount of memory.

A microprocessor can be programmed to perform any task that can be written and programmed by the user. Without a program, the microprocessor unit is a piece of useless electronic circuit. The programmer must take care of all the resources of the microprocessor and use them efficiently for implementing the required functionality. So to work with the microprocessor, it is necessary for the programmer to know about its internal resources and features. The programmer

must also understand the instructions that a microprocessor can support. Every microprocessor has its own associated set of instructions; this list is given by all microprocessor manufacturers. The instruction set for microprocessors is in two forms—one in mnemonic, which is comparatively easy to understand and the other in binary machine code, which the microprocessor works with and is difficult for us to understand. Generally, programs are written using mnemonics called assembly-level language and then converted into binary machine-level language. This conversion can be done manually or using an application called assembler.

In general, programs are written by the user for the microprocessor to work with real world data. Data are available in many forms and from many sources. To input these data to the microprocessor, the microprocessor-based systems need some input interfacing circuits and some electronic processing circuits. These circuits include data converters and ports. After processing the real world data, the output from the microprocessor must be taken out to give to the output devices or circuits. This again needs interfacing circuits and ports. So a microprocessor-based system will need a set of memory units and interfacing circuits for inputs and outputs. The circuits, together with the microprocessor, make the *microcomputer system*. The physical components of the microcomputer system are called *hardware*. The program that makes this hardware useful is called *software*.

The semiconductor manufacturing technology for chips has developed from transistor–transistor logic (TTL) to complementary metal-oxide-semiconductor (CMOS). Microprocessor manufacturing also has gone through these technological changes. The other semiconductor manufacturing technology available is emitter-coupled logic (ECL). TTL technology is most commonly used for basic digital integrated circuits; CMOS is favoured for portable computers and other battery-powered devices because of its low power consumption.

1.2 EXPLANATION OF BASIC TERMS

The terms relevant to the use of microprocessors are explained in this section. These explanations will give the reader an understanding of various microprocessor-related terms, technologies, and topics.

Chip A chip or an integrated circuit is a small, thin piece of silicon with the required circuits and transistors etched on it to perform a particular function. Simpler processors may consist of a few thousand transistors etched onto a silicon base just a few millimeters square.

Bit A bit means a single binary digit. The bit is also the fundamental storage unit of computer memory. In binary form, a bit can have only two values, 0 or 1, whereas a decimal digit can have 10 values, represented by symbols 0 through 9.

Bit size The bit size of a microprocessor refers to the number of bits that can be processed simultaneously by the basic arithmetic circuits of the microprocessor.

Word A word is a number of bits grouped together for processing. In microprocessors, a word refers to the basic data size or bit size that can be processed

by the arithmetic and logic unit (ALU) of the processor. A 16-bit binary number is called a word in a 16-bit processor.

Memory word The number of bits that can be stored in a register or memory element is called memory word. Mostly, all memory units use eight bits for their memory word.

Byte An 8-bit word is referred to as a byte.

Nibble A 4-bit word is referred to as a nibble.

Kilobyte A collection of 1024 bytes is called a kilobyte (2^{10} bytes).

Megabyte A collection of 1024 kilobytes is called a megabyte (2^{20} bytes).

RAM or R/W memory Random access memory or read/write memory is a type of semiconductor memory in which a particular memory location can be erased and written with new data at any time. These memory units are volatile, which means that the contents of the memory are erased when the power to the chip is disrupted. The access of the individual memory location can be done randomly. In microprocessors, the RAM is used to store data.

DRAM Dynamic random access memory is a semiconductor memory in which the stored contents need to be refreshed repeatedly at about thousands of times per second. Without refreshing, the stored data will be lost. These memory chips are preferred in a computer system as these are slower but economical.

SRAM Static random access memory chips keep the data stored in it as long as power is available. There is no need for refreshing. In terms of speed, SRAM is faster.

ROM Read only memory devices are memory devices whose contents are retained even after removing the power supply.

Arithmetic and logic unit ALU is a digital circuit present in the microprocessor to perform arithmetic and logic operations on digital data. The typical operations performed by the ALU are addition, subtraction, logical AND, logical OR, and comparison of binary data. Generally, the functions of the ALU of a microprocessor will decide the processor's functionality.

Microcontroller A microcontroller is a chip that includes microprocessor, memory, and input/output signal ports. Microcontrollers can be called single-chip microcomputers.

Microcomputer The system formed by interfacing the microprocessor with the memory and I/O devices to execute the required programs is called microcomputer.

Bus A bus is a group of wires/lines that carry similar information.

System bus The system bus is a group of wires/lines used for communication between the microprocessor and peripherals.

Firmware Software written for a microprocessor application without provision for changes is called firmware. These are stored in the permanent storage or ROM of the computer system.

Input device The devices that are used for providing data and instructions to the microprocessor or microcomputer system are called input devices. Keyboard and mouse are the common input devices.

Output device The devices that are used for transferring data out of the microprocessor or microcomputer system are called output devices. Display screen, printer, and other forms of display are the common output devices.

Floppy disk A removable-type magnetic disk used for storing programs and data for transferring from and to the computer is called floppy disk.

Disk drive The hardware component that is used to read or write data to devices such as floppy disks is called disk drive.

Computer architecture The design, internal configuration, and accesses in a digital computer are together called computer architecture.

Von–Neumann architecture The architecture in which the same memory is used for storing programs as well as data.

Harvard architecture The architecture in which programs and data are stored in two separate memory units.

CISC processor Complex instruction set computer is a processor architecture that supports many machine language instructions.

RISC processor Reduced instruction set computer is a processor architecture that supports limited machine language instructions. RISC processors are expected to execute the programs faster than CISC processors.

High-level language A computer programming language in which programs are written without the knowledge of the processor in which the program will be executed. BASIC, Fortran, C, Pascal, and Java are examples of high-level languages.

Assembly language A programming language written using the mnemonics or the instruction set of a particular microprocessor is called assembly language. Assembly language programming is microprocessor-specific. It is not as easily understood as a high-level language program, but is easier than a machine language program.

Machine language Machine language refers to binary code programs that are specific to the processor and can be directly executed by the processor. Machine language is the lowest level language and cannot be easily understood.

Assembler A computer application program that converts the assembly language program into machine-level language program.

Compiler A computer program that converts the high-level language program into machine-level language program.

Interpreter A computer program that reads the high-level or assembly-level program one line at a time and converts it into machine-level program. Compiler and assembler can function only on the entire program in a file.

Algorithm A sequence of operations or instructions that defines how to solve a problem using a computer or microcomputer. An algorithm must be definite, must follow a clear instruction flow without ambiguity, and must have definite start and end points.

BIOS Basic input/output system is a set of programs that handles the input and output functions and interacts with the hardware directly. A new hardware installed must be provided with the corresponding BIOS routines.

Clock The circuit in the computer that generates the sequence of evenly spaced pulses to synchronize the activities of the processor and its peripherals is called clock. The clock speed determines the speed of the operation of the computer. The computer with a high frequency clock works faster. Normally the clock frequency is in the range of megahertz (MHz) or gigahertz (GHz).

MIPS Million instructions per second is a measure of the speed at which the instructions are executed in a processor.

Tri-state logic It is the logic used by digital circuits. The three logic levels used are high (1), low (0), and high impedance state (Z). The logic high state of a digital circuit can source current and the logic low can sink current in a computer system, but the high impedance state neither sources nor sinks current and so the other devices connected to it are not affected.

Operating system The program that controls the entire computer and its resources and enables users to access the computer and its resources is called operating system. It is required for any computer system to become operational and user friendly. Under the control of the operating system, the computer recognizes and obeys commands typed by the user. In addition, the operating system provides built-in routines that allow the user's program to perform input/output operations without specifying the exact hardware configuration of the computer. In low-level microprocessor-based systems, the program that controls the hardware is called *monitor routine* or *monitor software*.

1.3 MICROPROCESSORS AND MICROCONTROLLERS

The microprocessor (also called CPU) is the principal element of a computer as it executes lists of instructions. These instruction lists are commonly called *programs*. This programming language is complex to use since it is machine- or processor-specific and coded into hexadecimal and binary.

Two types of processors are manufactured—the microprocessor and the microcontroller. At the data processing level, the two are practically equivalent. The distinction comes from the established functionalities.

The general-purpose microprocessors give the computers all the necessary computing power. These microprocessors need additional circuitry elements such

as memory devices and I/O ports to connect the input and output devices. All microprocessor-based systems need two types of memories—RAM and ROM. RAM is used for storage of data while ROM is used for storage of programs, especially the start-up program that runs when the microprocessor is powered on.

There are numerous microprocessors developed by many companies. The evolution of microprocessors, from 4-bit microprocessors to 64-bit microprocessors, has been discussed later in this chapter. This book is devoted to the discussion of two groups of microprocessors—Intel's 8-bit 8085 microprocessor series in brief and 16-bit 8086 series in detail.

Microcontrollers are microprocessors designed specially for control applications. Microcontrollers contain memory units and I/O ports inside a chip, in addition to the CPU. Microcontrollers are otherwise called embedded controllers; they are generally used to control and operate smart machines. Some of the machines using microcontrollers are microwave ovens, washing machines, sewing machines, automobile ignition systems, computer printers, and fax machines. You will be amazed to know that out of 100 processor chips manufactured, 99 are embedded processors; only one goes into a general computer! A plethora of semiconductor companies are in the microcontroller market and any application development engineer is flooded with a variety of microcontrollers to choose from. This book discusses Intel's 8-bit 8051 series and 16-bit 8096 series as also other advanced microcontrollers.

1.4 MICROPROCESSOR-BASED SYSTEM

A computer system developed using a basic general-purpose microprocessor is called a microcomputer system. The system consists of CPU, memory, and I/O ports as shown in Fig. 1.1.

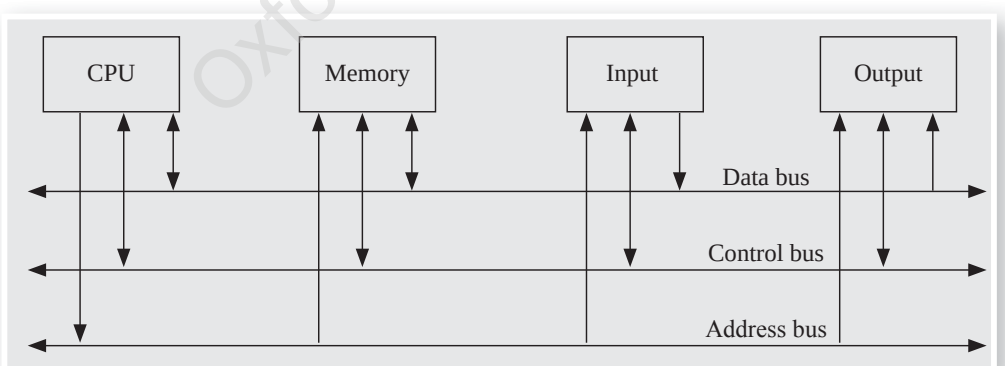


Fig. 1.1 Microcomputer system (Von–Neumann model)

Figure 1.2 shows a typical personal computer system. The interfacing of the processor with the other parts of the microcomputer system needs a three-bus architecture. The three buses are data bus, address bus, and control bus.

Each memory location or I/O port is identified by a specific address similar to a postal address. In microprocessor systems, the addresses are all in binary, and in general, represented in hexadecimal number format. The address is a

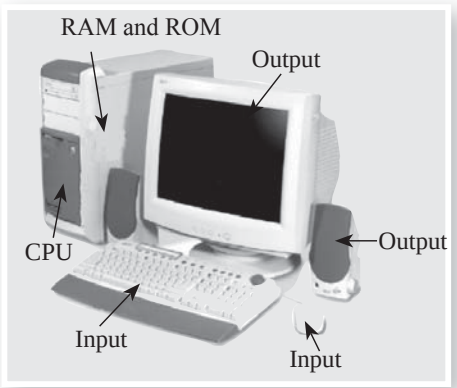


Fig. 1.2 Personal computer

unique pattern used to identify a location in the memory or an I/O port. The address bus consists of many lines that transport the digital data sent by the processor. An address bus of eight bits corresponds to eight lines of addresses and can thus address 2^8 different memory locations. These addresses are written in hexadecimal number format as 00H–FFH and can be used to for 256 different locations. Similarly, the 16-bit address bus can address 2^{16} different addresses. Its address range is 0000H–FFFFH. The greater the number of lines in the address bus, the greater the number of locations the processor is able to manage.

The address on the address bus can locate a specific memory or I/O location. After selecting the location, the data transfer between the memory and processor or between the I/O device and the processor is done through the data bus. The width of the data bus determines the data size that can be transferred. An 8-bit processor will generally have an 8-bit data bus and a 16-bit processor will have a 16-bit data bus. The memory locations in microprocessors are accessed as 8-bit or one-byte units only. So the transfer of a 16-bit data from memory needs two memory addresses. A 1 KB memory chip will have 1024 bytes of memory locations.

A control bus is needed for proper data transfer between the processor and the peripherals. The control bus basically consists of signals for selection of the correct memory or I/O device from the address, indication of the direction of data transfer, and synchronization of data transfer between slow devices. Many of the control signals are given by the processor itself because the processor is the master of the computer system. Some control signals such as selection of the correct memory chip can be generated externally by the logic circuits. The timing of the control signal is very important; the entire timing of the operation is controlled by the microprocessor in synchronization with the clock signal input.

1.5 ORIGIN OF MICROPROCESSORS

The microprocessor is the greatest invention of the 20th century. Its evolution started from the earlier mechanical calculating devices, in the 1930s. These devices used mechanical relays. Later, in the 1950s, these devices were replaced by vacuum tubes. The vacuum tubes were quickly replaced by transistors. The breakthrough in transistor technology led to the introduction of minicomputers in the 1960s and the personal computer revolution in the 1970s.

The transistor technology led to the development of complex devices called integrated circuits (ICs). The microprocessor, or microprocessing unit (MPU), later evolved as an IC and was designed to fetch instructions and execute the predefined arithmetic and logic functions. Intel was the first MPU producer and has been holding a large share of the world market for this product. The evolution

of microprocessors is categorized into five generations: first, second, third, fourth, and fifth.

1.5.1 First Generation (1971–1973)

The microprocessors that were introduced from 1971 to 1973 were referred to as the first-generation systems. First-generation microprocessors processed their instructions serially—they fetched the instruction, decoded it, and then executed it. The first microprocessor, the 4004, was introduced in 1971. It was co-developed by Busicom, a Japanese manufacturer of calculators, and Intel, a US manufacturer of semiconductors. The 4-bit 4004 microprocessors ran at 108 kHz and contained 2300 transistors. They were fabricated using p-channel metal-oxide-semiconductor (PMOS) technology, which provided low cost, slow speed, and low output currents. They were not compatible with TTL. In 1972, Intel made the 8-bit 8008 and 8080 microprocessors.

1.5.2 Second Generation (1974–1978)

As the technology evolved, the number of circuits that could be fabricated on a chip grew. Very large-scale integration (VLSI) led to chips that had speeds up to hundreds of millions of switchings per second. The second generation marked the beginning of very efficient 8-bit microprocessors. Some of the popular processors were Motorola's 6800 and 6809, Intel's 8085, and Zilog's Z80. The second-generation devices marked a sharp contrast with the use of newer semiconductor technology to fabricate chips. They were manufactured using n-channel metal-oxide-semiconductor (NMOS) technology. This technology offered faster speed and higher density than PMOS. It resulted in a five-fold increase in instruction execution speed and higher chip densities.

1.5.3 Third Generation (1978–1980)

The third generation, introduced in 1978, was dominated by Intel's 8086 and Zilog's Z8000, which were 16-bit processors with minicomputer-like performance. These processors had the technology of 16-bit arithmetic and pipelined instruction processing. The third generation came with IC transistor counts of about 250,000. In Motorola's MC68020, for example, an on-chip cache was incorporated for the first time and the depth of the pipeline was increased to five or more stages. It was designed using high density metal-oxide-semiconductor (HMOS) technology. HMOS provides some advantages over NMOS: Its speed–power product is four times better than that of NMOS; it can accommodate twice the circuit density of NMOS.

1.5.4 Fourth Generation (1981–1995)

The microprocessors entered their fourth generation with designs containing more than a million transistors in a single package. This era marked the beginning of 32-bit microprocessors. Intel introduced 80386 and Motorola introduced 68020/68030. They were fabricated using high density/high speed complementary metal-oxide-semiconductor (HCMOS), a low-power version of the HMOS technology.

1.5.5 Fifth Generation (1995–till date)

The fifth generation microprocessors employ decoupled super scalar processing and their design contains more than 10 million transistors. This generation marks the introduction of devices that carry on-chip functionalities. It has also paved the way for high speed memory I/O devices along with the introduction of 64-bit microprocessors. Intel leads the show here with Pentium, Celeron, and very recently, dual- and quad-core processors working with up to 3.5 GHz speed. This generation is characterized by a low-margin single-microprocessor PC business, which is complemented by high-volume sales. Table 1.1 gives the comparison of the major processors based on specific parameters such as clock speed and data word size.

Table 1.1 Comparison of general-purpose processors

General-purpose processors	Transistors	CPU speed	Data length (bits)
8080	6,000	2 MHz	8
8085	6,500	3 MHz	8
8088	29,000	3 MHz	16
8086	30,000	4 MHz	16
80286	1,34,000	6 MHz	16
80386	2,75,000	16 MHz	16/32
80486	12,00,000	33 MHz	16/32
Athalon XP	37,00,000	2.8 GHz	16/32/64
Celeron	75,00,000	1.06–2 GHz	32
Pentium II	75,00,000	233–450 MHz	32
Pentium III	95,00,000	450 MHz–1 GHz	32
Pentium III Xeon	2,81,00,000	500 MHz–1 GHz	32
Pentium 4	5,50,00,000	1.4–2.2 GHz	32
IBM PowerPC G3	65,00,000	233–333 MHz	32
PowerPC G4	1,05,00,000	400–800 MHz	32

1.5.6 Timeline of Microprocessor Evolution

(i) 1971—Intel 4004 microprocessor with 2300 transistors, working at a speed of 108 kHz

(ii) 1971—Intel 8008, twice as powerful as the 4004, with 3500 transistors and speed of 200 kHz

(iii) 1974—Intel 8080 processor with 6000 transistors and speed up to 2 MHz

(iv) 1976—Intel 8085 processor with about 6500 transistors and speed of 3–5 MHz came into existence. There were multiple versions of 8085 microprocessors. The original version of the 8085 microprocessor without suffix A was manufactured by Intel. It was quickly replaced with the 8085A, which had a bug-fixer. A few years later, in the 1980s, Intel introduced the 8085AH, the HMOS version of 8085A followed by the 80C85A, the CMOS version of the 8085A.

(v) 1978—Intel 80X86 families of microprocessors. The first generation of the 80X86 families included the 8086 and the 8088. It was followed by the 80186, 80286, 80386, and 80486.

(vi) 1979—Intel 8088, which was similar in architecture to the 8086; the difference was in the available number of data bits of the data bus. Number of transistors: 29,000; speed: 5 MHz, 8 MHz, 10 MHz

(vii) 1985—Intel 80386, the first 32-bit chip that contained 275,000 transistors, processing five million instructions per second, and running all popular operating systems, including Windows.

(viii) 1989—Intel 486 with an 8 KB cache memory (shared for data and instructions), operating at clock frequencies from 25 to 100 MHz

(ix) 1993—Intel Pentium processor retains the 32-bit address bus of the 80486 but doubles the data bus to 64 bits. It includes two 8 KB cache memories—one for instructions and the other for data. It was based on dual pipeline method known as superscalar architecture and currently operates with frequencies up to 1.75 GHz, 20-stage pipeline, and three-level cache memory architectures.

(x) 1997—Intel Pentium II processor was designed specifically to process video, MMX audio, and graphics data efficiently with speeds of 200 MHz, 233 MHz, 266 MHz, and 300 MHz.

(xi) 1999—Intel Celeron processor and Intel Pentium III processor

(xii) 2000—Intel Pentium 4 processor

Various other companies such as Motorola, NEC, Mitsubishi, Siemens, AMD, Toshiba, and Texas Instruments also manufacture processor chips. These companies have their own chips and architectures in addition to the regular Intel-based architectures.

1.6 CLASSIFICATION OF MICROPROCESSORS

Microprocessors can be classified based on their specifications, applications, and architecture.

Based on the size of the data that the microprocessors can handle, they are classified as 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit microprocessors.

Based on the application of the processors, they are classified as follows:

(i) General-purpose processors
(ii) Microcontrollers
(iii) Special-purpose processors

General-purpose processors are those that are used in general computer system integration and can be used by the programmer for any application. Common microprocessors from Intel 8085 to Intel Pentium are examples of general-purpose processors. *Microcontrollers* are microprocessor chips with built-in hardware for the memory and ports. These chips can be programmed by the user for any generic control application. *Special-purpose processors* are designed specifically

to handle special functions required for an application. Digital signal processors are examples of special-purpose processors; these have special instructions to handle signal processing. Application-specific integrated circuit (ASIC) chips are also examples of this category of microprocessors.

Based on the architecture and hardware of the processors, they are classified as follows:

(i) RISC processors
(ii) CISC processors
(iii) VLIW processors
(iv) Superscalar processors

RISC is a processor architecture that supports limited machine language instructions. *RISC processors* can execute programs faster than CISC processors. *CISC processors* have about 70 to a few hundred instructions and are easier to program. However, CISC processors are slower and more expensive than RISC processors. *Very long instruction word (VLIW) processors* have instructions composed of many machine operations. These instructions can be executed in parallel. This parallel execution is called instruction-level parallelism. VLIW processors also have a large number of registers. *Superscalar processors* use complex hardware to achieve parallelism. It is possible to have overlapping of instruction execution to increase the speed of execution.

1.7 TYPES OF MEMORY

Memory unit is an integral part of any microcomputer system. Its primary purpose is to hold program and data. The main objective of the memory unit design is to enable it to operate at a speed close to that of the processor. Although technology is available to design such a high speed memory, cost is the major limiting factor. To strike a balance between cost and operating speed, a memory system is usually designed using different materials such as solid state, magnetic, and optical materials.

A microcomputer memory can be logically divided into four groups:

(i) Processor memory/register
(ii) Cache memory
(iii) Primary or main memory
(iv) Secondary memory

Processor memory refers to a set of CPU registers. Processor registers are the first set of storage devices available for the programmers to store any data, but they are generally few in number—up to a few tens or hundreds. As these registers are available within the processor, they are the fastest memory registers. The main disadvantage is the cost involved, which restricts the number of registers and their bytes.

Cache memory is the fastest external memory; it is placed close to the processor. The instructions to be executed are placed in the cache memory for access by the processor. These are a few kilobytes in size. Cache memory contains volatile semiconductor RAMs. The processor fetches instructions from the cache memory and if an instruction is not in cache, it refers to the primary memory.

Primary memory is the storage area from which all the programs are executed. All the programs and corresponding data for execution must be within the primary memory. The primary memory is much larger than the processor memory and the cache memory but its operating speed is slower. The primary memory in a system varies from few KB to a few MB.

Secondary memory refers to the storage medium for huge files such as program source codes, compilers, operating systems, etc. These are not accessed directly or very frequently by the microprocessor in a computer system. Secondary memory consists of slow devices such as magnetic tapes and optical disks. Sometimes, they are referred to as auxiliary or backup store. Stored information in a magnetic tape or magnetic disk is not lost when power is turned off. Therefore, these storage devices are called non-volatile memories.

Classification of primary memory Primary memory normally includes ROM and RAM, which are further classified as shown in Fig. 1.3. Microprocessor-based systems have at least one RAM and one ROM chip.

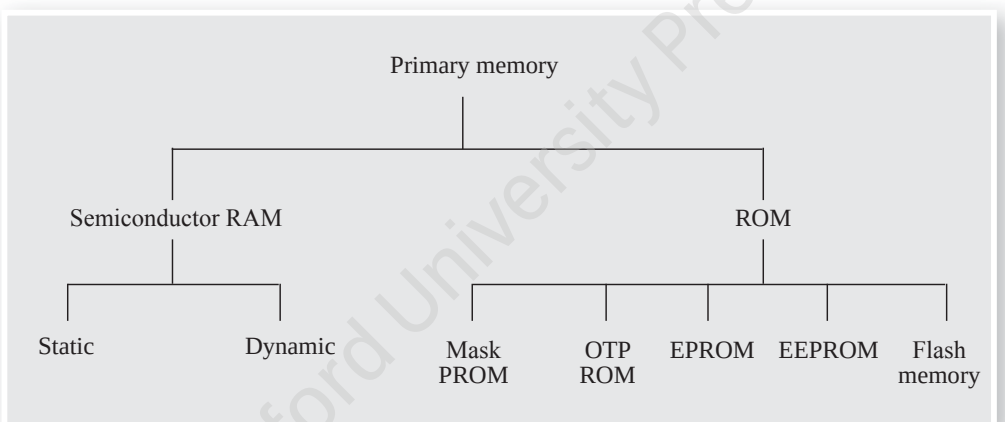


Fig. 1.3 Classification of primary memories

RAM devices allow both reading and writing to their memory cells. In static RAM devices, bits are stored as the status of on/off switches. There are no charges involved and hence, no charges to leak. However, static RAM devices have complex construction and hence larger size per unit storage. So they are more expensive. Static RAMs are comparatively faster and are used in cache memories.

In dynamic RAM devices, the data bits are stored as charge in capacitors. Since capacitor charge has a tendency to leak, these devices need refreshing even when they are powered. However, they have simpler construction and smaller size per unit storage. These devices are less expensive and comparatively slower.

As the name implies, a ROM permits only read access. There are many kinds of ROMs:

(i) Mask programmable ROMs (MPROMs) are custom-made for the customer; their contents are programmed by the manufacturer. Since they are mass produced, they are inexpensive. The customer cannot erase or program it afterwards.

(ii) Programmable ROMs (PROMs) or one-time programmable (OTP) ROMs are devices that can be programmed by the user in his/her place using special equipments. The main disadvantage of PROMs is that they cannot be erased and reprogrammed.

(iii) Erasable and programmable ROMs (EPROMs) allow the erasure and reprogramming of the content by the user. In an EPROM, programs are entered using electrical impulses and the stored information is erased using ultraviolet rays.

(iv) Electrically erasable PROMs (EEPROMs) or electrically alterable ROMs (EAROMs) allow the users to electrically erase and reprogram its contents. EEPROMs are different from RAMs in that electrical signals are required to erase and program them. EEPROMs require a higher voltage for erasing and programming than the normal 5 V supply.

(v) Flash memory devices are a group of single transistor cell EPPROMs. Cell sizes are about half the size of a two-transistor EEPROM. The operation requires bulk erasure of a large portion of the memory array.

1.8 INPUT AND OUTPUT DEVICES

Input and output devices permit the user to feed data to the computer and retrieve the computed result from it. Sometimes, the input and output devices can communicate among themselves. In general, computer systems have I/O ports; I/O devices are connected to these ports for data transfer. Basically, the ports are digital registers that allow the computer to transfer data between the I/O devices using additional control signals. These control signals allow error-free transfer of data.

The common input device used in almost all systems is the keypad. Microprocessor-based basic microcomputer systems use simple numeric keypads. However, advanced computer systems use keyboards with a large number of keys involving alphabets, numbers, and special characters. Nowadays, a number of optical devices and scanners such as mouse, joystick, and bar code scanners are also being used as input devices. Microcomputer systems also use different types of sensors for data input. These sensors need data converters such as analog to digital converters. Any introductory course on microprocessors should cover the interfacing of data converters, keypads, and switches.

An output device is a device through which the user can receive the results from the computer. The output can be a rapidly changing display or printed material. Other forms of output are sounds and alarms. The simplest output devices, used in almost all microprocessor-based systems and computer systems, are LEDs, seven-segment LED displays, and LCD displays. The advanced video display terminals (either cathode-ray tubes or LCDs) and ink-jet and laser printers are the common output devices nowadays. Some output devices can be used to directly control machineries. Some devices, such as display terminals with touch screen, may provide both input and output. Modems and other network interface cards can also be called output devices as they enable the transmission and reception of data between computers.

1.9 TECHNOLOGY IMPROVEMENTS ADAPTED TO MICROPROCESSORS AND COMPUTERS

Technological improvements are taking place rapidly in microprocessor, microcomputer, and personal computer systems. Some of these improvements are listed here:

(i) Increase in data bus/address bus width: The processing capability of the microprocessor can be drastically improved by increasing data size. This development can be seen clearly from the advancements in microprocessors (Section 1.5).

(ii) Increase in speed: As the data to be processed by the microprocessors and computers increased in volume, it became necessary to increase the speed of the processor. With high speed processors, the user can get results quickly, even with large data volumes.

(iii) Reduction in size and increase in capability: The trend in microprocessor technology is to include a large number of peripherals such as memory and I/O ports within a single chip. Microcontrollers are manufactured in this fashion. In addition, developments in large scale integration have led to the manufacture of small microprocessor chips with large built-in peripherals. Processors with a large amount of flash memory are now available in the market.

(iv) Development of external peripherals: The use of computers in all fields have resulted in the development of many fast and advanced peripheral devices. For example, the application of microprocessors in medicine has resulted in the development of many handheld electronic devices with specialized input sensors, output printers, etc. Faster peripherals can increase the speed of processor execution and provide a good user interface.

(v) Increase in memory unit size and speed: The developments in IC technology have led to a reduction in the size of the memory units and an increase in memory speed. This reduces the memory access time of the processor and results in higher speed of execution. More amount of memory per unit area is possible.

(vi) Microprocessors are largely used in handheld devices operated from a battery source. This has resulted in research on the reduction of power consumption in microprocessor chips. As power consumption is reduced, these devices work for more time once the batteries are fully charged. There are many devices operating at 3.3 V or even lower voltages and have low power consumption.

1.10 INTRODUCTION TO 8085 PROCESSOR

The microprocessor is a semiconductor device consisting of electronic logic circuits manufactured using either large-scale integration (LSI) or very large-scale integration (VLSI) technique. It basically contains registers, an arithmetic and logic unit, flip-flops, and timing and control circuits. All microprocessors work using Von–Neumann architecture. In this architecture, the CPU or the processor

fetches instructions from the memory, decodes it (i.e., interprets the nature of the instruction/command and develops clock-synchronized steps for execution), generates appropriate control signals, and finally executes it. The program is stored in consecutive memory locations. The execution steps are repeated for all the instructions of the program until the execution is terminated by hardware or software. The data required may be taken either from memory or from input ports; the results of the program may be either stored in the memory or transferred out through output ports.

A program is a list of instructions for the microprocessor to execute. Before the start of execution, the complete program must be stored in the memory. Let us assume that the starting address of the stored program is 8800H. While running the program, the microprocessor must be directed to 'go' from 8800H. Once it has executed the instruction in 8800H, it goes to the next address 8801H (assuming single-byte instructions) and so on until it reaches the end of the program.

Intel 8085 is an 8-bit microprocessor manufactured by Intel Corporation and is usually called a general-purpose 8-bit processor. It is upward compatible with microprocessor 8080, which was Intel's earlier product. There are several faster versions of the 8085 microprocessor such as 8085AH, 8085AH-1, and 8085AH-2.

A microprocessor system consists of three functional blocks—central processing unit (CPU), input and output units, and memory units, as shown in Fig. 1.4. The CPU contains several registers, an arithmetic and logic unit (ALU),

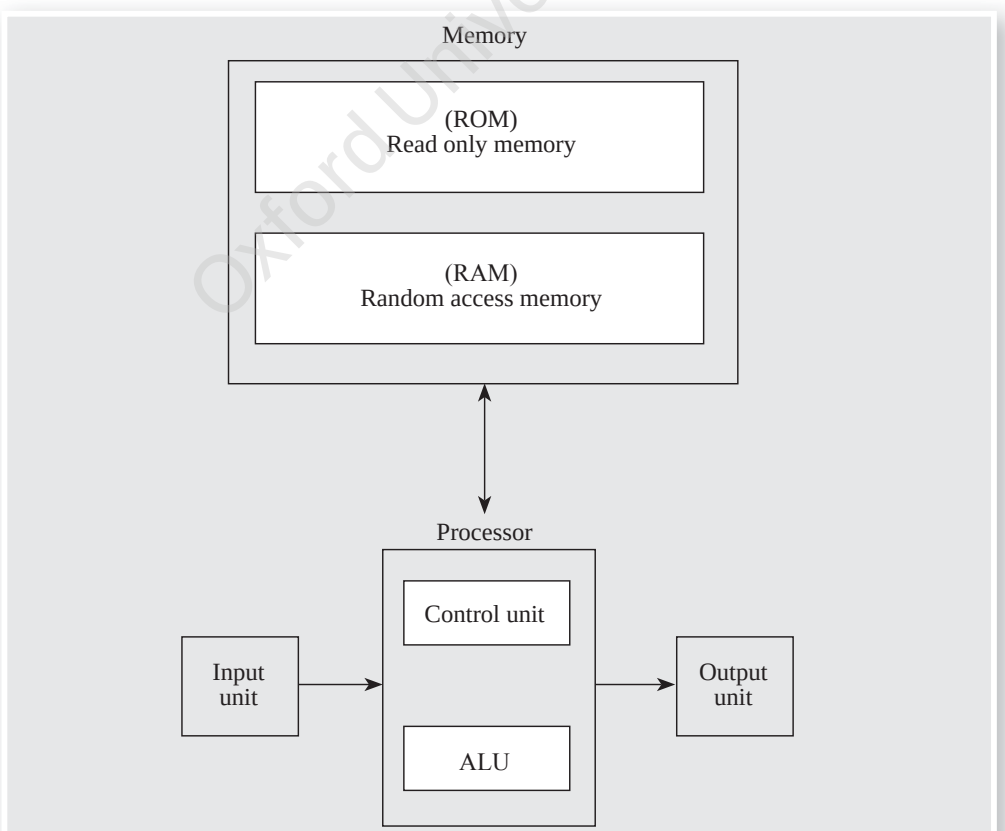


Fig. 1.4 A microprocessor system

and a control unit. The function of ALU, as the name implies, is to perform arithmetic and logical operations. The control unit translates the instructions and executes the desired task.

1.11 ARCHITECTURE OF 8085

The block diagram explaining the architecture of Intel 8085 microprocessor is shown in Fig. 1.5. It is generally available as a 40-pin IC package and uses +5 V for power. It can run at a maximum frequency of 3 MHz. The modified versions of the 8085 processor have these minimum common features and functional similarities.

The 8085 is called an 8-bit processor since its data length and data bus width is eight bits. It has an addressing capability of 16 bits, i.e., it can address $2^{16} = 64$ KB of memory (1 KB = 1024 bytes). The processor contains five functional units:

(i) Arithmetic and logic unit
(ii) General-purpose registers
(iii) Special-purpose registers
(iv) Instruction register and decoder
(v) Timing and control unit

1.11.1 Arithmetic and Logic Unit

ALU is the circuitry that performs the actual numerical and logical operations. Addition (ADD), subtraction (SUB), increment (INR), decrement (DCR), and comparison (CMP) are the arithmetic operations possible in the 8085

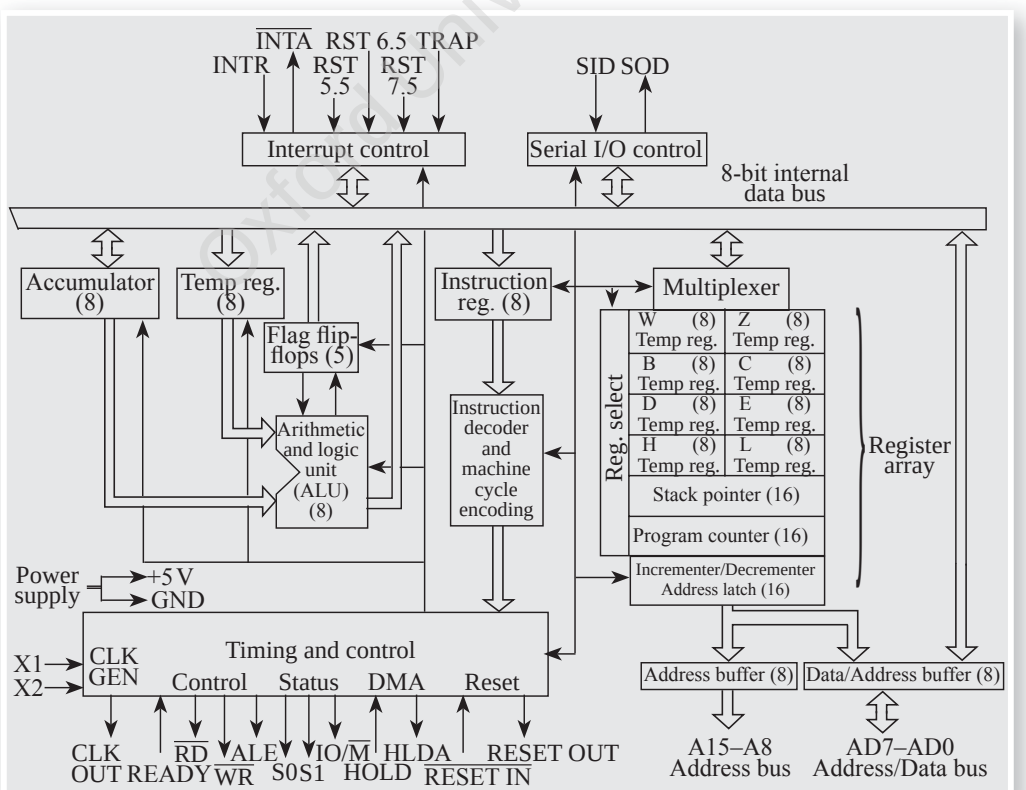


Fig. 1.5 Functional block diagram of Intel 8085

microprocessor. The possible logical operations are AND (AND), OR (OR), exclusive OR (EXOR), complement (CMA), etc.

The ALU of the 8085 processor is called accumulator-oriented ALU as one of the data used in arithmetic and logic operations must be stored in the accumulator. The other data is taken from a memory location or register. The results of the arithmetic and logical operations are stored in the accumulator. If the operation needs only one data, that data must be stored in the accumulator.

1.11.2 General-purpose Registers

A register is a collection of eight D-type flip-flops with parallel-in and parallel-out operation. A flip-flop can only store one bit at a time. Therefore, to handle eight bits at a time, eight flip-flops are required and hence the term 8-bit register. Though the registers are all storage areas inside the microprocessor, they differ in the purpose of storage. The general-purpose registers are used to store only the data that is being used by the program under execution and the results obtained from it. These general-purpose registers are user accessible through programs.

Registers B, C, D, E, H, and L are the general-purpose registers in the 8085, as shown in Fig. 1.6. They can also be called scratchpad registers. In almost all arithmetic and logical operations, these registers are used as the second operands, the first operand being the accumulator (A). The general-purpose registers are all 8-bit registers but they can be handled as 16-bit registers as well. This can be achieved by combining the register pairs B and C, D and E, and H and L to perform 16-bit operations. They are then named register pairs BC, DE, and HL, respectively.

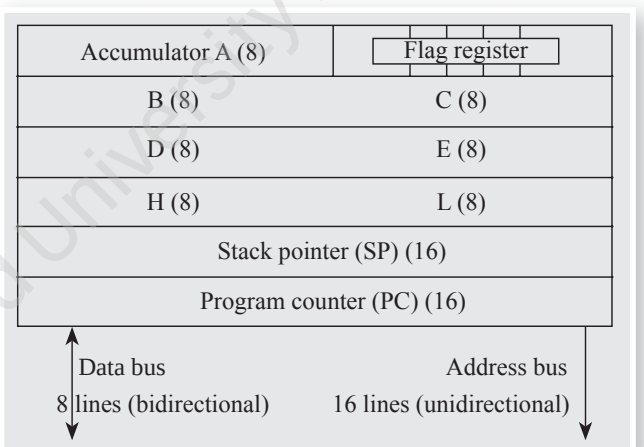


Fig. 1.6 Registers of Intel 8085

Among these pairs, HL has a special significance. A few memory-related instructions of the 8085 (refer instruction set) use the HL pair as a memory pointer. For example, the instruction MOV A, M transfers the content of the memory location to which the HL pair is pointing, to the accumulator. The HL pair is pre-loaded with the memory address in which data is available.

1.11.3 Special-purpose Registers

There are also special-purpose registers that are dedicated to a specific function. The accumulator, flag register, program counter (PC), and stack pointer (SP) constitute the special registers in the 8085 microprocessor.

1.11.3.1 *Accumulator*

The accumulator is an 8-bit register; it is a part of the ALU and is the most

important register. It is used to store 8-bit data and to perform arithmetic and logical operations. The output of an operation is also stored in the accumulator. The accumulator is identified as register A in the instruction set of the 8085. The programmer can use it at any time to store an 8-bit binary number. Being only eight bits long, it can only hold one byte at a time. Any previous data stored in this register will be overwritten as soon as new data is stored. The 8085 microprocessor communicates with input/output devices only through the accumulator.

1.11.3.2 *Flag Register*

This is a special 8-bit register. Each bit of the flag register is quite independent of the others. In all other registers, each bit is part of a single binary byte value and hence each bit would have a numerical value. The flag is an 8-bit register used to indicate the status of a recent arithmetic or logical operation. It may be set or reset after an arithmetic or logical operation according to the condition of the processed data. The five flag bits are zero (Z), carry (CY), sign (S), parity (P), and auxiliary carry (AC); their bit positions in the flag register are shown in Fig. 1.7. The remaining three bits (D1, D3, and D5) of the flag register remain unassigned; they are marked with an X to show that they are not used and are *don't cares*.

S	Z	X	AC	X	P	X	CY
D7	D6	D5	D4	D3	D2	D1	D0

Fig. 1.7 Flag register

Any flag register bit is said to be 'set' when its value is 1 and 'cleared' when its value is 0. The most commonly used flags are zero, carry, and sign. AC flag cannot be accessed externally.

Sign flag (S) The sign flag is just a copy of the bit D7 (most significant bit—MSB) of the accumulator. A negative number has a 1 in bit 7 and a positive number has a 0 in 2's complement representation. This flag indicates the sign of the number. (It may be recalled that signed magnitude numbers use 1 to indicate a negative number and 0 to indicate a positive number.) This flag can be used in signed arithmetic operations.

Zero flag (Z) The zero flag is set if an arithmetic operation results in a zero. It sets, i.e., it changes to binary 1 if the result in the accumulator is zero; if not, it remains reset, i.e., at binary 0.

Carry flag (C) The carry flag is set when a carry is generated in the process of an arithmetic operation in the accumulator. When addition is carried out, it sometimes results in a ninth bit being carried over to the next byte. The C flag copies the value of the carry, which is an extra bit, from D7. It also reflects the value of the borrow in subtractions.

Auxiliary carry flag (AC) The auxiliary carry flag is set when an auxiliary carry is generated in the process of an arithmetic operation in the accumulator, i.e., when a carry results from bit D3 and passes on to D4 (from the lower nibble to the higher nibble). This carry is also called half-carry. It may also occur in the process of a subtraction operation. In other words, this flag is set if the subtraction operation results in borrowing from the higher nibble.

Parity flag (P) The parity flag is set if the content of the accumulator after an arithmetic operation has an even number of 1s. Otherwise, the parity flag is reset. It is set for operation in the even parity mode.

1.11.3.3 *Program Counter*

Program counter (PC) is a 16-bit register that always points to the address of the next instruction to be executed. In other words, this register is used to sequence the execution of the instructions. After execution of every instruction, the content of the memory location indicated by the PC is moved to the instruction register and the PC is loaded with the next address. It keeps track of a program by counting the memory address from which the next byte is to be fetched, and hence the name program counter.

1.11.3.4 *Stack Pointer*

Stack is an array of memory locations organized in last-in, first-out (LIFO) or first-in, last-out (FILO) fashion. It is accessed using a 16-bit pointer register called stack pointer (SP), which holds the address of the memory location of the top of the stack. The programmer can reserve and allocate a series of RAM locations to be used as a stack and accordingly initialize the stack pointer. The range of stack memory locations must be chosen carefully so that it does not affect the program space. In all microprocessor-based systems, the stack is mainly used to store the return address of the main program when a subroutine is called. While the programmer uses the stack for storage and retrieval of data, the microprocessor uses the stack during subroutine calls. Care must be taken by the programmer to ensure that the data stored in the stack is retrieved properly, so that the data stored in the stack by the processor is not affected.

1.11.4 Instruction Register and Decoder

It is an 8-bit register that temporarily stores the instructions drawn from memory locations, before their actual execution. The content of the register is decoded by the decoder circuitry, where the nature of the operation to be performed is decided (interpreted). In addition, there are two temporary registers W and Z, which are controlled internally and not available for user access.

1.11.5 Timing and Control Unit

The timing and control unit gets commands from the instruction decoder and issues signals on the data bus, address bus, and control bus. The following sections explain the operation of the various buses and the timing.

A typical microprocessor communicates with memory and input/output devices using buses. There are three types of buses—the address bus, the data bus, and the control bus.

1.11.5.1 *Data Bus*

The microprocessor performs its functions using wires or lines called buses. For example, an 8-bit microprocessor normally uses eight wires to carry data between the microprocessor and the memory. To make their representation simple, the data wires with common functions are grouped together and referred to as the data bus.

The data bus (D0–D7) is a two-way bus carrying data around the system. Information going into the microprocessor and results coming out of the microprocessor are through this data bus. It is used for transfer of binary information between the microprocessor, memory, and peripherals. The lower group of eight address lines A0–A7 is multiplexed with the data bus in order to reduce the pin count. Therefore, the multiplexed lower group of address lines and data lines is more generally denoted as AD0–AD7.

1.11.5.2 *Address Bus*

The address bus carries addresses and is a one-way bus from the microprocessor to the memory or other devices. It is a group of sixteen unidirectional lines that allows flow of address from the processor to its peripheral devices. Each peripheral and memory location is identified by a 16-bit binary number called address. It follows that the maximum number of memory locations that can be addressed by the 8085 processor is 2^{16} bytes = 64 KB. Its basic function is to identify a peripheral or memory location.

The address bus lines are generally identified as A0–A15. The address bus has eight higher-order address lines (A8–A15), which are unidirectional. The lower-order eight lines (A0–A7) are multiplexed (time-shared) with the eight data bits (D0–D7) and hence, they are bidirectional. When the instruction is executed, these lines carry the address bits during the early part, and the eight data bits during the later part. To separate the address from the data, a latch is used externally to save the address before the function of the bits changes.

1.11.5.3 *Control Bus*

The control bus carries control signals that are partly unidirectional and partly bidirectional. For a microprocessor to function correctly, these control signals are vital. The control bus typically consists of a number of single lines that coordinate and control microprocessor operations. For example, a read/write control signal will indicate whether memory is being written into or read from. Thus, they are individual lines that provide a pulse to indicate the operation of the microprocessor. In fact, the microprocessor generates specific control signals for every operation, which in turn are used to identify the type of device the processor intends to communicate with. The following points describe the control and status signals of the 8085 processor:

(i) ALE (output): Address Latch Enable is a pulse that is provided when an address appears on the AD0–AD7 lines, after which it becomes 0. This signal can be used to enable a latch to save the address bits from the AD lines, thereby de-multiplexing the address bus and data bus.

(ii) $\overline{\text{RD}}$ (active low output): The Read signal indicates that data are being read from the selected I/O or memory device and that they are available on the data bus.

(iii) $\overline{\text{WR}}$ (active low output): The Write signal indicates that the data on the data bus are to be written into a selected memory or I/O location.

(iv) IO/$\overline{\text{M}}$ (output): It is a signal that distinguishes between a memory operation and an I/O operation. An active low on this signal shows it is a memory

operation (IO/$\overline{M}$ = 0) and a high on this line indicates an I/O operation (IO/$\overline{M}$ = 1).

(v) S1 and S0 (output): These are status signals used to specify the kind of operation being performed. The status signals combine with I/O signals to govern various operations; they are listed in Table 1.2. If both S0 and S1 are low, the operation of the processor tends to halt. If S0 is low and S1 is high, the processor reads data. If S0 is high and S1 is low, the processor writes data onto a memory or I/O device. If both S0 and S1 are high, the fetch operation is performed.

Table 1.2 Status signals and associated operations

S1	S0	States
0	0	Halt
0	1	Write
1	0	Read
1	1	Fetch

The schematic representation of the 8085 bus structure, shown in Fig. 1.8, explains how the movement of data within the computer is accomplished by a series of buses. Address information, data, and control signals have to be carried around inside the microprocessor as well as in the external system. Hence, the buses are present both internally and externally.

(vi) Interrupts: These signals are used to make the microprocessor respond to high priority externally initiated signals. When an interrupt signal is detected by the processor, it suspends the execution of the current program and executes the program corresponding to the interrupt signal instead. Five interrupt signals (INTR, RST 5.5, RST 6.5, RST 7.5, and Trap) are available to facilitate the processor to receive and acknowledge the interrupt call of peripherals. The 8085 processor accepts three more externally initiated signals—$\overline{\text{RESET IN}}$, Hold, and Ready as inputs. The following points explain these signals in brief:

(a) INTR (input): It is a general-purpose interrupt request signal. It is an active high signal.

(b) $\overline{\text{INTA}}$ (output): It is used to acknowledge an interrupt. It is an active low signal.

(c) Restart interrupts (input): These are vectored interrupts that transfer the

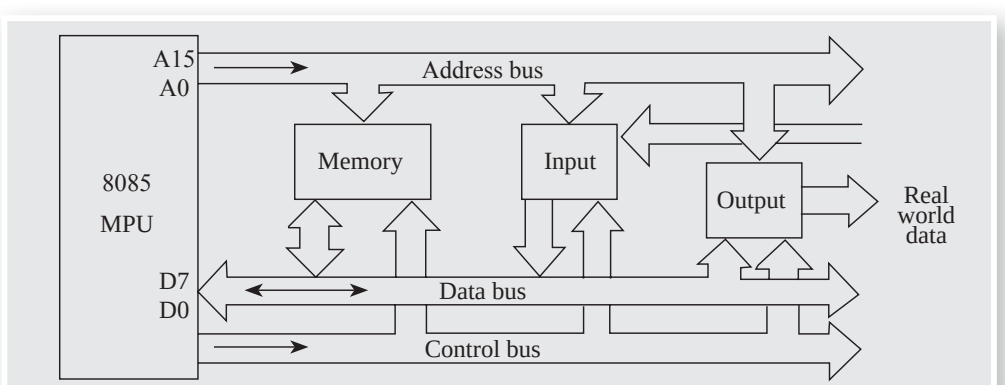


Fig. 1.8 Schematic representation of the 8085 bus structure

program control to specific memory locations. They have higher priority than INTR interrupts. The priority order is RST 7.5, RST 6.5, and RST 5.5.

(d) Trap (input): It is a non-maskable interrupt, i.e., it cannot be stopped or overridden by any command. It has the highest priority among all 8085 interrupts.

(e) $\overline{\text{RESET IN}}$ (input): When the signal on this pin goes low the program counter is set to zero and the processor is reset. It is an active low signal.

(f) RESET OUT (output): This signal can be used to reset other devices that are connected to the processor. It is an active high signal.

(g) Hold (input): This signal indicates that a peripheral such as a direct memory access (DMA) controller is requesting the use of the address and data buses.

(h) HLDA (output): It is an acknowledge signal that is sent in response to the Hold request. During the Hold state, the peripheral (I/O) devices get control over the data and address buses for data transfer to and from memory. This operation is called direct memory access (DMA). DMA is useful when high-speed peripherals want to transfer data to and from memory. The processor does not intervene during this period.

(i) Ready (input): It is a signal that serves to delay the microprocessor read/write signals until a slow-responding peripheral is ready to send or accept data. If this signal goes low, then the processor is allowed to wait for an integral number of clock cycles until Ready becomes high. The Ready signal must be synchronized with the processor clock.

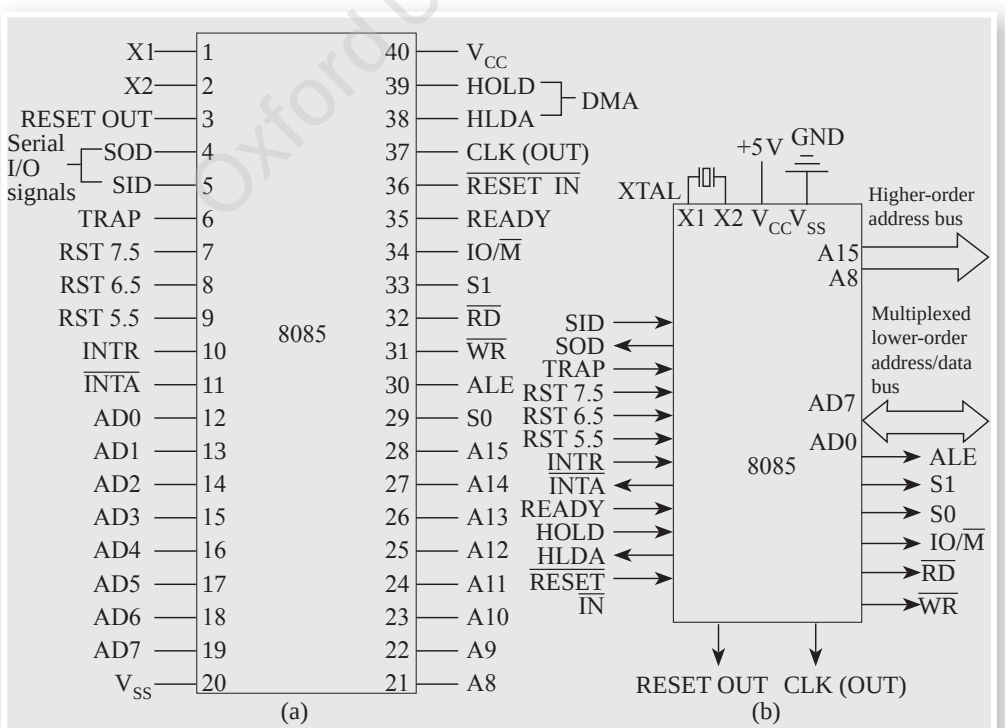


Fig. 2.7 8085 details (a) Pin diagram (b) Signal groups

The typical pin layout and signal groups of the 8085 microprocessor are shown in Figs 1.9 (a) and 1.9 (b), respectively. The 8085 is available with 40 pins as a dual in-line package (DIP).

Intel 8085 has 40 pins, operates at 3 MHz clock frequency, and requires +5 V for power supply.

1.11.5.4 *Serial I/O Signals*

There are two signals to implement serial transmission. They are serial input data (SID) and serial output data (SOD). The data bits are sent over a single line, one bit at a time, in serial transmission.

(i) SID (input): The bit data on this line is loaded in the seventh bit of the accumulator whenever a RIM instruction is executed.

(ii) SOD (output): The output SOD is set or reset as specified by the SIM instruction.

RIM and SIM instructions have been explained in detail in Chapter 5.

1.11.5.5 *Power Supply and System Clock*

The following pins are available in the 8085 chip to provide power and clock signal to the processor:

(i) X1, X2 (input): A microprocessor needs a square wave (clock) signal to ensure that all internal operations are synchronized. A crystal or R–C or L–C network is connected to these two pins. The crystal frequency is internally divided by two to give the operating system frequency. There are three advantages in increasing the frequency of a crystal—as frequency increases, the crystal size becomes smaller, and the crystal becomes lighter and cheaper. Therefore, clock circuits include a divide-by-two circuit so that a double-frequency crystal can be used. So, to run the microprocessor at 3 MHz, a 6 MHz crystal should be connected to the X1 and X2 pins. The crystal is preferred as a clock source because of its high stability, large Q (quality factor), and absence of frequency drifting with aging. Without a clock signal, the microprocessor cannot execute any program.

(ii) CLK (output): This output clock pin is used to provide the clock signal to the rest of the system.

Power supplies: V_{CC}—+5 V supply; V_{SS}—ground reference.

1.12 MICROPROCESSOR INSTRUCTIONS

Every microprocessor has its own instruction set. Based on the design of the ALU and the decoding unit, microprocessor manufacturers generally list out the instructions for every microprocessor manufactured. The instruction set consists of both assembly language mnemonics and the corresponding machine code.

The purpose of the instruction set is to facilitate the development of efficient programs by the users. The instruction set is based on the architecture of the processor. So to understand the instruction set of a processor, it is necessary to understand the basic architecture of the microprocessor and the user-accessible

registers in it. An *instruction* is a bit pattern that is decoded inside a microprocessor to perform a specific function. The assembly language mnemonics are the codes for these binary patterns so that the user can easily understand the functions performed by these instructions. The entire group of instructions that a microprocessor can handle is called its *instruction set*; this determines the microprocessor's functionality. The Intel 8085 processor has its own set of instructions listed both in mnemonics and machine code, also called as object code. As the 8085 is an 8-bit processor, the machine codes for the instructions are also 8 bits wide.

The syntax for 8085 instructions may contain one or more of the following notations:

R = 8-bit register (A, B, C, D, E, H, and L)
Rs = Source register
Rd = Destination register } (A, B, C, D, E, H, and L)
Rp = Register pair (BC, DE, HL, and SP)
P = Port address (8-bit binary number or two hex digits)
8-bit = 8-bit data or two hex digits
16-bit = 16-bit data/address or four hex digits
( ) = Contents of

1.13 CLASSIFICATION OF INSTRUCTIONS

Microprocessor instructions can be classified based on parameters such as functionality, length, and operand addressing.

1.13.1 Based on Functionality

Based on the functionality, the instructions are classified into the following five categories:

(i) Data transfer (copy) operations
(ii) Arithmetic operations
(iii) Logical operations
(iv) Branching operations
(v) Machine control operations

1.13.1.1 *Data Transfer (Copy) Operations*

This group of instructions copies data from a location called source register to another location called destination register. Generally, the contents of the source register are not modified. Although the term *data transfer* is used for the copy operation, it is misleading because it implies that the contents of the source memory location are destroyed. The various types of data transfer are listed in Table 1.3 along with examples of each type.

Table 1.3 Types of data transfer

Type	Example
Transferring data between one register and another	MOV A, D—Copies the content of register D to the accumulator
Storing a data byte in a register or memory location	MVI C, 66H—Loads register C with the data 66H

(Contd)

Table 1.3 Types of data transfer *(Contd)*

Type	Example
Transferring data between a memory location and a register	LDA 8800H—Loads the contents of memory location 8800H in the accumulator
Transferring data between an I/O device and the accumulator	IN PORT1—Transfers data from an input device to the accumulator

1.13.1.2 *Arithmetic Operations*

Arithmetic operations include addition, subtraction, increment, and decrement. As the 8085 has an accumulator-oriented ALU, one of the data used in the arithmetic operations is stored in the accumulator; the result is also stored in the accumulator. Arithmetic and logical operations cannot be executed without the accumulator.

Addition (ADD) The addition instructions of the 8085 add the contents of a register or memory location with the contents of the accumulator. The result is stored in the accumulator. The Intel 8085 instruction set supports two types of addition instructions—with and without addition of the carry flag content to the least significant bit of the numbers. The instruction set also supports 16-bit addition, i.e., the content of the HL register pair can be added to that of another register pair and the result stored in the HL register pair.

Subtraction (SUB) The instruction set of the 8085 supports two types of subtraction—with borrow and without borrow. Like addition, the subtraction operation also uses the accumulator as reference, i.e., it subtracts the content of a register or memory location from that of the accumulator and stores the result in the accumulator.

Increment/Decrement These operations can be used to increment or decrement the contents of any register, register pair, or memory location. Unlike the arithmetic and logical operations, the increment and decrement operations need not be based upon the accumulator.

1.13.1.3 *Logical Operations*

Logical instructions are also accumulator-oriented, i.e., they require one of the operands to be placed in the accumulator. The other operand can be any register or memory location. The result is stored in the accumulator. The operations that use two operands are logical AND, OR, and EXOR. The operation that uses a single operand (i.e., the accumulator) is the logical complement or NOT operation.

The instruction set of the 8085 supports rotation of the data stored in accumulator. The data can be rotated left or right, through the carry or without the carry.

The most important 8085 instruction is the compare instruction. This instruction is used to compare register or memory content with the accumulator content. The result of comparison such as equal to, greater than, or less than is reflected in the flag register bits.

1.13.1.4 *Branching Operations*

Branching instructions are important for programming a microprocessor. These

instructions can transfer control of execution from one memory location to another, either conditionally or unconditionally. Branching can take place in the following two ways:

(i) Execution control cannot return to the point of branching. Example: Jump instructions
(ii) Execution control can return to the point of branching, which is stored by the 8085. Example: Subroutine call instructions

1.13.1.5 *Machine Control Operations*

These instructions can be used to control the execution of other instructions. They include halting the operation of the microprocessor, interrupting program execution, etc. Detailed explanations for 8085 instructions are given in Section 1.14.

1.13.2 Based on Length

Based on the length of the machine language code, 8085 instructions can be classified into the following three types:

(i) One-byte instructions (ii) Two-byte instructions
(iii) Three-byte instructions

Assembly language instructions should be converted into machine code for storage and execution by the processor. So the length of the machine language code instructions determines the length of the program. This in turn determines the amount of memory required for the program.

1.13.2.1 *One-byte Instructions*

Instructions that require only one byte in machine language are called one-byte instructions. These instructions just have the machine code or opcode alone to represent the operation to be performed. The common examples are the instructions that have their operands within the processor itself. Some examples of one-byte instructions are given in Table 1.4. Even though the instruction ADD M adds the content of a memory location to that of the accumulator, its machine code requires only one byte.

Table 1.4 One-byte instructions

Opcode	Operand	Machine code/Opcode/ Hex code
MOV	A, B	78
ADD	M	86
XRA	A	AF

Let us now understand the instruction MOV Rd, Rs. This instruction copies the contents of source register Rs to destination register Rd. (Rd ← Rs)

It is coded as 01dddsss. Here, ddd is the binary code of one of the seven general-purpose registers that is the destination of the data and sss is the binary code of the source register.

Example:

```
MOV A, B  (coded as 01111000 = 78H)
```

1.13.2.2 *Two-byte Instructions*

Instructions that require two bytes in machine code are called as two-byte instructions. The first byte of the two-byte instructions is the opcode, which

specifies the operation to be performed. The second byte is the 8-bit operand, which is either an 8-bit number or an address. Some common examples of two-byte instructions are listed in Table 1.5.

Table 1.5 Two-byte instructions

Opcode	Operand	Machine code/Opcode/Hex code	Byte description
MVI	A, 7FH	3E	First byte
		7F	Second byte
ADI	0FH	C6	First byte
		0F	Second byte
IN	40H	DB	First byte
		40	Second byte

The instruction is stored in two consecutive memory locations.

MVI R, data—(R ← data)

Example:

```
MVI A, 32H (coded as 3E 32 in two contiguous bytes)
```

This is an example of immediate addressing.

The following two instructions are also examples of two-byte instructions:

(i) ADI data (A ← A + data)

(ii) OUT port (where port is an 8-bit device address. (Port) ← A) Since the byte is not the data itself, but points directly to where it is located, this is called direct addressing. For a detailed account of addressing modes, see Section 1.13.3.

1.13.2.3 *Three-byte Instructions*

Instructions that require three bytes in machine code are called three-byte instructions. In 8085 machine language, the first byte of the three-byte instructions is the opcode which specifies the operation to be performed. The next two bytes refer to the 16-bit operand, which is either a 16-bit number or the address of a memory location. Some common examples of three-byte instructions are listed in Table 1.6.

Table 1.6 Three-byte instructions

Opcode	Operand	Machine code/Opcode/Hex code	Byte description
JMP	9050H	C3	First byte
		50	Second byte
		90	Third byte
LDA	8850H	3A	First byte
		50	Second byte
		88	Third byte
LXI	H, 0520H	21	First byte
		20	Second byte
		05	Third byte

The instruction LXI Rp, 16-bit data can be explained as follows:

Rp is one of the pairs of registers BC, DE, or HL, which are used as 16-bit registers. The two data bytes are to be stored as a 16-bit number in L and H in sequence. LXI H, 0520H is coded as 21H 20H 05H in three bytes. (This is an example of immediate addressing.)

In executing the instruction LDA addr, the accumulator is loaded with the memory content of the address given in the instruction. Addr is a 16-bit address. LDA 8850H is coded as 3AH 50H 88H. (This is an example of direct addressing.)

1.13.3 Addressing Modes in Instructions

Every instruction in a program has to operate on data. The process of specifying the data to be operated on by the instruction is called *addressing*. Efficient software development for the microprocessor requires complete familiarity with the addressing mode employed for each instruction. For example, the instructions MOV B, A and MVI A, 82H are used to copy data from a source to a destination. In these instructions, the source can be a register or an 8-bit number (00H to FFH); the destination is a register. The source and destination are operands. The various formats for specifying operands are called *addressing modes*. The 8085 has the following five types of addressing:

(i) Immediate addressing
(ii) Memory direct addressing
(iii) Register direct addressing
(iv) Indirect addressing
(v) Implied or implicit addressing

1.13.3.1 *Immediate Addressing*

Immediate addressing transfers the operand given in the instruction—a byte or word—to the destination register or memory location. The operand is part of the instruction. The format for immediate addressing is given in Fig. 1.10.

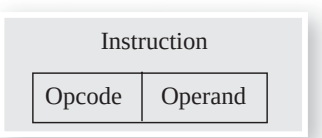


Fig. 1.10 Format of immediate addressing

Example:

```
MVI A, 9AH
```

(a) The operand is part of the instruction.
(b) The operand is stored in the register mentioned in the instruction.

Example:

```
ADI 05H
```

(a) Add 05H to the contents of the accumulator.
(b) 05H is the operand.

Immediate addressing has no memory reference to fetch data. It executes faster, but has limited data range.

1.13.3.2 *Memory Direct Addressing*

Memory direct addressing moves a byte or word between a memory location and register. The memory location address is given in the instruction. The instruction set does not support memory-to-memory transfer. Memory direct addressing is illustrated in Fig. 1.11.

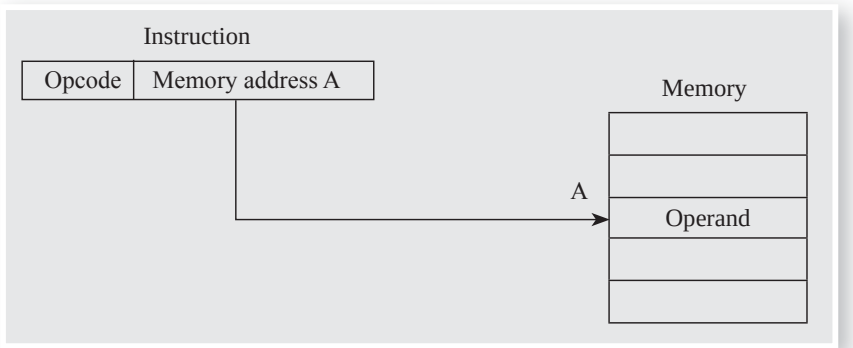


Fig. 1.11 Format of memory direct addressing

Example:

```
LDA 850FH
```

This instruction is used to load the contents of the memory location 850FH in the accumulator.

Example:

```
STA 9001H
```

This instruction is used to store the contents of the accumulator in the memory address 9001H.

In these instructions, the memory address of the operand is given in the instruction.

Direct addressing is also used for data transfer between the processor and input/output devices. For example, the IN instruction is used to receive data from the input port and store it in the accumulator; the OUT instruction is used to send the data from the accumulator to the output port.

Example:

```
IN 00H and OUT 01H
```

1.13.3.3 *Register Direct Addressing*

Register direct addressing transfers a copy of a byte or word from the source register to the destination register. The operand is in the register named in the instruction. It executes very fast, has very limited register space, and requires good assembly programming. The operand is within in the processor itself; so the execution is faster. Register direct addressing is illustrated in Fig. 1.12.

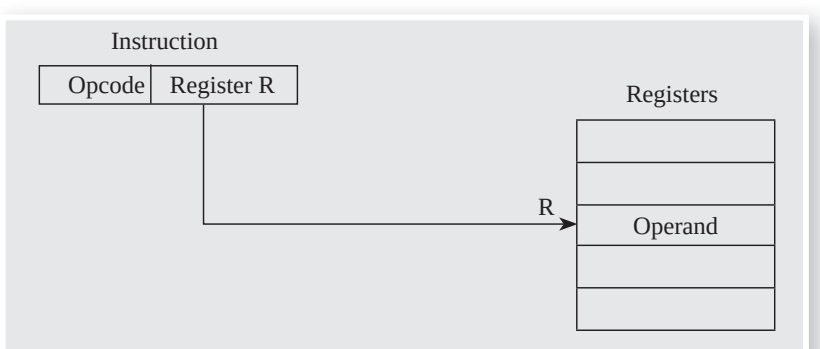


Fig. 1.12 Format of register direct addressing

Example:

```
MOV Rd, Rs
MOV B, C
```

It copies the contents of register C to register B.

Example:

```
ADD B
```

It adds the contents of register B to the accumulator and saves it in the accumulator.

1.13.3.4 *Indirect Addressing*

Indirect addressing transfers a byte or word between a register and a memory location. The address of a memory location is stored in a register and that register is specified in the instruction. This is illustrated in Fig. 1.13.

In indirect addressing, the effective address is calculated by the processor using the contents of the register specified in the instruction. This type of addressing employs several accesses—two accesses to retrieve the 16-bit address and a further access (or accesses) to retrieve the data which is to be loaded in the register.

Example:

```
MOV A, M
```

Here, the data is in the memory location pointed to by the contents of the HL pair. The data is moved to the accumulator.

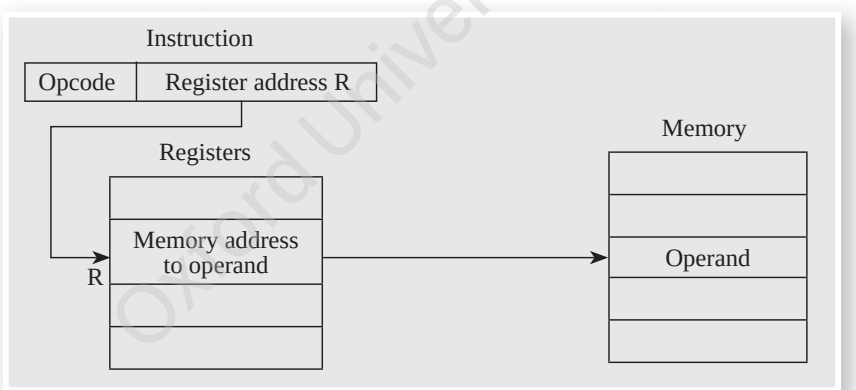


Fig. 1.13 Format of indirect addressing

1.13.3.5 *Implied or Implicit Addressing*

In implied addressing mode, the instruction itself specifies the data to be operated upon. For example, CMA complements the contents of the accumulator. No specific data or operand is mentioned in the instruction.

1.14 INSTRUCTION SET OF 8085

The 8085 microprocessor instruction set has 74 operation codes and 246 instructions. It is compatible with that of its predecessor, the 8080A, but has two additional instructions—SIM (set interrupt mask) and RIM (read interrupt mask)—related to serial I/O. The complete instruction set is listed in Appendix 1 with additional information such as number of clock states required for execution and the flags affected.

1.14.1 Format of Assembly Language Instructions and Programs

Assembly language programs are written for performing specific functions, converted into machine language code, and then stored in the memory of the microprocessor-based system. The conversion of an assembly language program into machine language code is called assembling; the application that performs this task is called assembler. This conversion or assembling can also be done manually by the programmers. To facilitate the process of assembling, the assembly language programs are written in a specific format as shown in Fig. 1.14.

Memory address	Machine code/Opcode	Label	Mnemonics with operands	Comments

Fig. 1.14 Format for writing assembly language programs

In general, the assembly language mnemonics with their operands are written first. The address where the instructions are stored is given a dummy name called label. The purpose of labels is to give the correct branch addresses in instructions. Labels are separated from mnemonics with a colon.

The comments column is essential for any program as it helps the programmer understand the logic of the program at any point in time. Without comments, it is difficult to understand an assembly language program. Comments are separated from the mnemonics with a semicolon.

The first two columns correspond to the physical memory address and the actual machine code. These two columns are filled in after completing the assembly language programming. These columns must contain only binary numbers, but for easy understanding, hexadecimal numbers are used. For manual assembling, these two columns are filled in by the programmer. An assembler can generate these columns automatically.

An example of the assembly language program format is given in Table 1.7.

Table 1.7 Sample assembly language program

Memory address	Machine code/ Opcode	Label	Mnemonics with operands	Comments
8000 8001 8002	3E 5F	START:	MVI A, 5FH	; Load data in the accumulator. ; Address of the next memory location

The instruction in Table 1.7 moves the data 5FH to the accumulator.

1.14.2 Data Transfer Instructions

Data transfer instructions are used to transfer data between two registers in the microprocessor or between a peripheral device and the microprocessor. Some instructions and their features are given in the following points. The complete list with explanations is given in Table 1.8.

(i) MVI instruction is used for storing 8-bit data in a microprocessor register.
(ii) LXI instruction is used for storing 16-bit data in a register pair.

(iii) In direct addressing mode, MOV instruction is used for data transfer between registers. In indirect addressing mode, MOV is used for data transfer between a memory location and a register. If the instruction has M in the operand field, the memory location pointed to by the HL pair is considered for data transfer.

Table 1.8 Data transfer instructions

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
MVI R, 8-bit	Moves the 8-bit data to the register	Immediate	Two bytes	MVI B, 3FH
LXI Rp, 16-bit	Loads the 16-bit data in the register pair	Immediate	Three bytes	LXI B, 5AF3H
MOV Rd, Rs	Copies the data from the source register to the destination register	Register direct	One byte	MOV A, B
LDA 16-bit	Loads the accumulator with the data from the memory location indicated by the 16-bit address	Memory direct	Three bytes	LDA 905FH
LHLD 16-bit	Loads the H and L registers directly from the two consecutive memory locations indicated by the 16-bit address	Memory direct	Three bytes	LHLD 900AH
STA 16-bit	Stores the contents of the accumulator in the memory location indicated by the 16-bit address	Memory direct	Three bytes	STA 9050H
SHLD 16-bit	Stores the contents of the H and L registers in two consecutive memory locations indicated by the 16-bit address	Memory direct	Three bytes	SHLD 809FH
PUSH Rp	Pushes the contents of the register pair onto a stack	Register direct	One byte	PUSH B
POP Rp	Pops the top two memory locations of the stack onto a register pair	Register direct	One byte	POP H
OUT 8-bit	Outputs the data in the accumulator to the port indicated by the 8-bit address	I/O	Two bytes	OUT 40H

(Contd)

Table 1.8 Data transfer instructions *(Contd)*

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
IN 8-bit	Inputs the data from the port indicated by the 8-bit address to the accumulator	I/O	Two bytes	IN 30H
MOV Rd, M	Copies the contents of the memory location pointed to by the HL register pair to the register	Indirect	One byte	MOV B, M
MOV M, Rs	Copies the contents of the register to the memory location pointed to by the HL register pair	Indirect	One byte	MOV M, C
LDAX Rp	Loads accumulator with the contents of the memory location pointed to by the register pair	Indirect	One byte	LDAX B
STAX Rp	Stores the contents of the accumulator in the memory location pointed to by the register pair	Indirect	One byte	STAX D
XCHG	Exchanges the contents of the HL register pair with that of the D and E register pair	Implicit	One byte	XCHG
SPHL	Copies the contents of the H and L registers to the stack pointer	Implicit	One byte	SPHL
XTHL	Exchanges the contents of the HL register pair with the top of stack	Implicit	One byte	XTHL

(iv) LDA and STA use memory direct addressing mode and a 16-bit memory address as operand.

(v) LDAX and STAX use indirect addressing mode for data transfer. The operand given in the instruction is one of the register pairs BC or DE. Register pair HL is not used with LDAX due to the availability of the alternative instruction MOV A, M.

(vi) LHLD and SHLD are the instructions used to transfer 16-bit data between the HL register pair and two consecutive memory locations. For example, executing SHLD 9000H instruction will store the contents of L register in 9000H and the contents of H register in 9001H.

(vii) PUSH and POP instructions are used for data transfer between a register

pair and a stack. The stack is a set of memory locations configured as a last-in, first-out (LIFO) or first-in, last-out (FILO) array. The top of the stack locations is pointed to by a special register, the stack pointer, which is within the microprocessor. PUSH instruction will store the register pair given in the instruction to the top two memory locations of the stack. Similarly, POP instruction will copy the last two bytes stored in the stack to the register pair mentioned in the instruction. Care must be taken in using these instructions as the stack is configured as a LIFO array. Another instruction to store data in the stack is XTHL, which exchanges the top two memory locations of the stack with the contents of the HL register pair.

(viii) Stack pointer can be initialized using LXI or SPHL instructions. SPHL instruction will copy the contents of the HL register pair to the stack pointer.

(ix) IN and OUT instructions use 8-bit port addresses as operand. IN instruction is used to get data from the input port and the data obtained is stored in the accumulator. OUT instruction is used to issue data from the accumulator to an output port.

(x) XCHG instruction is used to exchange the contents of the HL and DE register pairs.

1.14.3 Arithmetic Instructions

The arithmetic instructions supported by the 8085 are addition, subtraction, and their variants. The arithmetic instructions are listed in Table 1.9.

Table 1.9 Arithmetic instructions

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
ADI 8-bit	Adds the 8-bit data to the contents of the accumulator	Immediate	Two bytes	ADI 30H
ACI 8-bit	Adds the 8-bit data and the carry flag to the contents of the accumulator	Immediate	Two bytes	ACI 4FH
SUI 8-bit	Subtracts the 8-bit data from the contents of the accumulator	Immediate	Two bytes	SUI 2AH
SBI 8-bit	Subtracts the 8-bit data and the borrow from the contents of the accumulator	Immediate	Two bytes	SBI 5CH
ADD R	Adds the contents of the register to the contents of the accumulator	Register direct	One byte	ADD C

(Contd)

Table 1.9 Arithmetic instructions *(Contd)*

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
ADC R	Adds the contents of the register and the carry to the contents of the accumulator	Register direct	One byte	ADC E
SUB R	Subtracts the contents of the register from that of the accumulator	Register direct	One byte	SUB B
SBB R	Subtracts the contents of the register and the borrow from that of the accumulator	Register direct	One byte	SBB C
DAD Rp	Adds the contents of the register pair to that of the H and L registers	Register direct	One byte	DAD B
INR R	Increments the register by 1	Register direct	One byte	INR B
INX Rp	Increments the register pair by 1	Register direct	One byte	INX B
DCR R	Decrements the register by 1	Register direct	One byte	DCR E
DCX Rp	Decrements the register pair by 1	Register direct	One byte	DCX D
ADD M	Adds the contents of the memory location pointed to by the HL register pair to that of the accumulator	Indirect	One byte	ADD M
ADC M	Adds the contents of the memory location pointed to by the HL register pair and the carry to that of the accumulator	Indirect	One byte	ADC M
SUB M	Subtracts the contents of the memory location pointed to by the HL register pair from that of the accumulator	Indirect	One byte	SUB M
SBB M	Subtracts the borrow and the contents of the memory location pointed to by the HL pair from that of the accumulator	Indirect	One byte	SBB M

(Contd)

Table 1.9 Arithmetic instructions *(Contd)*

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
INR M	Increments the memory location pointed to by the HL register pair by 1	Indirect	One byte	INR M
DCR M	Decrements the memory location pointed to by the HL register pair by 1	Indirect	One byte	DCR M
DAA	Converts the contents of the accumulator from binary to BCD (Decimal-Adjust Accumulator)	Implicit	One byte	DAA

The following points list some key features of arithmetic operations:

(i) For arithmetic operations, one of the data must be stored in the accumulator and the other given or addressed in the instruction.
(ii) Add-with-carry instructions are used for multi-byte and higher-order byte addition.
(iii) Similarly, subtract-with-borrow instructions are used in multi-byte and higher-order byte subtraction.
(iv) Increment and decrement instructions can be operated not only on the accumulator, but also on other registers including memory locations.
(v) The contents of a register pair can be incremented or decremented using INX and DCX instructions.
(vi) DAA is the 8085 instruction that supports BCD addition. The addition of BCD data is done like binary addition, using the ADD instruction. DAA is used to convert the result of the binary addition of BCD numbers into a BCD number. This instruction cannot be used to directly convert binary numbers into BCD numbers.

1.14.4 Logical Instructions

The most important logical instructions supported by the 8085 are AND, OR, EXOR, and NOT. The complete list is given in Table 1.10.

Table 1.10 Logical instructions

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
ANI 8-bit	The 8-bit data is logically ANDed with the contents of the accumulator.	Immediate	Two bytes	ANI 0FH
XRI 8-bit	The 8-bit data is logically EXORed with the contents of the accumulator.	Immediate	Two bytes	XRI 01H
ORI 8-bit	The 8-bit data is logically ORed with the contents of the accumulator.	Immediate	Two bytes	ORI 80H

(Contd)

Table 1.10 Logical instructions *(Contd)*

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length	Example
ANA R	The contents of the register are logically ANDed with the contents of the accumulator.	Register direct	One byte	ANA C
XRA R	The contents of the register are logically EXORed with the contents of the accumulator.	Register direct	One byte	XRA D
ORA R	The contents of the register are logically ORed with the contents of the accumulator.	Register direct	One byte	ORA E
ANA M	The contents of the memory location pointed to by the HL register pair is logically ANDed with the contents of the accumulator.	Indirect	One byte	ANA M
XRA M	The contents of the memory location pointed to by the HL register pair is logically EXORed with the contents of the accumulator.	Indirect	One byte	XRA M
ORA M	The contents of the memory location pointed to by the HL register pair is logically ORed with the contents of the accumulator.	Indirect	One byte	ORA M
RLC	Rotates the bits of the accumulator left by one position	Implicit	One byte	RLC
RRC	Rotates the bits of the accumulator right by one position	Implicit	One byte	RRC
RAL	Rotates the bits of the accumulator left by one position, through the carry	Implicit	One byte	RAL
RAR	Rotates the bits of the accumulator right by one position, through the carry	Implicit	One byte	RAR
CPI 8-bit	Compares the 8-bit data with the contents of the accumulator	Immediate	Two bytes	CPI FFH
CMP R	Compares the contents of the register with that of the accumulator	Register direct	One byte	CMP B
CMP M	Compares the contents of the memory location pointed to by the HL register pair with that of the accumulator	Indirect	One byte	CMP M
CMA	Complements the contents of the accumulator	Implicit	One byte	CMA
CMC	Complements the carry	Implicit	One byte	CMC
STC	Sets the carry	Implicit	One byte	STC

For logical operations, one of the data must be stored in the accumulator and the other given or addressed in the instruction. Logical operations can be performed with immediate data, data stored in a register, or indirectly addressed memory location content.

Besides the instructions already mentioned, two types of rotate instructions are available in the 8085. One set—RLC and RRC—rotates the accumulator contents within itself. The RLC instruction shifts the accumulator content left by one bit. In the process, the most significant bit of the accumulator becomes the least significant bit. The RRC instruction shifts the accumulator content right by one bit.

The other set of rotate instructions—RAL and RAR—rotates the accumulator content along with the carry flag. The RAL instruction shifts the accumulator content left by one bit and in the process, the most significant bit will be shifted to the carry flag and the carry flag content will be shifted to the least significant bit of the accumulator.

The instruction set of the 8085 supports a compare instruction for comparing the magnitude of two binary numbers. The compare instructions are used to compare the accumulator content with the operand specified in the instruction. CPI instruction uses immediate addressing and CMP uses registers or indirectly addressed memory location for comparing with the accumulator. The result of the compare instruction is indicated in the flag register, as follows:

If [(A) − operand] = 0, i.e., (A) = operand, the zero flag is set.

If [(A) − operand] < 0, i.e., (A) < operand, the carry flag is set.

If [(A) − operand] > 0, i.e., (A) > operand, the zero and carry flags are reset.

1.14.5 Branching Instructions

Branching instructions are used to transfer the program execution to a different address. Branching instructions are of two types—jump instructions and subroutine instructions. The jump instructions merely transfer the execution from one location in the program to another, whereas the subroutine instructions in the main program transfer execution to a new location and also return to the main program. Return instructions are used for this purpose. The branching can take place unconditionally or conditionally, based on the flag conditions shown in Table 1.11. PCHL instruction is a special instruction used to branch to the address stored in the HL register pair.

RST *n* is the restart instruction supported by the 8085. Upon execution of the RST *n* instruction, the program execution will be transferred to the address given by $n \times 8$. For example, RST 4 instruction will transfer the execution to the address 0020H which is the hexadecimal equivalent of 32 (in decimal form).

In machine code or opcode, the16-bit or 4 hex digit addresses in the branching instructions are given such that the lower-order byte of the address follows the higher-order byte. For example, JMP 8030H is coded as C3 30 80. The opcode for JMP, C3, is stored first, followed by 30 and then by 80.

Table 1.11 Branching instructions

Mnemonics	Tasks performed on execution	Instruction length	Example
JMP 16-bit	Jump unconditionally	Three bytes	JMP 9500
JC 16-bit	Jump if carry is set	Three bytes	JC 9500
JNC 16-bit	Jump on no carry	Three bytes	JNC 9500
JP 16-bit	Jump on positive	Three bytes	JP 9500
JM 16-bit	Jump on minus	Three bytes	JM 9500
JZ 16-bit	Jump on zero	Three bytes	JZ 9500
JNZ16-bit	Jump on no zero	Three bytes	JNZ 9500
JPE 16-bit	Jump on parity even	Three bytes	JPE 9500
JPO 16-bit	Jump on parity odd	Three bytes	JPO 9500
CALL 16-bit	Call unconditionally	Three bytes	CALL 9500
CC 16-bit	Call on carry	Three bytes	CC 9500
CNC 16-bit	Call on no carry	Three bytes	CNC 9500
CP 16-bit	Call on positive	Three bytes	CP 9500
CM 16-bit	Call on minus	Three bytes	CM 9500
CZ 16-bit	Call on zero	Three bytes	CZ 9500
CNZ 16-bit	Call on no zero	Three bytes	CNZ 9500
CPE 16-bit	Call on parity even	Three bytes	CPE 9500
CPO 16-bit	Call on parity odd	Three bytes	CPO 9500
RET	Return unconditionally	One byte	RET
RC	Return on carry	One byte	RC
RNC	Return on no carry	One byte	RNC
RP	Return on positive	One byte	RP
RM	Return on minus	One byte	RM
RZ	Return on zero	One byte	RZ
RNZ	Return on no zero	One byte	RNZ
RPE	Return on parity even	One byte	RPE
RPO	Return on parity odd	One byte	RPO
PCHL	Copy HL contents to the program counter	One byte	PCHL
RST 0/1/2/3/4/5/6/7	Restart	One byte	RST 5

1.14.6 Machine Control Instructions

Machine control instructions are used to control the microprocessor execution and functioning and are listed in Table 1.12. They are explained in detail in the following points:

(i) NOP means no operation. When this instruction is executed, nothing is done; no changes occur in the contents of the registers. The program counter alone is incremented to fetch and execute the next instruction.

(ii) HLT instruction is used to halt the execution of the program. The operation of the microprocessor is suspended when HLT instruction is executed. The only way to exit the halt state is to apply the hardware reset signal.

(iii) Interrupts are disabled and enabled using DI and EI signals, respectively. Once the DI instruction has been executed, the processor ignores any interrupt request received. To enable interrupts again, the EI instruction has to be executed.

(iv) The SIM instruction is used to send serial data on the serial output data (SOD) line of the microprocessor and the RIM instruction is used to receive serial data on the serial input data (SID) line of the processor. The SIM and RIM instructions are also associated with the setting and reading of interrupt masks for RST hardware interrupts.

Table 1.12 Machine control instructions

Mnemonics	Tasks performed on execution	Addressing mode	Instruction length
NOP	No operation	Implicit	One byte
HLT	Halts the microprocessor execution	Implicit	One byte
DI	Disables interrupts	Implicit	One byte
EI	Enables interrupts	Implicit	One byte
RIM	Reads interrupt mask	Implicit	One byte
SIM	Sets interrupt mask	Implicit	One byte

1.15 SAMPLE PROGRAMS

1. Write an assembly language program to add two numbers.

The program given in Table 1.13 uses immediate addressing for the two data to be added. The data to be added are stored in memory locations 8001H and 8003H. The sum is stored in the memory location 8500H. This program assumes that no carry is generated from the addition.

Table 1.13 Program for adding two 8-bit numbers

Memory address	Machine code/ Opcode	Labels	Mnemonics with operands	Comments
8000	3E	START:	MVI A, 32H	; Load the first number in
8001	32			the accumulator.
8002	C6		ADI 64H	; Add the second number with
8003	64			the contents of the accumulator.
8004	32		STA 8500H	; Store the sum in the memory
8005	00			location 8500H.
8006	85			
8007	76		HLT	; Terminate program execution.

2. Write an assembly language program to add two numbers of 16 bits each.

This program also uses immediate addressing for loading the data in the processor registers. The sum is stored in the memory locations 8500H and 8501H, as shown in Table 1.14.

Table 1.14 Program for adding two 16-bit numbers

Memory address	Machine code/ Opcode	Labels	Mnemonics with operands	Comments
8000	21	START:	LXI H, 805FH	; Load the first 16-bit number in
8001	5F			the HL register pair.
8002	80			
8003	01		LXI B, 123AH	; Load the next number in the BC
8004	3A			register pair.
8005	12			
8006	09		DAD B	; Add the two numbers using double addition instruction.
8007	22		SHLD 8500H	; Store the result in the HL pair in
8008	00			the memory locations 8500H and
8009	85			8501H.
800A	76		HLT	; Terminate program execution.

3. Write an assembly language program to add the two numbers stored in the memory locations 8500H and 8501H and store the result in 8502H.

This program uses indirect addressing instructions to load the numbers to be added in the processor registers. The carry, if generated, is ignored. The program is shown in Table 1.15.

Table 1.15 Program for adding two numbers from memory

Memory address	Machine code/ Opcode	Labels	Mnemonics with operands	Comments
8000	21	START:	LXI H, 8500H	; Initialize HL register pair to point
8001	00			to the memory location of the
8002	85			first number.
8003	7E		MOV A, M	; Load the first number in the accumulator.
8004	23		INX H	; Increment the HL pair to point to the memory location of the next number.
8005	86		ADD M	; Add the two numbers.
8006	23		INX H	; Increment the HL pair to point to the next memory location.
8007	77		MOV M, A	; Store the contents of the accumulator in the memory location pointed to by the HL register pair.
8008	76		HLT	; Terminate program execution.

1.16 INSTRUCTION EXECUTION

The 8085 microprocessor is designed to fetch the instruction pointed to by the program counter, and then decode and execute the instruction within the processor. If necessary, further operand fetch takes place before completing the execution. Each instruction, as we have already seen, 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.

Instruction cycle is the time taken by the processor to complete the execution of an instruction. An instruction cycle consists of one to six machine cycles.

Machine cycle is the time required to complete one operation—accessing either the memory or an I/O device. A machine cycle consists of three to six T-states.

T-state is the time corresponding to one clock period. The T-state is the basic unit used to calculate the time taken for execution of instructions and programs in a processor.

To execute a program, the 8085 performs various operations such as opcode fetch, operand fetch, and memory read/write or I/O read/write. The microprocessor's external communication function can be divided into three categories:

(i) Memory read/write (ii) I/O read/write
(iii) Interrupt request acknowledge

POINTS TO REMEMBER

- The microprocessor is an electronic circuit that functions as the central processing unit (CPU) of a computer, providing computational control.
- The microprocessor is the controlling element in a computer system. The microprocessor performs data transfers, does simple arithmetic and logical operations, and makes simple decisions.
- The basic operation of the microprocessor is to fetch instructions stored in the memory and execute them one by one in sequence.
- Microprocessors are used in almost all advanced electronic systems.
- Microcontrollers are advanced forms of microprocessors, with memory and ports present within the chip.
- A microcomputer system is made by interfacing memory and I/O devices to a microprocessor.
- Microprocessor evolution is classified into five generations. The processors that are currently in use belong to the fifth generation.
- The microprocessor is a semiconductor device consisting of electronic logic circuits manufactured using either large-scale integration (LSI) or very large-scale integration (VLSI) technique. It works at a fixed clock frequency.
- A bus is a collection of wires connecting two or more chips.
- A typical microprocessor communicates with memory and other input/output devices using three buses—address bus, data bus, and control bus.
- Salient features of the 8085 microprocessor manufactured by Intel
  - It is an 8-bit microprocessor.
  - It has a 16-bit address bus (A0–A15) and hence, can address up to $2^{16} = 65,536$ bytes (64 KB).

  - The 8085 has a multiplexed bus (AD0–AD7), which is used as the lower-order address bus and the data bus. It can be de-multiplexed using a latch and the ALE signal.
  - The data bus is a group of eight lines (D0–D7).
  - It supports external interrupt request.
  - It has a 16-bit program counter (PC) and a 16-bit stack pointer (SP).
  - It has six 8-bit general-purpose registers, which can be arranged in pairs as BC, DE, and HL.
  - It requires a +5 V power supply and operates at 3 MHz clock frequency.
  - It contains 40 pins and is available as a dual in-line package (DIP).
  - It has five flags—sign, zero, auxiliary carry, parity, and carry.
- The microprocessor operations related to data manipulation can be summarized in the following four functions:
  (i) Transferring data
  (ii) Performing arithmetic operations
  (iii) Performing logical operations
  (iv) Testing for a given condition and altering the program sequence
- The instructions are classified into three groups according to word size: one-, two-, and three-byte instructions.
- An instruction has two parts—opcode (operation to be performed) and operand (data to be operated on). The operand can be data (8-bit or 16-bit), addresses, registers, or implicit in the opcode. The method of specifying an operand (directly, indirectly, etc.) is called addressing mode.
- The instructions are executed in steps of machine cycles and each machine cycle requires many T-states.

KEY TERMS

Accumulator It is an 8-bit register; it is a part of the ALU and is the most important register. It is used to store 8-bit data and to perform arithmetic and logical operations. The output of an operation is also stored in the accumulator. The accumulator is identified as register A.

Address bus This bus carries the binary number (i.e., the address) used to access a memory location. Binary data can then be written into or read from the addressed memory location. The address bus consists of 16 wires and can, therefore, handle 16 bits.

Addressing mode It is the method of specifying the data to be operated on by the instruction.

Bus It is a group of conducting lines that carry data, address, and control signals

Clock speed This determines how many instructions per second the processor can execute. It is specified in megahertz (MHz).

Control bus This bus has various lines for coordinating and controlling microprocessor operations. For example, $\overline{RD}$ and $\overline{WR}$ lines.

Data bus This bus carries data in binary form between the microprocessor and external units such as memory. Typical size is eight or 16 bits.

DMA controller It is used to take control of the system bus by placing a high signal on the Hold pin.

Flag It is a flip-flop used to store information about the status of the processor and the status of the instruction executed most recently.

Hold and HLDA These signals are used for direct memory access (DMA) type of data transfer. The Hold request makes the 8085 drive all its tri-stated pins to high impedance state. The HLDA signal goes high to acknowledge the receipt of the Hold signal.

Immediate addressing It transfers the operand given in the instruction—a byte or word—to the destination register or memory location.

Implied addressing In this addressing mode, the instruction itself specifies the data to be operated on.

IN This instruction is used to move data from an I/O port to the accumulator.

Indirect addressing It transfers a byte or word between a register and a memory location addressed by another register.

Instruction cycle It is the time required to execute an instruction.

IO/$\overline{\text{M}}$ signal This signal is used to differentiate memory access and I/O access. For input/output instructions it is high; for memory reference instructions it is low.

JMP and **CALL** JMP instruction permanently changes the program counter. CALL instruction leaves information on the stack so that the original program execution sequence can be resumed.

Machine cycle It is the time required to access the memory or input/output devices.

Memory direct addressing It moves a byte or word between a memory location and a register.

Opcode It is the part of the instruction that specifies the operation to be performed.

Operand It is the data on which the operation is performed.

OUT This instruction is used to move data from the accumulator to an I/O port.

Ready It is an input signal to the processor. It is used by the memory or I/O devices to get extra time for data transfer or to introduce wait states in the bus cycles.

Register direct addressing It transfers a copy of a byte or word from a source register to a destination register.

Timing diagram It is a graphical representation of the time taken by each instruction for execution. The execution time is represented in T-states.

Trap It is a non-maskable interrupt of the 8085 and is not disabled by processor reset or after reorganization of interrupt.

T-state It is the basic unit used to calculate the time taken for execution of instructions and programs in a processor. It is the time corresponding to one clock period.

REVIEW QUESTIONS

1. What is the main function of a computer?
2. Name any three input devices of a computer.
3. Name any two output devices of a computer.
4. Name any three storage devices of a computer.
5. Name any three places where computers can be used.
6. Draw a block diagram of a computer and label its components.
7. Who developed the world's first microprocessor?
8. What is the data bus width of the 8085 microprocessor?
9. When did Intel introduce the Pentium 4 microprocessor?
10. What is the amount of memory that the Pentium 4 processor can address?
11. What are the basic units of a microprocessor?
12. What is the function of microprocessor in a system?

13. How many memory locations can be addressed by a microprocessor with 14 address lines?
14. Name any two types of memories that are used in a computer.
15. Define computer hardware.
16. Define computer software.
17. What is the role of CPU in a computer?
18. What are input and output devices?
19. Describe and draw the diagram of Von–Neumann model.
20. Define the following abbreviations: CPU, RAM, and ROM
21. Name any three features of the 8085.
22. What are the operations performed by the ALU of the 8085?
23. What are the various registers in the 8085?
24. What is a flag? List its types. What is the structure of the flag register? Explain each flag with an example.
25. List the 16-bit registers of the 8085 microprocessor.
26. What is a bus?
27. Why is the data bus bidirectional?
28. How are the signals of the 8085 classified?
29. How are clock signals generated in the 8085? What is the frequency of the internal clock?
30. How does the 8085 processor differentiate a memory access (read/write) signal from an I/O access (read/write) signal?
31. Why is crystal a preferred clock source?
32. Which interrupt has the highest priority in the 8085? What is the priority of the other interrupts?
33. When and where is the Ready signal used?
34. What are Hold and HLDA? How are they used?
35. Draw a general block diagram of a microprocessor-based system. Explain briefly the various blocks of the system. Give some examples of the types of devices used for each block.
36. What is a microprocessor? Sketch and explain the various pins of the 8085.
37. Explain the operation of these 8085 signals: Ready, S1 and S0, Hold and HLDA, and ALE.
38. Explain the architecture of the 8085 with the help of its internal block schematic diagram.
39. List the four categories of 8085 instructions that are used for data manipulation.
40. Define opcode and operand. Identify the opcode and the operand in the instruction MOV H, L.
41. Explain the instruction XCHG.
42. What is an instruction? List any four arithmetic instructions and their uses.
43. Define stack. Explain the instructions related to stack operations.
44. When is the instruction XRA A used?
45. How many operations are there in the instruction set of the 8085 microprocessor?
46. Explain with examples the different instruction formats, based on the length of the instructions.
47. List the four instructions which control the interrupt structure of the 8085 microprocessor.
48. What is the last instruction executed in a program? Why?

49. What is the significance of XCHG and SPHL instructions?
50. Explain the operation carried out when the 8085 executes the instruction RST 0.
51. What is addressing? What are the various addressing modes available in the 8085?
52. Explain direct addressing with an example.
53. Explain implied addressing with an example.
54. What are the machine cycles in the 8085 microprocessor?

THINK AND ANSWER

1. Compare the instructions CALL and PUSH.
2. What is the difference between the shift and rotate instructions?
3. How many address lines are there in a 4096 × 8 EPROM chip?
4. Explain the difference between the instructions JMP and CALL.
5. If the instructions CALL and RET were not available in the 8085, would it still be possible to write subroutines? How would the subroutine be called? How would one return to the main program?

2

Methods of Data Transfer and Serial Transfer Protocols

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Different approaches to data transfer
- Different data transfer mechanisms
- Serial data transfer protocols
- Data transfer for slow peripheral devices
- Interrupt structure in microprocessors

2.1 DATA TRANSFER MECHANISMS

Data transfer is essential in any microprocessor-based system. It can take place between the processor and the memory, the processor and an input/output device, or the memory and an input/output device.

Data can be transferred in several ways. The mechanism differs based on characteristics such as the addressing of the devices, amount of data transferred, method of data transfer, and interaction among the devices. The data transfer mechanism is divided into the following types:

1. Based on the addressing of the device
   (i) I/O-mapped I/O access
   (ii) Memory-mapped I/O access
2. Based on the program and hardware involved
   (i) Programmed data transfer
      (a) Polled mode of data transfer
      (b) Interrupt-driven data transfer
   (ii) Direct memory access
      (a) Burst mode
      (b) Cycle stealing mode
3. Based on the method of data transfer and access
   (i) Parallel data transfer
      (a) Simple data transfer
      (b) Handshake mode data transfer
   (ii) Serial data transfer
      (a) Synchronous data transfer
      (b) Asynchronous data transfer

2.2 MEMORY-MAPPED AND I/O-MAPPED DATA TRANSFER

In I/O-mapped device data transfer method, I/O devices and memory are handled separately. A separate address range is assigned for input/output devices. Separate control signals are used for memory access and for I/O device read/write operation.

The microprocessor has separate instructions for input and output device access, such as the IN and OUT instructions of the 8085. As memory and I/O device accesses are governed by separate control signals, a single address can be assigned to both an I/O device and a memory location.

In memory-mapped I/O, each input/output device is treated like a memory location. The same control signals are used for I/O device read/write operation and for memory access. Each input or output device is identified by a unique address in the memory address range. All memory-related instructions that are used to read data from memory are used to access input and output devices. Since the I/O devices use some of the memory address space, the maximum memory addressing capacity is reduced in this system.

2.3 PROGRAMMED DATA TRANSFER

The instructions for programmed data transfer are written and controlled by the programmer and executed by the processor. The data transfer between the processor and I/O devices (and vice versa) takes place by executing the corresponding instructions. Programmed I/O data transfers are identical to read and write operations for memories and device registers. An example of programmed I/O is a device driver writing one data byte at a time directly into the device's memory.

Programmed data transfer can take place at a time determined by the programmer. Based on the time of execution of the data transfer instruction, programmed data transfer is divided into two types—polled mode of data transfer and interrupt-driven data transfer.

In polled mode of data transfer, data is read from an input device when the processor or CPU is ready. The processor then executes the data transfer instruction. If the input device is not ready, the processor waits until the device is ready with data. Similarly, data is written into an output device by the processor when it executes the write instruction corresponding to that output device. The program is written in such a way that the processor waits in a loop until the output device is ready to receive data. Clearly, in waiting for the device to be ready, processor time is wasted in this mode of data transfer.

In interrupt-driven data transfer, data is read from the input device only when it is ready with data. When the device is ready, it gives an interrupt signal to the processor, indicating that the data is ready. In the interrupt service routine (ISR), a program is executed to read the data from the corresponding input device. Similarly, the output device gives an interrupt to the processor when it can accept data. The programmers have to write an ISR for data transfer to the corresponding output device. Interrupt-driven data transfer is advantageous as data transfer is done only when the device is ready; the processor need not wait until the device is ready. The processor can execute some other main routines and data transfer program will be executed as ISRs. It is an efficient technique because processor time is not wasted in waiting while an I/O device is getting ready or not ready. Slow I/O devices can be interfaced for data transfer using the interrupt-driven technique.

2.4 DIRECT MEMORY ACCESS

In programmed I/O data transfer, the processor is actively involved in the entire data transfer process. So, the data transfer rate is limited. The processor is tied up and processor time is wasted. To overcome these disadvantages, the direct memory access (DMA) method of data transfer is used.

Direct memory access is a technique to transfer data between the peripheral I/O devices and the memory, without the intervention of the processor. The basic idea is to transfer *blocks of data* directly between the memory and the peripherals. Even though the transfer is done without the processor, the processor initiates the DMA operation. This technique is generally used to transfer large blocks of data between memory and I/O. During DMA data transfer, the processor/CPU is kept in an idle suspended state called as Hold state. DMA performs high-speed data transfer to and from mass storage peripheral devices such as hard disk drives, magnetic tapes, CD ROMs, and video controllers. A hard disk may have a transfer rate of 5 Mbps, i.e., one byte every 200 ns. Performing such data transfer using the CPU is not only undesirable but also unnecessary, since the CPU transfer rate is limited by the speed of the memory and peripheral devices.

Under normal circumstances, the CPU has full control of the address and data buses in the system. When direct memory access occurs, an external device or DMA controller takes over the temporary control of the system bus from the CPU. The CPU writes necessary control words into the DMA controller, to indicate the following details about the data transfer: read or write operation, device address involved, starting address of the data memory block and the amount of data to be transferred. After this initialization, the DMA controller takes care of the data transfer. In the 8085, the hold request is received and acknowledged using the HOLD and HLDA pins, respectively.

The sequence of events in a typical DMA process is as follows:

(i) The peripheral or the DMA controller asserts one of the request pins (such as HOLD) for holding the processor.

(ii) The processor completes its current instruction and enters into the Hold state. In the Hold state, the processor temporarily stops the execution of the instruction and releases the address and data buses by making them enter into a high impedance state.

(iii) The processor issues a Hold Acknowledge (HLDA) signal to indicate the release of bus control to the peripheral or the DMA controller.

(iv) The DMA operation starts.

(v) Upon completion of the DMA operation, the peripheral or the DMA controller removes the Hold signal applied to the processor and relinquishes bus control.

In general, a DMA controller can interface several peripherals that may request DMA with the processor. It is the controller that decides the priority of DMA requests that are received simultaneously from many peripherals. It then communicates with the peripheral device and the processor, and provides memory addresses for transferring data. The 8237 programmable DMA controller is the

controller device that is most commonly used with the 8085 and 8088. It is a four-channel device, with each channel being dedicated to a particular peripheral device. In addition, each channel is capable of addressing 64 KB of memory.

DMA data transfer can be divided into two types:

(i) Burst or block transfer mode

(ii) Cycle stealing or interleaved mode

In burst mode of DMA data transfer, a complete block of data is transferred in a single DMA cycle. The system bus is released by the peripheral or DMA controller only after the required bytes of data are transferred. In cycle stealing mode of data transfer, a block of data is transferred over many DMA cycles. The system bus is released to the processor after a byte or a set of bytes are transferred in one DMA cycle. Thus, the processor is not suspended from its activities for a long time. It takes several DMA cycles to complete the transfer of one block of data.

2.5 PARALLEL DATA TRANSFER

In parallel mode of data transfer, all the bits in a word are simultaneously transmitted. Since the 8085 word consists of eight bits, all the eight bits are transmitted and received in parallel form. In some special cases, the number of data bits transferred will be lesser than eight. In general, parallel data transfer is used for transfer of data over short distances such as within a system, within a printed circuit board (PCB), etc. It can be done either in polled mode or in interrupt-driven mode. In polled method, data is read from the input device by the processor at a time determined by the processor. This polled mode of data transfer can be done in two ways—synchronous or simple I/O and handshake I/O.

In simple or synchronous mode, data is read from the input device by the processor irrespective of the status of the input device. It is assumed that the input device is in synchronism with the processor and that it is ready with data whenever the processor reads the data. Similarly, the data is written into the output device irrespective of its status. The processor assumes that the output device is in synchronism with the processor.

In handshake I/O mode, the processor checks for the status of the I/O devices before data transfer. An input device gives a signal to the processor, indicating that it is ready with the data. The processor checks continuously for the reception of this signal and upon reception can read the data. Similarly, an output device gives a signal to the processor, indicating that it can accept data. The processor, before writing data to the output device, checks for this signal. If the signal indicating readiness of the output device is available, the processor can write the data to the output device. The signals that are transferred between the devices and the processor are called handshake signals.

2.6 SERIAL DATA TRANSFER

Parallel data transfer has the drawback of needing several wires to transfer all the bits of data. So, it can not be used effectively for long distance transfers. As one wire is used for each bit, byte-wise data transfers are eight times more expensive

than a single bit transfer. Serial data transfer is the solution for data transfers over long distances. It is a low-cost way to send data over long distances. In serial data transfer, only one bit is transferred over a data transfer line. All the bits in a data word can be transmitted by using a shift register and transferring the data bit by bit. Parallel-to-serial data conversion is done by a device called universal asynchronous receiver–transmitter (UART).

In serial data transfer, the following three aspects are important: First, the speed or frequency at which the bits are transmitted into the serial data line. The frequency at which the data is transmitted serially is technically called baud rate. Baud rate is the measure of the number of bits transmitted over a second. Second, the mode of data transfer. Serial data transfer can be done in two modes—synchronous and asynchronous. Third, the voltage levels for logic 1 and logic 0 for the data being transmitted. Various serial communication protocols define these aspects as standards for proper communication.

In synchronized data transfer, the device that sends the data and the device that receives the data are synchronized with the common clock. In synchronous mode, data transfer takes place with a fixed and known time frame. In asynchronous data transfer, data words are transmitted with a random time frame between them. Most microprocessor- and computer-related data communications are based on the asynchronous mode of transmission. Microprocessors use interrupts and other software techniques to synchronize random timing between data words, so as to receive the data completely.

The modem plays an important role in serial transmission. It is a device that allows transmission of serial data over communication lines such as telephone lines. In general, communication lines are incapable of carrying the voltage changes required for a direct digital connection. A modem overcomes this limitation by modulating digital information into analog signals using one of the modulation techniques and demodulating it back into digital information upon reception.

The computer or a microprocessor terminal that initiates the serial communication is called *data terminal equipment* (DTE). The final equipment that receives the serial data is also called data terminal equipment. *Data communication equipment (DCE)* is a device that connects the DTE to a transmission line. So, the transmitting DTE sends the serial data to the DCE. The DCE is generally a modem. This helps in level shifting and transmitting the serial data over the chosen transmission line. Similarly, at the receiving station, a DCE (generally a modem) receives the signal and transfers the serial data to the receiving DTE.

This section introduces the RS-232, RS-485, general-purpose interface bus (GPIB), and IEEE 488 standards, which are used for data transfer between two computer or processor systems. RS-232 is a common serial communication protocol used in computer systems.

2.6.1 Introduction to RS-232 Standard

RS-232 is a serial communication standard given by the Electronic Industries Association (EIA), an organization represented by a group of electronic industries.

It is used for one-to-one communication between two computers or processor systems. RS-232 standard can also be used with modems. RS-232 can be used to interface a processor system or DTE with a modem/DCE. It is the standard used on personal computers' COM port. The maximum possible speed with RS-232 is 20 kbps and the maximum possible cable length is 50 feet.

Logic 1 is represented by voltages in the range −3 V to −25 V, and typically by −12 V. Logic 0 uses the voltage range from 3 V to 25 V, and typically 12 V. When no data is sent over the transmission line and the transmitter is inactive, the voltage level on the line is kept at a logic high level, i.e., −12 V. Figure 2.1 shows the RS-232 voltage levels.

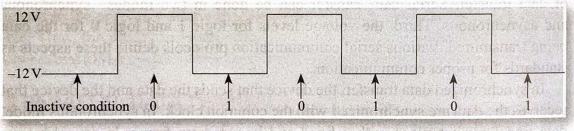


Fig. 2.1 RS-232 voltage levels

RS-232 is a serial communication standard for asynchronous communication. The transmitter places logic 1 on the data bus when it is inactive. To start transmission, the transmitter sends a logic 0 as the start bit. The start bit makes the receiver wake up from idle mode and start receiving data. After the start bit, data bits are transmitted on the serial transmission line. The length of the data bits can be five, six, seven, or eight depending on the transmitting equipment. The least significant bit of the data byte is transmitted first in the data line. The data bits are succeeded by a parity bit or any other error correcting bit set by the programmer. After this, the stop bit is sent by the transmitter to indicate the end of the data bits. Logic 1 is used as the stop bit in RS-232 communication standard. The format of the signal transmitted is shown in Fig. 2.2. Here, the ASCII code for the character A is shown being transmitted on the line with the parity bit as 1. The format uses two stop bits of logic 1 consecutively.

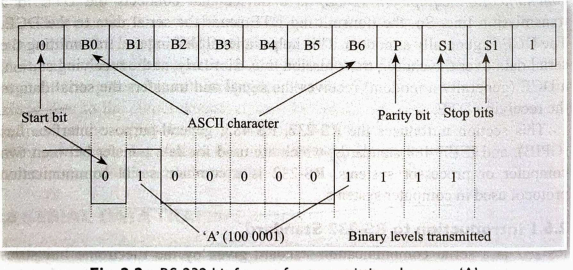


Fig. 2.2 RS-232 bit format for transmitting character 'A'

Another parameter specified by the RS-232 communication standard is the baud rate. It is the rate at which data is transmitted and received. The baud rate and the timing for each bit is related by the following formula:

Time period for each bit in seconds = 1/baud rate

Table 2.1 lists the standard baud rate used by the RS-232 communication standard and the corresponding bit duration.

RS-232 communication connection is done through standard connectors. Two types of RS-232 connectors are available. One has 25 pins and the other has nine. Details of the DB25S and DB9S connectors are shown in Fig. 2.3. A cable with any one of these connectors is used to connect the DTE (computer) with the DCE (modem).

Table 2.1 Bit timings for standard baud rates

Baud rate	Time for each bit in microseconds
1200	833
2400	417
9600	104
19,200	52

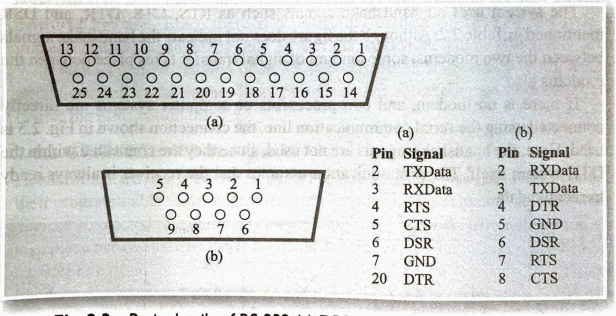


Fig. 2.3 Basic details of RS-232 (a) DB25S and (b) DB9S connectors

The basic signals used in these connectors are given in Table 2.2.

Table 2.2 Signals of RS-232 connection

Signal name	Function
Receive data line (RXD)	Data is received by the processor on this line
Transmit data line (TXD)	Data is sent by the processor through this line
Data Terminal Ready (DTR)	Signal sent out by the processor to indicate that it is ready for communication
Data Set Ready (DSR)	Signal sent by the modem to the processor to indicate that it is ready to transmit or receive
Request to Send (RTS)	Signal sent out by the processor to the modem to indicate that it is ready to send data
Clear to Send (CTS)	Signal sent by the modem to the processor to indicate that it can accept data for transmission

Figure 2.4 shows the standard connection for RS-232 communication between two DTEs through two modems.

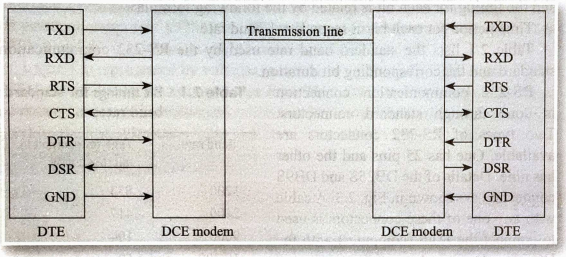


Fig. 2.4 RS-232 communication connection using modems and line

The system uses all handshake signals such as RTS, CTS, DTR, and DSR mentioned in Table 2.2. Although the figure does not indicate the handshake signals between the two modems, some amount of signal transfer takes place between the modems also.

If there is no modem, and two processors or computer systems are directly connected using the serial communication line, the connection shown in Fig. 2.5 is used. Here, the handshake signals are not used, since they are connected within the DTE system itself. The communication assumes that the receiver is always ready to receive data.

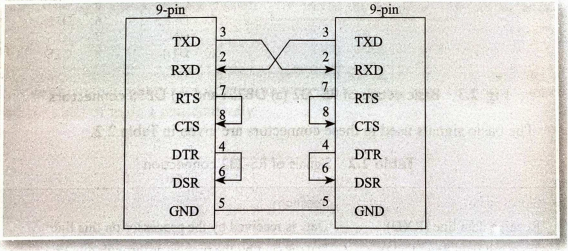


Fig. 2.5 RS-232 connections with no handshaking and DCE (modem)

2.6.2 Introduction to RS-485 Standard

RS-485 is another serial communication standard defined by the EIA. The major difference between RS-232 and RS-485 is that RS-232 is used for one-to-one communication, whereas RS-485 is used in a network environment. The major features of RS-485 standard are as follows:

(i) RS-485 can connect several processors or DTEs in a network structure for communication.

(ii) RS-485 can be used for communication over longer distances than RS-232.
(iii) RS-485 can communicate with higher baud rates, i.e., faster than RS-232.
(iv) One RS-485 transmitter can drive up to 32 receivers in a network.
(v) RS-485 transmitter uses two signal lines—+sig and −sig. The RS-485 receiver senses the voltage difference between these lines. So, any voltage difference on the ground line between the transmitter and the receiver does not affect the reception. However, RS-232 receiver senses the voltage level of the signal with respect to the ground and so, the noise voltage level may affect the data sensed.

By default, all the senders on the RS-485 bus are tri-stated, i.e., in high impedance state. In higher-level protocols, one of the nodes is defined as a master that sends queries or commands over the RS-485 bus. All other nodes receive these data. Depending of the information contained in the data sent, zero or more nodes on the line respond to the master. In this situation, bandwidth of almost 100% can be used. There are other ways of implementing the RS-485 network, where every node can start a data session on its own. This is comparable to the way Ethernet networks function. Since there is a possibility of data collision in this type of implementation, theoretically only 37% of the bandwidth will be effectively used. With such an implementation of an RS-485 network, it is necessary to implement error detection in the higher-level protocol so as to detect data corruption and resend the information later.

2.6.3 GPIB/IEEE 488 Standards

Hewlett-Packard designed the Hewlett-Packard Interface Bus (HP-IB) to connect their programmable smart instruments to computers. This standard supports many devices connected to a common bus and forming a network. Communication can take place between all the devices connected to the bus. This standard has a higher transfer rate of up to 1 Mbyte/s, in comparison with the RS-232 and RS-485. This standard has been named IEEE Standard 488. HP-IB is also called GPIB. The devices in the GPIB bus can be connected in a linear network, star configuration, or a combination of both.

GPIB standard categorizes the devices connected together into three types—talkers, listeners, and controllers. A talker can send data to other devices. A listener is a device that can receive data from other devices connected in the bus. A controller is a device that determines which of the devices should be listeners and which of them should be talkers. In general, a GPIB bus has one controller and many talkers and listeners. Some of the devices in the bus network can act as both talkers and listeners. Communication can take place from one talker to one listener or from one talker to many listeners in the bus. The controller decides the data transfers and also issues commands to other devices. A bus system with only one talker does not need any controller. The talkers and listeners are generally computer or microprocessor systems. The microprocessor systems can be configured as talkers or listeners by interfacing it to the Intel 8291 GPIB talker–listener. Similarly, Intel 8292 GPIB controller can be interfaced to the microprocessor or computer systems to manage the GPIB communication.

The GPIB interface system uses 24-pin connectors, as shown in Fig. 2.6.

Among the 24 pins, eight lines are bidirectional data lines and eight are ground lines. Among the remaining eight lines, three pins are for handshake signals and five are for bus interface management signals. The eight data lines are used for transmission and reception of data, addresses, commands, and status bytes.

The five bus interface management lines and their functions are as follows:

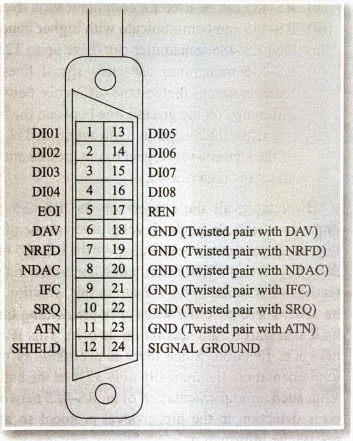


Fig. 2.6 GPIB bus connector and signals

(i) IFC (Interface Clear)—The controller in the bus sends this signal to all other devices in the bus to initialize the bus and reset the system communication upon powering on.

(ii) ATN (Attention)—The controller sends an active low ATN signal to indicate that it is sending a universal command or an address on the bus. This signal is made high for data transfers.

(iii) REN (Remote Enable)—The controller makes this signal active to directly control a device instead of the front panel controllers in the device.

(iv) EOI (End or Identify)—The EOI signal is issued by the talker to indicate the end of block transfer of data. The controller uses the EOI line to make devices identify themselves in a parallel poll.

(v) SRQ (Service Request)—This signal is made active by any device that requires to transfer data on the bus.

Three lines are used as handshake signals to control the transfer of message bytes between devices. The process is called a three-wire interlocked handshake and it is used to transfer data from different devices at different transfer rates. It guarantees that message bytes on the data lines are sent and received without transmission error.

(i) NRFD (Not Ready for Data)—This signal is sent by all devices when they are not ready to receive a message byte. When receiving data, the devices make this line inactive by making it low.

(ii) DAV (Data Valid)—The controller makes DAV low while sending commands, and the talker drives DAV low while sending data messages.

(iii) NDAC (Not Data Accepted)—The active low NDAC signal is low until the

transmitted data is received by the slowest listener. This signal indicates that a device in the bus has not received a message byte. Usually, the talkers wait for this signal and once the NDAC is high, the DAV signal is removed by the talkers.

The GPIB uses active low logic with standard TTL levels. For example, when DAV is active, the devices send a TTL low level ($\leq$0.8 V), and when DAV is made inactive, the line has a TTL high level ($\geq$2.0 V).

2.7 INTERRUPT STRUCTURE OF A MICROPROCESSOR

Interrupt is a mechanism by which the processor (CPU) is made to transfer control from its current program execution to another program of more importance or higher priority. The interrupt signal may be given to the processor by any external peripheral device. In general, interrupts are generated by a variety of sources, either internal or external, to the CPU. Interrupts are the primary means by which input and output devices obtain the services of the CPU.

The program or the routine that is executed upon interrupt is called *interrupt service routine* (ISR). The processor must temporarily stop its current task and execute the ISR, which relates specifically to the event or device that issues the interrupt signal. After execution of the ISR, the processor must return to the interrupted program. Processors have many interrupt signals and proper identification of interrupt signals is done internally by the processor.

The key features in the interrupt structure of any microprocessor are as follows:

(i) The number and types of interrupt signals available.
(ii) The address of the memory where the ISR is located for a particular interrupt signal. This address is called *interrupt vector address*.
(iii) The masking and unmasking feature for the interrupt signals. This feature allows the programmer to execute the ISR only when required.
(iv) The priority of interrupts when more than one interrupt signals are available
(v) The timing of the interrupt signals
(vi) The handling and storing of information about the interrupted program (status information). This information must be loaded into the CPU when the ISR is executed. When the return instruction is executed, control is transferred back to the interrupted program.

2.8 TYPES OF INTERRUPTS

Interrupts are classified based on their maskability, interrupt vector address, and source. These classifications are discussed in Sections 2.8.1–2.8.3.

2.8.1 Vectored and Non-vectored Interrupts

The vectored and non-vectored interrupts are as follows:

(i) Non-vectored interrupts have fixed interrupt vector address for ISRs of different interrupt signals. They are useful for small systems, where there are few interrupt sources and the software structure is not complicated.
(ii) Vectored interrupts require the interrupt vector address to be supplied by

the external device that gives the interrupt signal. This technique, called *vectoring*, is implemented in a number of ways.

2.8.2 Maskable and Non-maskable Interrupts

The maskable and non-maskable interrupts are as follows:

(i) Maskable interrupts are interrupts that can be blocked; the corresponding ISRs are not executed. The masking can be done by software or hardware means.

(ii) Non-maskable interrupts (NMIs) are interrupts that are always recognized; the corresponding ISRs are executed.

2.8.3 Software and Hardware Interrupts

The software and hardware interrupts are as follows:

(i) Software interrupts are special instructions, which after execution transfer the control to a predefined ISR. These instructions are included in the program by the programmer.

(ii) Hardware interrupts are signals given to the processor from external devices, for recognition as an interrupt and execution of the corresponding ISR.

2.9 INTERRUPT HANDLING PROCEDURE

When an interrupt signal is recognized, the processor will have to store information about the current program before executing the ISR. The processor checks for the interrupt request signals at the end of every instruction execution. If the interrupt is masked, it will not be recognized until interrupts are re-enabled. The CPU responds to an interrupt request by a transfer of control to another program, in a manner similar to a subroutine call. This is shown pictorially in Fig. 2.7. The sequence of operations that take place when an interrupt signal is recognized is as follows:

(i) Save the program counter (PC) contents (address of the next instruction) and supplementary information about the current state (flags, registers, etc.) in the stack.

(ii) Load PC with the beginning address of an ISR and start to execute it.

(iii) Finish ISR when the return instruction is executed.

(iv) Return to the point in the interrupted program where execution was interrupted by reloading the saved program counter content from the stack.

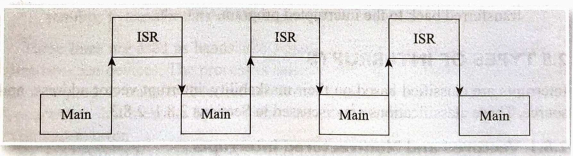


Fig. 2.7 Transfer of control from main memory to ISR

Interrupts and stack memory Stack is a special memory organization that operates on the last-in, first-out (LIFO) principle. The data stored recently is retrieved first. Similarly, data stored first in the stack is read last. Stack is a temporary storage memory in the RAM area. It is basically administered by a special register called

stack pointer (SP). SP register always contains the address of the top of the stack (ToS). Storing a data in the stack memory pointed to by the stack pointer is called push operation. Reading a data from the stack is called pop operation.

Stack is used by the interrupt system of the microprocessor for implementing the subroutine call and return mechanism, passing parameters to subroutines, etc. When the transfer of control takes place from the interrupted program to the ISR, the program counter content is stored in the stack, because after the execution of the ISR, the control must return to the program counter content. To facilitate this control transfer, the stack pointer must be properly initialized to a physically available memory with sufficient memory range. In the 8085, the stack memory grows towards lower addresses and so, the stack pointer must be initialized with the highest memory address allotted for the stack operation.

The stack can be accessed by the instructions PUSH and POP. The ISRs should not disturb the return address stored by the processor in the stack. So, the ISRs should have equal number of PUSH and POP instructions. This condition ensures that the return address stored in the stack is retrieved properly by the processor.

POINTS TO REMEMBER

- Different data transfer schemes are available for data transfer between two processors or between a processor and an I/O device.
- The different data transfer schemes are programmed data transfer and DMA, polled and interrupt-driven, and serial and parallel data transfer.
- Various serial port standards such as RS-232, RS-485, IEEE488, and GPIB are used for data transfer in different applications.
- Interrupts are an important mechanism available in the processors to temporarily stop current program execution and execute a program of higher priority.
- Interrupt vector addresses and the source, priorities, and timing of interrupts are very important to program and understand the operation of interrupts in a processor.
- Interrupts can be either hardware generated and random, or software generated and programmed.
- The processor can be interrupted before the completion of an interrupt service routine (ISR) if the program has executed the EI instruction. This enables nested ISR execution.

KEY TERMS

DMA It is a special method of data transfer between I/O devices and memory without the need of processor for data transfer.

I/O-mapped I/O scheme This scheme uses special control lines and different address space for accessing I/O devices. The processor needs separate instructions for I/O-mapped I/O access.

Interrupt priorities The sequence or order in which the interrupts are sensed by the microprocessor. This order decides which ISR will be executed first, when more than one interrupt is applied simultaneously to the processor.

Interrupt service routine The routine executed by the processor upon sensing an interrupt signal is called interrupt service routine.

Interrupt vector address It is the location to which program control is transferred, upon receipt of an interrupt.
Interrupt-driven I/O scheme This scheme uses a special signal from the I/O devices to initiate a data transfer by the processor.
Memory-mapped I/O scheme This scheme uses the same instructions and hardware used for memory accesses, for accessing I/O devices.
Parallel data transfer It is the method in which all the bits of a word are transmitted simultaneously.
Polled I/O transfer This method uses a software routine to access and transfer data between processor and I/O devices.
Serial data transfer It is the method of transferring a single bit at a time over a transmission line.

REVIEW QUESTIONS

1. Explain memory-mapped I/O.
2. What is I/O-mapped I/O?
3. Compare memory-mapped I/O with peripheral-mapped I/O.
4. What are the various schemes of data transfer?
5. Discuss interrupt-driven data transfer scheme.
6. Explain DMA method of data transfer.
7. If the speed of the I/O devices is lesser than that of the processor, what type of data transfer scheme can be used?
8. What are the advantages of serial data transfer?
9. Compare synchronous and asynchronous modes of data transfer.
10. Explain the RS-232 method of serial data transfer.
11. What is meant by 'priority of interrupts'? Explain the operation of the interrupts structure of the 8085, with the help of a circuit diagram.
12. Distinguish between (i) vectored and non-vectored interrupt, (ii) maskable and non-maskable interrupt, (iii) software and hardware interrupt.
13. Explain interrupt-driven I/O technique. How does the 8085 respond to the INTR interrupt?

THINK AND ANSWER

What are the ways to identify the device that has interrupted the processor in a microprocessor-based system?

Part I

INTEL 8086—16-BIT MICROPROCESSORS

- Intel 8086 Microprocessor Architecture, Features, and Signals
- Addressing Modes, Instruction Set, and Programming of 8086
- 8086 Interrupts
- Memory and I/O Interfacing
- Features and Interfacing of Programmable Devices for 8086-based Systems
- Multiprocessor Configuration
- 8086-based Systems

Intel 8086 Microprocessor Architecture, Features, and Signals

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Internal architecture of the 8086, which consists of an execution unit and a bus interface unit
- Different general-purpose and segment registers and their functions
- Accessing of instructions and data from the memory using the segment and offset addresses
- Pin details of the 8086
- Functions of the maximum mode and minimum mode signals
- Differences between the 8086 and 8088

3.1 INTRODUCTION

In 1978, Intel released its first 16-bit microprocessor, the 8086, which executes the instructions at 2.5 MIPS (million instructions per second). The execution time for one instruction is 400 ns (= 1/MIPS = $1/(2.5 \times 10^6)$). The 8086 can address 1 MB (1 MB = 2^{20} bytes) of memory, as it has a 20-bit address bus. The width of the data bus in the 8086 is 16 bits. This higher execution speed and larger memory size have enabled the 8086 to replace the smaller minicomputers in many applications. Another feature in the 8086 is the presence of a small six-byte instruction queue in which the instructions fetched from the memory are placed before they are executed.

3.2 ARCHITECTURE OF 8086

The functional block diagram of the 8086 is shown in Fig. 3.1. It is subdivided into the following two units:

(i) An execution unit (EU), which includes the ALU, eight 16-bit general-purpose registers, a 16-bit flag register, and a control unit.

(ii) A bus interface unit (BIU), which includes an adder for address calculations, four 16-bit segment registers (CS, DS, SS, and ES), a 16-bit instruction pointer (IP), a six-byte instruction queue, and bus control logic.

3.2.1 Execution Unit

The EU consists of eight 16-bit general-purpose registers—AX, BX, CX, DX, SP, BP, SI, and DI. Among these registers, AX, BX, CX, and DX can be further divided into two 8-bit registers—AH and AL, BH and BL, CH and CL, and DH and DL, respectively, as shown in Fig. 3.1. The general-purpose registers can

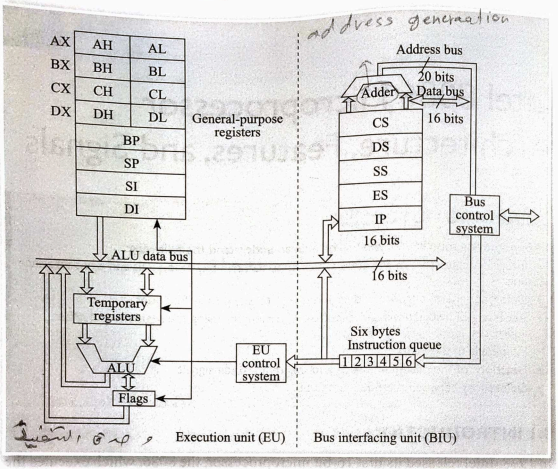


Fig. 3.1 Functional block diagram of the 8086

be used to store 8-bit or 16-bit data during program execution. In addition, each register has the following special functions:

(i) AX/AL: AX or AL is used as the accumulator. It is used in the multiply, divide, and input/output (I/O) operations, and in some decimal and ASCII adjustment instructions.

(ii) BX: The BX register holds the offset address of a location in the memory. It is also used to refer to the data in the memory using the look-up table technique, with the help of the XLAT instruction.

(iii) CX/CL: CX is used to hold the count value while executing the repeated string instructions (REP/REPE/REPNE) and the LOOP instruction. CL is used to hold the count value while executing the shift/rotate instructions. The count value indicates the number of times the same code has to be executed when the LOOP instruction is used, and the number of times the data item has to be shifted/rotated when the shift/rotate instruction is used.

(iv) DX: DX is used to hold a part of the result during a multiplication operation and a part of the dividend before a division operation. It is also used to hold the I/O device address while executing the IN and OUT instructions.

(v) SP: The SP register or the stack pointer is used to hold the offset address of the data stored at the top of the stack segment. SP is used along with the SS register to decide the address at which the data is to be pushed or popped, during the execution of the PUSH or POP instruction, respectively.

(vi) BP: The BP register is called base pointer. It is also used to hold the offset address of the data to be read from or written into the stack segment.

(vii) SI: The SI register is called source index register. It is used to hold the offset address of the source data in the data segment, while executing string instructions.

(viii) DI: The DI register is called destination index register. It is used to hold the offset address of the destination data in the extra segment, while executing string instructions.

Here, the term *segment* refers to a portion of the memory where the data, code, or stack for a program is stored. In the 8086, the maximum size of a segment can be 64 KB and the minimum size can be even 1 byte. A segment always begins at a memory address divisible by 16. This means that the starting address of a segment in the memory in hexadecimal form is XXXX0H. The reason for this is explained in Section 3.2.2.

The flag register of the 8086 is shown in Fig. 3.2.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	OF	DF	IF	TF	SF	ZF	–	AF	–	PF	–	CF

Note: Bits marked '–'— Intel-reserved bits (normally set to 0)

Fig. 3.2 Flag register of the 8086

The flags in the flag register can be classified into status flags and control flags. The flags CF, PF, AF, ZF, SF, and OF are called *status flags,* as they indicate the status of the result that is obtained after the execution of an arithmetic or logic instruction. The flags DF, IF, and TF are called *control flags*, as they control the operation of the CPU. The functions of the different flags are as follows:

(i) CF (carry flag): CF holds the carry after an 8-bit or 16-bit addition or the borrow after an 8-bit or 16-bit subtraction operation.

(ii) PF (parity flag): If the lower eight bits of the result have an odd parity (i.e., odd number of 1s), PF is set to 0. Otherwise, it is set to 1.

(iii) AF (auxiliary carry flag): AF holds the carry after addition or the borrow after subtraction of the bits in the bit position 3 (the LSB is treated as bit position 0). This flag is used by the DAA or the DAS instruction to adjust the value in AL after a BCD addition or subtraction, respectively.

(iv) ZF (zero flag): ZF indicates that the result of an arithmetic or logic operation is zero. If Z = 1, the result is zero and if Z = 0, the result is not zero.

(v) SF (sign flag): SF holds the arithmetic sign of the result after an arithmetic or logic instruction is executed. If S = 0, the sign bit is 0 and the result is positive.

(vi) TF (trap flag): TF is used to debug a program using the single-step technique. If it is set (i.e., TF = 1), the 8086 gets interrupted (trap or single-step interrupt) after the execution of each instruction in the program. If TF is cleared (i.e., TF = 0), the trapping or debugging feature is disabled.

(vii) DF (direction flag): DF selects either the increment or decrement mode for the DI and/or SI register, during the execution of string instructions. If D = 0, the registers are automatically incremented; if D = 1, the registers are automatically decremented. This flag can be set and cleared using the STD and CLD instructions, respectively.

(viii) IF (interrupt flag): IF controls the operation of the INTR interrupt pin of the 8086. If IF = 0, the INTR pin is disabled and if IF = 1, the INTR pin is enabled. This flag can be set and cleared using the STI and CLI instructions, respectively.

(ix) OF (overflow flag): Signed negative numbers are represented in the 2's complement form in the microprocessor. When signed numbers are added or subtracted, an overflow may occur. An overflow indicates that the result has exceeded the capacity of the machine. For example, if the 8-bit signed data 7EH (= +126) is added with the 8-bit signed data 02H (= +2), the result is 80H (= −128 in the 2's complement form). This result indicates an overflow condition and the overflow flag is set during the given signed addition operation. In an 8-bit register, the minimum and maximum value of the signed number that can be stored is −128 (= 80H) and +127 (= 7FH), respectively. In a 16-bit register, the minimum and maximum value of the signed number that can be stored is −32,768 (= 8000H) and +32,767 (= 7FFFH), respectively. For operations on unsigned data, OF is ignored.

3.2.2 Bus Interface Unit

There are four segment registers CS, DS, SS, and ES in the 8086. The function of these registers is to indicate the starting or base address of the code segment, data segment, stack segment, and extra segment, respectively, in the memory. The code segment contains the instructions of a program and the data segment contains data for the program. The stack segment holds the stack of the program, which is needed while executing the CALL and RET instructions and also to handle interrupts. The extra segment is an additional data segment that is used by some string instructions.

The base address of any segment can be obtained by appending four binary 0s to the farthest right portion of the content of the corresponding segment register, which is the same as appending the hexadecimal digit 0. It is also equivalent to shifting the content of the segment register left by four bits. Hence, a segment in the 8086 always starts at a memory address that is divisible by the decimal number 16 (also known as 16-byte boundary). This is illustrated with an example as follows:

Example 3.1

Let us assume that the segment registers have the following values stored in them:

CS	DS	SS	ES
2000H	4000H	6000H	8000H

The base address of the code segment is obtained by appending four binary 0s (same as the hexadecimal digit 0) to the content of CS. Therefore, the base

address is 20000H. Similarly, the base address of the data segment, stack segment, and extra segment are 40000H, 60000H, and 80000H, respectively. Figure 3.3 shows the location of these segments in the memory.

If the size of two different segments is less than 64 KB, it is possible that the two segments may overlap (i.e., another segment may begin within the 64 KB allocated to a segment). For example, let a particular application in the 8086 require a code segment of size 1 KB and a data segment of size 2 KB. If the code segment is stored in the memory from the address 20000H, it will end at the memory address 203FFH. The data segment can be stored from the address 20400H (which is the next immediate 16-byte boundary in the memory). The CS and DS registers are loaded with the values 2000H and 2040H, respectively, for running this application in the 8086.

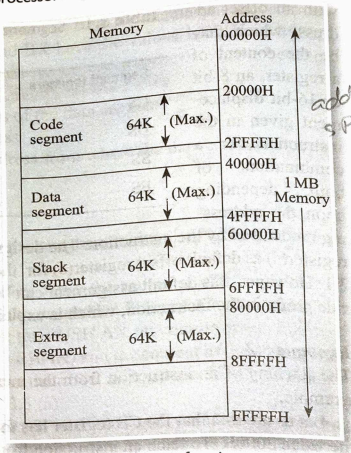


Fig. 3.3 Location of various segments in memory

3.2.3 Minimum and Maximum Mode Operations

The 8086 can be operated in either minimum or maximum mode. By connecting the MN/$\overline{MX}$ pin to logic 1, the 8086 is operated in minimum mode. In the minimum mode of operation, the 8086 itself generates all the control signals. There is a single 8086 in the minimum mode system. The other components in a minimum mode 8086 system are latches, transceivers, clock generator, memory, and I/O devices. Chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

The 8086 is operated in the maximum mode by connecting the MN/$\overline{MX}$ pin to the ground. In this mode, the 8086 generates the status signals and another chip called bus controller (8288) generates the control signals using this status information. In the maximum mode, there may be more than one 8086 in the system configuration. The other components in the system are the same as in the minimum mode 8086 system.

3.3 ACCESSING MEMORY LOCATIONS

Each address in the physical memory (ROM/EPROM) is called a physical address. To access an operand (either data or instruction) from a particular segment of the memory, the 8086 has to first calculate the physical address of that operand. To accomplish this task, the 8086 adds the base address of the corresponding segment

with an offset address, which may be the content of a register, an 8-bit or 16-bit displacement given in the instruction, or a combination of both, depending upon the addressing mode used by the instruction. The designers of the 8086 have assigned certain register(s) as default offset register(s) for the segment registers, as shown in Table 3.1. However, this default assignment can be changed by using the segment override prefix in the instruction, which is explained in Chapter 4 (Section 4.2).

Table 3.1 Segment registers and default offset registers in the 8086

Segment registers	Default offset registers
CS	IP
DS	BX, SI, DI, 8- or 16-bit displacement
SS	SP and BP
ES	DI for string instructions

Example 3.2

The fetching of an instruction from the memory in the 8086 is explained in this example.

Let us assume that the CS register has the value 3000H and the IP register has the value 2000H. To fetch an instruction from the memory, the CPU calculates the memory address from which the next instruction is to be fetched, as follows:

CS × 10H = 30000H Base address of the code segment
+ IP = 2000H → Offset address
32000H → Memory address from where the next instruction is taken

Example 3.3

Let us see the fetching of data from the memory using the DS and BX registers, with an example. Consider the execution of the instruction MOV AX, [BX].

The square bracket around BX in this instruction indicates that the data specified by the BX register is in the memory; the BX register holds the offset address of

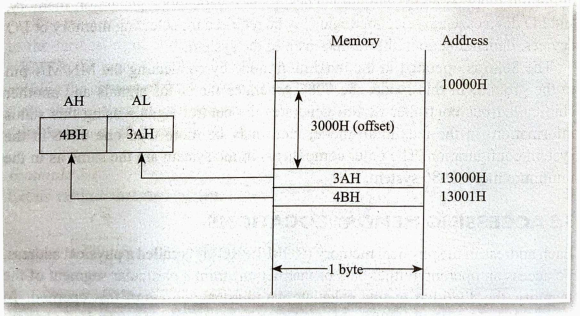


Fig. 3.4 Execution of the instruction MOV AX, [BX]

the data in the data segment. The data obtained from the memory is moved to the AX register. Let us assume that DS and BX have the values 1000H and 3000H, respectively. To calculate the memory address from where the data has to be taken, the CPU does the following operation:

DS × 10H =10000H ⟶ Base address of the data segment
+ BX = 3000H ⟶ Offset address
13000H ⟶ Memory address from where the data is taken

This is also explained in Fig. 3.4.

Example 3.4

Let us see the pushing of data into the stack segment using the PUSH instruction, with an example.

Assume that the SS and SP registers have the values 3000H and 0105H, respectively. Consider the execution of the instruction PUSH AX by the 8086. The steps carried out by the 8086 to execute the PUSH AX instruction are as follows:

(i) SP is decremented by 1 (i.e., SP = 0104H) and the content of the AH register (higher byte of AX) is pushed into the offset address specified by SP in the stack segment, as shown in Fig. 3.5 (a).

(ii) SP is again decremented by 1 (i.e., SP = 0103H) and the content of the AL register (lower byte of AX) is pushed into the offset address specified by SP in the stack segment, as shown in Fig. 3.5 (b).

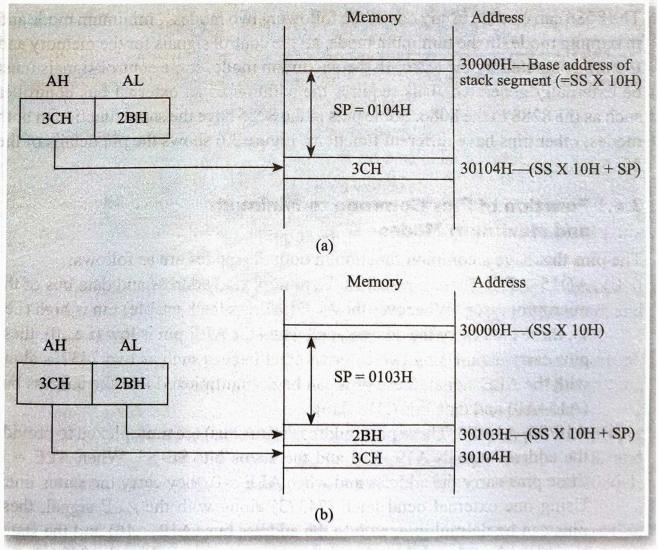


Fig. 3.5 PUSH AX (a) Pushing the first byte of AX onto the stack segment (b) Pushing the second byte of AX onto the stack segment

The instruction queue is six bytes long and stores the pre-fetched instructions from the code segment. From there, the instruction is taken to the instruction decoder, where it is decoded. The decoder passes the decoded information to the timing and control circuit, which in turn generates the various control signals to execute the instruction. Whenever this decoded instruction requires branching (which arises when conditional or unconditional jump instructions are decoded), the instruction queue is flushed and the instruction bytes from the branch address are fetched into the queue. The BIU fetches the instruction bytes from the memory whenever the EU is not using the address/data bus and puts them in the instruction queue. Hence, fetching and execution of instructions can take place simultaneously. Thus the instruction queue reduces the execution time of a program.

The segment and offset mechanism for accessing the memory in the 8086 allows the programmer to write relocatable programs or data structures. A *relocatable program* or data structure is one that can be placed anywhere in the memory map of the 8086 and executed without any modification. This is not possible in the 8085 microprocessor. In a relocatable program, the jump instructions use only relative values (positive or negative) with respect to the program counter, using which the jump address is calculated. In addition, in a relocatable data structure, the data is referred to using the offset address in the data segment or the extra segment.

3.4 PIN DETAILS OF 8086

The 8086 can operate in any one of the following two modes—minimum mode and maximum mode. In the minimum mode, all the control signals for the memory and I/O are generated by the 8086. In the maximum mode, some control signals must be externally generated. This requires the addition of an external bus controller such as the 8288 to the 8086. Some pins in the 8086 have the same function in both modes; other pins have different functions. Figure 3.6 shows the pin details of the 8086.

3.4.1 Function of Pins Common to Minimum and Maximum Modes

The pins that have a common function in both the modes are as follows:

(i) AD15–AD0: These pins act as the multiplexed address and data bus of the microprocessor. Whenever the ALE (address latch enable) pin is high (i.e., 1), these pins carry the address, and when the ALE pin is low (i.e., 0), these pins carry data. Using two external octal latches such as two 74373s along with the ALE signal, these pins can be de-multiplexed into the address bus (A15–A0) and data bus (D15–D0).

(ii) A19/S6–A16/S3: These pins (address/status bus) are multiplexed to provide the address signals A19–A16 and the status bits S6–S3. When ALE = 1, these pins carry the address and when ALE = 0, they carry the status lines. Using one external octal latch (74373) along with the ALE signal, these pins can be de-multiplexed into the address bus (A19–A16) and the status bus (S6–S3). S3 and S4 indicate the segment accessed by the 8086 during the current bus cycle. This is shown in Table 3.2.

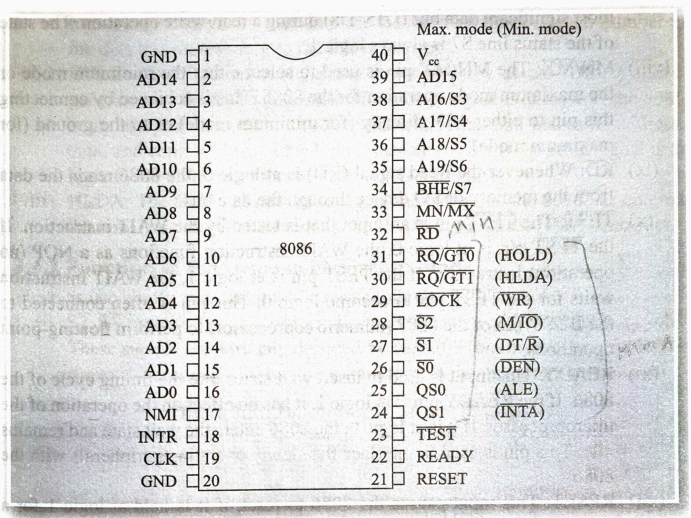


Fig. 3.6 Pin details of the 8086

Table 3.2 Function of status bits S4 and S3

S4	S3	Segment accessed
0	0	Extra segment
0	1	Stack segment
1	0	Code segment or no segment
1	1	Data segment

The status bit S5 indicates the condition of the IF bit; S6 always remains at logic 0.

(iii) NMI: The non-maskable interrupt (NMI) input is a hardware interrupt. It cannot be disabled by software. It is a positive edge-triggered interrupt and when it occurs, the type 2 interrupt occurs in the 8086.

(iv) INTR: The interrupt request (INTR) is a level-triggered hardware interrupt, which depends on the status of IF. When IF = 1, if INTR is held high (i.e., logic 1), the 8086 gets interrupted. When IF = 0, INTR is disabled.

(v) CLK: The clock signal must have a duty cycle of 33% to provide proper internal timing for the 8086. Its maximum frequency can be 5, 8, and 10 MHz for different versions of the 8086—the 8086, 8086-2, and 8086-1, respectively.

(vi) V_{CC}: This power supply pin provides a +5 V signal to the 8086. The variation allowed in the power supply input is ±10%.

(vii) $\overline{BHE}$/S7: The bus high enable ($\overline{BHE}$) pin is used in the 8086 to enable the

most significant data bus (D15–D8) during a read/write operation. The state of the status line S7 is always logic 1.

(viii) MN/$\overline{MX}$: The MN/$\overline{MX}$ pin is used to select either the minimum mode or the maximum mode operation for the 8086. This is achieved by connecting this pin to either +5 V directly (for minimum mode) or to the ground (for maximum mode).

(ix) $\overline{RD}$: Whenever the Read signal ($\overline{RD}$) is at logic 0, the 8086 reads the data from the memory or I/O device through the data bus.

(x) $\overline{TEST}$: The $\overline{TEST}$ pin is an input that is tested by the WAIT instruction. If the $\overline{TEST}$ pin is at logic 0, the WAIT instruction functions as a NOP (no operation) instruction. If the $\overline{TEST}$ pin is at logic 1, the WAIT instruction waits for the $\overline{TEST}$ pin to become logic 0. This pin is often connected to the BUSY pin of the 8087 (numeric coprocessor) to perform floating-point operations.

(xi) READY: This input is used to insert wait states into the timing cycle of the 8086. If the READY pin is at logic 1, it has no effect on the operation of the microprocessor. If it is at logic 0, the 8086 enters the wait state and remains idle. This pin is used to interface the slowly operating peripherals with the 8086.

(xii) RESET: This input causes the 8086 to reset, if it is held at logic 1 for a minimum of four clocking periods. Whenever the 8086 is reset, CS and IP are initialized to FFFFH and 0000H, respectively, and all other registers are initialized to 0000H. This causes the 8086 to begin executing instructions from the memory address FFFF0H.

(xiii) GND: The GND connection is the return for the power supply (V_{CC}). The 8086 has two GND pins and both must be connected to ground for proper operation.

3.4.2 Function of Pins used in Minimum Mode

The pins used in the minimum mode are as follows:

(i) M/$\overline{IO}$: This pin indicates whether the 8086 is performing memory read/write operation (M/$\overline{IO}$ = 1) or I/O read/write operation (M/$\overline{IO}$ = 0).

(ii) $\overline{WR}$: The Write signal indicates that the 8086 is sending data to a memory or I/O device. When $\overline{WR}$ is at logic 0, the data bus contains valid data for the memory or I/O.

(iii) DT/$\overline{R}$: The Data Transmit/Receive signal indicates that the 8086 data bus is transmitting (DT/$\overline{R}$ = 1) or receiving (DT/$\overline{R}$ = 0) data. This signal is used to control the data flow direction in external data bus buffers.

(iv) $\overline{DEN}$: The Data Bus Enable signal activates external data bus buffers. When data is transferred through the data bus of the 8086, this signal is at logic 0. When $\overline{DEN}$ is high, no data flows in the data bus.

(v) ALE: When the Address Latch Enable (ALE) signal is high, it indicates that the 8086 multiplexed address/data bus (AD15–AD0) and multiplexed address/status bus (A19/S6–A16/S3) contain an address, which can be either a memory address or an I/O port address.

(vi) $\overline{INTA}$: The Interrupt Acknowledge signal is a response to the INTR input

pin. The $\overline{INTA}$ signal is used to place the interrupt type or vector number in the data bus, in response to the INTR interrupt.

(vii) HOLD: The Hold input requests a direct memory access (DMA) and is generated by the DMA controller. If the Hold signal is at logic 1, the 8086 completes the execution of the current instruction and places its address, data, and control buses in the high impedance state. If the Hold signal is at logic 0, the 8086 executes the instructions normally.

(viii) HLDA: The Hold Acknowledge signal indicates that the 8086 has entered the hold state and is connected to the HLDA input of the DMA controller.

3.4.3 Function of Pins used in Maximum Mode

The pins used in the maximum mode are as follows:

(i) $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$: The status bits indicate the function of the current bus cycle. These signals are normally decoded by the 8288 (bus controller). Table 3.3 shows the function of these three status bits in the maximum mode.

Table 3.3 Function of $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$ pins

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Function
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive (inactive)

(ii) $\overline{LOCK}$: The Lock output is used to lock peripherals off the system. This pin is activated by using the LOCK prefix on any instruction.

(iii) $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$: The request/grant pins request DMA during the maximum mode operation of the 8086. These lines are bidirectional and are used to request and grant a DMA operation.

(iv) QS1 and QS0: The queue status bits show the status of the internal instruction queue in the 8086. These pins are provided for access by the numeric coprocessor (8087). Table 3.4 shows the function of the QS1 and QS0 bits.

Table 3.4 Function of QS1 and QS0 pins

QS1	QS0	Function
0	0	Queue is idle (or no operation).
0	1	First byte of opcode is read from the queue.
1	0	Queue is empty.
1	1	Subsequent byte of opcode is read from the queue.

3.5 DIFFERENCES BETWEEN 8086 AND 8088

Intel 8088 is the predecessor of the 8086 processor. Both the processors are 16-bit processors with identical architectures and instruction sets, but they have minor differences. Both the processors are made with the high performance metal oxide semiconductor (HMOS) technology with the 40-pin dual in-line package. The data in the 8088 is 8 bits whereas it is 16 bits in the 8086. The 8088 is developed with the provision of connecting external interfaces such as the 8255, 8253, 8259, etc. so that all the existing circuits built around the 8085 can work as before with the 8088 but with more flexibility in programming and all other features of the 8086. The differences between the 8086 and 8088 are listed in Table 3.5.

Table 3.5 Differences between the 8086 and the 8088

Intel 8086	Intel 8088
• Data bus is 16 bits wide.	• Data bus is 8 bits wide.
• The 8086 has the signal $\overline{M}$/IO in pin 28.	• The corresponding signal in the 8088 is the $\overline{IO}$/M pin (complement of that in the 8086).
• Pin number 34 is $\overline{BHE}$/S7—bus high enable/status signal.	• Pin number 34 is $\overline{SSO}$—status output signal. In the maximum mode, it is always high. In the minimum mode, the pin is logically equivalent to $\overline{S0}$ in the 8086.
• The 8086 has a 6-byte instruction queue. At least two bytes must be free to fetch the next instruction into the queue.	• The 8088 has a 4-byte instruction queue. A single free byte in the instruction queue is enough to fetch the next instruction into the queue.
• There are two memory banks in the 8086, namely, odd and even banks, and the total memory size is 1 MB, which is accessed by segment-offset mechanism.	• There is a single memory bank in the 8088, and the total memory size is 1 MB, which is accessed by segment-offset mechanism.
• There are two I/O banks in the 8086, namely, odd and even banks, and the total I/O address space is 64 KB. Both 8-bit and 16-bit I/O ports can be interfaced with the 8086.	• There is a single I/O bank in the 8088, and the total I/O address space is 64 KB. Only 8-bit I/O ports can be interfaced with the 8088.

Figures 3.7 and 3.8 show the pin details of the 8086 and 8088 processors, respectively.

Both the processors can be operated in the minimum and maximum modes. The difference between the two processors lies in the pin numbers 28 and 34. The 8088 has only an 8-bit data bus and so, the bus AD0–AD7 acts as the multiplexed address and data bus. In the 8086, the bus AD0–AD15 acts as the multiplexed address and data bus as the data bus is 16 bits wide.

The instruction sets for the 8086 and the 8088 are common; hence the programs written for one processor can be executed in the other. Both the processors are said to have software compatibility.

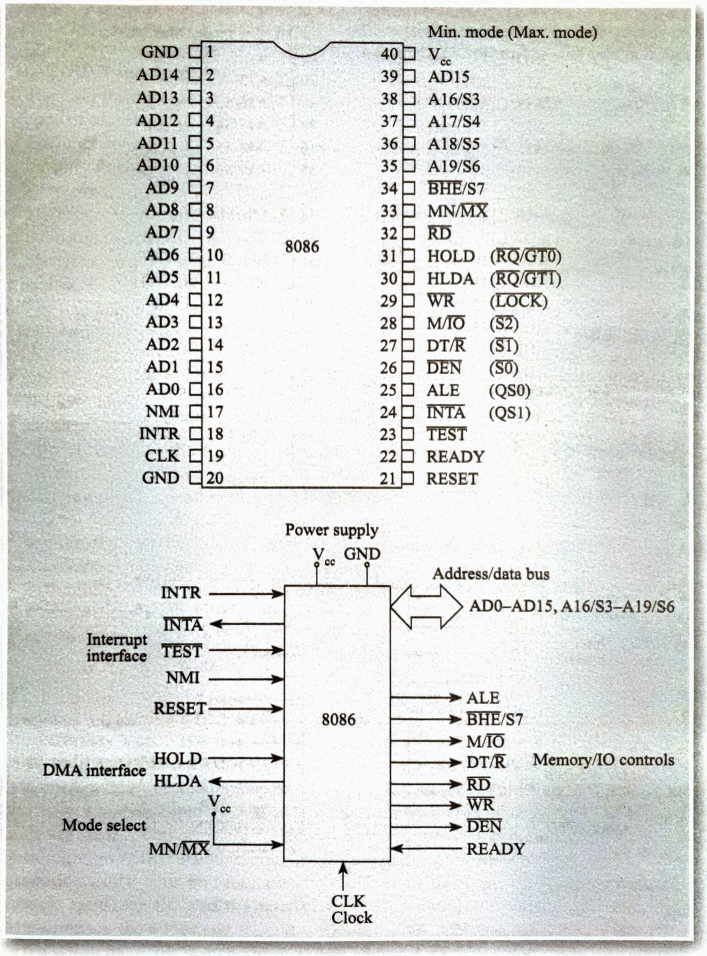


Fig. 3.7 Pin details of the 8086 processor

The major difference between the two processor-based systems lies in the design of the interface between the memory and the I/O devices. The 8086 uses two banks of memory, namely, lower or even memory bank connected to the data bus D7–D0, and higher or odd memory bank connected to the data bus D15–D8. This is because the 8086 has a 16-bit data bus and the memory chips are available with 8-bit data bus only. In the 8088, only one memory bank is interfaced with the

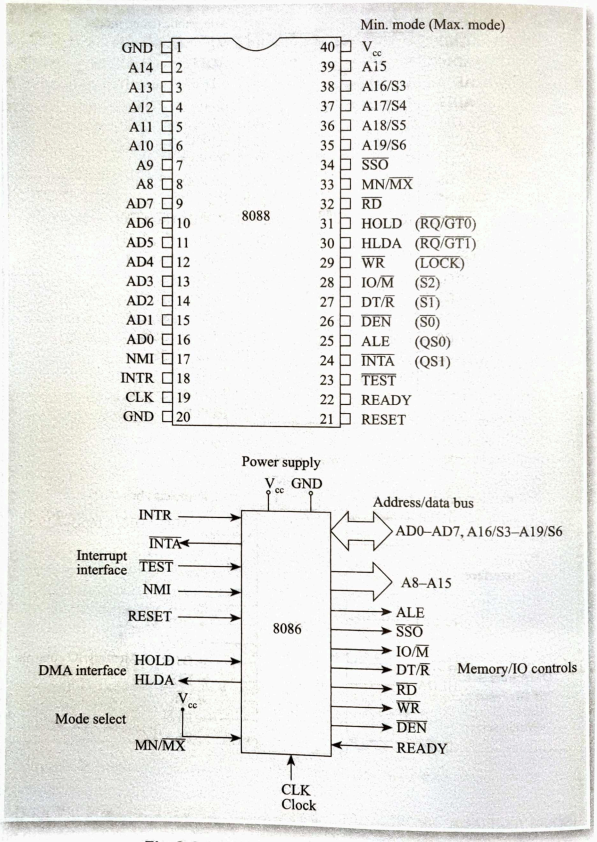


Fig. 3.8 Pin details of the 8088 processor

processor with the available 8-bit data bus. Similarly, there are two I/O banks in the 8086, namely, odd and even banks, and the total I/O address space is 64 KB. There is a single I/O bank in the 8088, and the total I/O address space is 64 KB. In this book, only the 8086 is considered for programming and interfacing. All the programming aspects are common to both 8086 and 8088 processors. The difference in interfacing is explained in the Section 6.9 of Chapter 6.

POINTS TO REMEMBER

- The internal architecture of the 8086 mainly contains two units—the bus interface unit (BIU) and the execution unit (EU).
- The BIU fetches instructions and data from the memory to the processor, using the content of a segment register and an offset.
- There exists a six-byte instruction queue in the 8086, which is used to store the recently fetched instructions in the CPU. This is used to speed up the execution of a program.
- There are four memory segments—code, data, stack, and extra segments in the 8086 and their base address is indicated by adding four binary 0s to the right of the corresponding segment register's content. The maximum size of a memory segment is 64 KB.
- For fetching either an instruction byte or a data, the 8086 adds the base address of the particular segment with an offset address present in a register or available as an 8- or 16-bit displacement in the instruction, or obtained by a combination of both.
- The designers of the 8086 have fixed the default offset register(s) for every segment register. However, this can be changed using the segment override prefix in the instruction.
- The EU contains the ALU, general-purpose registers, and the flag register, which are used during the execution of an instruction.
- The flag register contain different flags, which can be classified as status flags and control flags. The status flags reflect the result of arithmetic and logical operations, and the control flags control the operation during execution of instructions.
- The 8086 can be operated in minimum mode or maximum mode.
- In the 8086, the size of the address bus and data bus is 20 bits and 16 bits, respectively. The 8086 can access a maximum memory size of 1 MB ($= 2^{20}$), as it has a 20-bit address bus.

KEY TERMS

Bus interface unit This unit BIU includes an adder for address calculations, four 16-bit segment registers (CS, DS, SS, and ES), a 16-bit instruction pointer (IP), a six-byte instruction queue, and bus control logic. This unit is responsible for fetching the instructions and data into the 8086 from the memory or I/O device.

Code segment This segment contains the instructions of a program.

Data segment This segment contains the data for a program.

Execution unit This unit includes the ALU, eight 16-bit general-purpose registers, a 16-bit flag register, and the control unit. This unit is responsible for executing instructions in the 8086.

Extra segment This is an additional data segment used by some string instructions.

Flags These show information related to the result of the arithmetic or logic operation performed in the ALU. Flags in the flag register can be classified as status flags and control flags.

Instruction queue It is six bytes long in the 8086 and stores the pre-fetched instructions from the memory. It is used to speed up the execution of a program.

Maximum mode operation In this mode, some control signals must be externally generated, using a bus controller such as the 8288.

Minimum mode operation In this mode, all control signals for the memory and I/O are generated by the microprocessor itself.
Offset This is a 16-bit number that is added to the base address of a segment, to select a byte of instruction or data from the memory.
Relocatable program It is the one that can be placed anywhere in the memory map of the 8086 and executed without any modification.
Segment register This register indicates the starting or base address of a segment in the memory.
Stack segment This segment holds the stack of a program.

REVIEW QUESTIONS

1. What is the size of the address bus and data bus in the 8086?
2. What is meant by multiplexed address and data bus?
3. Draw the register organization of the 8086 and explain typical applications of each register.
4. How is the 20-bit physical memory address calculated in the 8086 processor?
5. Write the different memory segments used in the 8086 and their functions.
6. List the segment registers and their default offset registers in the 8086.
7. What are the steps involved when PUSH BX is executed by the 8086?
8. Write the function of the DF, IF, and TF bits in the 8086.
9. The content of the different registers in the 8086 is CS = F000H, DS = 1000H, SS = 2000H, and ES = 3000H. Find the base address of the different segments in the memory.
10. If the current content of the CS and IP registers is FFFFH and 0000H, respectively, from which memory location will the 8086 fetch the next instruction?
11. If the content of the DS and BX registers is 2500H and 1000H, respectively, from which memory location will the 8086 fetch the data, while executing the instruction MOV CX, [BX]?
12. If the content of the SS and SP registers is 5000H and 1000H, respectively, in which memory location is the content of DX saved, when the 8086 executes the instruction PUSH DX?
13. What is the difference between the minimum and maximum mode operation of the 8086?
14. What is the supply to be given to the V_{CC} input of the 8086?
15. What is the maximum frequency and duty cycle of the clock signal given to the 8086?
16. What is the function of the $\overline{BHE}$ and ALE signals in the 8086?
17. Which pins of the 8086 are used to enable and control the external data bus buffers?
18. What is the minimum time for which the Reset input must be activated for proper reset of the 8086?
19. What are the contents of the CS and IP registers immediately after the reset of the 8086?
20. What is meant by DMA operation? Which pins of the 8086 are used to perform the DMA operation in the minimum and maximum modes of the 8086?
21. What is the role of the status lines S4 and S3 in the 8086?
22. What is the function of the $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$ signals in the maximum mode operation of the 8086?

23. What is the role of the $\overline{\text{TEST}}$ pin in the 8086?
24. Explain the architecture of the 8086 with a neat functional block diagram.
25. Explain the function of the different flags in the 8086.
26. What are the differences between the 8086 and 8088 processors?

THINK AND ANSWER

1. How much memory, in terms of bytes, can be interfaced with the 8086? Why?
2. What is the minimum and maximum size of a segment in terms of bytes? Why?
3. Why is memory divided into segments in the 8086? What are its advantages?
4. How many 8K × 8 memory chips are required to construct a 1 MB memory?
5. Which pin of the 8086 determines the mode of operation? How?
6. What are the differences between NMI and INTR interrupts in the 8086?
7. Which pin of the 8086 is used to synchronize the slowly operating peripherals with the 8086? How?
8. Is it possible for a segment to begin at a memory address that is not divisible by 16 (i.e., the address that does not end with the digit 0H) in the 8086? Why?
9. Is it possible for two segments to overlap in the 8086? Why?
10. Why is the stack segment said to be growing downwards in the 8086?
11. Mention the differences between 8085 and 8086 microprocessors.

4

Addressing Modes, Instruction Set, and Programming of 8086

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Different addressing modes and instruction formats in the 8086
- Function of data transfer, arithmetic, logical, shift/rotate, flag manipulation, string, program control transfer, and processor control instructions in the 8086
- Assembly language programming of the 8086
- 8086 assembler and function of assembler directives

4.1 ADDRESSING MODES IN 8086

There are different addressing modes in the 8086. The addressing mode indicates the way in which the operand or data for an instruction is accessed and the way in which the microprocessor calculates the branch address for the jump, call, and return instructions. We can classify the addressing modes in the 8086 under five categories:

(i) Register addressing mode
(ii) Immediate addressing mode
(iii) Data memory addressing modes
(iv) Program memory addressing modes
(v) Stack memory addressing mode

Let us see each addressing mode in detail.

4.1.1 Register Addressing Mode

In this addressing mode, the data present in the register is moved or manipulated and the result is stored in the register.

Example:

```
(a) MOV AL, BL        ; Move the content of BL to AL.
(b) MOV CX, BX        ; Move the content of BX to CX.
(c) ADD CL, BL        ; Add the contents of CL and BL and store the
                        result in CL.
(d) ADC BX, DX        ; Add the contents of BX, the carry flag, and
                        DX, and store the result in BX.
```

4.1.2 Immediate Addressing Mode

In this mode, the destination can be either a memory location or a register. The data can be 8 bits or 16 bits wide and is directly given in the instruction.

Example:

```
(a) MOV AL, 50H        ; Move the data 50H to AL.
(b) MOV BX, 23A0H      ; Move the data 23A0H to BX.
(c) MOV [SI], 43C0H    ; Move the data 43C0H to the memory at [SI].
```

In the last example, [SI] represents the memory location in the data segment at the offset address specified by the SI register.

4.1.3 Data Memory Addressing Modes

The term *effective address* (EA) represents the offset address of the data within a segment, which is obtained by different methods, depending upon the addressing mode that is used in the instruction. Let us assume that the various registers in the 8086 have the following values (Table 4.1) stored in them, throughout the discussion of data memory addressing modes.

Table 4.1 Values stored in different registers of the 8086

Register	CS	DS	SS	ES	BX	BP	SI	DI
Stored value	1000H	3000H	4000H	6000H	2000H	1000H	1000H	3000H

The different data memory addressing modes are as follows:

(i) Direct addressing: In this mode, the 16-bit offset address of the data within the segment is directly given in the instruction.

Example:

(a) `MOV AL, [1000H]`

In this instruction, the effective address is 1000H. Since the destination is an 8-bit register (i.e., AL), a byte is taken from the memory at the address given by DS × 10H + EA (= 31000H) and stored in AL.

(b) `MOV BX, [2000H]`

EA = 2000H in this instruction. Since the destination is a 16-bit register (i.e., BX), a word is taken from the memory address DS × 10H + EA (= 32000H) and stored in BX. (Note: Since a word contains two bytes, the bytes present at the memory addresses 32000H and 32001H are moved to BL and BH, respectively.)

(ii) Base addressing: In this mode, EA is the content of the BX or BP register. When the BX register is present in the instruction, data is taken from the data segment and when BP is present, data is taken from the stack segment.

Example:

(a) `MOV CL, [BX]`

EA = (BX) = 2000H

Memory address = DS × 10 + (BX) = 32000H. The byte from the memory address 32000H is read and stored in CL.

(b) `MOV DX, [BP]`

EA = (BP) = 1000H

Memory address = SS × 10 + (BP) = 41000H. The word from the memory address 41000H is read and stored in DX.

(iii) Base relative addressing: In this mode, EA is obtained by adding the content of the base register with an 8-bit or 16-bit displacement. The displacement is a signed number with negative values represented in 2's complement form. The 16-bit displacement can have values from −32768 to +32767 and the 8-bit displacement can have values from −128 to +127.

Example:

(a) `MOV AX, [BX + 5]`

EA = (BX) + 5

Memory address = DS × 10H + (BX) + 5
= 30000H + 2000H + 5 = 32005H

The word from the memory address 32005H is read and stored in AX.

(b) `MOV CH,[BX – 100H]`

EA = (BX) − 100H

Memory address = DS × 10H + (BX) − 100H
= 30000H + 2000H − 100H = 31F00H

The byte from the memory address 31F00H is read and stored in CH.

(iv) Index addressing: In this mode, EA is the content of the SI or DI register, which is specified in the instruction. The data is taken from the data segment.

Example:

(a) `MOV BL, [SI]`

EA = (SI) = 1000H

Memory address = DS × 10H + SI
= 30000H + 1000H = 31000H

A byte from the memory address 31000H is read and stored in BL.

(b) `MOV CX, [DI]`

EA = (DI) = 3000H

Memory address = DS × 10H + (DI)
= 30000H + 3000H = 33000H

A word from the memory address 33000H is read and stored in CX.

(v) Index relative addressing: This mode is the same as the base relative addressing mode, except that instead of the BP or BX register, the SI or DI register is used.

Example:

(a) `MOV BX, [SI – 100H]`

EA = (SI) − 100H

Memory address = DS × 10H + (SI) − 100H
= 30000H + 1000H − 100H = 30F00H

A word from the memory address 30F00H is read and stored in BX.

(b) `MOV CL, [DI + 10H]`

EA = (DI) + 10H

Memory address = DS × 10H + (DI) + 10H
= 30000H + 3000H + 10H = 33010H

A byte from the memory address 33010H is read and stored in CL.

(vi) Base plus index addressing: In this mode, EA is obtained by adding the content of a base register and an index register.

Example:

```
MOV AX, [BX + SI]
```

EA = (BX) + (SI)

Memory address = DS × 10H + (BX) + (SI)
= 30000H + 2000H + 1000H = 33000H

A word from the memory address 33000H is taken and stored in AX.

Base relative, index relative, and base plus index addressing modes are used to access a byte or word type data one by one, from a table or an array of data stored in the data segment.

(vii) Base relative plus index addressing: In this mode, EA is obtained by adding the content of a base register, an index, and a displacement.

Example:

(a)
```
MOV CX, [BX + SI + 50H]
```

EA = (BX) + (SI) + 50H

Memory address = DS × 10H + (BX) + (SI) + 50H
= 30000H + 2000H + 1000H + 50H
= 33050H

A word from the memory address 33050H is read and stored in CX.

Base relative plus index addressing is used to access a byte or a word in a particular record of a specific file in the memory. An application program may process many files stored in the data segment. Each file contains many records and a record contains a few bytes or words of data. In base relative plus index addressing, the base register may be used to hold the offset address of a particular file in the data segment; the index register may be used to hold the offset address of a particular record within that file; the relative value is used to indicate the offset address of particular byte or word within that record.

4.1.4 Program Memory Addressing Modes

Program memory addressing modes are used with the JMP and CALL instructions and consist of three distinct forms—direct, relative, and indirect.

(i) Direct addressing: Direct program memory addressing stores both the segment and the offset address where the control has to be transferred with the opcode, as shown in Fig. 4.1.

This instruction is equivalent to JMP 32000H. When it is executed, the 16-bit offset value 2000H is loaded in the IP register and the 16-bit segment value 3000H is loaded in CS. When the microprocessor calculates the memory address from where it has to fetch an instruction using the relation CS × 10H + IP, the address 32000H is obtained using the given CS and IP values.

This type of jump is known as *inter-segment jump,* using which the microprocessor can jump to any memory location within the memory system (i.e., within 1 MB). It is also known as *far jump*. The inter-segment or FAR CALL

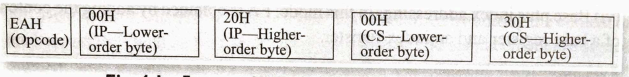


Fig. 4.1 Format of JMP instruction (direct addressing)

instruction also uses direct program memory addressing. While using the assembler to develop the 8086 program, the assembler directive FAR PTR is sometimes used to indicate the inter-segment jump instruction.

Example:

(a) JMP FAR PTR COMPUTE
(b) JMP FAR PTR SIMULATE

In these examples, COMPUTE and SIMULATE are the labels of memory locations that are present in code segments other than the ones in which these instructions are present.

(ii) Relative addressing: The term *relative* here means relative to the instruction pointer (IP). Relative JMP and CALL instructions contain either an 8-bit or a 16-bit signed displacement, which is added to the current instruction pointer. Based on the new value of IP thus obtained, the address of the next instruction to be executed is calculated using the relation CS × 10H + IP.

The 8-bit or 16-bit signed displacement allows a forward or a reverse memory reference, depending on the sign of the displacement. If the displacement is positive, PC is incremented by the displacement value and if it is negative, PC is decremented by the magnitude of the displacement value. A one-byte displacement is used in the short jump and call instructions, and a two-byte displacement is used in the near jump and call instructions. Both types are considered *intra-segment jumps,* since the program control is transferred anywhere within the current code segment.

An 8-bit displacement has a jump range between +127 and −128 bytes from the next instruction, while a 16-bit displacement has a jump range between −32,768 and +32,767 bytes from the instruction following the jump instruction in the program. The opcode of the relative short jump and near jump instructions are EBH and E9H, respectively.

While using an assembler to develop the 8086 program, the assembler directives SHORT and NEAR PTR are used to indicate the short jump and near jump instructions, respectively.

Example:

(a) JMP SHORT OVER
(b) JMP NEAR PTR FIND

In these examples, OVER and FIND are the labels of memory locations that are present in the same code segment in which these instructions are present.

(iii) Indirect addressing: The indirect jump or CALL instructions use a 16-bit register (AX, BX, CX, DX, SP, BP, SI, or DI), a relative register ([BP], [BX], [DI], or [SI]), or a relative register with displacement. The opcode of the indirect jump instruction is FFH. It can be either an inter-segment indirect jump or an intra-segment indirect jump.

If a 16-bit register holds the jump address in an indirect JMP instruction, the operation is a near jump. If the CX register contains 2000H and the JMP CX instruction present in a code segment is executed, the microprocessor jumps to the offset address 2000H in the current code segment to take the next instruction for execution (this is done by loading the IP with the content of CX, without changing the content of CS).

When the instruction JMP [DI] is executed, the microprocessor first reads a word in the current data segment from the offset address specified by DI and places that word in the IP register. Now, with this new value of IP, the 8086 calculates the address of the memory location to which it has to jump, using the relation CS × 10H + IP.

Example:
Let us assume that the registers DS, DI, and CS have the values 1000H, 2000H, and 3000H, respectively. When JMP [DI], present at the offset address 1500H in the code segment 3000H is executed, the microprocessor reads a word from the address given by DS × 10H + DI (= 12000H) in the memory, and loads it in the instruction pointer (IP). Let us assume that the word that is stored in the address 12000H is 4000H. Hence, the program counter will be loaded with the value 4000H. Now, the microprocessor fetches the next instruction for execution from the address given by CS × 10H + IP (= 3000H × 10H + 4000H = 34000H).

4.1.5 Stack Memory Addressing Mode

The stack is used to hold data temporarily during program execution and also store the return address for procedures and interrupt service routines. The stack memory is a last-in, first-out (LIFO) memory. Data are placed into the stack using the PUSH instruction and taken out using the POP instruction. The CALL instruction uses the stack to hold the return address for procedures and the RET instruction is used to remove the return address from the stack.

The stack segment is maintained by two registers—the stack pointer (SP) and the stack segment (SS) register. Data is pushed into or popped from the stack as words (16-bit data), since bytes (8-bit data) cannot be used with the PUSH and POP instructions. Whenever a word of data is pushed into the stack, the higher-order eight bits of the word are placed in the memory location specified by SP − 1 (i.e., at the address SS × 10H + SP − 1) and the lower-order eight bits of the word are placed in the memory location specified by SP − 2 in the current stack segment (i.e., at the address SS × 10H + SP − 2). SP is then decremented by 2. The data pushed into the stack may be the content of a 16-bit register, a segment register, or a 16-bit data in the memory.

Since SP gets decremented for every push operation, the stack segment is said to be growing downwards, as for successive push operations, data are stored in the lower memory addresses in the stack segment. Due to this, SP is initialized with the highest offset address, according to the user's requirement, at the beginning of the program.

Example:

```
(a) PUSH AX          ; Push the content of AX into the stack.
(b) PUSH DS          ; Push the content of DS into the stack.
(c) PUSH [BX]        ; Push the content of the memory location at
                       the offset address specified by BX in the
                       current data segment, into the stack.
```

The PUSHF instruction is used to push the flag register's content into the stack.

Whenever a word is popped from the stack, the lower-order eight bits of the word are removed from the memory location specified by SP and the higher-order eight bits of the word are removed from the memory location specified by SP + 1 in the current stack segment. SP is then incremented by two.

Example:

```
(a) POP BX           ; Pop the content of BX from the stack.
(b) POP ES           ; Pop the content of ES from the stack.
(c) POP [BP]         ; Pop the content of the memory location at
                       the offset address specified by BP in the
                       current stack segment, from the stack.
```

The POPF instruction is used to pop a word stored in the stack and move it to the flag register.

4.2 SEGMENT OVERRIDE PREFIX

The segment override prefix, which can be added to almost any instruction in any memory related addressing mode, allows the programmer to deviate from the default segment and offset register mechanism. The segment override prefix is an additional byte that appears in at the beginning of an instruction, to select an alternative segment register. The JMP and CALL instructions cannot be prefixed with the segment override prefix, since they use only the code segment (CS) register for address generation.

Example:

The MOV AX, [BP] instruction accesses data within the stack segment by default, since BP is the offset register for the stack segment. However, if the programmer wants to get data from the data segment using BP as the offset register in this instruction, the instruction should be modified as MOV AX, DS: [BP].

Table 4.2 shows the instructions that address memory segments other than the default ones.

Table 4.2 Instructions that include the segment override prefix

Instruction	Default segment	Accessed segment
MOV BX, ES:[BP]	SS	ES
MOV BX, SS:[DI]	DS	SS
MOV CX, ES:[BX]	DS	ES

(Contd)

Table 4.2 Instructions that include the segment override prefix *(Contd)*

Instruction	Default segment	Accessed segment
MOV CX, ES:[SI]	DS	ES
MOV AX, CS:[BX]	DS	CS

4.3 INSTRUCTION FORMAT OF 8086

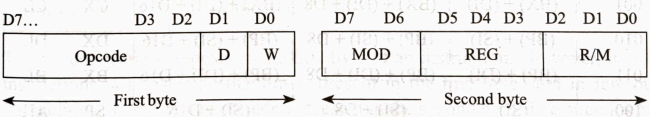
The instruction format, which is the representation of an instruction in machine language, has one or more fields associated with it. The first field is called opcode or operation code field, which indicates the type of the operation to be performed by the 8086. The other fields in the instruction format are known as operand fields. The 8086 executes the instruction using the information present in these fields. There are six general instruction formats in the 8086. The length of an instruction may vary from one byte to six bytes. The instruction formats of the 8086 are explained here:

4.3.1 One-byte Instruction

This format is only one byte long, and it may have implied register or data operands. The least significant three bits of the opcode are used to specify the register operand, if any. Otherwise, all the eight bits in the instruction form an opcode and the operands are implied.

4.3.2 Register to Register

This format is two bytes long. The first byte of the code specifies the opcode. The width of the operand is specified by the W bit. The second byte of the instruction indicates the register operand and R/M (register/memory) field, as given here:



The register represented by the REG field is one of the operands and is given in Table 4.3. When the MOD field's bits (i.e., bits D7 and D6) are 1, the R/M field is also treated as a REG field. The direction (D) bit indicates whether the data is transferred from the register (if D = 0) or to the register (if D = 1).

Table 4.3 Assignment of codes for different registers in the 8086

W bit	Register code	Register	W bit	Register code	Register
0	000	AL	0	010	DL
0	001	CL	0	011	BL
0	100	AH	1	010	DX
0	101	CH	1	011	BX
0	110	DH	1	100	SP

(Contd)

Table 4.3 Assignment of codes for different registers in the 8086 *(Contd)*

W bit	Register code	Register	W bit	Register code	Register
0	111	BH	1	101	BP
1	000	AX	1	110	SI
1	001	CX	1	111	DI

Register code	Register	Register code	Register
00	ES	10	SS
01	CS	11	DS

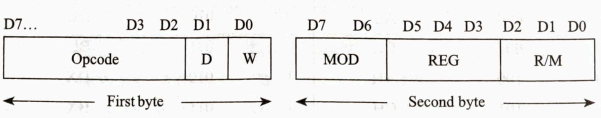
4.3.3 Register to/from Memory with No Displacement

This format is two bytes long. The MOD field indicates the mode of addressing. The MOD, REG, R/M, and W fields are decided as per Table 4.4.

Table 4.4 MOD, REG, R/M, and W fields for different addressing modes

Operands	Memory operand			Register operand	
	No displacement	8-bit Displacement	16-bit Displacement		
MOD / R/M	00	01	10	11	
				W = 1	W = 0
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AX	AL
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CX	CL
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DX	DL
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BX	BL
100	(SI)	(SI) + D8	(SI) + D16	SP	AH
101	(DI)	(DI) + D8	(DI) + D16	BP	CH
110	D16	(BP) + D8	(BP) + D16	SI	DH
111	(BX)	(BX) + D8	(BX) + D16	DI	BH

In Table 4.4, D16 and D8 represent 16-bit and 8-bit displacements, respectively. The instruction format is given here:



4.3.4 Register to/from Memory with Displacement

This type of instruction format contains two bytes as in the previous instruction format and one or two additional bytes for displacement, namely, DISP-LOW (third byte) and DISP-HIGH (fourth byte), which are the lower-order and higher-order bytes of displacement, respectively.

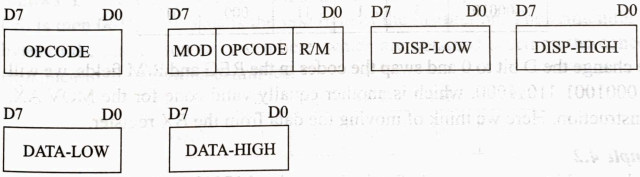
4.3.5 Immediate Operand to Register

In this format, the first byte and the bits D3–D5 in the second byte are used for opcode. It also contains one or two bytes of immediate data as shown here:

DATA-LOW and DATA-HIGH are the lower-order and higher-order bytes of data, respectively.

4.3.6 Immediate Operand to Memory with 16-Bit Displacement

This instruction format is five or six bytes long. The first two bytes contain the opcode, MOD, and R/M fields. Then two bytes of displacement and two bytes of data are present as shown here:



The opcode usually appears in the first byte of an instruction. However, in a few instructions, a register destination is present in the first byte, and in few other instructions, their 3 bits of opcode is present in the second byte. The opcodes have different single-bit indicators, which are as follows:

(i) W bit—This bit indicates whether the instruction operates on 8-bit data for which W = 0 or 16-bit data for which W = 1.

(ii) D bit—This bit is present in double operand instructions. One of the operands in the instruction must be a register specified by the REG field, which will be the source operand if D = 0. Otherwise, it is a destination operand for which D = 1. D bit is also called direction bit.

(iii) S bit—This bit is the sync-extension bit. S bit is always used with the W bit to show the different types of operations as given here:

(a) When S = W = 0, it indicates 8-bit operation with an 8-bit immediate data.

(b) When S = 0 and W = 1, it indicates 16-bit operation with a 16-bit immediate data.

(c) When S = 1 and W = 1, it indicates 16-bit operation with a sign-extended immediate data.

(iv) V bit—This bit is used in shift and rotate instructions. It is set to 0 if shift count is 1 and to 1 if the CL register contains the shift count.

(v) Z bit—This bit is used by the REP instruction to control the loop. If the Z bit is 1, the string instruction with REP prefix is executed until the zero flag matches the Z bit.

The following examples give the machine language coding of a few instructions in the 8086:

Example 4.1

Find the machine language code for the instruction MOV AX, BX.

Solution:

This instruction will move a word from the BX register to the AX register. The 6-bit opcode for the MOV instruction is 100010. Since a word is moved, W = 1. The D bit for this instruction code is made either 0 or 1, depending on how we interpret the instruction. If we think of the instruction as moving a word to AX, then make D = 1 and put 000 in the REG field to represent the AX register. The MOD field is made 11 to represent register addressing mode. The R/M field is made 011 to represent BX register. The resultant code for the MOV AX, BX instruction will be as follows:

Opcode	D	W	MOD	REG	R/M
100010	1	1	11	000	011

If we change the D bit to 0 and swap the codes in the REG and R/M fields, we will get 10001001 11011000, which is another equally valid code for the MOV AX, BX instruction. Here we think of moving the data from the BX register.

Example 4.2

Find the machine language code for the instruction MOV DL, [BX].

Solution:

The opcode of the MOV instruction is 100010. The bit D is made 1 because the data is being moved to DL. The W bit is made 0, because a byte is moved into DL. Next the 3-bit code for the DL register, which is 010, is put in the REG field of the second byte of the instruction code. The MOD and R/M fields are filled with bits 00 and 111, respectively. Assembling all these bits together, the machine language code of the MOV DL, [BX] instruction is obtained as follows:

Opcode	D	W	MOD	REG	R/M
100010	1	0	00	010	111

Example 4.3

Find the machine language code for the instruction MOV [SI + 50H], CL.

Solution:

The opcode of the MOV instruction is 100010. The value 001 is put in the REG field to represent the CL register. D is made 0 because we are moving data from the CL register. W is made 0 since we are moving a byte. The R/M and MOD fields

are set to 100 and 01, respectively, since the addressing mode is of the general form [SI + D8). Putting all these bits together, we get the first two bytes of the instruction, as follows:

Opcode	D	W	MOD	REG	R/M
100010	0	0	01	001	100

The displacement 50H is inserted after these two bytes, as the third byte of the instruction.

Example 4.4

Find the machine language code for the instruction MOV CS: [BX], AL.

Solution:

This instruction moves the data in the AL register to the memory location whose address is given by CS X 10H + BX. The CS: in the instruction is called a segment override prefix. When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put before the code for the rest of the instruction. The code byte for the segment override prefix is 001XX110, in which we insert a 2-bit code in place of the X's to indicate which segment base has to be added to the effective address. As given in Table 4.3, the 2-bit codes are as follows: ES = 00, CS = 01, SS = 10, and DS = 11. The segment override prefix for CS: is then 00101110. This is the first byte of the instruction. The remaining two bytes of the instruction are given here, which are obtained based on the concepts explained in the previous examples.

Opcode	D	W	MOD	REG	R/M
100010	0	0	00	000	111

4.4 INSTRUCTION SET OF 8086

The instructions of the 8086 are classified as data transfer, arithmetic, logical, flag manipulation, control transfer, shift/rotate, string, and machine control instructions.

4.4.1 Data Transfer Instructions

The data transfer instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LSS, LAHF, and SAHF. These instructions are discussed here in detail:

(i) MOV: The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number. The general format of the MOV instruction is MOV destination, source.

Example:

```
(a) MOV BL, 50H        ; Move immediate data 50H to BL.
(b) MOV CX, [BX]       ; Copy the word from the memory at [BX] to CX.
(c) MOV AX, CX         ; Copy the contents of CX to AX.
```

Note: [BX] indicates the memory location at the offset address specified by BX in the data segment.

(ii) PUSH: The PUSH instruction is used to store the word in a register or a memory location into the stack, as explained in the stack addressing mode. SP is decremented by two after the execution of PUSH.

Example:

```
(a) PUSH CX          ; PUSH the content of CX into the stack.
(b) PUSH DS          ; PUSH the content of DS into the stack.
(c) PUSH [BX]        ; PUSH the word in the memory at [BX] into
                       the stack.
```

(iii) POP: The POP instruction copies the top word from the stack to a destination specified in the instruction. The destination can be a general-purpose register, a segment register, or a memory location. After the word is copied to the specified destination, SP is incremented by two.

Example:

```
(a) POP BX           ; Pop the content of BX from the stack.
(b) POP DS           ; Pop the content of DS from the stack.
(c) POP [SI]         ; Pop a word from the stack and store it in
                       the memory at [SI].
```

Note: [SI] indicates the memory location in the data segment at the offset address specified by SI.

(iv) XCHG: The XCHG instruction exchanges the contents of a register with the contents of a memory location. It cannot exchange the contents of two memory locations directly. The source and destination must both be either words or bytes. The segment registers cannot be used in this instruction.

Example:

```
(a) XCHG AL, BL      ; Exchanges the content of AL and BL.
(b) XCHG CX, BX      ; Exchanges the content of CX and BX.
(c) XCHG AX, [BX]    ; Exchanges the content of AX with the content
                       of the memory at [BX].
```

(v) XLAT: The XLAT instruction is used to translate a byte in AL from one code to another code. The instruction replaces a byte in the AL register with a byte in the memory at [BX], which is one of the data items present in a look-up table.

Before XLAT is executed, the look-up table containing the desired codes must be put in the data segment and the offset address of the starting location of the look-up table is stored in BX. The code byte to be translated is put in AL. When XLAT is executed now, it adds the content of the AL with BX to find the offset address of the data in the look-up table. Further, the byte in that offset address will get copied to AL.

(vi) IN: The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data is stored in AL and if a 16-bit port is read, the data is stored in AX. The IN instruction has two formats—fixed port and variable port.

In the *fixed port type* IN instruction, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

Example:

```
IN AL, 80H      ; Input a byte from the port with address 80H to AL.
IN AX, 40H      ; Input a word from the port with address 40H to AX.
```

For the *variable port type* IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Hence, we will be able to address up to 65,536 ports in this mode. The following example shows a part of a program having the IN instruction. The operations done when the instructions are executed are given in the corresponding comment fields.

Example:

```
MOV DX, 0FE50H          ; Initialize DX with the port address FE50H.
IN AL, DX               ; Input a byte from the 8-bit port with port
                          address FE50H into AL.
IN AX, DX               ; Input a word from the 16-bit port with port
                          address FE50H into AX.
```

The drawback of the fixed port type IN instruction is that the port address cannot be changed once the program is stored in the ROM. The variable port type IN instruction has the advantage that the port address can be computed in the program during execution, and by loading it in DX, the corresponding port can be accessed using the IN instruction.

(vii) OUT: The OUT instruction transfers a byte from AL or a word from AX to the specified port. Similar to the IN instruction, the OUT instruction has two forms—fixed port and variable port.

Examples for fixed port OUT instruction:

```
(a) OUT 48H, AL         ; Sends the content of AL to the port with
                          address 48H.
(b) OUT 0F0H, AX        ; Sends the content of AX to the port with
                          address FOH.
```

Examples for variable port OUT instruction:

The following example shows a part of a program having the OUT instruction.

```
MOV DX, 1234H  ; Load the port address 1234H in DX.
OUT DX, AL     ; Send the content of AL to the port with address
                 1234H.
OUT DX, AX     ; Send the content of AX to the port with address
                 1234H.
```

(viii) LEA (load effective address): The general format of the LEA instruction is LEA register, source. This instruction determines the offset address of the variable or memory location called the source and puts this offset address in the indicated 16-bit register.

Example:

(a) `LEA BX, COST` ; Load BX with the offset address of COST in the data segment, where COST is the name assigned to a memory location in the data segment.

(b) `LEA CX, [BX + SI]` ; Load CX with the value equal to (BX) + (SI), where (BX) and (SI) represent the content of BX and SI, respectively.

(ix) LDS: This instruction loads the register and DS with words from the memory. The general form of this instruction is LDS register, memory address of first word.

The LDS instruction copies a word from the memory location specified in the instruction into the register, and then copies a word from the next memory location into the DS register. LDS is useful in initializing the SI and DS registers to point to the start of a string before using one of the string instructions.

Example:

```
LDS SI, [2000H]     ; Copy the content of the memory at the
                      offset address 2000H in the data segment to
                      the lower-order byte of SI, and the content
                      of 2001H to the higher-order byte of SI. Copy
                      the content at the offset address 2002H in
                      the data segment to the lower-order byte of
                      DS and the content of 2003H to the higher-
                      order byte of DS.
```

(x) LES and LSS: The LES and LSS instructions are similar to the LDS instruction, except that instead of the DS register, the ES and SS registers, respectively, are loaded, along with the register specified in the instruction.

(xi) LAHF: This instruction copies the lower-order byte of the flag register into AH.

(xii) SAHF: This instruction stores the content of AH in the lower-order byte of the flag register.

Except the SAHF and POPF instructions, no other data transfer instruction affects the flag register.

4.4.2 Arithmetic Instructions

The arithmetic instructions in the 8086 are used to perform addition, addition with carry, subtraction, subtraction with borrow, increment, decrement, negation (changing sign), comparison, multiplication, division, decimal-adjust after addition, decimal-adjust after subtraction, and processing of ASCII data. Let us now discuss each instruction in detail.

(i) ADD: The general format of the ADD instruction is ADD destination, source.

The data from the source and destination are added and the result is placed in the destination. The source may be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the

source and destination cannot both be memory locations. The data from the source and destination must be of the same type (either bytes or words).

Example:

```
(a) ADD BL, 80H          ; Add the immediate data 80H to BL.
(b) ADD CX, 12B0H        ; Add the immediate data 12B0H to CX.
(c) ADD AX, CX           ; Add the content of AX and CX and store the
                           result in AX.
(d) ADD AL, [BX]         ; Add the content of AL and the byte from the
                           memory at [BX] and store the result in AL.
(e) ADD CX, [SI]         ; Add the content of CX and the word from the
                           memory at [SI] and store the result in CX.
(f) ADD [BX], DL         ; Add the content of DL with the byte from the
                           memory at [BX] and store the result in the
                           memory at [BX].
```

The flags AF, CF, OF, PF, SF, and ZF are affected by the execution of the ADD instruction.

ADD with carry

(ii) ADC: This instruction adds the data in the source and destination with the content of the carry flag and stores the result in the destination. The general format of this instruction is ADC destination, source.

All the rules specified for ADD are applicable to the ADC instruction.

(iii) SUB: The general form of the subtract (SUB) instruction is SUB destination, source. It subtracts the number in the source from the number in the destination and stores the result in the destination. Like the ADD instruction, the source may be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and destination cannot both be memory locations. The data from the source and destination must be of the same type (either bytes or words).

For subtraction, the carry flag (CF) functions as the borrow flag. If the result is negative after subtraction, CF is set. Otherwise, it is reset. The flags AF, CF, OF, PF, SF, and ZF are affected by the SUB instruction.

Example:

```
(a) SUB AL, BL           ; Subtract BL from AL and store the result in
                           AL.
(b) SUB CX, BX           ; Subtract BX from CX and store the result in
                           CX.
(c) SUB BX, [DI]         ; Subtract the word in the memory at [DI]
                           from BX and store the result in BX.
(d) SUB [BP], DL         ; Subtract DL from the byte in the memory at
                           [BP] and store the result in the memory at
                           [BP].
```

(iv) SBB: Subtract with borrow—The general form of this instruction is SBB destination, source. The SBB instruction subtracts the content of the source and the carry flag from the content of the destination and stores the result in the

destination. The rules for the source and the destination are same as that for the SUB instruction. AF, CF, OF, PF, SF, and ZF are affected by this instruction.

(v) INC: The increment (INC) instruction adds 1 to the content of a specified register or a memory location. The data incremented may be a byte or word. While the carry flag is not affected by this instruction, the flags AF, OF, PF, SF, and ZF are affected.

Example:

```
(a) INC CL              ; Increment the content of CL by 1.
(b) INC AX              ; Increment the content of AX by 1.
(c) INC BYTE PTR [BX]   ; Increment the byte in the memory at [BX] by 1.
(d) INC WORD PTR [SI]   ; Increment the word in the memory at [SI] by 1.
```

In these examples, the terms BYTE PTR and WORD PTR are assembler directives, which are used to specify the type of data (byte or word) to be incremented in the memory.

(vi) DEC: The decrement (DEC) instruction subtracts 1 from the content of a specified register or memory location. The data decremented may be a byte or a word. CF is not affected, but AF, OF, PF, SF, and ZF flags are affected by this instruction.

(vii) NEG: The negate (NEG) instruction replaces the byte or word in the specified register or memory location by its 2's complement (i.e., changes the sign of the data). The CF, AF, SF, PF, ZF, and OF flags are affected by this instruction.

Example:

```
(a) NEG AL              ; Take 2's complement of the data in AL and
                          store it in AL.
(b) NEG CX              ; Take 2's complement of the data in CX and
                          store it in CX.
(c) NEG BYTE PTR [BX]   ; Take 2's complement of the byte in the memory
                          at [BX] and store the result in the same
                          place.
(d) NEG WORD PTR [SI]   ; Take 2's complement of the word in the memory
                          at [SI] and store the result in the same place.
```

(viii) CMP: The general form of the compare (CMP) instruction is CMP destination, source. This instruction compares a byte or word in the source with a byte or word in the destination and affects only the flags, according to the result. The content of the source and destination are not affected by the execution of this instruction. The comparison is done by subtracting the content of the source from that of the destination. The AF, OF, SF, ZF, PF, and CF flags are affected by the instruction. The rules for the source and destination are the same as those for the SUB instruction.

Example:

After the instruction CMP AX, DX is executed, the status of CF, ZF, and SF will be as follows:-

	CF	ZF	SF
If AX = DX	0	1	0

If AX > DX	0	0	0
If AX < DX	1	0	1

(ix) MUL: The multiply (MUL) instruction is used for multiplying two unsigned bytes or words. The general form of the MUL instruction is MUL source. The source can be a byte or a word from a register or memory location, which is considered as the multiplier. The multiplicand is taken by default from AL and AX for byte and word type data, respectively. The result of multiplication is stored in AX and DX–AX (i.e., the most significant word of the result in DX and the least significant word of the result in AX) for byte and word type data, respectively. (Note: Multiplying two 8-bit data gives a 16-bit result and multiplying two 16-bit data gives a 32-bit result.)

Example:

```
(a) MUL CH              ; Multiply AL and CH and store the result in
                          AX.
(b) MUL BX              ; Multiply AX and BX and store the result in
                          DX–AX.
(c) MUL BYTE PTR [BX]   ; Multiply AL with the byte in the memory at
                          [BX] and store the result in AX.
```

If the most significant byte of the 16-bit result is 00H or the most significant word of a 32-bit result is 0000H, both CF and OF will be 0. Checking these flags allows us to decide whether the leading 0s in the result have to be discarded or not. The AF, PF, SF, and ZF flags are undefined (i.e., a random number is stored in these bits) after the execution of the MUL instruction.

(x) IMUL: The IMUL instruction is used for multiplying the signed byte or word in a register or memory location with AL or AX, and store the result in AX or DX–AX, respectively. If the magnitude of the result does not require all the bits of the destination, the unused bits are filled with copies of the sign bit.

If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0s or all 1s), CF and OF will both be 0. Otherwise, both will be 1. AF, PF, SF, and ZF are undefined after IMUL.

To multiply a signed byte by a signed word, the byte is moved into a word location and the upper byte of the word is filled with copies of the sign bit. If the byte is moved into AL, using the CBW (convert byte to word) instruction, the sign bit in AL is extended into all the bits of AH. Thus, AX contains the 16-bit sign-extended word.

Example:

```
(a) IMUL BL             ; Multiply AL with BL and store the
                          result in AX.
(b) IMUL AX             ; Multiply AX and AX and store the
                          result in DX-AX.
(c) IMUL BYTE PTR [BX]  ; Multiply AL with the byte from the
                          memory at [BX] and store the result in
                          AX.
```

```
(d) IMUL WORD PTR [SI]        ; Multiply AX with the word from the
                                memory at [SI] and store the result in
                                DX–AX.
```

(xi) DIV: The divide (DIV) instruction is used for dividing unsigned data. The general form of the DIV instruction is DIV source, where 'source' is the divisor. It can be a byte or word in a register or memory location. The dividend is taken by default from AX and DX–AX for byte and word type data division, respectively. Table 4.5 shows the complete details of the DIV instruction.

Table 4.5 Details of DIV instruction

Dividend (bits)	Divisor (bits)	Quotient (bits)	Remainder (bits)
AX (16)	Source (8)	AL (8)	AH (8)
DX–AX (32)	Source (16)	AX (16)	DX (16)

If an attempt is made to divide by 0 or if the quotient is too large to fit in AL or AX (i.e., if the result is greater than FFH in 8-bit division or FFFFH in 16-bit division), the 8086 automatically generates a type 0 interrupt. All flags are undefined after a DIV instruction.

Example:

```
(a) DIV DL          ; Divide the word in AX by the byte in DL.
                      The quotient is stored in AL and the remainder
                      in AH.
(b) DIV CX          ; Divide the double word (32 bits) in DX-AX by
                      the word in CX. The quotient is stored in AX
                      and the remainder in DX.
(c) DIV BYTE PTR [BX] ; Divide the word in AX by the byte from the
                        memory at [BX]. The quotient is stored in AL
                        and the remainder in AH.
```

(xii) IDIV: The IDIV instruction is used for dividing signed data. The general form and the rules for the IDIV instruction are same as those for the DIV instruction. The quotient is a signed number and the sign of the remainder is the same as the sign of the dividend.

To divide a signed byte by a signed byte, the dividend byte is put in AL and using the CBW (convert byte to word) instruction, the sign bit of the data in AL is extended to AH. Thus, the byte in AL is converted to a signed word in AX. To divide a signed word by a signed word, the dividend byte is put in AX and using the CWD (convert word to double word) instruction, the sign bit of the data in AX is extended to DX. Thus, the word in AX is converted to a signed double word in DX–AX.

If an attempt is made to divide by 0 or if the quotient is too large or too small to fit in AL and AX for 8- and 16-bit division, respectively (i.e., either the result is greater than the decimal value +127 in 8-bit division or the decimal value +32,767 in 16-bit division, or the result is less than the decimal value −128 in 8-bit division

or the decimal value −32,767 in 16-bit division), the 8086 automatically generates a type 0 interrupt. All flags are undefined after a DIV instruction.

(xiii) DAA: Decimal adjust AL after BCD addition—This instruction is used to get the result of addition of two packed BCD numbers (in a packed BCD number, two decimal digits are represented as eight bits) as a BCD number. The result of addition must be in AL for DAA to work correctly. If the lower nibble (four bits) in AL is greater than 9 after addition or if the AF flag is set by the addition, the DAA instruction adds 6 to the lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the carry flag is set by the addition, the DAA instruction adds 60H to AL.

Example:

(a) Let AL = 01011000 = 58 BCD
CL = 00110101 = 35 BCD

Consider the execution of the following instructions:

```
ADD AL, CL      ; AL = 10001101 = 8DH and AF = 0 after execution
DAA             ; Add 0110 (decimal 6) to AL, since lower nibble in AL
                  is greater than 9
                ; AL = 10010011 = 93 BCD and CF = 0
```

Therefore, the result of addition is 93 BCD.

(b) Let AL = 10001000 = 88 BCD
CL = 01001001 = 49 BCD

Consider the execution of the following instructions:

```
ADD AL, CL      ; AL = 11010001 and AF = 1 after execution
DAA             ; Add 0110 (decimal 6) to AL
                ; AL = 11010111 = D7H
                ; Upper nibble 1101 > 9. So add 60H (0110 0000) to AL.
                ; AL = 0011 0111 = 37 BCD and CF = 1
```

The final result is 137 BCD, taking into account the carry generated. The DAA instruction affects AF, CF, PF, and ZF. OF is undefined after the DAA instruction is executed.

(xiv) DAS: Decimal adjust after BCD subtraction—DAS is used to get the result in packed BCD form after subtracting two packed BCD numbers. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or if the AF is set by subtraction, the DAS instruction subtracts 6 from the lower nibble of AL. If the result in the upper nibble is now greater than 9 or if the carry flag is set, the DAS instruction subtracts 60H from AL.

Example:

(a) Let AL = 86 BCD = 10000110
CH = 57 BCD = 01010111

Consider the execution of the following instructions:

```
SUB AL, CH      ; AL = 00101111 = 2FH and CF = 0 after execution
```

```
DAS             ; Lower nibble of the result is 1111. So DAS subtracts
                  06H from AL to make AL = 00101001 = 29 BCD and CF =
                  0 to indicate that there is no borrow.
```

The result is 29 BCD.

(b) Let AL = 49 BCD = 01001001
CH = 72 BCD = 01110010

Consider the execution of the following instructions:

```
SUB AL, CH      ; AL = 1101 0111 = D7H and CF = 1 since result is
                  negative
DAS             ; Subtract 0110 0000 (60H) from AL because upper nibble
                  in AL is greater than 9. This makes AL = 01110111 =
                  77 BCD and CF = 1, indicating that a borrow is
                  needed.
```

The answer is 77 BCD as 149 BCD − 72 BCD = 77 BCD. The value 149 BCD is mentioned here, considering the borrow that is generated after the subtraction.

There are four arithmetic instructions that are used to perform operations on unpacked BCD numbers. In an unpacked BCD number, one decimal digit is represented as an 8-bit number in which the upper four bits are always zero. For example, the decimal digit 3 is represented as 03H in unpacked BCD form.

(xv) AAA: The AAA (ASCII adjust after addition) instruction must always follow the addition of two unpacked BCD operands in AL. When AAA is executed, the content of AL is changed to a valid unpacked BCD number; the upper four bits of AL are cleared. CF is set and AH is incremented if a decimal carry-out from AL is generated.

Example:

Let AL = 05 (decimal) = 00000101
BH = 06 (decimal) = 00000110
AH = 00H

Consider the execution of the following instructions:

```
ADD AL, BH      ; AL = 11 (decimal) and CF = 0
AAA             ; AL = 01 and AH = 01 and CF = 1
```

Addition of 5 and 6 gives a decimal result of 11, which is equal to 0101H in unpacked BCD form. It is stored in AX. When this result is to be sent to the printer, the ASCII code of each decimal digit is easily found by adding 30H to each byte.

(xvi) AAS: ASCII adjust after subtraction—This instruction always follows the subtraction of one unpacked BCD operand from another in AL. It changes the content of AL to a valid unpacked BCD number and clears the top four bits of AL. CF is set and AH is decremented if a decimal borrow occurs.

Example:

(a) Let AL = 09 BCD = 00001001
CL = 05 BCD = 00000101
AH = 00H

Consider the execution of the following instructions:

```
SUB AL, CL      ; AL = 04 BCD
AAS             ; AL = 04 BCD and CF = 0
                ; AH = 00H
```

(b) Let AL = 05 BCD
CL = 09 BCD
AH = 00H

Consider the execution of the following instructions:

```
SUB AL, CL      ; AL = -4 BCD (in 2's complement form AL = FCH) and
                  CF = 1
AAS             ; AL = 04 BCD
                ; CF = 1 indicating that a borrow is needed and
                  AH = FFH = 2's complement of -1
```

AAA and AAS affect the AF and CF flags and OF, PF, SF, and ZF are left undefined. Another salient feature of these two instructions is that it is possible to take input data in the ASCII form of the unpacked decimal number, obtain the result as an unpacked decimal number, and then convert it to ASCII form by adding 30H to it.

(xvii) AAD: ASCII-adjust before the division instruction modifies the dividend in AH and AL, to prepare for the division of two valid unpacked BCD operands. After the execution of AAD, AH is cleared and AL contains the binary equivalent of the original unpacked two-digit numbers. Initially, AH contains the most significant unpacked digit and AL contains the least significant unpacked digit.

Example:
To perform the operation 32 (decimal)/08 (decimal)
Let AH = 03H ; Upper decimal digit in the dividend
AL = 02H ; Lower decimal digit in the dividend
CL = 08H ; Divisor
Consider the execution of the following instructions:

```
AAD             ; AX = 0020H (binary equivalent of the decimal value
                  32 in 16-bit form)
DIV CL          ; Divide AX by CL. AL contains the quotient and AH the
                  remainder.
```

AAD affects the PF, SF, and ZF flags. AF, CF, and OF are undefined after execution of AAD.

(xviii) AAM: The AAM (ASCII adjust AX after multiplication) instruction corrects the value obtained by multiplication of two valid unpacked decimal numbers. The higher-order digit is placed in AH and the lower-order digit in AL.

Example:
Let AL = 05 (decimal)
CL = 09 (decimal)
Consider the execution of the following instructions:

```
MUL CH          ; AX = 002DH = 45 (decimal)
```

```
AAM              ; AH = 04 and AL = 05 (unpacked BCD form of the decimal
                   number 45)
OR AX, 3030H     ; To get the ASCII code of the result in AH and AL
                   (Note: This instruction is used only when the result
                   is needed in ASCII form.)
```

AAM affects the same flags as AAD.

4.4.3 Logical Instructions

The logical instructions in the 8086 include AND, OR, XOR, NOT, and TEST. Let us now discuss each instruction in detail.

(i) AND: The AND instruction performs a logical AND operation between the corresponding bits in the source and destination and stores the result in the destination. The source and the destination can be either bytes or words. The general form of the AND instruction is AND destination, source.

The rules for the destination and source for the AND instruction are the same as those for the ADD instruction. CF and OF are both 0, and PF, SF, and ZF are updated after the AND instruction is executed. AF is undefined. PF is affected only when the AND operation is performed on an 8-bit operand.

(ii) OR: The OR instruction performs a logical OR operation between the corresponding bits in the source and destination and stores the result in the destination. The source and the destination can be either bytes or words. The general form of the OR instruction is OR destination, source.

The rules for the source and destination and the way flags are affected are the same as the AND instruction.

(iii) XOR: The XOR instruction performs a logical XOR operation between the corresponding bits in the source and destination and stores the result in the destination. The source and the destination can be either bytes or words. The general form of the XOR instruction is XOR destination, source.

The rules for the source and destination and the way flags are affected are the same as the AND instruction.

(iv) NOT: The NOT instruction inverts each bit (i.e., performs 1's complement) of the byte or word at a specified destination. The destination can be a register or a memory location. The NOT instruction does not affect any flags.

Example:

```
(a) NOT AL          ; Take 1's complement of AL.
(b) NOT BX          ; Take 1's complement of BX.
(c) NOT [SI]        ; Take 1's complement of the data in the memory
                      at [SI].
```

(v) TEST: This instruction ANDs the content of a source byte or word with the content of the specified destination byte or word. The flags are updated, but neither operand is changed. The TEST instruction is often used to set flags before a conditional jump instruction. The general form of TEST instruction is TEST

destination, source. The rules for the source and destination and the way flags are affected are the same as the AND instruction.

Example:
Let AL = 0111 1111 = 7FH

```
TEST AL, 80H            ; AL = 7FH (unchanged)
```

ZF = 1 since (AL) AND (80H) = 00H; SF = 0; PF = 1

4.4.4 Flag Manipulation Instructions

The 8086 has a few instructions exclusively for performing operations on the flags in the flag register. They are used to set or clear specific flags in the flag register, to push or pop the flag register content into or from the stack, and to transfer the lower-order byte of the flag register to the AH register and vice versa. Table 4.6 indicates the function of the different flag manipulation instructions in the 8086.

Table 4.6 Flag manipulation instructions

Mnemonics	Function
LAHF	Load the lower-order byte of the flag register in AH
SAHF	Store AH in the lower-order byte of the flag register
PUSHF	Push the flag register's content onto the stack
POPF	Pop the top word of the stack onto the flag register
CMC	Complement the carry flag (CF = complement of CF)
CLC	Clear the carry flag (CF = 0)
STC	Set the carry flag (CF = 1)
CLD	Clear the direction flag (DF = 0)
STD	Set the direction flag (DF = 1)
CLI	Clear the interrupt flag (IF = 0)
STI	Set the interrupt flag (IF = 1)

4.4.5 Control Transfer Instructions

The control transfer instructions of the 8086 are used to call a subroutine, return from a subroutine, and branch conditionally or unconditionally in a program. In conditional branching, there are two categories depending on whether unsigned or signed data is involved. The terms 'above' and 'below' are used when referring to the magnitude of unsigned numbers. The binary number 10000000 (= 128 in decimal form) is above the binary number 01000000 (= 64 in decimal form). The terms 'greater' and 'lesser' are used when referring to the relationship between two signed numbers. 'Greater' means more positive. The signed binary number 00001111 (= +15 in decimal form) is greater than the signed binary number 10000001 (= −127 in decimal form). The control transfer instructions of the 8086 are given in Table 4.7.

Table 4.7 Control transfer instructions

Mnemonics	Description
	Unconditional transfers
JMP addr	Jump unconditionally to addr
CALL addr	Call the procedure or subroutine starting at addr
RET	Return from the procedure or subroutine
	Conditional transfers
JA addr	Jump if above to addr (jump if CF = ZF = 0)
JAE addr	Jump if above or equal to addr (jump if CF = 0)
JB addr	Jump if below to addr (jump if CF = 1)
JBE addr	Jump if below or equal to addr (jump if CF = 1 or ZF = 1)
JC addr	Jump if carry to addr (jump if CF = 1)
JCXZ addr	Jump if CX = 0 to addr
JE addr	Jump if equal to addr (jump if ZF = 1)
JG addr	Jump if greater to addr (Jump if ZF = 0 and SF = OF)
JGE addr	Jump if greater or equal to addr (Jump if SF = OF)
JL addr	Jump if lesser to addr (Jump if SF ≠ OF)
JLE addr	Jump if lesser or equal to addr (Jump if ZF = 1 or SF ≠ OF)
JNA addr	Jump if not above to addr (Jump if CF = 1 or ZF = 1)
JNAE addr	Jump if not above or equal to addr (Jump if CF = 1)
JNB addr	Jump if not below to addr (Jump if CF = 0)
JNBE addr	Jump if not below or equal to addr (Jump if CF = ZF = 0)
JNC addr	Jump if no carry to addr (Jump if CF = 0)
JNE addr	Jump if not equal to addr (jump if ZF = 0)
JNG addr	Jump if not greater to addr (jump if ZF = 1 or SF ≠ OF)
JNGE addr	Jump if not greater or equal to addr (jump if SF ≠ OF)
JNL addr	Jump if not lesser to addr (Jump if SF = OF)
JNLE addr	Jump if not lesser or equal to addr (Jump if ZF = 0 and SF = OF)
JNO addr	Jump if no overflow to addr (Jump if OF = 0)
JNP addr	Jump if no parity to addr (Jump if PF = 0)
JNS addr	Jump if no sign to addr (jump if SF = 0)
JNZ addr	Jump if no zero to addr (jump if ZF = 0)

(Contd)

Table 4.7 Control transfer instructions *(Contd)*

Mnemonics	Description
JO addr	Jump if overflow to addr (jump if OF = 1)
JP addr	Jump if parity to addr (jump if PF = 1)
JPE addr	Jump if parity is even to addr (jump if PF = 1)
JPO addr	Jump if parity is odd to addr (jump if PF = 0)
JS addr	Jump if sign to addr (jump if SF = 1)
JZ addr	Jump if zero to addr (jump if ZF = 1)

In this table, 'addr' is the target address in the memory, to which the 8086 has to jump, if the condition is satisfied while executing conditional jump instructions. 'addr' is also the target address to which the 8086 has to jump while executing unconditional jump instructions. In the CALL instruction, 'addr' indicates the address where the subroutine is located. In the case of conditional jump instructions, the target address must be located at a relative address, which is in the range of +127 bytes to −128 bytes from the instruction following the conditional jump instruction.

Some of the conditional jump instructions have identical effects as follows:

JE—JZ	JNE—JNZ	JL—JNGE	JNL—JGE
JG—JNLE	JNG—JLE	JB—JNAE	JNB—JAE
JA—JNBE	JNA—JBE	JP—JPE	JNP—JPO

There are a few instructions in the 8086 that are used to implement loops. These are given in Table 4.8.

Table 4.8 Loop instructions

Mnemonics	Description
LOOP addr	Decrement CX. Go to addr if CX ≠ 0.
LOOPE addr	Loop while equal (Decrement CX. Go to addr if CX ≠ 0 and ZF = 1.)
LOOPZ addr	Same as LOOPE
LOOPNE addr	Loop while not equal (Decrement CX. Go to addr if CX ≠ 0 and ZF = 0.)
LOOPNZ addr	Same as LOOPNE

In this table, 'addr' is the target address, which must be located at a relative address in the range of +127 bytes to −128 bytes from the instruction following the LOOP instruction.

The use of the LOOP instruction in a program is explained here with an example:

```
MOV CX, 100
AGAIN : MOV AL, BL
```

```
...
LOOP AGAIN              ; Decrement CX and if CX ≠ 0, go to AGAIN.
```

In this example, the loop starting from the address AGAIN is repeated 100 times, since CX is initialized to 100.

Finally, the software interrupt-related instructions (given in Table 4.9) can also be used to cause the 8086 to jump to another place in the memory and execute the interrupt service routine for a particular interrupt. The IRET instruction is used to return the control from the interrupt service routine to the main program.

Table 4.9 Interrupt-related instructions

Mnemonics	Description
INT n	Software interrupt instruction where n is the interrupt type. n can be any number between 00H and FFH. This instruction causes the 8086 to execute the interrupt service routine (ISR) of interrupt type n.
INTO	This instruction interrupts the 8086 if there is an overflow (i.e., OF = 1).
IRET	This instruction returns the control from the interrupt service routine to the main program.

The 8086 interrupts are discussed in detail in Chapter 5.

4.4.6 Shift/Rotate Instructions

The shift/rotate instructions perform logical left-shift and right-shift, and arithmetic left-shift and right-shift operations. The arithmetic left-shift (SAL) and logical left-shift (SHL) have the same function, but the former is used on signed data, whereas the latter is used on unsigned data.

(i) SAL/SHL: The general format of the SAL/SHL instruction is SAL/SHL destination, count. The destination can be a register or a memory location and a byte or a word. This instruction shifts each bit in the destination a specified number of bit positions to the left. As a bit is shifted out of the LSB position, a 0 is placed in the LSB position. The MSB is shifted into the carry flag (CF) as follows:

$$CF \longleftarrow MSB \longleftarrow LSB \longleftarrow 0$$

If the number of shifts to be done is 1, it can be directly specified in the instruction, with a count value equal to 1. For shifts of more than one bit position, the desired number of shifts is loaded into the CL register and CL is placed in the count position of the instruction. CF, SF, and ZF are affected according to the result. PF has meaning only when AL is used as the destination. The SAL and SHL instructions can be used to multiply a signed number and unsigned number, respectively, by a power of 2. Shifting a number left by one bit and two bits multiplies the number by two and four, respectively, and so on.

Example:

(a)
```
SAL AX, 1               ; Shift left the content of AX by one
                          bit.
```

(b)
```
SAL BL, 1               ; Shift left the content of BL by one
                          bit.
```

```
(c) SAL BYTE PTR [SI], 1        ; Shift left the byte content of the
                                  memory at [SI] by one bit.
(d) SAL WORD PTR [BX], 1        ; Shift left the word content of the
                                  memory at [BX] by one bit.
(e) MOV CL, 05
    SAL AX, CL                  ; Shift left the content of AX by five
                                  bits.
(f) MOV CL, 03
    SAL BYTE PTR [SI], CL       ; Shift left the byte content of the
                                  memory at [SI] by three bits.
```

(ii) SAR: The general format of the SAR instruction is SAR destination, count. The destination can be a register or a memory location and a byte or a word. This instruction shifts each bit in the destination a specified number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position (i.e., the sign bit is copied into the MSB). The LSB will be shifted into the carry flag (CF) as follows:

$$\text{MSB} \longrightarrow \text{MSB} \longrightarrow \text{LSB} \longrightarrow \text{CF}$$

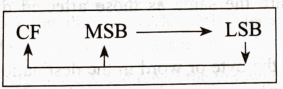
The rules for the count value in the instruction are the same as those for the SAL instruction. CF, SF, and ZF are affected according to the result. PF has meaning only when AL is used as the destination.

(iii) SHR: The general format of the SHR instruction is SHR destination, count. The destination can be a register or a memory location and a byte or a word. This instruction shifts each bit in the destination a specified number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is placed in the MSB position. The LSB is shifted into the carry flag (CF) as follows:

$$0 \longrightarrow \text{MSB} \longrightarrow \text{LSB} \longrightarrow \text{CF}$$

The rules for the count value in the instruction are same as those for the SHL instruction. CF, SF, and ZF are affected according to the result. PF has meaning only when an 8-bit destination is used.

(iv) ROR: This instruction rotates all the bits of the specified byte or word by a specified number of bit positions to the right. The operation done when ROR is executed is as follows:



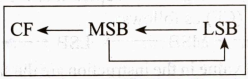
The general format of the ROR instruction is ROR destination, count. The data bit moved out of the LSB is copied into CF. ROR affects only CF and OF. In the single-bit rotate operation, if the sign bit (i.e., the MSB) changes after the execution of ROR, OF is set. This is applicable only for the single-bit rotate operation. ROR is used to swap nibbles in a byte and to swap bytes in a word. It can also be used to rotate a bit in a byte/word into CF, where it can be checked and acted upon by the JC and JNC instructions. CF contains the bit most recently rotated out of the LSB,

in the case of a multiple bit rotate operation. The rules for the count value are same as those for the shift instruction.

Example:

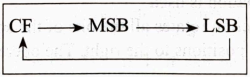
```
(a) ROR CH, 1                ; Rotate right the byte in CH by one bit
                               position.
(b) ROR BX, CL               ; Rotate right the word in BX by the
                               number of bit positions given by CL.
(c) ROR BYTE PTR [SI], 1     ; Rotate right the byte in the memory at
                               offset [SI] by one bit position.
(d) ROR WORD PTR [BX], CL    ; Rotate right the word in the memory at
                               offset [BX] by the number of bit
                               positions given by CL.
```

(v) ROL: ROL rotates all the bits in a byte or word in the destination to the left, by one or more bit positions, using CL, as follows:



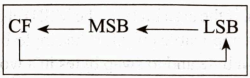
The data bit moved out of the MSB is copied into CF. ROL affects only CF and OF. In the single-bit rotate operation, if the sign bit (i.e., the MSB) changes after the execution of ROL, OF is set. This is applicable only for the single-bit rotate operation. ROL is used to swap nibbles in a byte or swap bytes in a word. It can also be used to rotate a bit in a byte/word into CF, where it can be checked and acted upon by the JC and JNC instructions. CF contains the bit most recently rotated out of the LSB, in the case of the multiple bit rotate operation.

(vi) RCR: RCR rotates the byte or word in the destination right, through the carry flag (CF), either by one bit position or by the number of bit positions given by CL, as follows:



The flags affected are the same as those affected during the execution of ROR.

(vii) RCL: RCL rotates the byte or word in the destination left through the carry flag (CF), either by one bit position or by the number of bit positions given by CL, as follows:



The flags affected are the same as those affected during the execution of ROL.

4.4.7 String Instructions

The 8086 string manipulation instructions are given in Table 4.10. The string instructions operate on elements of strings, bytes, or words. The register SI contains the offset address of an element (byte or word) in the source string, which is present in the data segment. The register DI contains the offset address of an element (byte or word) in the destination string, which is present in the extra segment. The source string is in the data segment at the offset address given by SI; the destination string is in the extra segment at the offset address given by DI. After each string operation, SI and/or DI are automatically incremented or decremented by 1 or 2 (for byte or word operation), according to the D flag in the flag register. If D = 0, SI and/or DI are automatically incremented and if D = 1, SI and/or DI are automatically decremented.

Table 4.10 String instructions in the 8086

Mnemonics	Function
MOVSB	Move the string byte from DS:[SI] to ES:[DI].
MOVSW	Move the string word from DS:[SI] to ES:[DI].
CMPSB	Compare string bytes (done by subtracting the byte at ES: [DI] from the byte at DS: [SI]). Only flags are affected; the content of the bytes compared is unaffected.
CMPSW	Compare string words (done by subtracting the word at ES: [DI] from the word at DS: [SI]). Only flags are affected; the content of the words compared is unaffected.
LODSB	Load the string byte at DS:[SI] into AL.
LODSW	Load the string word at DS:[SI] into AX.
STOSB	Store the string byte in AL at ES:[DI].
STOSW	Store the string word in AX at ES:[DI].
SCASB	Compare string bytes (done by subtracting the byte at ES: [DI] from the byte at AL). Only flags are affected; the content of the bytes compared is unaffected.
SCASW	Compare string words (done by subtracting the word at ES: [DI] from the byte at AX). Only flags are affected; the content of the words compared is unaffected.
REP	Decrement CX and repeat the following string operation if CX ≠ 0.
REPE or REPZ	Decrement CX and repeat the following string operation if CX ≠ 0 and ZF = 1.
REPNE or REPNZ	Decrement CX and repeat the following string operation if CX ≠ 0 and ZF = 0.

The REP (repeat) prefix placed before a string instruction causes the string instruction to be executed CX times.

Example:

```
MOV CX, 32H          ; Load 32H (= decimal 50) in CX.
REP MOVSW            ; Execute MOVSW instruction 50 times.
```

Execution of these two instructions causes the moving of a string having 50 words from the data segment to the extra segment.

4.4.8 Machine or Processor Control Instructions

The machine/processor control instructions in the 8086 include HLT, LOCK, NOP, ESC, and WAIT. Let us discuss each instruction in detail.

(i) HLT: The halt instruction stops the execution of all instructions and places the processor in the halt state. An interrupt or a Reset signal causes the processor to resume execution from the halt state.

(ii) LOCK: The lock instruction provides the processor an exclusive hold on the use of the system bus. It activates an external locking signal (LOCK) of the processor and is placed as a prefix to the instruction for which a lock is to be asserted. The lock functions only with the XCHG, ADD, OR, ADC, SBB, AND, SUB, XOR, NOT, NEG, INC, and DEC instructions, when they involve a memory operand. An undefined opcode trap interrupt is generated, if a LOCK prefix is used with any instruction not listed here.

(iii) NOP: No operation—This instruction is used to insert a delay in software delay programs.

(iv) ESC: This instruction is used to pass instructions to a coprocessor such as the 8087, which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the escape instruction.

As the 8086 fetches instruction bytes from the memory, the coprocessor catches these bytes from the data bus and puts them in a queue. However, the coprocessor treats all the normal 8086 instructions as NOP instructions. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction.

(v) WAIT: When this instruction is executed, the 8086 checks the status of its $\overline{\text{TEST}}$ input pin and if the $\overline{\text{TEST}}$ input is high, it enters an idle condition during which it does not do any processing. The 8086 remains in this state until the 8086's $\overline{\text{TEST}}$ input pin is made low or an interrupt signal is received on the INTR or NMI pins. If a valid interrupt occurs while the 8086 is in this idle state, it returns to the idle state after the interrupt service routine is executed. The WAIT instruction does not affect flags. It is used to synchronize the 8086 with external hardware such as the 8087 coprocessor.

4.5 8086 ASSEMBLY LANGUAGE PROGRAMMING

A large number of assembly language programming examples for the 8086 have been provided in this section. These programs can be converted into machine language programs either by manually finding the opcode for each instruction in the program, or by using an assembler, and executing in an 8086-based system. Since manually finding the opcode of each instruction of the 8086 is time consuming,

the *line assembler* or *assembler* is normally used in converting assembly language programs into machine language programs. The line assembler converts each mnemonic of an instruction immediately into an opcode as it is entered into the system, and is used in microprocessor trainer kits. The line assembler is stored in any one of the ROM-type memories in the trainer kit. The assembler is a software that needs a personal computer for generating the opcodes of an assembly language program. The generated opcodes can be downloaded to the microprocessor-based system such as the microprocessor trainer kit or the microprocessor-based prototype hardware, through the serial or parallel port of the computer.

Many assemblers, such as Microsoft Macro Assembler (MASM), Turbo assembler (TASM), and DOS assembler, are used to convert the 8086 assembly language programs into machine language programs. While using these assemblers, the assembly language program is written using assembler directives. Assembler directives are commands to the assembler to indicate the size of a variable (either byte or word), number of bytes or words to be reserved in the memory, value of a constant, name of a segment, etc., in a program. Assembler directives are not converted into opcode, but are used to generate the proper opcode of an instruction. The use of Microsoft's assembler is discussed in this section.

The immediate data given in the instruction ends with 'H' for hexadecimal data, 'B' for binary data, and 'D' for decimal data. In some assemblers, immediate data without any of these alphabets is treated as decimal data. For hexadecimal data that begins with the alphabets A–F, a zero must be placed before the data (e.g., 0F5H).

4.5.1 Writing Programs using Line Assembler

The following examples illustrate the writing of 8086 assembly language programs using a line assembler, which is used in the 8086-based trainer kits for executing 8086 assembly language programs.

Example 4.5

Write a program to add a word-type data, located at the offset 0800H (least significant byte, LSB) and 0801H (most significant byte, MSB) in the segment address 3000H, to another word-type data located at the offset 0700H (LSB) and 0701H (MSB) in the same segment. Store the result at offset 0900H and 0901H in the same segment. Store the carry generated in the addition in the same segment at offset 0902H.

Flowchart:

The flowchart for this problem is shown in Fig. 4.2.

Program:

```
MOV AX, 3000H    ; Load the value 3000H in AX.
MOV DS, AX       ; Initialize DS with the value 3000H.
MOV AX, [800H]   ; Move the first data word to AX.
ADD AX, [700H]   ; Add AX with the second data word.
MOV [900H], AX   ; Store AX at the offset 900H and 901H.
JC CARRY         ; If carry = 1, jump to CARRY.
```

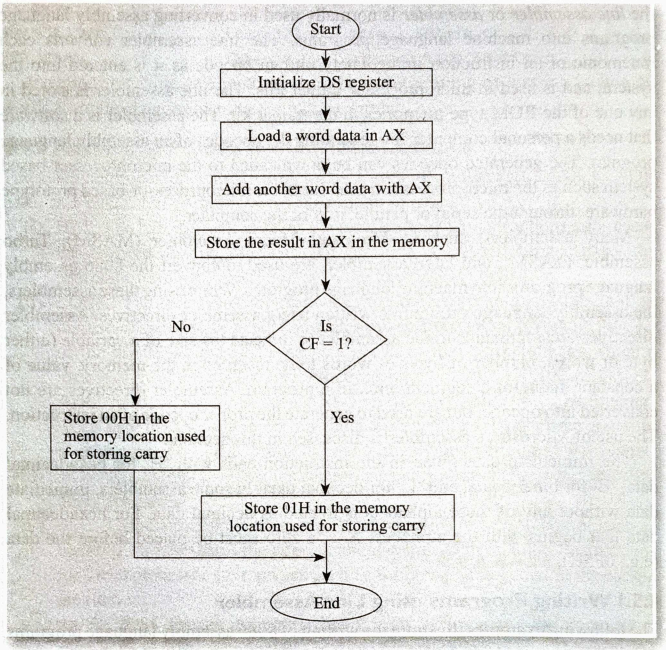


Fig. 4.2 Flowchart for adding two word-type data

```
        MOV [902H], 00H  ; If there is no carry, store 00H at the offset
                           902H.
        JMP END          ; Jump to END.
CARRY:  MOV [902H], 01H  ; Store 01H at the offset 902H.
END:    HLT              ; Terminate program execution.
```

Note:

(i) To initialize a segment register with a value, the value is first loaded in one of the general-purpose registers such as AX or BX. It is then moved to the segment register. In this example, AX is used to load 3000H into DS.

(ii) Instead of the AX register, any other 16-bit general purpose register (BX, CX, etc.) can be used for performing addition.

(iii) Sometimes, instead of using the HLT instruction at the end, the software interrupt instruction (INT) may be used to return control to the monitor program after execution of the program.

Example 4.6

Write a program to add a byte-type data located at the offset address 0800H in the segment address 3000H to another byte-type data located at the offset address

0700H in the same segment. Store the result and the carry generated in the offset addresses 0900H and 0901H in the same segment, respectively.

Flowchart:
The flowchart for this problem is the same as in Fig. 4.2, except that instead of word-type data, byte-type data is used. Hence, instead of AX, AL can be used.

Program:

```
        MOV AX, 3000H     ; Load the value 3000H in AX.
        MOV DS, AX        ; Initialize DS with the value 3000H.
        MOV AL, [800H]    ; Move the first data byte to AL.
        ADD AL, [700H]    ; Add AL with the second data byte.
        MOV [900H], AL    ; Store AL at the offset address 900H.
        JC CARRY          ; If carry = 1, jump to CARRY.
        MOV [901H], 00H   ; If there is no carry, store 00H.
        JMP END           ; Jump to END.
CARRY:  MOV [901H], 01H   ; Store 01H at the offset address 901H.
END:    HLT               ; Terminate program execution.
```

Note: Instead of the AL register, any other 8-bit general purpose register (BL, BH, CL, etc.) can be used in the program.

Example 4.7

Write a program to subtract the byte content of the memory location 3000H: 4000H from the byte content of the memory location 4000H:5000H, and store the result at the location 2000H: 3000H. Assume that the input data is signed data and the negative numbers are represented in 2's complement form. (*Note:* 3000H: 4000H represents the segment address of 3000H and the offset 4000H in that segment.)

Flowchart:
The flowchart for this problem is shown in Fig. 4.3.

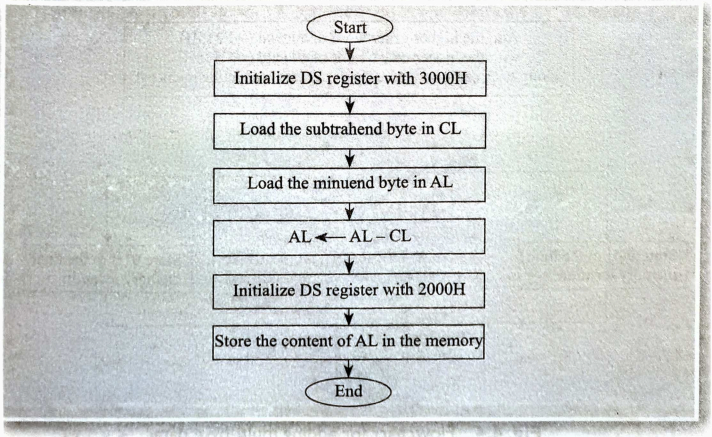


Fig. 4.3 Flowchart for subtracting one byte-type data from another

Program:

```
MOV BX, 3000H    ; Load the value 3000H in BX.
MOV DS, BX       ; Initialize DS with the segment address 3000H.
MOV CL, [4000H]  ; Get the subtrahend from the offset 4000H.
MOV BX, 4000H    ; Load the value 4000H in BX.
MOV DS, BX       ; Initialize DS with the segment address 4000H.
MOV AL, [5000H]  ; Move the minuend at the offset 5000H to AL.
SUB AL, CL       ; AL = AL - CL
MOV BX, 2000H    ; Load the value 2000H in BX.
MOV DS, BX       ; Initialize DS with the segment address 2000H.
MOV [3000H], AL  ; Store AL at the offset 3000H.
HLT              ; Terminate program execution.
```

Note: The result is also signed data and the negative result will be in 2's complement form. If the result is positive, the MSB of the result will be 0, and if the result is negative, the MSB will be 1.

Example 4.8

Write a program to add the multi-byte data F2354687H with C545689FH and store the result starting from the address 1000H: 2000H in the memory, with the lower-order byte of the result stored first.

Flowchart:

The flowchart for this problem is given in Fig. 4.4.

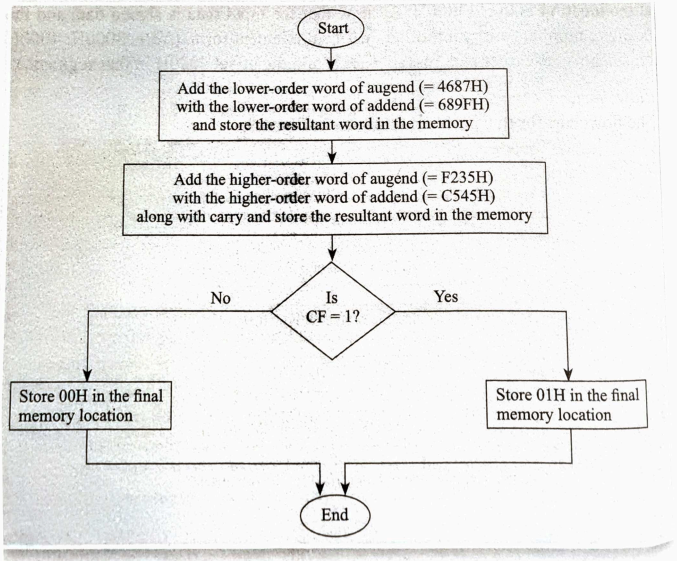


Fig. 4.4 Flowchart for adding multi-byte data

Program:

```
        MOV AX, 4687H       ; Move the word 4687H to AX.
        ADD AX, 689FH       ; Add AX with the word 689FH.
        MOV BX, 1000H       ; Initialize BX with the value 1000H.
        MOV DS, BX          ; Move the content of BX to DS.
        MOV BX, 2000H       ; Move the offset address 2000H to BX.
        MOV [BX], AX        ; Store the result in AX in the memory
        MOV AX, 0F235H      ; Move the word F235H to AX.
        ADC AX, 0C545H      ; Add the word C545H to AX along with previous
                              carry.
        MOV [BX +2], AX     ; Store the result in AX in the memory at
                              offset [BX + 2].
        JC CARRY            ; If CF = 1, go to the place CARRY.
        MOV [BX + 4], 00H   ; Store 00H in the offset [BX + 4] since
                              carry is 0.
        JMP END             ; Jump to the place END.
CARRY:  MOV [BX + 4], 01H   ; Store 01H in the offset [BX + 4] since
                              carry is 1.
END:    HLT                 ; Terminate program execution.
```

Example 4.9

Write a program to subtract the multi-byte data 2035A21CH from the multi-byte data 10F3C2B6H, and store the result starting from the address 2000H:3000H in the memory with the lower-order byte of the result stored first. Assume that the data is signed data.

Flowchart:

The flowchart for this problem is given in Fig. 4.5. If the result is positive, the MSB of the result will be 0, and if the result is negative, the MSB of the result will be 1, and the result will be in 2's complement form.

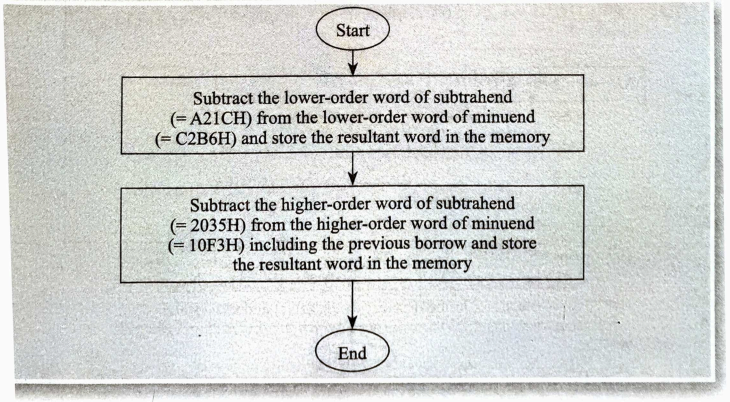


Fig. 4.5 Flowchart for subtracting one multi-byte data from another

Program:

```
MOV AX, 0C2B6H    ; Move the word C2B6H to AX.
SUB AX, 0A21CH    ; Subtract the word A21CH from AX.
MOV BX, 2000H     ; Move the segment address 2000H to BX.
MOV DS, BX        ; Move the content of BX to DS.
MOV BX, 3000H     ; Move the offset address 3000H to BX.
MOV [BX], AX      ; Store the result in AX in the memory at offset [BX].
MOV AX, 10F3H     ; Move the word 10F3H to AX.
SBB AX, 2035H     ; Subtract the word 2035H from AX with previous
                    borrow.
MOV [BX + 2], AX ; Store the result in AX in the memory at offset
                    [BX + 2].
HLT               ; Terminate program execution.
```

Example 4.10

Write a program to add one hundred byte-type data stored in an array starting from the address 1000H:2000H in the memory. The result has to be stored at the offset address 2100H in the same segment.

Flowchart:

The flowchart for this problem is given in Fig. 4.6.

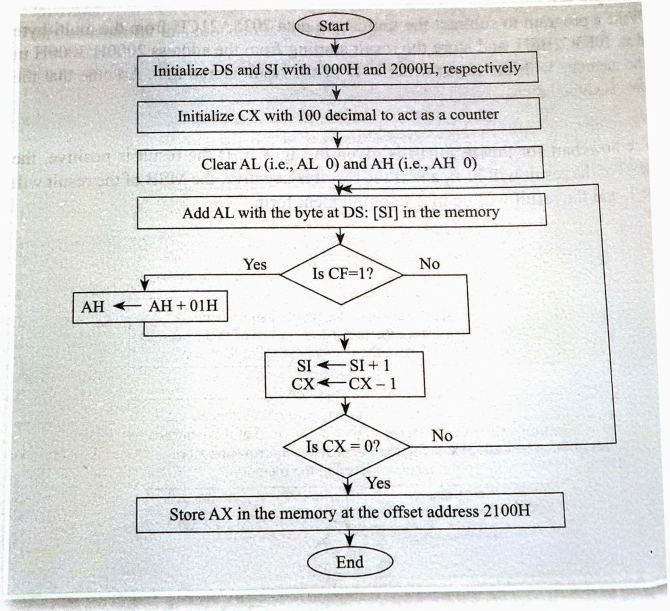


Fig. 4.6 Flowchart for adding hundred byte-type data

Program:

```
        MOV BX, 1000H     ; Initialize DS with 1000H.
        MOV DS, BX        ; Move the content of BX to DS
        MOV SI, 2000H     ; Initialize SI with 2000H.
        MOV CX, 100       ; Initialize CX with decimal 100 (= 64H).
        XOR AX, AX        ; Clear AX (i.e., AH = AL = 0) and carry
                            flag.
AGAIN:  ADD AL, [SI]      ; Add byte at [SI] in the memory with AL.
        JNC NO_CARRY      ; If CF = 0, go to the place NO_CARRY.
        INC AH            ; Increment AH when CF = 1.
NO_CARRY: INC SI          ; Increment SI by 1 to access the next
                            byte in the memory.
        LOOP AGAIN        ; Repeat the loop AGAIN CX times.
        MOV [2100H], AX   ; Store the result in AX at the offset
                            address 2100H in the memory.
        HLT               ; Terminate program execution.
```

Example 4.11

Write a program to move one hundred bytes of data from the offset address 2000H to the offset address 3000H in the segment 4000H.

Flowchart:

The flowchart for this problem is shown in Fig. 4.7.

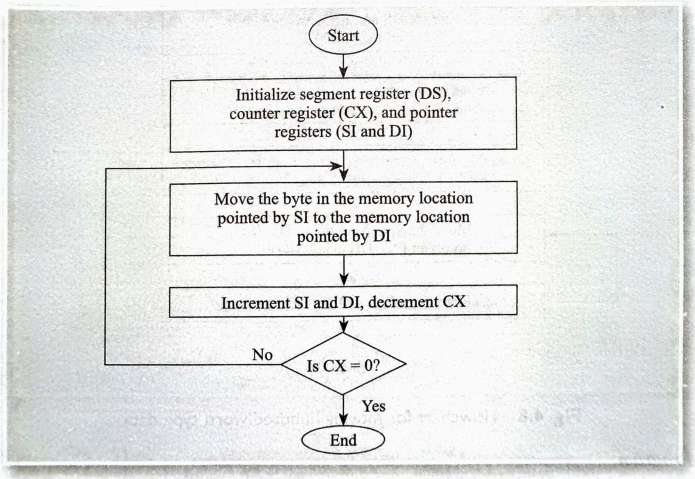


Fig. 4.7 Flowchart for moving hundred bytes of data

Program:

```
        MOV AX, 4000H  ; Initialize DS with 4000H.
        MOV DS, AX     ; Move the content of AX to DS.
```

```
        MOV SI, 2000H  ; Initialize SI with 2000H.
        MOV DI, 3000H  ; Initialize DI with 3000H.
        MOV CX, 64H    ; Initialize CX with 64H (= 100D).
AGAIN:  MOV AL, [SI]   ; Move data from offset [SI] to AL.
        MOV [DI], AL   ; Store data in AL at offset [DI].
        INC SI         ; Increment SI.
        INC DI         ; Increment DI.
        LOOP AGAIN     ; Repeat the loop AGAIN CX times.
        HLT            ; Terminate program execution.
```

Note: This program can also be written using MOVSB or MOVSW instruction and also be written in such a way that 50 word-type data is moved from one place to another place in the memory, since two bytes constitute one word. The following example indicates the use of string instruction for moving data from one place to another place in the memory.

Example 4.12

Write a program to move one hundred word-type data from the offset address 1000H to the offset address 3000H in the segment 5000H using MOVSW instruction.

Flowchart:

The flowchart for this problem is shown in Fig. 4.8.

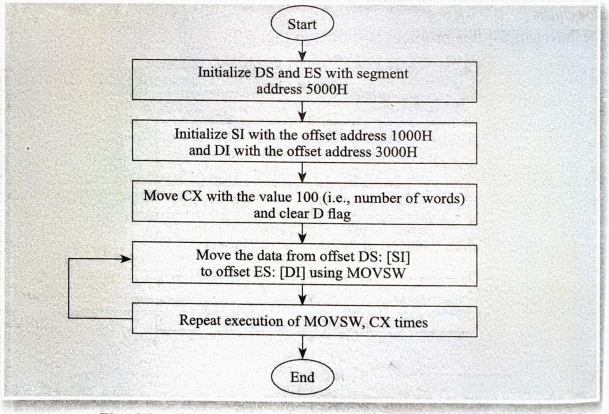


Fig. 4.8 Flowchart for moving hundred word-type data

Program:

```
MOV AX, 5000H  ; Store the segment address 5000H in AX.
MOV DS, AX     ; Initialize DS with the segment address 5000H.
MOV ES, AX     ; Initialize ES with the segment address 5000H.
MOV SI, 1000H  ; Initialize SI with the offset of the source's starting
                 address.
```

```
MOV DI, 3000H  ; Initialize DI with the offset of the destination
                 address.
MOV CX, 100    ; Initialize CX with the number of words in the string
                 (decimal value of 100 or 64H).
CLD            ; Clear the D flag for auto-increment mode.
REP MOVSW      ; Execute MOVSW instruction CX times.
HLT            ; Terminate program execution.
```

Note: In this program, the segment addresses of the source and destination are the same, and hence DS and ES registers are loaded with the same value. If they are different, ES and DS registers are loaded with the segment address of the destination and source, respectively. As D is 0, every time MOVSW is executed, the SI and DI registers are incremented by 2 to point to the next word in the string.

Example 4.13

Write a program to find the smallest word in an array of 100 words stored sequentially in the memory, starting at the offset 1000H in the segment address 5000H. Store the result at the offset 2000H in the same segment.

Flowchart:

The flowchart for the problem is given in Fig. 4.9.

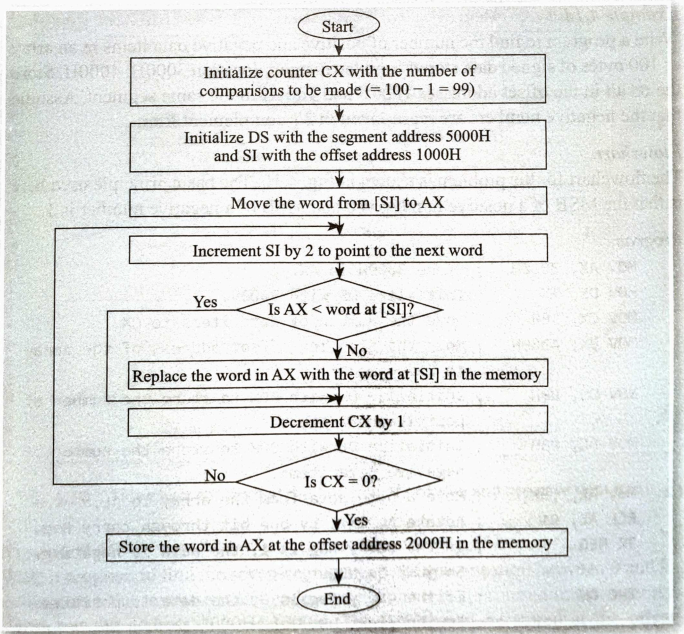


Fig. 4.9 Flowchart for finding the smallest word in an array of 100 words

Program:

```
        MOV CX, 99          ; Initialize CX with the number of comparisons
                              (= 100 - 1).
        MOV AX, 5000H       ; Store the segment address 5000H in AX.
        MOV DS, AX          ; Initialize DS with the segment address
                              5000H.
        MOV SI, 1000H       ; Initialize SI with the offset 1000H.
        MOV AX, [SI]        ; Move the first word to AX.
START:  ADD SI,02           ; Increment SI twice to point the next
                              word.
        CMP AX, [SI]        ; Compare the next word with the word in
                              AX.
        JC REPEAT           ; If AX is smaller, jump to REPEAT.
        MOV AX, [SI]        ; Replace the word in AX with the smaller
                              word.
REPEAT: LOOP START          ; Repeat the loop START, CX times.
        MOV [2000H], AX     ; Store the smallest number in AX at the
                              offset 2000H.
        HLT                 ; Terminate program execution.
```

Example 4.14

Write a program to find the number of positive and negative data items in an array of 100 bytes of signed data stored from the memory location 3000H: 4000H. Store the result in the offset addresses 1000H and 1001H in the same segment. Assume that the negative numbers are represented in 2's complement form.

Flowchart:

The flowchart for the problem is shown in Fig. 4.10. The basic principle used here is that the MSB of a positive number is 0 and MSB of a negative number is 1.

Program:

```
        MOV AX, 3000H   ; Store 3000H in AX.
        MOV DS, AX      ; Initialize DS with 3000H.
        MOV CX, 100     ; Move the number of data items to CX.
        MOV BX, 4000H   ; Move the starting offset address of the array
                          to BX.
        MOV DH, 00H     ; Initialize DH with 00H to store the number of
                          positive data items.
        MOV DL, 00H     ; Initialize DL with 00H to store the number of
                          negative data items.
L2:     MOV AL, [BX]    ; Move a byte data from the array to AL.
        RCL AL, 01      ; Rotate AL left by one bit through carry flag.
        JC NEG          ; If the carry flag is 1, the data is negative.
                          So jump to NEG.
        INC DH          ; If the carry flag is 0, the data is positive.
                          So increment DH.
        JMP L1          ; Jump to L1.
```

```
NEG: INC DL             ; Increment DL.
L1:  INC BX             ; Increment BX to point to the next data in the
                          memory.
     LOOP L2            ; Repeat loop L2 to check all data items in the
                          array.
     MOV [1000H], DH    ; Store the content of DH at the offset address
                          1000H.
     MOV [1001H], DL    ; Store the content of DL at the offset address
                          1001H.
     HLT                ; Terminate program execution.
```

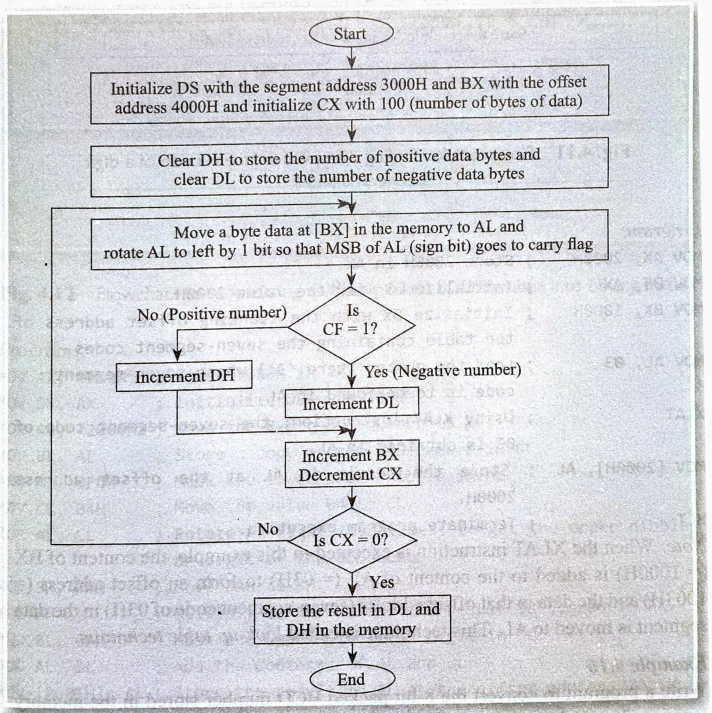


Fig. 4.10 Flowchart for finding the number of positive and negative data items

Example 4.15

Write a program to find the seven-segment code of any one digit between 0 and F. Assume that the seven-segment code of the digits 0 to F is stored in the memory starting at the address 2000H: 1000H. The result must be stored at the offset address 2000H in the same segment.

Flowchart:

The flowchart for the problem is shown in Fig. 4.11.

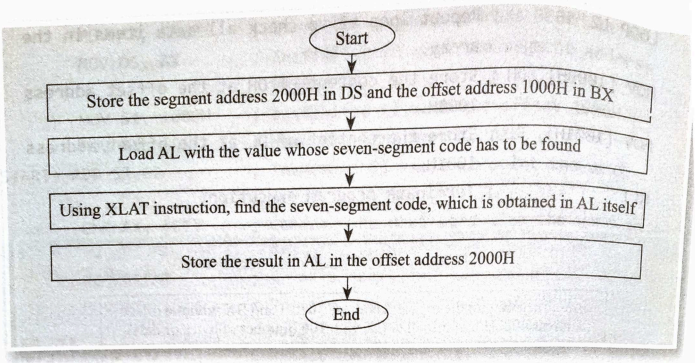


Fig. 4.11 Flowchart for finding the seven-segment code of a digit between 0 and F

Program:

```
MOV AX, 2000H      ; Store 2000H in AX.
MOV DS, AX         ; Initialize DS with the value 2000H.
MOV BX, 1000H      ; Initialize BX with the starting offset address of
                     the table containing the seven-segment codes.
MOV AL, 03         ; Load the number (here '3') whose seven-segment
                     code is to be found in AL.
XLAT               ; Using XLAT instruction, the seven-segment code of
                     03 is obtained in AL.
MOV [2000H], AL    ; Store the result in AL at the offset address
                     2000H.
HLT                ; Terminate program execution.
```

Note: When the XLAT instruction is executed in this example, the content of BX (= 1000H) is added to the content of AL (= 03H) to form an offset address (= 1003H) and the data in that offset address (seven-segment code of 03H) in the data segment is moved to AL. This technique is called *look-up table technique.*

Example 4.16

Write a program to convert the 8-bit packed BCD number stored in the memory location 3000H:2000H into a binary number and store it in the offset address 2001H in the same segment.

The binary number corresponding to an 8-bit packed BCD number is obtained by multiplying the decimal value 10 (= 0AH) with the upper digit of the BCD number and adding the result with the lower digit of the BCD number. Since the maximum 8-bit BCD number is 99 and the corresponding binary number is 63H (= 9 × 0AH + 9), the result in this program is also 8 bits.

Flowchart:
The flowchart for the problem is shown in Fig. 4.12.

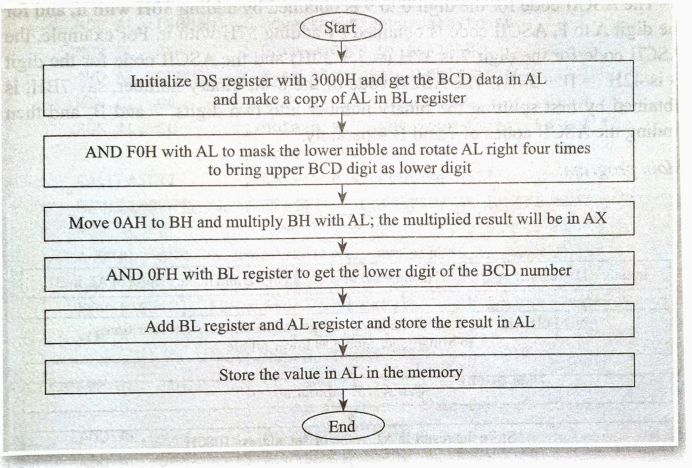


Fig. 4.12 Flowchart for converting an 8-bit packed BCD number into binary form

Program:

```
MOV AX, 3000H    ; Store 3000H in AX.
MOV DS, AX       ; Initialize DS with 3000H.
MOV AL, [2000H]  ; Move the 8-bit BCD number to AL.
MOV BL, AL       ; Store a copy of the BCD number in BL.
AND AL, 0F0H     ; Mask the lower-order nibble in AL.
MOV CL, 04H      ; Move the value 04 to CL.
ROR AL, CL       ; Rotate AL right four times, to get the upper nibble
                   or digit of the BCD number.
MOV BH, 0AH      ; Move 0AH to BH.
MUL BH           ; Multiply AL and BH and the result is stored in AX.
AND BL, 0FH      ; Mask the upper nibble or digit in BL.
ADD AL, BL       ; Add the contents of AL and BL.
MOV [2001H], AL  ; Store the result in AL at the offset address 2001H.
HLT              ; Terminate program execution.
```

Note: Since the maximum two-digit BCD number is 99H and the corresponding binary number is 63H (8 bits only), the AH value after MUL BH instruction is executed will be 00H. Hence it is not considered for the next addition.

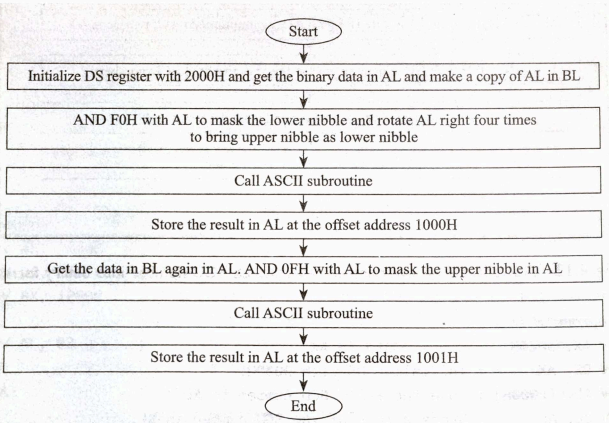
Example 4.17
Write a program to convert the given 8-bit binary number into ASCII codes. The 8-bit binary number is present in the memory location 2000H: 5000H, and the result is to be stored at the offset address 1000H and 1001H in the same segment.

Flowchart:

The flowchart for the problem is shown in Fig. 4.13.

The ASCII code for the digit 0 to 9 is obtained by adding 30H with it, and for the digit A to F, ASCII code is obtained by adding 37H with it. For example, the ASCII code for the digit 7 is 37H (= 7 + 30H) and the ASCII code for the digit B is 42H (= B + 37H). The ASCII code of the 8-bit binary number, say 7BH, is obtained by first splitting the binary number into two digits, 7 and B, and then finding the ASCII codes of 7 and B separately.

Main program:



Subroutine:

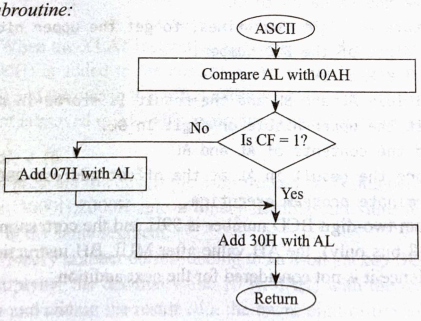


Fig. 4.13 Flowchart for converting an 8-bit binary number into ASCII code (a) Main program (b) Subroutine

Program:

```
        MOV AX, 2000H     ; Store 2000H in AX.
        MOV DS, AX        ; Initialize DS with 2000H.
        MOV AL, [5000H]   ; Move the binary data to AL.
        MOV BL, AL        ; Save a copy of AL in BL.
        AND AL, 0F0H      ; Mask the lower nibble in AL.
        MOV CL, 04
        ROR AL, CL        ; Rotate AL right four times, to get the
                          upper nibble.
        CALL ASCII        ; Call the subroutine ASCII.
        MOV [1000H], AL   ; Store the result in AL in the memory.
        MOV AL, BL        ; Move the binary data again to AL.
        AND AL, 0FH       ; Mask the upper nibble in AL.
        CALL ASCII        ; Call the subroutine ASCII.
        MOV [1001H], AL   ; Store the result in AL in the memory.
        JMP L1            ; Jump to L1.
                          ; Subroutine ASCII
ASCII:  CMP AL, 0AH       ; Compare AL with the value 0AH.
        JC L2             ; If AL is lesser than 0AH, go to L2.
        ADD AL, 07H       ; Add 07H with AL.
L2:     ADD AL, 30H       ; Add 30H with AL.
        RET               ; Return to the main program.
L1:     HLT               ; Terminate program execution.
```

Example 4.18

Write a program to add the two BCD data 87H and 98H and store the result in BCD form in the memory locations 2000H: 3000H and 2000H: 3001H.

Flowchart:

The flowchart for this problem is given in Fig. 4.14.

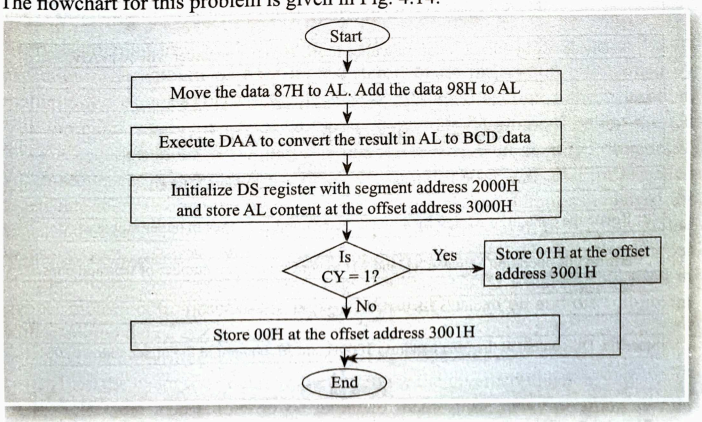


Fig. 4.14 Flowchart for adding two BCD data and storing the result in BCD form

Program:

```
    MOV AL, 87H        ; Move the first BCD data to AL.
    ADD AL, 98H        ; Add the second BCD data with AL.
    DAA                ; Decimal-adjust AL to get the result in BCD form.
    MOV BX, 2000H      ; Store 2000H in BX.
    MOV DS, BX         ; Initialize DS with 2000H.
    MOV [3000H], AL    ; Store the content of AL, which is the lower
                         byte of the result in the memory.
    JC L1              ; If the carry flag is 1, go to L1.
    MOV [3001H], 00H   ; Store 00H in the memory, which is the higher
                         byte of the result.
    JMP L2             ; Go to L2.
L1: MOV [3001H], 01H   ; Store 01H in the memory, which is the higher
                         byte of the result.
L2: HLT                ; Terminate program execution.
```

Example 4.19

Write a program to convert the 8-bit binary number FFH into a BCD number. The result is to be stored at the memory locations 3000H: 2000H and 3000H: 2001H.

Flowchart:

The flowchart for this problem is given in Fig. 4.15. The 8-bit binary number is converted into a BCD number by first dividing the binary number by decimal 100 to get the number of hundreds in it. Then the remainder obtained in the first division is divided by decimal 10 to get the number of tens in it. The reminder in the second division is the number of ones in the binary number. The number of tens and ones can be concatenated by proper shifting and performing OR operation.

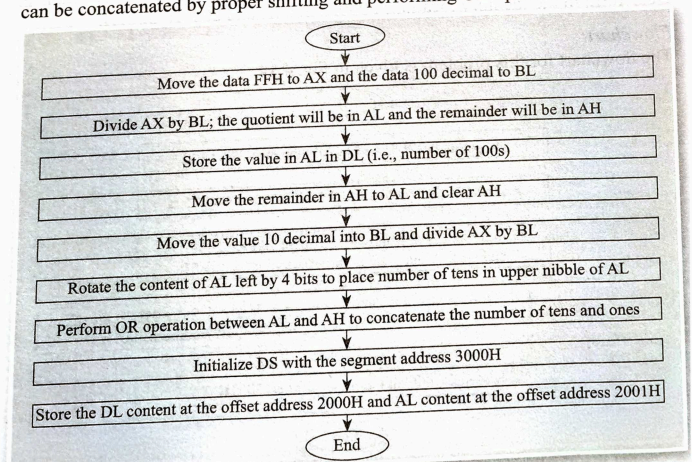


Fig. 4.15 Flowchart for converting an 8-bit binary number into BCD form

Program:

```
MOV AX, 00FFH      ; Move the data 00FFH to AX.
MOV BL, 100        ; Store the decimal value 100 (or 64H) in BL.
DIV BL             ; Divide AX by BL to find the number of hundreds in
                     the binary number.
MOV DL, AL         ; Move the quotient in AL (number of hundreds) to
                     DL.
MOV AL, AH         ; Move the remainder in AH to AL.
MOV AH, 00         ; Clear AH.
MOV BL, 10         ; Store the decimal value 10 (or 0AH) in BL.
DIV BL             ; Divide AX by BL to find the number of tens in the
                     binary number.
                   ; AH has the remainder, which is the number of ones
                     in the binary number.
MOV CL,04
ROL AL, CL         ; Rotate the content of AL left four times to make the
                     lower nibble the upper nibble.
OR AL, AH          ; Perform OR operation on AL and AH to concatenate
                     the number of tens and ones.
MOV BX, 3000H      ; Store 3000H in BX.
MOV DS, BX         ; Initialize DS with 3000H.
MOV [2000H], DL    ; Move the value of DL to the memory.
MOV [2001H], AL    ; Move the value of AL to the memory.
HLT                ; Terminate program execution.
```

Note: The binary number FFH when converted to BCD gives the result 255, as there are two hundreds, five tens, and five ones in it. In this program, 02H is stored in the offset address 2000H, and 55H is stored in the offset address 2001H in the data segment.

4.5.2 Writing Time Delay Programs

Every instruction in the 8086 requires a definite number of clock cycles for its execution. The amount of time for execution of an instruction is obtained by multiplying the number of clock cycles required for the execution of the instruction, with the clock period at which the 8086 is running. The time duration needed for the execution of an instruction can be used to derive the required time delay. When sequences of instructions are executed by the 8086, the total time needed to execute them is obtained by adding the individual time durations required for execution of each instruction. In a program loop, the number of instructions in the loop may be less but the 8086 depends on the loop count, which is the number of times the program loop has to be executed. Time delay programs are used in stepper motor control and square wave generation, to turn on and off equipment with specified delay, etc.

The steps for writing a time delay program are as follows:

(i) . Find the exact time delay (t_d) required for the given application.

(ii) Select the instructions to be included in the time delay program. While

selecting the instructions and registers to be used in the delay program, care must be taken that the execution of these instructions does not affect the main program execution. That is, any memory location or register used by the main program must not be altered by the time delay program.

If a register used in the main program is needed in the delay program, the content of that register is pushed into a stack before executing the time delay program. At the end of the execution of the time delay program, its original value will be popped from the stack and then control will be transferred to the main program.

(iii) Find the period of the clock at which the microprocessor is running by taking the reciprocal of the 8086's clock frequency. T is the duration of one clock period or clock state.

(iv) Find the number of clock states required for the execution of each of the instructions in the time delay program. Then find the number of clock states (m) needed to execute the loop in the delay program once, by adding the clock states required for each instruction in the delay program.

(v) Find the number of times (i.e., count n) the loop in the delay program has to be executed by dividing the required time delay (t_d) by the time taken to execute the loop once, which is $m \times T$.

$$\text{Count } (n) = t_d / (m \times T)$$

The time delay obtained using this method is sufficiently accurate to be used in many problems. When more accurate delays are required, the programmable timer IC 8253 or the 8254 can be used.

Example 4.20

Write a time delay program to generate a delay of 120 ms in an 8086-based system that runs on a 10 MHz frequency clock.

Solution:

The time delay program is as follows:

	Instruction	T-states for execution
	MOV BX, Count	4
L1:	DEC BX	2
	NOP	3
	JNZ L1	16
	RET	8

In this program, the instructions DEC BX, NOP, and JNZ LI form the loop as they are executed repeatedly until BX becomes zero. Once BX becomes zero, the 8086 returns to the main program.

Number of clock cycles for execution of the loop once (m) = 2 + 3 + 16 = 21

Time required for the execution of the loop once = $m \times T = 21 \times 1/(10 \times 10^6)$

$= 2.1\ \mu s$

Count = $t_d/(m \times T) = 120 \times 10^{-3}/(2.1 \times 10^{-6})$

$= 57143 = DF37H$

By loading DF37H in BX, the time taken to execute the delay program is approximately 120 ms. The NOP included in the delay program is to increase the

execution time of the loop. To get more delay, the number of NOP instructions in the delay loop can be increased. The exact delay obtained using this time delay subroutine can be calculated as shown here. The MOV BX, Count and RET instructions in the delay program are executed only once. The JNZ instruction takes 16 T-states when the condition is satisfied (i.e., Z = 0) and four T-states when the condition is not satisfied, which occurs only once.

Exact delay = $[4 \times 0.1 + (2 + 3) \times 57143 \times 0.1 + 16 \times 57142 \times 0.1 + 4 \times 0.1 + 8 \times 0.1]$ μs

$= 0.4 + 28571.5 + 91427.2 + 0.4 + 0.8$

$= 120000.3$ μs $= 120.0003$ ms

The error in the previous calculation is very less as the exact delay is also very close to 120 ms. When the 16-bit count register is used in the delay program, the maximum count value that can be loaded in it is FFFFH. This may put a limitation on the maximum time delay that can be generated using the above delay subroutine. Whenever large time delays are required, more than one count register may be used in the time delay subroutine. Example 4.21 illustrates this.

Example 4.21

Write a delay program to create a time delay of five minutes. Assume that a 10 MHz clock is used with the 8086.

	Instruction	**T-states for execution**
	MOV AX, COUNT1	4
L2:	MOV BX, COUNT2	4
L1:	NOP	3
	DEC BX	2
	JNZ L1	16
	DEC AX	2
	JNZ L2	16
	RET	8

Here there are two nested counter loops for decrementing the two counter registers. Let the value of COUNT2 be FFFFH, which is equal to 65535 decimal.

Let the execution time for inner loop be t_1.

$t_1 = [0.1 \times 4 + (2 + 3 + 16) \times 65535 \times 0.1]$ μs

$= 0.137605$ s

Let the execution time of outer loop once be t_2.

$t_2 = t_1 + (16 + 2) \times 0.1 \times 10^{-6}$

$= 0.1376068$ s

Required delay = $t_d = 5 \times 60 = 300$ s

COUNT1 = $t_d / t_2 = 300/0.1376068 = 2180 = 884$H

4.5.3 8086 Assembler Directives

An assembler is a program that is used to convert an assembly language program into an equivalent machine language program. The assembler finds the address of each label and substitutes the value of each constant and variable in the assembly language program during the assembly process, to generate the machine language code. While performing these operations, the assembler may find syntax errors.

They are reported to the programmer at the end of the assembly process. The logical and other programming errors are not found by the assembler.

For completing these tasks, an assembler needs some commands from the programmer—the required storage class for a particular constant or a variable such as byte, word, or double word, the logical name of the segments such as CODE, STACK, or DATA, the type of procedures or routines such as FAR, NEAR, PUBLIC, or EXTRN, the end of a segment, etc. These types of commands are given to the assembler using predefined alphabetical strings called *assembler directives*, which help the assembler to correctly generate the machine codes for the assembly language program.

In addition, there are a few operators that perform the addition or subtraction operation on constants or labels. The assembler directives commonly used in Microsoft Macro Assembler or Turbo Assembler are as follows:

4.5.3.1 Assembler Directives for Variable and Constant Definition

The assembler directives for variable and constant definition are as follows:
(i) DB, DW, DD, DQ, and DT: The directives DB (define byte), DW (define word), DD (define double word), DQ (define quad word), and DT (define ten bytes) are used to reserve one byte, one word (i.e., 2 bytes), one double word (i.e., 2 words), one quad word (i.e., 4 words), and ten bytes in the memory, respectively, for storing constants, variables, or strings.

Example:

```
(a)  DATA1 DB 20H                          ; Reserve one byte for storing
                                             DATA1 and assign the value 20H
                                             to it.
(b)  ARRAY1 DB 10H, 20H, 30H               ; Reserve three bytes for storing
                                             ARRAY1 and initialize it with
                                             the values 10H, 20H, and 30H.
(c)  CITY DB "MADURAI"                     ; Store the ASCII code of the
                                             characters specified within
                                             double quotes in the array or
                                             list named CITY.
(d)  DATA2 DW 1020H                        ; Reserve one word for storing
                                             DATA2 and assign the value
                                             1020H to it.
(e)  ARRAY2 DW 1030H, 2000H,
     3000H, 4000H                          ; Reserve four words for storing
                                             ARRAY2 and initialize them with
                                             the specified values.
(f)  DATA3 DD 1234ABCDH                    ; Initialize DATA3 as a double
                                             word with 1234ABCDH.
(g)  DATA4 DQ 1234ABCD5678EFBBH            ; Initialize DATA4 as a quad word
                                             with 1234ABCD5678EFBBH.
(h)  DATA5 DT 123456789ABCDEF12345H ; Initialize DATA5 as a series
                                      of 10 bytes having the value
                                      123456789ABCDEF12345H.
```

The directive DUP (duplicate) is used to reserve a series of bytes, words, double words, or ten bytes and is used with DB, DW, DD, and DT, respectively. The reserved area can be either filled with a specific value or left uninitialized.

Example:

```
(a)  Array DB 20 DUP (0)          ; Reserves 20 bytes in the memory
                                    for the array named ARRAY and
                                    initializes all the elements
                                    of the array to 0 (due to the
                                    presence of 0 within the
                                    bracket near the DUP
                                    directive).
(b)  ARRAY1 DB 25 DUP (?)         ; Reserves 25 bytes in the memory
                                    for the array named ARRAY1 and
                                    keeps all the elements of array
                                    uninitialized (due to the
                                    question mark present within
                                    the bracket near the DUP
                                    directive).
(c)  ARRAY2 DB 50 DUP (64H)       ; Reserves 50 bytes in the memory
                                    for the array named ARRAY2 and
                                    initializes all the elements
                                    of the array to 64H.
```

(ii) EQU: The directive EQU (equivalent) is used to assign a value to a data name.

Example:

```
(a)  NUMBER EQU 50H          ; Assign the value 50H to NUMBER.
(b)  NAME EQU "RAMESH"       ; Assign the string "RAMESH" to NAME.
```

4.5.3.2 *Assembler Directives Related to Code (Program) Location*

The assember directives related to code location are as follows:

(i) ORG: The ORG (origin) directive directs the assembler to start the memory allocation for a particular segment (data, code, or stack) from the declared offset address in the ORG statement. While starting the assembly process for a memory segment, the assembler initializes a location counter (LC) to keep track of the allotted offset addresses for the segment. When the ORG directive is not mentioned at the beginning of the segment, LC is initialized with the offset address 0000H. When the ORG directive is mentioned at the beginning of the segment, LC is initialized with the offset address specified in the ORG directive.

Example:

```
ORG 100H
```

When this directive is placed at the beginning of the code segment, the location counter is initialized with 0100H and the first instruction is stored from the offset address 0100H within the code segment. If it is placed in the data segment, the next data storage starts from the offset address 0100H within the data segment.

(ii) EVEN: The EVEN directive updates the location counter to the next even address, if the current location counter content is not an even number.

Example:

```
EVEN
ARRAY2 DW 20 DUP (0)
```

These statements in a segment declare an array named ARRAY2 having 20 words, starting at an even address. The advantage of storing an array of words starting at an even address is that the 8086 takes just one memory read/write cycle to read/write the entire word, if the word is stored starting at an even address. Otherwise, the 8086 takes two memory read/write cycles to read/write the word.

Example:

The EVEN directive can also be used at the beginning of a procedure, so that the instructions in it can be fetched quickly by the 8086 during execution.

```
EVEN
RESULT PROC NEAR
   …                    ; Instructions in the RESULT procedure
RESULT ENDP
```

Here the procedure RESULT, which is of type NEAR, is stored starting at an even address in the code segment. The ENDP directive indicates the end of the RESULT procedure.

(iii) LENGTH: This directive is used to determine the length of an array or string in bytes.

Example:

```
MOV CX, LENGTH ARRAY
```

CX is loaded with the number of bytes in the ARRAY.

(iv) OFFSET: This operator is used to determine the offset of a data item in a segment containing it.

Example:

```
MOV BX, OFFSET TABLE
```

If the data item named TABLE is present in the data segment, this statement places the offset address of TABLE, in the BX register.

(v) LABEL: The LABEL directive is used to assign a name to the current value in the location counter. It is used to specify the destination of the branch-related instructions such as jump and call. When LABEL is used to specify the destination, it is necessary to specify whether it is NEAR or FAR. When the destination is in the same segment, the label is specified as NEAR and when the destination is in another segment, it is specified as FAR.

Example:

```
REPEAT LABEL NEAR
CALCULATE LABEL FAR
```

LABEL can also be used to specify a data item. When it is used to specify a data item, the type of the data item must be specified. The data may have the type —byte or word.

Example:
A stack segment having 100 words of data is defined using the following statements:

```
STACK SEGMENT
DW 100 DUP (0)          ; Reserve 100 words for stack
STACK_TOP LABEL WORD
STACK ENDS
```

The second statement reserves 100 words in the stack segment and fills them with 0. The third statement assigns the name STACK_TOP to the location present just after the hundredth word. The offset address of this label can then be assigned to the stack pointer in the code segment using the following statement:

```
MOV SP, OFFSET STACK_TOP
```

4.5.3.3 *Assembler Directives for Segment Declaration*

The assembler directives for segment declaration are as follows:
(i) SEGMENT and ENDS: The SEGMENT and ENDS directives indicate the start and end of a segment, respectively. In some cases, the segment may be assigned a type such as PUBLIC (i.e., it can be used by other modules of the program while linking) or GLOBAL (i.e., it can be accessed by any other module).

Large assembly language programs are usually developed as separate assembly modules. Each assembly module is individually assembled, tested, and debugged. When all the assembly modules are working correctly, their object code files are linked together to form the complete program. For the modules to link together correctly, any segment, label, or variable name referred to in other modules must be declared PUBLIC in the module in which it is defined. For example, the statement DATA1 SEGMENT WORD PUBLIC makes the segment named DATA1 available to other assembly modules. Here, the term WORD is used to inform the linker to locate the segment in the first available even address. Similarly, the statement PUBLIC X1, X2 makes the two variables X1 and X2 available to other assembly modules. If an instruction in an assembly module refers to a variable or label which is present in another assembly module, the assembler must be told that it is external, using the EXTRN directive.

The GLOBAL directive can be used in place of the PUBLIC or EXTRN directive. For a symbol or name defined in the current assembly module, the GLOBAL directive is used to make that symbol or name available to other assembly modules. For example, the statement GLOBAL MULTIPLIER makes the variable MULTIPLIER public so that it can be accessed from other assembly modules. The statement GLOBAL MULTIPLIER: WORD informs the assembler that MULTIPLIER is a variable of type 'word', which is in another assembly module.

Example:

```
CODE1 SEGMENT
...                ; Instructions of CODE 1 segment
CODE1 ENDS
```

This example indicates the declaration of a code segment named CODE1.

(ii) ASSUME: The ASSUME directive is used to inform the assembler, the name of the logical segments to be assumed for different segments used in the program.

Example:

```
ASSUME CS: CODE1, DS: DATA1
```

This statement informs the assembler that the segment address where the logical segments CODE1 and DATA1 are loaded in memory during execution is to be stored in the CS and DS registers, respectively.

(iii) GROUP: This directive is used to form a logical group of segments with a similar purpose. The assembler passes information to the linker/loader to form the code, such that the group declared segments or operands lie within a 64 KB memory segment. All such segments can be addressed using the same segment address.

Example:

```
PROGRAM1 GROUP CODE1, DATA1, STACK1
```

This statement directs the loader/linker to prepare an executable (EXE) file such that the CODE1, DATA1, and STACK1 segments lie within a 64 KB memory segment that is named PROGRAM1. Now, for the ASSUME statement, we can use the label PROGRAM1 rather than CODE1, DATA1, and STACK1, as follows:
ASSUME CS: PROGRAM1, DS: PROGRAM1, SS: PROGRAM1

(iv) SEG: The segment operator is used to decide the segment address of the label, variable, or procedure and substitute the segment address in place of the SEG label.

Example:

```
MOV AX, SEG ARRAY1    ; Load the segment address in which ARRAY1 is
                        present, in AX.
MOV DS, AX            ; Move the content of AX to DS.
```

4.5.3.4 *Assembler Directives for Declaring Procedures*

The assember directives for declaring procedures are as follows:

(i) PROC: The PROC directive indicates the start of a named procedure. The NEAR and FAR directives specify the type of the procedure.

Example:

```
SQUARE_ROOT PROC NEAR
```

This statement indicates the beginning of a procedure named SQUARE_ROOT, which is to be called by a program located in the same segment. The FAR directive is used for the procedures to be called by the programs present in code segments other than the one in which this procedure is present. For example, SALARY PROC FAR indicates the beginning of a FAR type procedure named SALARY.

(ii) ENDP: The ENDP directive is used to indicate the end of a procedure. To mark the end of a particular procedure, the name of the procedure may appear as a prefix with the directive ENDP.

Example:

```
SALARY PROC NEAR
...                    ; Code of SALARY procedure
SALARY ENDP
```

(iii) EXTRN and PUBLIC: The directive EXTRN (external) informs the assembler that the procedures, label/labels, and names declared after this directive has/have already been defined in some other segments and in the segments where they actually appear, they must be declared public, using the PUBLIC directive.

Example:

```
MODULE1 SEGMENT
PUBLIC SQUARE_ROOT
SQUARE_ROOT PROC FAR
...                    ; Code of SQUARE_ROOT procedure
SQUARE_ROOT ENDP
MODULE1 ENDS

MODULE2 SEGMENT
EXTRN SQUARE_ROOT FAR
...                    ; Code of MODULE2
CALL SQUARE_ROOT
...
MODULE2 ENDS
```

If one wants to call the procedure named SQUARE_ROOT appearing in MODULE1 from MODULE2, it must be declared public using the statement PUBLIC SQUARE_ROOT in MODULE1 and it must be declared external using the statement EXTRN SQUARE_ROOT in MODULE2. If a jump or call address is external, it must be represented as NEAR or FAR. If data are defined as external, their size must be represented as BYTE, WORD, or DWORD.

4.5.3.5 *Other Assembler Directives*

(i) PTR: The PTR (pointer) operator is used to declare the type of a label, variable, or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, the particular label, variable, or memory operand is treated as an 8-bit quantity, while if the prefix is WORD, it is treated as a 16-bit quantity.

Example:

```
(a)   INC BYTE PTR [SI]      ; Increment the byte contents of the
                               memory location addressed by SI.
(b)   INC WORD PTR [BX]      ; Increment the word contents of the
                               memory location addressed by BX.
```

The PTR directive is also used to declare a label either as FAR or NEAR type. The FAR PTR directive indicates to the assembler that the label following FAR

PTR is not available within the same segment and the address of the label is of size 32 bits (2 bytes offset, followed by 2 bytes segment address).

Example:

(a) `JMP FAR PTR DIVIDE`

(b) `CALL FAR PTR CONVERT`

where DIVIDE and CONVERT are the names of a label and procedure, respectively.

The NEAR PTR directive indicates that the label following NEAR PTR is in the same segment and needs only 16 bits (2 bytes offset) to address it.

(ii) GLOBAL: The labels, variables, constants, or procedures declared GLOBAL may be used by the other modules of the program.

Example:

The following statement declares the procedure ROOT as a GLOBAL label.

```
ROOT PROC GLOBAL
```

Example:

The following statement declares the variables DATA1, DATA2, and ARRAY1 as GLOBAL variables.

```
GLOBAL DATA1, DATA2, ARRAY1
```

(iii) LOCAL: The label, variables, constants, or procedures declared LOCAL in a module are to be used only by that particular module. After some time, some other module may declare a particular data type LOCAL, which was previously declared as LOCAL by another module or modules. Thus, the same label may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared LOCAL as follows:

```
LOCAL DATA1, DATA2, ARRAY1, A1, A2
```

(iv) NAME: The NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by its declared name. The names, if selected properly, may indicate the function of the different modules, and hence help in good documentation.

(v) SHORT: The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e., the displacement is within −128 to +127 bytes from the address of the byte present next to the JMP instruction). This method of specifying the jump address saves memory. Otherwise, the assembler may reserve 2 bytes for the displacement in the jump instructions.

Example:

```
JMP SHORT MULTIPLY
```

where MULTIPLY is a label.

(vi) TYPE: The TYPE operator directs the assembler to decide the data type of the specified label and replaces the TYPE label with the decided data type. For the word type variable, the data type is 2. For the double word type, it is 4, and for the byte type, it is 1.

Example:
If DATA1 is an array having word type data, the instruction `MOV BX, TYPE DATA1` moves the value 0002H to BX.

(vii) MACRO and ENDM: Suppose a number of instructions occur repeatedly in the main program, the program listing becomes lengthy. In such a situation, a macro definition, i.e., a label, is assigned with the repeatedly appearing string of instructions. The process of assigning a label or macro name to the repeatedly appearing string of instructions is called macro definition. The macro name is then used throughout the main program to refer to that string of instructions.

The difference between a macro and a subroutine is that in the macro, the complete code of the instructions in the macro is inserted at each place where the macro name appears during the assembly process. Hence, the length of the EXE file is larger and the macro does not utilize the service of the stack. However, a subroutine is present only in one place in a program and the control of execution is transferred to the subroutine by calling that subroutine whenever necessary. Hence, the length of the EXE file is smaller while using subroutines. A subroutine uses the stack for storing the return address when it is called. The drawback with subroutines is the overhead time needed to push the return address into the stack, while calling the subroutine, and to retrieve the same from the stack, while returning from the subroutine to the main program.

Defining a MACRO A MACRO can be defined anywhere in a program, using the directives MACRO and ENDM. The label prior to the MACRO is the macro name, which is used in the main program wherever needed. The ENDM directive marks the end of the instructions or statements assigned to the macro name.

Example:

```
CALCULATE MACRO
MOV AX, [BX]
ADD AX, [BX + 2]
MOV [SI], AX
ENDM
```

In this example, CALCULATE is the macro name and the macro is used to add two successive data in the memory, whose offset address is present in BX and the result is stored in the memory at the offset address present in SI. In the program, which uses the above macro definition, wherever the instructions defined in the above macro are repeating, we can simply use the macro name (CALCULATE) instead of those instructions and this is called *macro reference*. When that program is assembled using the assembler, the assembler replaces each macro reference by the corresponding string of instructions defined in the macro and finds the opcode of each instruction. This is called *macro expansion*.

Passing parameters to a MACRO Using parameters in a macro definition, the programmer specifies the parameters of the macro that are likely to be changed each time the macro is called. The macro given here (CALCULATE) can be modified to calculate the result for different sets of data and store it in different

memory locations as follows:

```
CALCULATE MACRO OPERAND, RESULT
MOV BX, OFFSET OPERAND
MOV AX, [BX]
ADD AX, [BX + 2]
MOV SI, OFFSET RESULT
MOV [SI], AX
ENDM
```

The parameters OPERAND and RESULT can be replaced by OPERAND1, RESULT1 and OPERAND2, RESULT2 while calling the macro, as follows:

```
...
CALCULATE          OPERAND1, RESULT1
...
CALCULATE          OPERAND2, RESULT2
...
```

4.5.4 Writing Assembly Language Programs using MASM

MASM (Microsoft Macro Assembler) is one of the assemblers commonly used along with the LINK (linker) program to structure the machine codes generated by MASM in the form of an executable (EXE) file. The MASM reads the assembly language program, which is called source program and produces an object file as output. The LINK program accepts the object file produced by MASM along with library files if needed, and produces an EXE file.

While writing a program for MASM, the program listing is first typed using a text editor in the computer, such as Norton's Editor (NE) and Turbo C editor. After the program editing is done, it is saved with the extension .ASM. For example, MSI.ASM is a valid file name that can be assigned to an assembly language program. The programmers have to ensure that all the files—the editor, MASM.EXE (MASM assembler), and LINK.EXE (linker)—are available in the same directory. After editing, the assembling of the program has to be done using MASM. If all the above mentioned software is present in the root directory of the C drive in the computer, to assemble the file MSI.ASM, the programmer has to type the following at the DOS prompt in the computer:

C:\ > MASM MSI.ASM or C:\ > MASM MSI

After entering this command, the assembler asks for the names of the following types of files, which it generates after the assembly process:

Object file name [.OBJ]:

List file name [NUL.LST]:

Cross reference [NUL.CRF]:

The programmer can type a name against every file name and press the enter key after each name. If no name is entered against the file name before pressing the enter key, all the three files will have the same name as the source file. The OBJ (object) file contains the machine codes of the program that is assembled. The .LST (list) file contains the total offset map of the source file, including labels, opcodes, offset addresses, memory allotment for different labels, and directives.

The cross reference (.CRF) file is used for debugging the source program, as it contains information such as size of the file in bytes, list of labels, number of labels, and routines to be called in the source program.

After the cross reference file name is entered, the assembly process starts. If the program contains syntax errors, they are displayed using the error code number and the corresponding line numbers at which the errors have occurred. Once these errors are corrected by the programmer, the assembly process is completed successfully.

The DOS linking program LINK.EXE is used to link the different object modules of a source program and the function library routines to produce an integrated executable code for the source program. The linker is invoked using the following command:

C :\> LINK MSI.OBJ

After entering this command, the linker asks for the name of the following files:

Run file [.EXE]:
List files [NUL.MAP]:
Libraries [LIB]:

If no file names are entered for these files, by default, the source file name is considered. The optional input 'Libraries' expects the name of a special library (if any) from which the functions were used by the source program. The output of the linker program is an executable file with either the file name entered by the user or the default file name, and .EXE extension. The executable file name can be entered at the DOS prompt to execute the file as follows:

C :\> MSI.EXE

In the advanced version of MASM, both assembling and linking are combined under a single menu-invokable compile function.

DEBUG.com is a DOS utility program that is used for debugging and troubleshooting 8086 assembly language programs. The DEBUG utility enables us to have control over the hardware resources and the memory in the computer (PC) up to a certain extent, as the PC uses one of the INTEL processors (80486, Pentium, etc.) as the CPU. DEBUG enables us to use the PC as a low-level 8086 microprocessor kit. Typing the DEBUG command at the DOS prompt and pressing the enter key invokes the debugging facility. A '-' (dash) appears DEBUG is successfully invoked, as follows:

C :\> DEBUG
-

Now, by typing 'R' at the '-' line and pressing the enter key, we can see the content of the different registers and flags present in the CPU of the PC, as follows:

```
-R
AX = 0000H        BX = 0005H      CX = 000DH      DX = S000H
SP = 8500H        BP = 9800H      SI = 2000H      DI = 7000H
DS = S000H        ES = 3000H      SS = 4000H      CS = 2000H
IP = 2000H        FLAGS = 0024H
```

Table 4.11 shows the list of generally used DEBUG commands, along with their syntax, in alphabetical order.

Table 4.11 Generally used DEBUG commands

S. No.	Command character	Format(s)	Function
1.	-a	<ENTER>	Assemble from the current CS:IP.
2.	-a	SEG:OFFSET <ENTER>	Assemble the entered instruction from SEG:OFFSET address.
3.	-c	SEG:OFFSET1 OFFSET2 N <ENTER>	Copy *N* bytes from OFFSET1 to OFFSET2 in segment SEG.
4.	-d	<ENTER>	Display 128 memory locations of RAM, starting from the current CS:IP address.
5.	-d	SEG: OFFSET1 OFFSET2 <ENTER>	Display memory contents in segment SEG from OFFSET1 to OFFSET2.
6.	-e	<ENTER>	Enter Hex data at current display pointer SEG:OFFSET.
7.	-e	SEG: OFFSET1 <ENTER>	Enter Hex data at SEG:OFFSET1 byte by byte by pressing the space key for giving each data one by one. The data entry is to be completed by pressing the enter key.
8.	-f	SEG: OFFSET1 OFFSET2 BYTE <ENTER>	Fill the memory area starting from SEG: OFFSET1 to OFFSET2 with the byte given.
9.	-f	SEG: OFFSET1 OFFSET2 BYTE1, BYTE2, BYTE3, BYTE4, etc. <ENTER>	Fill the memory area starting from SEG: OFFSET1 to OFFSET2 with the byte sequence BYTE1, BYTE2, BYTE3, BYTE4, etc.
10.	-g	<ENTER>	Execute from current CS:IP. By modifying CS and IP using R command, this can be used for execution from any address.
11.	-g	= OFFSET <ENTER>	Execute from the OFFSET in the current CS.
12.	-l	<ENTER>	Load the file FNAME.EXE as set by the –n command in the RAM and set the CS:IP at the address at which the file is loaded.
13.	-m	SEG: OFFSET1 OFFSET2 N <ENTER>	Move *N* bytes from OFFSET1 to OFFSET2 in segment SEG.
14.	-n	FNAME.EXE <ENTER>	Set filename pointer to FNAME. Here FNAME represents an executable file name.

(Contd)

Table 4.11 Generally used DEBUG commands *(Contd)*

S. No.	Command character	Format(s)	Function
15.	-q	<ENTER>	Quit the DEBUG and return to DOS.
16.	-r	<ENTER>	Display all registers and flags.
17.	-r	reg <ENTER> Old content:New content	Display specified register 'reg' content and modify it with the entered new content.
18.	-s	SEG: OFFSET1 OFFSET2 BYTE/BYTES <ENTER>	Searches a byte or string of bytes separated by ',' in the memory region from SEG: OFFSET1 to OFFSET2 and displays all the offsets at which the byte or string of bytes is found.
19.	-t	SEG: OFFSET <ENTER>	Trace the program execution by single stepping starting from the address SEG: OFFSET.
20.	-u	<ENTER>	Unassemble from the current CS:IP.
21.	-u	SEG: OFFSET <ENTER>	Unassemble from the address SEG: OFFSET.
22.	-?	<ENTER>	List all the commands in DEBUG.

The remaining DEBUG commands can be referred to from any book that discusses assembly language programming in personal computers. In this section, a few examples for writing 8086 assembly language programs while using an assembler are given.

Example 4.22

Write a program to add two 8-bit data (F0H and 50H) in the 8086 and store the result in the memory, when MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT          ; Beginning of data segment
        OPER1 DB 0F0H         ; Store the first operand.
        OPER2 DB 50H          ; Store the second operand.
        RESULT DB 01 DUP (0); Reserve a byte of memory for the
                              result.
        CARRY DB 01 DUP (0) ; Reserve a byte for storing the
                              carry.
        DATA ENDS             ; End of data segment
        CODE SEGMENT          ; Beginning of code segment
START:  MOV AX, DATA          ; Load AX with the segment address of
                              DATA.
        MOV DS, AX            ; Move the content of AX to DS.
```

```
        MOV BX, OFFSET OPER1    ; Move the offset address of OPER1 to
                                  BX.
        MOV AL, [BX]            ; Move the first operand to AL.
        ADD AL, [BX+1]          ; Add the second operand to AL.
        MOV SI, OFFSET RESULT   ; Store the offset address of RESULT
                                  in SI.
        MOV [SI], AL            ; Store the content of AL in the
                                  location RESULT.
        INC SI                  ; Increment SI to point to the location
                                  of carry.
        JC CAR                  ; If carry = 1, jump to CAR.
        MOV BYTE PTR [SI], 00H  ; Store 00H in the location CARRY.
        JMP LOC1                ; Jump to LOC1.
CAR:    MOV BYTE PTR [SI], 01H  ; Store 01H in the location CARRY.
LOC1:   MOV AH, 4CH
        INT 21H                 ; Return to DOS prompt.
        CODE ENDS               ; End of code segment
        END START               ; Program ends.
```

Example 4.23

Write a program to subtract the word 2350H from the word 1ACFH and store the result in the memory, when MASM assembler is used. Assume that the data is signed data.

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT            ; Beginning of data segment
        OPR1 DW 2350H           ; Store the subtrahend.
        OPR2 DW 1ACFH           ; Store the minuend.
        RESULT DW 01 DUP (?)    ; Reserve a word of the memory for the
                                  result.
        DATA ENDS               ; End of data segment
        CODE SEGMENT            ; Beginning of data segment
START:  MOV AX, DATA            ; Load AX with the segment address of
                                  DATA.
        MOV DS, AX              ; Move the content of AX to DS.
        MOV AX, OPR2            ; Move minuend to AX.
        SUB AX, OPR1            ; Subtract subtrahend from minuend.
        MOV RESULT, AX          ; Store content of AX in location
                                  RESULT.
        MOV AH, 4CH             ; Return to DOS prompt.
        INT 21H
        CODE ENDS               ; End of code segment
        END START               ; Program ends.
```

Note: In this program, if the result is positive, its MSB will be 0. If the result is negative, its MSB will be 1 and the result will be in 2's complement form.

Example 4.24

Write a program to add two hundred words of data when MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
           DATA SEGMENT              ; Data segment starts.
           NUMBERS DW 1234H, 5CD5H, 4BE0H, …
                                     ; Array of 200 words
           COUNT EQU 200             ; COUNT is assigned the value 200
                                       decimal.
           RESULT_HW DW 01 DUP (?)   ; Reserve a word for storing
                                       higher word of the result.
           RESULT_LW DW 01 DUP (?)   ; Reserve a word for storing lower
                                       word of the result.
           DATA ENDS                 ; Data segment ends.
           CODE SEGMENT              ; Code segment starts.
START:     XOR AX, AX                ; Clear AX and carry flag.
           MOV BX, 0000H             ; Clear BX.
           MOV CX, COUNT             ; Load count (= 200) in CX.
           MOV SI, OFFSET NUMBERS    ; Move offset address of NUMBERS
                                       in SI.
L2:        ADD AX, [SI]              ; Add one word data at [SI] with
                                       AX.
           JNC NO_CARRY              ; If CF = 0, go to NO_CARRY
           INC BX                    ; If CF = 1, increment BX.
NO_CARRY:  INC SI                    ; Increment SI by 2 to point to
                                       next word.
           INC SI
           LOOP L2                   ; Repeat loop L2 CX times.
           MOV RESULT_HW, BX         ; Move data in BX to RESULT_HW.
           MOV RESULT_LW, AX         ; Move data in AX to RESULT_LW.
           MOV AH, 4CH               ; Return to DOS prompt.
           INT 21H
           CODE ENDS                 ; End of code segment
           END START                 ; Program ends.
```

Example 4.25

Write a program to find whether the byte FFH is present in the given string. In addition, find the relative address of that byte from the starting location of the string. If the byte FFH is not present, store 00H in DL, and if it is present, store 01H in DL. Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
           DATA SEGMENT          ; Data segment starts
           COUNT EQU 08H         ; No. of characters in the string
```

```
        STRING DB 30H, 35H, 32H, 34H, 0FFH, 43H, 46H, 21H
                                ; String for searching
        LOCAT DB 01 DUP (?)     ; Storage for location of the search
                                  byte FFH in the string
        SEARCH_BYTE DB 0FFH     ; Search byte is FFH.
        DATA ENDS               ; Data segment ends.
        CODE SEGMENT            ; Code segment starts.
START:  MOV AX, DATA            ; Move segment address of DATA to AX.
        MOV DS, AX              ; Move content of AX to DS.
        MOV ES, AX              ; Move content of AX to ES.
        MOV CX, COUNT           ; Move COUNT to CX.
        MOV DI, OFFSET STRING   ; Move offset of STRING to DI.
        MOV DH, 00H             ; Move 00H to DH to find the location
                                  of FFH.
        MOV AL, SEARCH_BYTE     ; Move FFH to AL.
        MOV DL, 00H             ; Move 00H to DL.
        CLD                     ; Clear D flag.
CHECK:  SCASB                   ; Compare AL content with the content
                                  of ES: [DI].
        JZ FOUND                ; If Z = 1, which indicates a match, go
                                  to FOUND.
        INC DH                  ; Increment DH to point to the next
                                  relative address.
        LOOP CHECK              ; Go to CHECK CX times.
        JMP L1                  ; Jump to location to L1.
FOUND:  MOV DL, 01H             ; Move 01H to DL as FFH is found.
        MOV LOCAT, DH           ; Move value in DH to LOCAT.
L1:     MOV AH, 4CH             ; Go to DOS prompt.
        INT 21H
        CODE ENDS               : End of the code segment
        END START               ; Program ends.
```

Note: In this program, the result stored in LOCAT will be equal to 04H after execution, since the relative address is 00H for the first byte in the string.

Example 4.26

Write a program to find the parity of a given word-type data. If the parity is even, store 00H in BL register, and if the parity is odd, store 01H in BL register. Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT    ; Data segment starts.
        NUMBER DW 5260H; Store the number for which parity is to be
                         found.
        COUNT EQU 16    ; Assign COUNT to the number of the bits in a
                          word.
```

```
          DATA ENDS          ; Data segment ends.
          CODE SEGMENT       ; Code segment starts.
START:    MOV AX, DATA       ; Move segment address of DATA to AX.
          MOV DS, AX         ; Move the content of AX to DS.
          MOV CX, COUNT      ; Load COUNT in CX.
          MOV AX, NUMBER     ; Move NUMBER to AX.
          CLR DL             ; Clear DL, which is used to find number of
                               1s in the NUMBER.
L1:       RCR AX, 1          ; Rotate AX right through carry by 1 bit.
          JNC NO_CAR         ; If CF = 0, go to NO_CAR.
          INC DL             ; Increment DL as a 1 is encountered in the
                               number.
NO_CAR:   LOOP L1            ; Execute loop L1, CX times.
          MOV AL, DL         ; Move DL to AL to perform division.
          MOV AH, 00H        ; Clear AH to perform division.
          MOV BL, 02H        ; Move 02H to BL, which is divider.
          DIV BL             ; Divide AX by BL; if the reminder is 0,
                               number is even parity.
          CMP AH, 00H        ; Compare AH, which is having remainder, with
                               00H.
          JZ EVEN            ; If Z = 1, go to location EVEN.
          MOV BL, 01H        ; Move 01H to BL to indicate odd parity.
          JMP L2             ; Go to location L2.
EVEN:     MOV BL, 00H        ; Move 00H to BL to indicate even parity.
L2:       MOV AH, 4CH        ; Return to DOS prompt.
          INT 21H
          CODE ENDS          ; End of the code segment.
          END START          ; Program ends.
```

Note: Parity indicates number of 1s in a data. If the number of 1s is even, the number is said to be even parity number, and if the number of 1s is odd, the number is said to be odd parity number. In this program, the number of 1s in the number is found by rotating that number through the carry flag one bit at a time and checking whether the carry flag and hence a particular bit in the number is 0 or 1. If the bit is 1, a register (here DL) is incremented, which is initially loaded with 00H, and if the bit is 0, the same register is not incremented. After testing all bits in the number, if the value in DL is even, the number is said to be even-parity number, which is checked by dividing the DL content by 2 and seeing the remainder. If the remainder is 0, the value in DL is even and the number has even parity. Otherwise, the number has odd parity.

Example 4.27

Write a program to find the sum of two 2 × 2 matrices whose elements are byte-type data and store the result in the memory. Assume that MASM assembler is used.

Flowchart:

Let the two matrices be **A** and **B** and their elements as follows:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Let us assume that the data and the result are stored in the memory as follows:

Data 1	Data 2	Result
a_{11}	b_{11}	c_{11}
a_{12}	b_{12}	c_{12}
a_{21}	b_{21}	c_{21}
a_{22}	b_{22}	c_{22}

Let the resultant matrix obtained by adding **A** and **B** be **C**, whose elements are as follows:

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Program:

```
ASSUME CS: CODE, DS: DATA
           DATA SEGMENT                    ; Data segment starts.
           DATA1 DB 50H, 40H, 30H, 20H     ; Store the elements of matrix A.
           DATA2 DB 40H, 20H, 80H, 70H     ; Store the elements of matrix B.
           RESULT DB 04 DUP (0)            ; Reserve four bytes for storing
                                             resultant matrix C.
           COUNT EQU 04                    ; COUNT is equal to 4 as there are
                                             four elements in each matrix.
           DATA ENDS                       ; Data segment ends.
           CODE SEGMENT                    ; Code segment starts.
START:     MOV AX, DATA                    ; Segment address of DATA is
                                             moved to DS.
           MOV DS, AX                      ; Move the content of AX to DS.
           MOV CX, COUNT                   ; Load COUNT in CX.
           MOV BX, OFFSET DATA1            ; Move the offset address of
                                             DATA1 to BX.
           MOV SI, OFFSET DATA2            ; Move the offset address of
                                             DATA2 to SI.
           MOV DI, OFFSET RESULT           ; Move the offset address of
                                             RESULT to DI.
AGAIN:     MOV AL, [BX]                    ; Move one element of matrix A
                                             to AL.
           ADD AL, [SI]                    ; Add AL with corresponding
                                             element of matrix B.
```

```
        MOV [DI], AL            ; Store the result in the
                                  memory.
        INC BX                  ; Increment BX to point to the
                                  next element of A.
        INC SI                  ; Increment SI to point to the
                                  next element of B.
        INC DI                  ; Increment DI to point to the
                                  next location to store the
                                  result.
        LOOP AGAIN              ; Repeat the loop CX times.
        MOV AH, 4CH             ; Return to DOS prompt.
        INT 21H
        CODE ENDS               ; Code segment ends.
        END START               ; Program ends.
```

Note: In this program, the elements for matrices **A** and **B** are chosen such that the sum of the corresponding elements of **A** and **B** is also 8-bit data (i.e., byte). If it is more than 8 bits, two bytes must be reserved for storing each element of resultant matrix **C**, thus totally requiring 8 bytes for storing all elements of matrix **C**. The carry generated in the addition is stored as the MSB for each of the elements in matrix **C**. It is left to the reader to write the program for the same.

Example 4.28

Write a program to find the smallest word in the given array having three word-type data. Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS:DATA
        DATA SEGMENT            ; Data segment starts.
        ARRAY DW 2500H, 1600H, 0032H
                                ; Three words of data in ARRAY
        COUNT EQU 03H           ; Assign 03 to COUNT, as three words
                                  are compared.
        SMALLEST DW 01 DUP (0) ; Reserve one word to store the
                                  smallest word.
        DATA ENDS               ; Data segment ends.
        CODE SEGMENT            ; Code segment starts.
START:  MOV AX, DATA            ; Move the segment address of DATA to
                                  DS.
        MOV DS, AX              ; Move the content of AX to DS.
        MOV SI, OFFSET ARRAY    ; Move the offset of the array to SI.
        MOV CX, COUNT           ; Load COUNT in CX.
        DEC CX                  ; Decrement CX as the number of
                                  comparisons is one less than COUNT.
        MOV AX, [SI]            ; Move the first word to AX.
AGAIN:  ADD SI, 02              ; Add 2 to SI to point the next word.
        CMP AX, [SI]            ; Compare the word in AX with the
                                  word at [SI].
```

```
        JC NEXT                 ; If AX is small, go to NEXT.
        MOV AX, [SI]            ; Move the small word in [SI] to AX.
NEXT:   LOOP AGAIN              ; Repeat the loop AGAIN, CX times.
        MOV SMALLEST, AX        ; Store AX in the location SMALLEST.
        MOV AH, 4CH
        INT 21H                 ; Return to the DOS prompt.
        CODE ENDS               ; End of code segment
        END START               ; Program ends.
```

Example 4.29

Write a program to find the number of even and odd data bytes present in the given array having five byte-type data. Assume that MASM assembler is used.

In this program, the array has five bytes of data (40H, 31H, 23H, 52H, 39H).

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT            ; Data segment starts.
        ARRAY DB 40H,31H,23H,52H,39H
                                ; Enter all data in the array here.
        COUNT EQU 05H           ; Initialize COUNT with 05, which is the
                                  number of data.
        EVEN_NOS DB 00H         ; Reserve a byte for storing number
                                  of even data.
        ODD_NOS DB 00H          ; Reserve a byte for storing number
                                  of odd data.
        DATA ENDS               ; Data segment ends.
        CODE SEGMENT            ; Code segment starts.
START:  MOV AX, DATA            ; Move segment address of DATA to DS.
        MOV DS, AX              ; Move the content of AX to DS.
        MOV BL, 00H             ; Initialize BL with 00H, to store
                                  the number of even data.
        MOV DL, 00H             ; Initialize DL with 00H, to store
                                  the number of odd data.
        MOV CX, COUNT           ; Initialize CX with COUNT.
        MOV SI, OFFSET ARRAY    ; Move the offset address of ARRAY to
                                  SI.
AGAIN:  MOV AL, [SI]            ; Move one byte from ARRAY to AL.
        RCR AL, 1               ; Rotate AL right through the carry
                                  by 1 bit.
        JC ODD                  ; If carry = 1, the number is odd. So go
                                  to ODD.
        INC BL                  ; Otherwise, the number is even;
                                  increment BL.
        JMP L1                  ; Jump to L1.
ODD:    INC DL                  ; Increment DL by 1 as the number is
                                  odd.
```

```
L1:     INC SI              ; Increment SI to point to the next
                              data.
        LOOP AGAIN          ; Go to LOOP AGAIN, CX times.
        MOV EVEN_NOS, BL    ; Store the content of BL in EVEN_NOS.
        MOV ODD_NOS, DL     ; Store the content of DL in ODD_NOS.
        MOV AH, 4CH
        INT 21H             ; Return to the DOS prompt.
        CODE ENDS           ; End of code segment
        END START           ; Program ends.
```

Example 4.30

Write a program to arrange the given array having four word-type data in ascending order. Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT            ; Data segment starts.
        ARRAY DW 3200H, 4F35H, 2350H, 1FC2H
                                ; Store the elements of ARRAY here.
        COUNT EQU 04H           ; COUNT is initialized with the
                                  number of data items.
        DATA ENDS               ; Data segment ends.
        CODE SEGMENT            ; Code segment starts.
START:  MOV AX, DATA            ; Move segment address of DATA
                                  to DS.
        MOV DS, AX              ; Move the content of AX to DS.
        MOV CL, COUNT           ; Load the number of data items
                                  in CX.
        DEC CL                  ; Decrement CX as the number of
                                  passes is one less than the
                                  number of data.
NEXT_PASS: MOV BL, CL           ; Initialize BL with the number
                                  of comparisons to be done in
                                  each pass.
        MOV SI, OFFSET ARRAY    ; Move the offset address of
                                  ARRAY to SI register.
COMPARE: MOV AX, [SI]           ; Move one data word from array
                                  to AX.
        CMP AX, [SI+2]          ; Compare AX with the next word
                                  in the array.
        JC L1                   ; If the first data is lesser
                                  than the second, go to L1.
        XCHG AX, [SI+2]         ; Otherwise, exchange the data
                                  in AX and the memory at [SI+2].
        XCHG AX, [SI]           ; Exchange the content of AX
                                  (smaller data) and the data in
                                  the memory at [SI].
```

```
        ADD SI, 02              ; Increment SI by 2 to compare
                                  the next data with AX.
L1:     DEC BL                  ; Decrement the number of
                                  comparisons in BL by 1.
        JNZ COMPARE             ; If BL is not 0, go to COMPARE
                                  for the next comparison.
        LOOP NEXT_PASS          ; If BL is 0, go to NEXT_PASS.
        MOV AH, 4CH
        INT 21H                 ; Return to the DOS prompt.
        CODE ENDS               ; End of segment
        END START               ; Program ends.
```

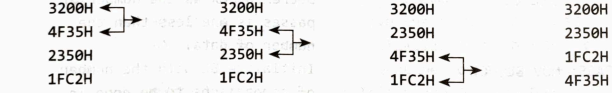
The algorithm used here is explained with the following simple example. Let us consider arranging four words stored in an array in ascending order. Since there are 4 (= N) words, 3 (= N − 1) passes have to be done. In the first pass, 3 (= N − 1) comparisons are made and the highest number is brought to the end of the array. In the second pass, 2 (= N − 2) comparisons are made since only the top three words of the array need to be compared, and in the third pass, only one comparison is needed to compare the first two data in the array.

Let us assume that the data in the array is as follows:

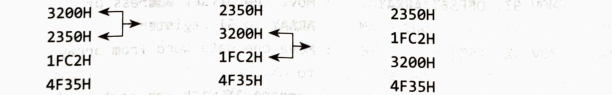
3200H
4F35H
2350H
1FC2H

The comparisons done in each pass and the exchange of data for arranging them in ascending order are shown here:

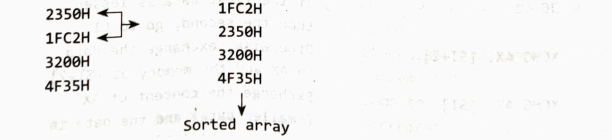
PASS I:



PASS II:



PASS III:



Example 4.31

Write a program to find the average of 100 byte-type data stored in an array in data segment. Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE1, DS: DATA1
           DATA1 SEGMENT              ; Data segment starts.
           ARRAY DB 10H, 20H, 30H     ; 100 bytes are stored.
           COUNT EQU 100              ; COUNT is the number of bytes in the
                                        array.
           AVERAGE DB 01 DUP (0)      ; Reserve one byte to store the
                                        result.
           DATA1 ENDS                 ; Data segment ends.
           CODE1 SEGMENT              ; Code segment starts.
START:     MOV AX, DATA1              ; Segment address of DATA1 is moved
                                        to AX.
           MOV DS, AX                 ; Move AX content to DS.
           MOV SI, OFFSET ARRAY       ; Move offset address of ARRAY to SI.
           XOR AX, AX                 ; Clear AX and carry flag.
           MOV BX, 0000H              ; Clear BX.
           MOV CX, COUNT              ; Move COUNT to CX.
L1:        MOV BL, [SI]               ; Move one byte from array into BL.
           ADD AX, BX                 ; Add AX and BX.
           INC SI                     ; Increment SI to point to the next
                                        byte.
           LOOP L1                    ; Repeat Loop L1 CX times.
           MOV DH, COUNT              ; Move COUNT to DH.
           DIV DH                     ; Divide AX by DH.
           MOV AVERAGE, AL            ; Store AL content in AVERAGE.
           MOV AH, 4CH                ; Return to DOS prompt.
           INT 21H
           CODE1 ENDS                 ; Code segment ends.
           END START                  ; Program ends.
```

Note: In this program, even though the array contains byte-type data, registers AX and BX are used. They are cleared initially, to keep the contents of AH and BH at 00H. The content of BH is not changed throughout the program execution. The byte to be added is first moved to BL and then BX is added with AX. The use of AX and BX results in storing the carry generated in the byte addition automatically in AH, when ADD AX, BX instruction is executed.

Example 4.32

Write a program to find the sum of many multi-byte numbers. Let us assume that we want to add four multi-byte numbers, each having four bytes. Let the numbers to be added be F0B0C010H, 203050C0H, 40453080H, and 807060B0H. The numbers are stored in the memory as shown, starting from the LSB of the first number. To store the result, five bytes are reserved, since the addition of these

numbers will generate carry, which will be stored in the fifth byte (i.e., MSB). Initially, the five bytes in the result are cleared (i.e., made 00H). Assume that MASM assembler is used.

Data in the memory (in Hex)

10	(LSB)
C0	Number 1
B0	
F0	(MSB)
C0	(LSB)
50	Number 2
30	
20	(MSB)
80	(LSB)
30	Number 3
45	
40	(MSB)
B0	(LSB)
60	Number 4
70	
80	(MSB)
00H	
00H	(LSB)
00H	Result
00H	
00H	(MSB)

Program:

```
ASSUME CS: CODE1, DS: DATA1
        DATA1 SEGMENT              ; Data segment starts.
        BYTES EQU 04H              ; BYTES indicates number of bytes
                                     in a multi-byte number.
        NUMBER EQU 04H             ; NUMBER indicates number of multi-
                                     byte numbers.
        NUM_LIST DB 10H, 0C0H, 0B0H, 0F0H, 0C0H, 50H, …
                                   ; Store the data in the multi-byte
                                     numbers in the memory.
        RESULT DB 05 DUP (0)       ; Reserve five bytes to store the
                                     result.
        DATA1 ENDS                 ; Data segment ends.
```

```
        CODE1 SEGMENT                ; Code segment starts.
START:  MOV AX, DATA1                ; Segment address of DATA1 is moved
                                       to DS.
        MOV DS, AX                   ; Move the content of AX to DS.
        MOV SI, OFFSET NUM_LIST      ; Load SI with offset address of
                                       NUM_LIST.
        MOV CX, NUMBER               ; Load CX with the value of NUMBER
                                       (= 4).
L3:     MOV BL, BYTES                ; Load BL with the value of BYTES
                                       (= 4).
        MOV DI, OFFSET RESULT        ; Load DI with the offset address
                                       of RESULT.
        XOR AL, AL                   ; Clear AL and the carry flag.
L1:     MOV AL, [SI]                 ; Move the byte at [SI] to AL.
        ADC AL, [DI]                 ; Add with carry, the byte at [DI]
                                       and AL.
        MOV [DI], AL                 ; Store the result in AL at [DI].
        INC SI                       ; Increment SI to point to the next
                                       byte.
        INC DI                       ; Increment DI to point to the next
                                       byte.
        DEC BL                       ; Decrement BL as one byte is
                                       added.
        JNZ L1                       ; If BL ≠ 0, go to L1 to perform
                                       addition of next bytes.
        JNC L2                       ; If final carry is 0, go to L2.
        INC BYTE PTR [DI]            ; Otherwise increment data at
                                       [DI].
L2:     LOOP L3                      ; Repeat loop L3, CX times.
        MOV AH, 4CH                  ; Return to DOS prompt.
        INT 21H
        CODE1 ENDS                   ; Code segment ends.
        END START                    ; Program ends.
```

Note: In this program, the first multi-byte number and the multi-byte data in the location RESULT (which initially has 00H in all bytes) are added, and the result is stored in the location RESULT itself. The final carry generated in the addition is stored in the MSB of the result. Then the next multi-byte number is added with the multi-byte number stored in RESULT, and the result of that addition is stored in RESULT again. This is repeated until all multi-byte numbers are added.

Example 4.33

Write a program to perform subtraction of two BCD data (say 80H – 17H). Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT                 ; Data segment starts.
```

```
          MINU DB 80H              ; Sort the minuend.
          SUBT DB 17H              ; Store the subtrahend.
          RESULT DB 01 DUP (0) ; Reserve one byte for storing the
                                     result.
          BORROW DB 01 DUP (0) ; Reserve one byte for storing borrow.
          DATA ENDS                ; Data segment ends.
          CODE SEGMENT             ; Code segment starts.
START:    MOV AX, DATA             ; Move segment address of DATA to DS.
          MOV DS, AX               ; Move the content of AX to DS.
          XOR AL, AL               ; Clear AL and carry flag.
          MOV AL, MINU             ; Move minuend to AL.
          MOV BL, SUBT             ; Move subtrahend to BL.
          SUB AL, BL               ; Subtract BL from AL.
          DAS                      ; Decimal-adjust AL after subtraction
                                     to get the result in BCD form.
          MOV RESULT, AL           ; Store the content of AL in RESULT.
          JNC NO_CAR               ; If borrow = 0, go to NO_CAR.
          MOV BORROW, 01H          ; Move 01H in BORROW.
NO_CAR:   MOV AH, 4CH              ; Return DOS prompt.
          INT 21H
          CODE ENDS                ; Code segment ends.
          END START                ; Program ends.
```

Note: The data at RESULT and BORROW will be 63H and 00H, respectively, after execution of the program.

Example 4.34

Write a program to multiply two 8-bit data, namely, operand 1 and operand 2, and to divide the same two data (i.e., operand 1/operand 2). Assume that MASM assembler is used.

Program:

```
ASSUME CS: CODE, DS: DATA
          DATA SEGMENT             ; Data segment starts.
          OPR1 DB 40H              ; Operand 1 is stored.
          OPR2 DB 20H              ; Operand 2 is stored.
          PRODUCT DW 01 DUP (0)    ; Product is stored here.
          QUOTIENT DB 01 DUP (0) ; Quotient is stored here.
          REMAINDER DB 01 DUP (0); Remainder is stored here.
          DATA ENDS                ; Data segment ends.
          CODE SEGMENT             ; Code segment starts.
START:    MOV AX, DATA             ; Store segment address of DATA in DS.
          MOV DS, AX               ; Move the content of AX to DS.
          XOR AX, AX               ; Clear AH and AL.
          MOV AL, OPR1             ; Move OPR1 to AL.
          MOV BL, OPR2             ; Move OPR2 to BL.
          MUL BL                   ; Multiply AL & BL.
```

```
MOV PRODUCT, AX          ; Store AX content in PRODUCT.
XOR AX, AX               ; Clear AX and carry.
MOV AL, OPR1             ; Move OPR1 to AL.
DIV BL                   ; Divide AX by BL.
MOV QUOTIENT, AL         ; Move AL content to QUOTIENT.
MOV REMAINDER, AH        ; Move AH content to REMAINDER.
MOV AH, 4CH              ; Return to DOS prompt.
INT 21H
CODE ENDS                ; Code segment ends.
END START                ; Program ends.
```

Example 4.35

Write a program to convert the given 8-bit binary number into equivalent Gray code. Assume that MASM assembler is used.

Solution:

The equivalent Gray code of a binary number is obtained using the following relation:

Let the bits in the binary number be represented as follows:

```
B7      B6      B5      B4      B3      B2      B1      B0
(MSB)                                                   (LSB)
```

Let the bits in the equivalent Gray code be represented as follows:

```
G7      G6      G5      G4      G3      G2      G1      G0
(MSB)                                                   (LSB)
```

The bits in the Gray code are obtained by the following relations:

```
G7 = B7
and Gi = Bi XOR (Bi + 1), where i = 0 to 6
```

The algorithm used in this program is as follows:

(i) Move the binary number to AL, BL, and CL registers.

(ii) Rotate the content of BL right by 1 bit, not through the carry. The binary number bits in BL, after rotation, will be as follows:

BL

B0	B7	B6	B5	B4	B3	B2	B1

(iii) The content in CL will be as follows:

CL

B7	B6	B5	B4	B3	B2	B1	B0

(iv) XOR the content of BL with that of CL and store the result in BL.

(v) AND 80H with the content of AL to mask all the bits except MSB in AL.

(vi) If MSB in AL is 0, AND 7FH with BL to get Gray code. If MSB in AL is 1, OR 80H with BL to get Gray code since the MSB in binary code and Gray code are the same.

Program:

```
ASSUME CS: CODE, DS: DATA
          DATA SEGMENT          ; Data segment starts.
```

```
        BINARY DB 4BH          ; Sort the binary data.
        GRAY DB 01 DUP (0)     ; Reserve 1 byte for Gray code.
        DATA ENDS              ; Data segment ends.
        CODE SEGMENT           ; Code segment starts.
START:  MOV AX, DATA           ; Move segment address of DATA to DS.
        MOV DS, AX             ; Move the content of AX to DS.
        MOV AL, BINARY         ; Move binary number to AL.
        MOV BL, AL             ; Move AL to BL.
        MOV CL, AL             ; Move AL to CL.
        ROR BL, 1              ; Rotate right BL content by 1 bit.
        XOR BL, CL             ; XOR BL and CL content, the result
                                 stored in BL.
        AND AL, 80H            ; AND AL with 80H to mask the lower bits
                                 except MSB.
        CMP AL, 80H            ; Compare AL with 80H to test the MSB.
        JZ L1                  ; If MSB in AL is 1, go to L1.
        AND BL, 7FH            ; AND 7FH with BL content to make MSB in
                                 BL 0.
        JMP L2                 ; Go to L2.
L1:     OR BL, 80H             ; OR 80H with BL content to make MSB in
                                 BL to 1.
L2:     MOV GRAY, BL           ; Move BL content to GRAY.
        MOV AH, 4CH            ; Return to DOS prompt.
        INT 21H
        CODE ENDS              ; Code segment ends.
        END START              ; Program ends.
```

Example 4.36

Write a program to find the square root of the given byte-type data. Assume that the byte-type data is a perfect square. Assume that MASM assembler is used.

Algorithm:

Three registers can be used to find the square root of the number. Let the number whose square root is to be found be stored in the CL register, and the data 00H and 01H be stored in registers AL and DL, respectively. The algorithm used to find the square root is as follows:

(i) Check the content of CL, if CL = 0. Go to step (v).
(ii) Subtract the value in DL from the value in CL and store the result in CL.
(iii) Increment AL.
(iv) Add 2 with the content of DL and go to step (i).
(v) Store the value in AL in the location RESULT as AL contains the square root of the number.

Let the number for which square root has to be found be 09H. The value in different registers at the end of each iteration, during the execution of the program, is shown in Table 4.12:

Table 4.12 Values in different registers at the end of each iteration

Iteration	CL	AL	DL	Iteration	CL	AL	DL
0	09H	00H	01H	2	05H	02H	05H
1	08H	01H	03H	3	00H	03H	07H

Since CL becomes 0 at the end of the third iteration, the value in AL at the end of the third iteration (which is the result) is 03H, and is stored in the memory.

Program:

```
ASSUME CS: CODE, DS: DATA
           DATA SEGMENT          ; Data segment starts.
           NUMBER DB 09H         ; Store the number whose square root
                                   has to be found.
           RESULT DB 01 DUP (0) ; Reserve 1 byte for the result.
           DATA ENDS             ; Data segment ends.
           CODE SEGMENT          ; Code segment starts.
START:     MOV AX, DATA          ; Move segment address of DATA to DS.
           MOV DS, AX            ; Move the content of AX to DS.
           MOV CL, NUMBER        ; Move NUMBER to CL.
           MOV DL, 01H           ; Move 01H to DL.
           MOV AL, 00H           ; Move 00H to AL.
REPEAT:    CMP CL, 00H           ; Compare CL with 00H.
           JZ STORE              ; If Z = 1 (i.e., CL = 00H), go to STORE.
           SUB CL, DL            ; Subtract DL from CL and store the
                                   result in CL.
           INC AL                ; Increment AL.
           ADD DL, 02H           ; Add 02H to DL.
           JMP REPEAT            ; Go to REPEAT.
STORE:     MOV RESULT, AL        ; Store the value in AL at RESULT.
           MOV AH, 4CH           ; RESULT to DOS prompt.
           INT 21H
           CODE ENDS             ; Code segment ends.
           END START             ; Program ends.
```

Example 4.37

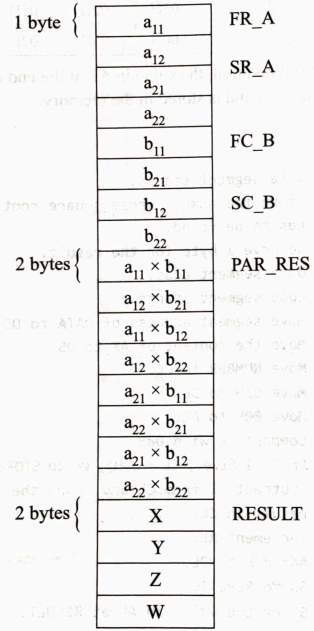
Write a program to multiply two 2 × 2 matrices whose elements are byte-type data.

Let the matrices to be multiplied be **A** and **B** and their elements as shown here:

$$\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{and} \quad \mathbf{B}=\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\mathbf{A}\times\mathbf{B}=\begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{pmatrix}=\begin{pmatrix} X & Y \\ Z & W \end{pmatrix}$$

Let all the elements of the two matrices be stored in the memory as shown. The labels have been mentioned for different memory locations. Let the elements X, Y, Z, and W of the resultant matrix all be 16-bit data (i.e., word data).



Program:

```
ASSUME CS: CODE, DS: DATA, SS: STACK
        DATA SEGMENT                    ; Data segment starts.
        FR_A DB 01H, 02H                ; Store the first-row elements
                                          of A.
        SR_A DB 03H, 04H                ; Store the second-row elements
                                          of A.
        FC_B DB 05H, 06H                ; Store the first-column elements
                                          of B.
        SC_B DB 04H, 02H                ; Store the second-column
                                          elements of B.
        PAR_RES DW 08 DUP (0)           ; Reserve eight words for storing
                                          the partial result.
        RESULT DW 04 DUP (0)            ; Reserve four words for storing
                                          the final result.
```

```
        DATA ENDS
        STACK SEGMENT                ; Stack segment starts.
        STORE DW DUP 100 (0)         ; Reserve 100 words for
                                       storage.
        STACK_TOP LABEL WORD         ; Give the label STACK_TOP to the
                                       top of the stack.
        STACK ENDS
        CODE SEGMENT
START:  MOV AX, DATA                 ; Store the segment address of
                                       DATA in DS.
        MOV DS, AX                   ; Move the content of AX to DS.
        MOV AX, STACK                ; Store the segment address of
                                       STACK in AX.
        MOV SS, AX                   ; Move the content of AX to SS.
        MOV SP, OFFSET STACK_TOP     ; Load offset of STACK_TOP in
                                       SPP.
        MOV SI, OFFSET FR_A          ; Move offset address of FR_A to
                                       SI.
        MOV DI, OFFSET FC_B          ; Move offset address of FC_B to
                                       DI.
        MOV CL, 02                   ; Move 02 to CL.
        MOV BX, OFFSET PAR_RES       ; Move offset address of PAR_RES
                                       to BX.
        CALL PAR_PROD                ; Call PAR_PROD subroutine.
        MOV SI, OFFSET FR_A          ; Move offset address of FR_A to
                                       SI.
        MOV DI, OFFSET SC_B          ; Move offset address of SC_B to
                                       DI.
        MOV CL, 02                   ; Move 02 to CL.
        CALL PAR_PROD                ; Call PAR_PROD subroutine.
        MOV SI, OFFSET SR_A          ; Move offset address of SR_A to
                                       SI.
        MOV DI, OFFSET FC_B          ; Move offset address of FC_B to
                                       DI.
        MOV CL, 02                   ; Move 02 to CL.
        CALL PAR_PROD                ; Call PAR_PROD subroutine.
        MOV SI, OFFSET SR_A          ; Move offset address of SR_A to
                                       SI.
        MOV DI, OFFSET SC_B          ; Move offset address of SC_B to
                                       DI.
        MOV CL, 02                   ; Move 02 to CL.
        CALL PAR_PROD                ; Call PAR_PROD subroutine.
        MOV BX, OFFSET PAR_RES       ; Move offset address of PAR_RES
                                       to BX.
        MOV DI, OFFSET RESULT        ; Move offset address of RESULT
                                       to DI.
```

```
          MOV CX, 04              ; Move 04 to CX.
L3:       MOV AX, [BX]            ; Move word at [BX] to AX.
          ADD BX, 02              ; Increment BX by 2 to point to
                                    the next word.
          ADD AX, [BX]            ; Add AX and the word at [BX].
          MOV [DI], AX            ; Move the content of AX to
                                    [DI].
          ADD BX, 02              ; Add 02 to BX to point to the
                                    next word, for addition.
          ADD DI, 02              ; Add 02 to DI to point to the
                                    next word, for storage.
          LOOP L3                 ; Repeat loop L3, CX times.
          JMP L4                  ; Go to L4.
PAR_PROD: XOR AX, AX              ; Clear AX and carry flag.
          MOV BP, SI              ; Move the content of SI to BP
                                    for use in multiplication of
                                    the next column.
L1:       MOV AL, [SI]            ; Move byte at [SI] to AL.
          MUL BYTE PTR [DI]       ; Multiply AL and the byte at
                                    [DI].
          MOV [BX], AX            ; Store the content of AX at [BX].
          INC SI                  ; Increment SI.
          INC DI                  ; Increment DI.
          ADD BX, 02              ; Add 02 with BX to point to the
                                    next word.
          MOV AH, 00H             ; Clear AH for the next
                                    multiplication.
          DEC CL                  ; Decrement CL.
          JNZ L1                  ; If CL ≠ 0, go to L1.
L2:       RET                     ; Return from subroutine.
L4:       MOV AH, 4CH             ; Return to DOS prompt.
          INT 21H
          CODE ENDS               ; Code segment ends.
          END START               ; Program ends.
```

Note: In this program, the partial products $a_{11} \times b_{11}$, $a_{12} \times b_{21}$, $a_{11} \times b_{12}$, and $a_{12} \times b_{22}$ are found by loading SI, DI, and BX registers with the correct offset address and calling the PAR_PROD subroutine. CL is loaded with 02 since two partial products belonging to a row of matrix **A** have to be found each time the PAR_PROD subroutine is called. Then the partial products $a_{21} \times b_{11}$, $a_{22} \times b_{21}$, $a_{21} \times b_{12}$, and $a_{22} \times b_{22}$ are found out by loading SI, DI, and CL with the correct values and calling the PAR_PROD subroutine again to find the four partial products for the second row of matrix **A** in a similar manner. Finally, the two adjacent partial products are added in sequence to obtain the words X, Y, Z, and W, which are stored from the location RESULT in the memory.

Example 4.38

Write a program to add a profit of ₹50 to the purchase cost of ten items stored in an array. The purchase cost of each item does not exceed ₹250. Find the selling price of each item. Assume that the MASM assembler is used.

Solution:

Since the maximum purchase cost of each item is less than 255, byte-type data is enough to store the purchase cost of the items. But when the profit is added, the selling price may exceed ₹255, and hence word-type array is used to store the selling price of the items.

```
ASSUME CS: CODE, DS: DATA
          DATA SEGMENT                    ; Data segment starts.
          PUR_COST DB F0H, 34H, 50H, ...  ; Store the purchase cost of
                                            10 items.
          PROFIT DB 50                    ; Store the profit.
          SELL_PRICE DW 10 DUP (0)        ; Store the selling price.
          COUNT EQU 10                    ; Assign a value of 10 to
                                            COUNT.
          DATA ENDS                       ; Data segment ends.
          CODE SEGMENT                    ; Code segment starts.
START:    MOV AX, DATA                    ; Move segment address of DATA
                                            to DS.
          MOV DS, AX                      ; Move the content of AX to
                                            DS.
          MOV SI, OFFSET PUR_COST         ; Move the offset address of
                                            PUR_COST to SI.
          MOV DI, OFFSET SELL_PRICE       ; Move the offset address of
                                            SELL_PRICE to DI.
          MOV CX, COUNT                   ; Move the value of COUNT to
                                            CX.
AGAIN:    MOV AX, 0000H                   ; Clear AX.
          MOV BX, 0000H                   ; Clear BX.
          MOV AL, [SI]                    ; Move data in [SI] to AL.
          MOV BL, PROFIT                  ; Move value in PROFIT to BL.
          ADD AX, BX                      ; Add AX and BX.
          MOV [DI], AX                    ; Store AX at [DI].
          INC SI                          ; Increment SI to point to the
                                            next data.
          ADD DI,02                       ; Increment DI to point to the
                                            next storage location.
          LOOP AGAIN                      ; Execute the loop AGAIN, CX
                                            times.
          MOV AH, 4CH                     ; Return to DOS prompt.
          INT 21H
          CODE ENDS                       ; Code segment ends.
          END START                       ; Program ends.
```

4.6 PROGRAM DEVELOPMENT PROCESS

Figure 4.16 illustrates the program development process. A word processor or editor program is used to generate an ASCII file for the program module, which is termed as a source file, for example, the source file named as (xxx) has an extension (xxx.asm).

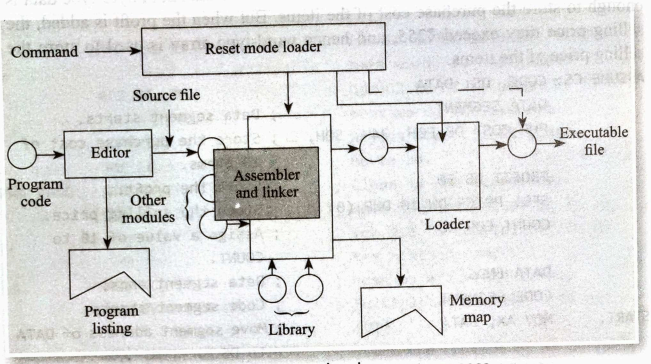


Fig. 4.16 Program development process

There are two types of statements in an assembly language program (ALP):

(i) *Instructions:* These are translated into object codes by the assembler.

(ii) *Directives:* These give direction to the assembler during the assembly process, but are not converted into object codes.

The assembler program converts a source module (xxx.asm) file into an object module, which is in hexadecimal file format (xxx.obj). Figure 4.17 depicts the assembly operation process.

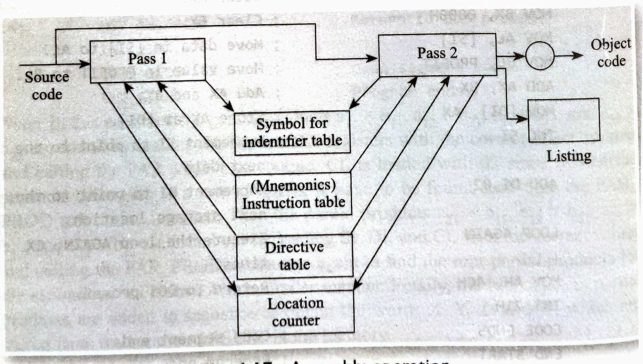


Fig. 4.17 Assembly operation

The assembler scans the program statements one by one from left to right starting with first statement to last statement indicated by an end statement. This process is called a pass. Most assemblers are two-pass assemblers.

The purpose of first pass is to provide the assembler the location of labels. In the first pass, location counter (LC) is used to construct a *symbol table.* During the first pass, the assembler does the following:

(i) Passes the source code, calculating the offset for each line in the program
(ii) Makes assumptions regarding undefined values
(iii) Does elementary error checking and displays error if necessary
(iv) Generates preliminary listing of source file

The purpose of second pass is to generate codes. The symbol table allows the second pass to use the offset of a label to generate the 'and' addresses. During the second pass, the assembler does the following:

(i) Scans the source code and converts it into machine codeusing the symbol table to insert addresses as needed
(ii) Attempts to reconcile the assumptions made in the first pass
(iii) Generates list file (if opted for)
(iv) Generates (.obj) and cross reference (.crf) files.
(v) Displays errors/warnings

The *linking process* is initiated by activating LINK or TINK command. The function of the linking process is to

(i) Combine separate object modules into one executable file
(ii) Resolve the references that are unresolved after second pass (due to public, global, or external variables)
(iii) Produce a list file showing how the object files are linked

The unresolved references after second pass are resolved with the help of the following assembler directives:

(i) PUBLIC directive is used to declare that the labels of code, data, or entire segments are available to other program modules. It is placed in the opcode field.
(ii) EXTRN (external) directive is used for indicating that the labels are external to a module. Label represents jump or data addresses. It may represent a segment. This directive must appear both in data and code segments to define labels as external to the segment. These statements are used by linker to link modules together to create a program using modular programming techniques. When segments are made public, they are combined with other public segments that contain data with the same segment name.

Libraries A library file stores a collection of related procedures. It is created with the LIB command. They are collections of assembled (xxx.obj) object files and each performs a procedure or task. They allow common procedures to be collected in aplace so that they can be shared by the users. These files have an extension (xxx.Lib) and are invoked when a program is linked with the linker program.

4.7 MODULAR PROGRAMMING

In a program it may be necessary to perform a particular task repeatedly. The formulation of complex programs from numerous complex sequences, called *program modules*, each of which performs a well-defined task, is referred to as *modular programming*. Large programs are broken down into small segments called modules. Each module implements a specific function and has its own code segment (CS) and data segment.

Procedures are useful in such situations. A procedure is a group of instructions that usually performs one task. It forms a reusable section of the software, which is stored once in memory, but used as often as necessary. This saves memory space and makes it easier to develop the software. A procedure is a sequence of instructions, which can be employed repeatedly, within a longer program.

To make software development faster, it is better to develop and test in the form of small program segments. This splitting does not cause any loss of capability to the program. The advantages of procedures over single programs are as follows:

(i) It reduces the code length and memory requirement.
(ii) It reduces development time as the modifications in the procedures are localized, which can be debugged and tested separately.
(iii) It supports modular programming methodology and improves the legibility of a program as the flow of logic is well-defined.
(iv) Since it is possible to develop library of procedures for most commonly used task, which can be shared by users from library, unnecessary duplication of codes is avoided. A list of library modules may be the only requirement for the user.
(v) Procedures provide a flexible and convenient way of exchanging information between the application programs.

It is necessary for the programmer to define and initialize the stack area (in the user memory) before a processor is expected to execute the procedure. Since the stack area is used by the processor and user, an extra care is necessary while using stack group of instructions. The data on the top of the stack can only be accessed using stack pointer (SP). It is essential to indicate the size of the stack in terms of number of bytes and starting address of the stack. The starting address of the stack area is loaded in the stack segment register (SS) and address of top of the stack in the SP.

A sample code for stack definition and initialization is as follows:

(i)
```
STACK_SEG                    ; Define a stack segment.
STACKDW 50      DUP (0)      ; Size is 50 words.
STACK_TOP LABEL WORD         ; Variable name STACK_TOP refers
                               to the stack.
```

(ii)
```
CODE_SEG                     ; Define a code segment.
MOV AX, CODE_SEG             ; Initialize CS register.
MOV CS, AX
MOV AX, STACK_SEG            ; Initialize SS register.
MOV SS, AX
LEA SP, STACK_TOP            ; Initialize top of the stack.
```

Example: If stack starts at → 6000H

(SS) = (6000H) and (SP) = (0050)

In order to handle the procedures, operations required by the processor systems are invoking or calling a procedure (called program) and returning from the module back to the main program (calling program).

In addition, it is necessary to provide input parameters to a procedure when it is called and to return back to the result or output parameter after execution. The number of input/output parameters passed to and from the procedure can vary. Even the parameters to be passed may be control/status information, which may not have a peripheral for data transfer.

Since the main module and procedure use processor registers/RAM without any difference, they can be used to hold the input as well as the output parameter information. However, it is important to take care of the data before and after calling procedural module. Original input data must be saved if the required location is modified by the module. This method is adequate as long as the amount of data/information to be exchanged is small. To exchange large amounts of data, information may be stored in memory location and it is necessary to pass the address pointer to the module. To reduce overheads, stack may also be used for parameter passing. A procedure may be classified as follows:

(i) *Intra-segment procedure (near):* These procedures are in the same segment of the main program module. They are identified by *near* directive.

(ii) *Inter-segment procedures (far):* These procedures are not in the segment of the main program module but in some other segment. They are identified by *far* directive.

(iii) *Reentrant procedures:* They define a procedure that can be interrupted, used, and reentered without losing or writing over anything. They push all the registers and flags used in the procedure and use only registers or the stack to pass parameters.

(iv) *Recursive procedures:* These procedures call themselves and are often used to work with complex data structures called trees. Recursion is a powerful tool that allows us to express our solution elegantly and can be used as an alternative to iteration as solutions methods for problems such as binary search, quick sort, as well as Fibonacci series. A recursive procedure calls itself, either directly or indirectly through another procedure.

In most cases, recursive versions tend to be inefficient as they induce more overheads to invoke and return from procedure calls. They may require duplicate computation. They have excessive demands for more memory or stack area.

The instructions used to handle procedures are as follows:

(i) CALL
(ii) RET
(iii) RETF—return from a far procedure
(iv) RETN—return form a near procedure

4.7.1 CALL Instruction

When executed, the CALL instruction performs the following two operations:

(i) Stores the return address to which the procedure will return to after execution

(ii) Modifies the contents of the instruction pointer (IP)/CS register so that it points to the starting address of the procedure, depending upon whether it is an intra-segment call or an inter-segment call

4.7.1.1 *Direct Call*

(i) *If the procedure is in the same segment:* The processor produces the starting address of the procedure by adding a 16-bit signed displacement contained in the instruction to the contents of the IP. If the displacement is negative, it is represented in 2's complement sign-and-magnitude form.
IP ← Mem16
(SP) ← Return address

(ii) *If the procedure is in another segment:* The IP and the CS register contents are changed to transfer control to the procedure. The information of new value of CS/IP is specified as bytes (4–5) and (2–3) in the instruction. It is to be noted that as usual low byte is written first.
CS ← New seg-base (Bytes 4–5)
IP ← Offset (Bytes 2–3)
(SP) ← Return address

4.7.1.2 *Indirect Call*

(i) *If the procedure is in the same segment:* The processor produces the starting address of the procedure by adding a 16-bit signed value specified by any of the general purpose register in the instruction to the contents of the IP.
IP ← Mem16
(SP) ← Return address

(ii) *If the procedure is in another segment:* It replaces the CS and the IP registers contents with 16-bit values from memory locations whose address is specified by MOD byte in the instruction. The first word from specified memory location is placed in the IP, and the next word is placed in the CS register.

Example:
CALL DWORD PTR [DX]
CS ← New seg-base address [BX+3], [BX+2]
IP ← [BX+1], [BX]
(SP) ← Return address

4.7.2 RET Instruction

The RET is the last instruction in the procedure. At the end of the procedure, the value saved in the stack is loaded back in the IP register to return execution to the calling program so that the control is transferred to the main line program. The assembler will automatically code a near RET for a near procedure and a far RET for a far procedure. Two more instructions (RETF and RETN) are provided for return from far/near procedure.

(i) RFTF at the end of the procedure copies return values from the stack back

into the IP, and CS registers to transfer control back to the next line in the main program.
IP ← (word from top of the stack)
CS ← (word from top of the stack + 2)
It may add a 16-bit immediate number contained in the instruction code to SP.

(ii) RETN at the end of the procedure copies a word from the top of the stack to the IP register.
IP ← (word from top of the stack)
It may add a 16-bit immediate number contained in the instruction code to SP.

4.7.3 Macro

A macro is a group of instructions that performs a task. It is inserted in the program during the assembly process. Macro instructions are placed in the program by the assembler at a point where they are invoked by using their name. A macro is a sequence of code that needs to be written only once, butwhose basic structure can be repeated several times within a module by giving it a name.

The code to be repeated is called the prototype code, and the prototype code along with the statements for referencing and terminating is called the macro definition. The procedure for using a macro is to give macro definition and then declare it at various appropriate points within a program by placing a statement that includes the macro's names at these points. These statements are known as macro calls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro's code. It is preceded by a macro definition and completed by a macro terminator.

The *macro* and *endm* directives are used to define a macro sequence. The first statement of a *macro* contains the name of the macro and any parameters associated with it. It is termed as a *definition*. The last statement *endm* is called a terminator. All the statements between name and terminator define a *macro body*. When *macro* is to be used its name is written. This is called macro call. The assembler replaces the call with the code. This is called macro expansion.

Example:

```
PUSH_ALLMACRO                ; Definition
PUSH AX                      ; Macro body
PUSH BX
PUSH CX
PUSH DX
PUSH DI
PUSH SI
ENDM                         ; Terminator
```

Macros may contain local variables (one which is used in the macro body, but is not available outside the macro). To define a local variable, we use the LOCAL directive. Table 4.9 depicts a comparison of procedure and macro features.

Table 4.9 Comparison of features of procedure and macro

Procedure	Macro
Called during execution	Inserted during assembly process
Assembled and executed separately	Cannot be executed separately
Reduces memory requirements	No change in memory requirements
May be anywhere and in any segment	It must be defined in the same program.
Requires a special call statement	Using the name is enough
Program control is transferred.	Program control is not transferred.
Can be used by any assembler	Used if assembler has a support for MACRO features
Parameters are passed through register, memory, or stack.	Parameters are passed as a part of the statement that calls MACRO.
Machine code is put only once in memory.	Machine code is generated each time when called.
Accessed by CALL and return mechanism during program execution	Accessed during assembly process when a name given to it is defined

4.7.4 Illustrative Example

Write a program segment to find the LCM of two numbers using a procedure to find the HCF/GCD of two numbers.

Solution:

Algorithm:

An algorithm for finding HCF/GCD of two numbers is as follows:

Step 1: Find remainder of the larger number divided by the smaller number
Dividend = larger number
Divisor = smaller number

Step 2: If rem # 0
Carry out division; where
Dividend = divisor
Divisor = remainder
Go to Step 2.
Else
Stop the process; divisor is GCD or HCF.

Program segment:

LCM of two numbers can be found using the formula (LCM = $x1 \times x2$/ HCF).

```
DATA
        X1 DWXXXX
        X2 DWYYYY
        LCM DW 2 DUP (0)
        HCF DW DUP(?)
```

```
            STACK_SEG              ; Define a stack segment.
            STACK DW50 DUP(0)      ; Size is 50 words.
            STACK_TOPLABEL WORD    ; Top of the stack can be referred
                                     using variable name STACK_TOP.
            CODE_SEG               ; Define a code segment.
ASSUME DS:  DATA_SEG
SS:         STACK_SEG
            MOV AX, CODE_SEG       ; Initialize CS register.
            MOVCS, AX
            MOV AX, STACK_SEG
            MOVSS, AX
            LEA SP, STACK_TOP      ; Initialize top of the stack.
LCM:        MOV AX, DATA
            MOV DS, AX             ; Initialize code segment register.
            MOV AX, X1
            MOVBX, X2
            CALLHCF                ; Find HCF of two numbers (x1 and x2)
            MULBX
            MOVBX, HCF
            DIV BX
            MOV LCM, AX            ; Save lower word of LCM.
            MOV LCM+2, DX          ; Save upper word of LCM.
            END LCM
            HCFPROCNEAR            ; Define a procedure with a name (HCF).
            PUSH AX
            PUSH BX
REPT:       CMP AX, BX
            JE EXIT
            JB BIG-BX
DIVAX_BX:   MOV DX, 0000H
            DIV BX
            CMP DX, 0000H
            JE EXIT
            MOV AX, DX
            JMPREPT
BIG_BX:     XCHG AX, BX
            JMPDIVAX_BX
EXIT:       MOVHCF, BX
            POP BX
            POP AX
            RET
            HCF ENDP               ; Indicates end of a procedure
```

POINTS TO REMEMBER

- The addressing modes in the 8086 are classified as register, immediate, data memory, stack memory, and program memory addressing modes.
- The data memory addressing modes are classified as direct, base, index, base plus indexed, base-relative, index-relative, and base-relative plus index addressing modes.
- The program memory addressing modes are classified as direct, relative, and indirect addressing modes.
- The 8086 instructions are classified as data transfer, arithmetic, logical, shift/rotate, flag manipulation, control transfer, string, and machine control instructions.
- The assembly language programming of the 8086 can be done with a line assembler or an assembler.
- Assembler directives are used while writing an assembly language program that is to be assembled by using an assembler.
- The formulation of complex programs from numerous complex sequences, called *program modules*, each of which performs a well-defined task, is referred to as modular programming.
- It is possible to develop a library of procedures for most commonly used tasks, which can be shared by users from library.
- *Procedures* provide a flexible and convenient way of exchanging information between the application programs.
- It is necessary for the programmer to define and initialize the stack area in the user memory area before a processor is expected to execute the procedure. Since the stack area is used by the processor and user, extra care is necessary while using stack group of instructions.
- The assembler automatically codes a near RET for a *near* procedure and a far RET for a *far* procedure.
- A macro is a group of instructions that performs a task which is inserted in the program during the assembly process.

KEY TERMS

Addressing mode This mode is the way in which the microprocessor addresses the operands while fetching data during the execution of an instruction or the way in which the microprocessor calculates the memory address from where the next instruction to be executed is taken, in the case of jump or call instructions.

Assembler It is a software that is used to convert assembly language programs into machine language programs.

Assembler directives These are commands to the assembler, which give various details in a program such as the required storage class for a particular constant or variable (byte, word, or double word), logical name of the segments (CODE, STACK, or DATA segment), type of procedures or routines (FAR, NEAR, PUBLIC, or EXTRN), end of a segment (ENDS), and macro definition (MACRO, ENDM).

Inter-segment jump This refers to the operation of jumping from one code segment to another.

Intra-segment jump This refers to the operation of jumping within the same code segment.

Library It is a collection of object files created with the LIB command and has extension .lib.

Line assembler It converts each line in an assembly language program into the corresponding machine language program, as soon as it is entered in the system.
Macro A macro is a sequence of code that needs to be written only once, but whose basic structure can be repeated several times within a module by giving it a name.
Procedure A procedure is a group of instructions that usually performs one task, which is stored once in memory, but used as often as necessary.

REVIEW QUESTIONS

1. What is the function of segment override prefix? Give two examples.
2. What is the difference between inter-segment and intra-segment jumps in the 8086?
3. What is the difference between short and near jumps in the 8086?
4. What is the function of the assembler directives FAR PTR, NEAR PTR, and SHORT PTR?
5. Write the different steps performed by the 8086 when it executes the instructions PUSH CX and PUSH [SI].
6. What are the different uses of stack in a microprocessor?
7. Write the different steps performed by the 8086 when it executes the instructions POP CX and POP [BX].
8. Write the operation performed by the 8086 when it executes the XLAT instruction. What is the use of XLAT?
9. What is the difference between fixed port and variable port addressing in the 8086?
10. Which instructions of the 8086 are used to communicate with the I/O devices in the I/O-mapped I/O scheme?
11. Write the function of the assembler directives BYTE PTR and WORD PTR.
12. What is the difference between the MUL and IMUL instructions in the 8086?
13. What is the difference between the DIV and IDIV instructions in the 8086?
14. What are the default operand and result locations for 8- and 16-bit data multiplication instructions in the 8086?
15. What are the default operand and result locations for 8- and 16-bit data division instructions in the 8086?
16. What is the function of the DAA instruction in the 8086?
17. Write the operations performed when the instruction AAD is executed in the 8086.
18. Which instructions of the 8086 are used to set and reset the D and I flags?
19. What is the range of the relative address that is used in the conditional jump instructions?
20. What is the function of the INT n instruction? Which instruction of the 8086 is used to return from the interrupt service routine to the main program?
21. What are the operations performed when the instructions LOOP and LOOPNE are executed in the 8086?
22. What is the function of the D and I flags in the 8086?
23. Which registers are used as offset registers and segment registers for pointing to the source and destination during the execution of the string instructions in the 8086?
24. What is the function of the REP and REPE prefixes used with string instructions in the 8086?

25. What is the function of the LOCK prefix used with an 8086 instruction?
26. What is the function of the assembler and assembler directives?
27. What is the function of the assembler directives ORG and DB?
28. What is a macro? Give an example.
29. What is the difference between a macro and a subroutine?
30. What is the need for passing parameters to a macro?
31. Describe the different data memory addressing modes in the 8086 giving an example for each.
32. Describe the different program memory addressing modes in the 8086 giving an example for each.
33. Explain the stack memory addressing modes in the 8086 giving examples.
34. Explain the different data transfer instructions in the 8086 giving examples for each.
35. Explain the different arithmetic instructions in the 8086 giving examples for each.
36. Describe the different logical instructions in the 8086 giving examples for each.
37. Write the function of assembler directives that are used to define variables and constant data with an example for each.
38. What are the assembler directives that are related to segment declaration? Explain with examples.
39. Write the function of assembler directives that are related to code location, with an example for each.
40. What are the assembler directives that are related to procedure declaration? Explain with examples.
41. Explain the function of the assembler directives PTR, TYPE, SHORT, GLOBAL, and LOCAL with an example for each.

PROGRAMMING EXERCISES

1. Write an 8086 assembly language program to find the sum of 100 words present in an array stored from the address 3000H: 1000H in the data segment and store the result from the address 3000H: 2000H.
2. Write an 8086 assembly language program to find the prime numbers among 100 bytes of data in an array stored from the address 4000H: 1000H in the data segment and store the result from the address 4000H: 3000H.
3. Write an 8086 assembly language program to find the number of occurrences of the character 'A' among 50 characters of a string-type data stored from the address 5000H: 1000H in the data segment and store the result in the address 2000H: 5000H.
4. Write an 8086 assembly language program to check whether the two strings, one stored from the address 2000H: 1000H in the data segment and the other stored from the address 2000H: 3000H, are equal or not. If they are equal, store the value 00H in AL. Otherwise, store the value 01H in AL.
5. Write an 8086 assembly language program to find the number of bytes that have the hexadecimal digit 'F' in their upper nibble among 100 bytes of data in an array stored from the address 8000H: 1000H in the data segment. Store the result in the address 8000H: 3000H.
6. Write an 8086 assembly language program to complement the lower nibble of

each byte in 100 bytes of data in an array stored from the address 8000H: 1000H in the data segment. Store the result from the address 8000H: 3000H.

7. Write an 8086 assembly language program to add two matrices having word-type data in each element of the matrix. Assume that each element of the result after addition of the corresponding elements of the matrix is also word-type data. The data for one matrix is present in an array stored from the address 8000H: 1000H in the data segment, and the corresponding data for another matrix is present in an array stored from the address 8000H: 2000H in the data segment. The result is to be stored from the address 7000H: 1000H.
8. Write an 8086 assembly language program to multiply two square matrices having word-type data in each element of the matrix. Assume that each element of the resultant matrix is of double word type. The data for one matrix is present in an array stored from the address 8000H: 1000H in the data segment, and the corresponding data for the other matrix is present in an array stored from the address 8000H: 1000H in the data segment. The result is to be stored from the address 7000H: 1000H.
9. Write an 8086 assembly language program to find the factorial of the given byte of data using a recursive algorithm. The result is to be stored in the address 7000H: 1000H.
10. Write a non-recursive assembly language subroutine for the 8086 to evaluate the number $F_n = F_{n-1} + F_{n-2}$ for any given $n > 1$ given that $F_0 = 0$ and $F_1 = 1$. Consider the number n in such a way that F_n is not more than a 16-bit number.
11. Solve problem 1 assuming that the program is to be assembled by an assembler.
12. Solve problem 7 assuming that the program is to be assembled by an assembler.
13. Solve problem 10 assuming that the program is to be assembled by an assembler.
14. Write a procedure chg_to exchange the contents of two memory locations. Write a main program which accepts a string terminated by full stop (.), from the keyboard, reverses it using chg and displays both the strings on the terminal.
15. Write a program segment to accept a string from the keyboard consisting of digit and non-digit characters and display the sum of the digits present in the input string.
16. Write a procedure str_match that receives two pointers to strings: string and substring and searches for substring in string and returns the starting position of the first match if operation is successful else (FF) is returned.

THINK AND ANSWER

1. Let the content of the different registers in the 8086 be as follows: DS = 1000H, SS = 2000H, ES = 3000H, BX = 4000H, SI = 5000H, DI = 6000H, and BP = 7000H. Find the memory address/addresses from where the 8086 accesses the data while executing the following instructions:
   (i) MOV AX, [BX]
   (ii) MOV BX, [SI]
   (iii) MOV CX, [BP]
   (iv) MOV AL, [DI]
   (v) MOV BH, SS: [SI]
   (vi) MOV CX, ES: [DI]
   (vii) MOV AX, [BX + DI]
   (viii) MOV BX, [BP + DI + 5]
   (ix) MOV AH, [BX + 10H]
   (x) MOV CX, DS: [BP + 4]
   (xi) MOV BX, [SI − 5]
   (xii) MOV AX, [BX + 10]
2. Which registers of the 8086 are modified while executing inter-segment and intra-segment jump instructions?

3. Is it possible to exchange the content of two memory locations or the content of two segment registers using the XCHG instruction? Why?
4. If the content of BP = 1000H and SI = 2000H, what is the value present in CX after the 8086 executes the instructions LEA CX, [BP + SI], and LEA CX, [SI].
5. Is it possible to use two memory operands in the ADD and SUB instructions?
6. Is the carry flag affected by the execution of the INC and DEC instructions in the 8086?
7. What is the difference between SUB and CMP instructions?
8. What is the difference between TEST and AND instructions?
9. Which instructions of the 8086 are used to handle procedure or subroutine?
10. What is the difference between arithmetic and logical right-shift?
11. What are the common applications of left-shift and right-shift operations?
12. When is the CL register used with the shift and rotate instructions?
13. Consider the following pair of partial programs:

(i)
```
MOV AX, 4000H
ADD AX, AX
ADC AX, AX
JC DOWN
```

(ii)
```
MOV AX, 4000H
ADD AX, AX
RCL AX, 1
JC DOWN
```

For each case, what is the data in AX after execution of the third instruction and from where does the processor fetch the next instruction after execution of the fourth instruction?

14. How is the WAIT instruction used to coordinate the operation between the 8086 and the 8087?

8086 Interrupts

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Different types of interrupts in the 8086, such as hardware and software interrupts
- Processing of an interrupt by the 8086
- Interrupt vector table and interrupt vectors in the 8086
- Functions of the different interrupts in the 8086
- Priority among the interrupts in the 8086
- Writing interrupt service routines
- A few BIOS (basic input/output system) interrupts or function calls

5.1 INTRODUCTION

The 8086 allows normal program execution to be interrupted in one of the following ways:

(i) An external signal given through one of its interrupt pins (INTR/NMI)
(ii) A special instruction in the program, such as the software interrupt instruction (INT N)
(iii) The occurrence of an error condition such as divide-by-0
(iv) A trap interrupt

After receiving the interrupt, the microprocessor stops the execution of the current program and calls a procedure called *interrupt service routine* (ISR), which services that interrupt. The IRET instruction executed at the end of the interrupt service routine returns the execution to the interrupted program.

5.2 INTERRUPT TYPES IN 8086

There are 256 interrupt types in the 8086. Among these, a few interrupt types are assigned for specific interrupts such as the divide-by-0 interrupt, trap interrupt, and the NMI interrupt. A few interrupt types are reserved by Intel for future expansion. The programmer is free to use the remaining interrupt types according to his/her requirement.

An 8086 interrupt can come from any one of the following four sources:

(i) An external signal applied to the non-maskable interrupt (NMI) pin or to the interrupt (INTR) pin. An interrupt caused by a signal applied to one of these inputs is called *hardware interrupt*.
(ii) The execution of the instruction INT n, where n is the interrupt type that can take any value between 00H and FFH. This is called *software interrupt*.
(iii) An error condition such as divide-by-0, which is produced in the 8086 by the execution of the DIV/IDIV instruction.
(iv) A trap interrupt.

5.3 PROCESSING OF INTERRUPTS BY 8086

After executing each instruction in a program, the 8086 checks if any interrupt has been requested. If an interrupt has been requested, the 8086 processes it by performing the following series of steps:

(i) Pushes the content of the flag register onto the stack to preserve the status of the interrupt flags (IF) and trap flags (TF), by decrementing the stack pointer (SP) by 2
(ii) Disables the INTR interrupt by clearing IF in the flag register
(iii) Resets TF in the flag register, to disable the single step or trap interrupt
(iv) Pushes the content of the code segment (CS) register onto the stack by decrementing SP by 2
(v) Pushes the content of the instruction pointer (IP) onto the stack by decrementing SP by 2
(vi) Performs an indirect far jump to the start of the interrupt service routine (ISR) corresponding to the received interrupt

Figure 5.1 shows the processing of an interrupt by the 8086.

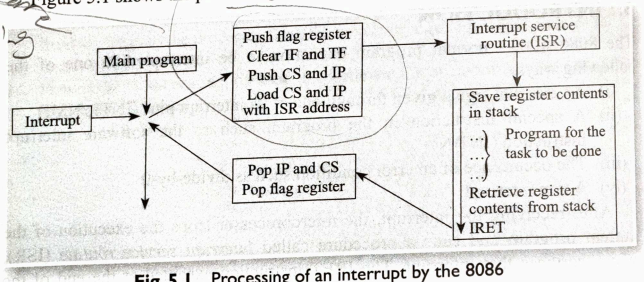


Fig. 5.1 Processing of an interrupt by the 8086

Now, let us see in detail how the 8086 does an indirect far jump to the start of the ISR of the received interrupt. When the 8086 responds to an interrupt, it refers to four memory locations present in the interrupt vector table (IVT), to get the new values of CS and IP. These memory locations are used to find the starting address of the ISR of the received interrupt in the memory. In an 8086 system, the first 1 KB of memory from the addresses 00000H–003FFH is set aside as a table called interrupt vector table (IVT) for storing the interrupt vectors (IVs). Each interrupt vector indicates the starting address of the ISR of a particular interrupt in the memory. It contains four bytes, in which the lower two bytes are called offset and the upper two bytes are called segment. The offset part of the interrupt vector is loaded in the IP register and the segment part is loaded in the CS register. While using interrupts in the 8086, the ISR of the different interrupts must be initially stored in the memory at the desired locations. Then the interrupt vectors corresponding to the various interrupts are stored in the IVT. For example, if the ISR of interrupt type 0 is stored in the memory starting at the address 30000H, the segment part of the interrupt vector is entered as 3000H and its offset part is

entered as 0000H in the IVT. When these two values are loaded in the CS and IP registers, respectively, the 8086 calculates the address of the next instruction to be executed using the relation CS × 10H + IP, and obtains 30000H, which is the starting address of the ISR of interrupt type 0. Since four bytes are required to store the CS and IP values for each ISR in the IVT, and the IVT must hold the interrupt vector for a maximum of 256 interrupts, the maximum size of the IVT is 1 KB. Each interrupt vector is also called interrupt pointer and the IVT is also referred to as the interrupt pointer table.

Figure 5.2 shows the 256 interrupt vectors arranged in the IVT in the memory. The IP value is inserted as the lower-order word of the interrupt vector and the CS value is inserted as the higher-order word of the interrupt vector. Each interrupt vector is identified by a number called its *type,* which is an 8-bit number. Hence the different interrupt types vary from 0 to 255 (00H–FFH). The lowest five types are dedicated to specific interrupts such as the divide-by-0 interrupt, the single step (trap) interrupt, the NMI interrupt, the one-byte INT instruction interrupt, and the overflow interrupt. Interrupt types 5–31 are reserved by Intel for use in advanced microprocessors such as the 80286 and the 80386. The upper 224 interrupt types are available for the programmer to use for hardware/software interrupts.

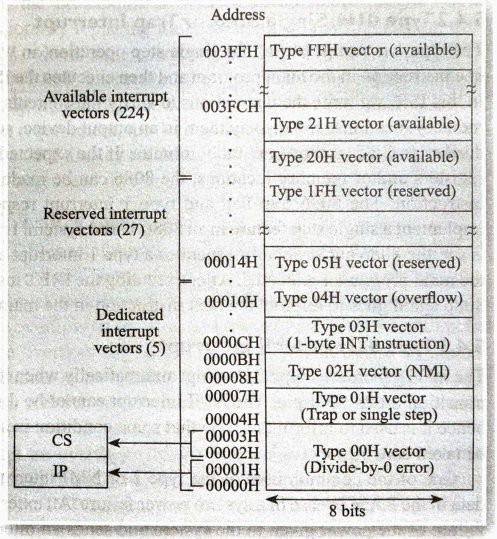


Fig. 5.2 Interrupt vector table in the 8086

The interrupt vector for each interrupt type requires four memory locations. For example, the interrupt vector for type 00H occupies the memory locations 00000H–00003H, the interrupt vector for type 01H occupies the memory locations 00004H–00007H, and so on. When the 8086 responds to a particular type of interrupt, it automatically multiplies the type of that interrupt by 4 to find the desired address in the vector table, from where it takes the interrupt vector and loads it in the IP and CS registers. For example, if the interrupt type 03H is currently received by the 8086, it goes to the memory address given by 03H × 04H = 000CH, to get the interrupt vector for type 03H.

5.4 DEDICATED INTERRUPT TYPES IN 8086

The lowest five interrupt types in the 8086 (i.e., types 00H–04H) are dedicated to specific interrupts such as the divide-by-0 interrupt, the single step (trap) interrupt, the NMI interrupt, the one-byte INT instruction interrupt, and the overflow interrupt. Let us now discuss these interrupt types in detail.

5.4.1 Type 00H or Divide-by-zero Interrupt

Whenever the quotient from a DIV or IDIV operation is too large to fit in the result register, which occurs while dividing a number by 0, or if the divisor is very small compared to the dividend, the 8086 automatically generates a type 0 interrupt.

5.4.2 Type 01H, Single Step, or Trap Interrupt

The type 1 interrupt is used for single step operation, in which the 8086 executes one instruction in the main program and then executes the ISR of the trap interrupt. In this ISR, we write the instructions to verify the contents of certain registers and memory locations, and display them in an output device, such as a seven-segment display or CRT (cathode ray tube) monitor. If the expected data are present in the registers and/or memory locations, the 8086 can be made to proceed to the next instruction. The 8086 trap flag and type 1 interrupt response make it easier to implement a single step feature in an 8086-based system. If the trap flag in the 8086 is set, the 8086 automatically generates a type 1 interrupt after each instruction in the main program is executed. After executing the IRET instruction in the ISR, the 8086 again goes to execute the next instruction in the main program.

5.4.3 Type 02H or NMI Interrupt

The 8086 generates a type 2 interrupt automatically when it receives a low-to-high transition on its NMI pin. The NMI interrupt cannot be disabled by software and hence it is used to inform the 8086 that some condition in an external system must be taken care of.

One of the common uses of the type 2 or NMI interrupt is to save important data in the RAM in case of a system power failure. An external circuitry detects the failure of the power given to the system and sends an interrupt signal to the NMI input of the 8086. Due to the large filter capacitor present in most power supplies, the DC power to the 8086 remains for a few ms (say 25 ms or 50 ms) after the AC power has failed. This time is sufficient for the NMI interrupt's ISR to copy the important data used in the program to a RAM chip with battery-backed power supply. When AC power is restored, the data stored in the battery-backed RAM can be retrieved and the program resumes execution from where it stopped.

The NMI interrupt is also used to sense hazardous situations such as fire, smoke, and unsafe pressure or temperature limits in an industrial environment, when the 8086 is used to control the industrial processes. In these applications, an appropriate sensor is used to detect the abnormal condition and its output is connected to the NMI interrupt. Whenever the NMI interrupt is activated, the 8086 runs the NMI interrupt's ISR, which is used to issue an alarm signal and shut off the process if needed.

5.4.4 Type 03H or One-byte INT Interrupt

The type 3 interrupt is produced by the execution of the INT 03H instruction. It is a single-byte instruction, which is mainly used to implement a breakpoint function in the 8086 system, for debugging a program. When we insert a breakpoint in the program, the 8086 system executes the instructions up to the breakpoint and then executes the ISR corresponding to the breakpoint interrupt. Unlike the single-step technique in which the execution is stopped after each instruction, the breakpoint technique allows us to execute all the instructions up to the inserted breakpoint in the main program. The processor then goes on to execute the ISR of the breakpoint interrupt.

In an 8086 system, the breakpoint is inserted in the main program at a particular place by temporarily replacing the instruction byte at the address with the instruction byte CCH, which is the opcode of the INT 03H instruction. When the 8086 executes the INT 03H instruction, the type 3 interrupt is produced. In the type 3 interrupt's ISR, all the register contents are saved in the stack. Then, depending on the system requirement, the desired register and/or memory location contents may be sent to a CRT display for debugging, while the system waits for a command from the user to proceed further.

5.4.5 Type 04H or Overflow Interrupt

The 8086 overflow flag (OF) is set if the result of an arithmetic operation on signed numbers is too large to be stored in the destination register or memory location. There are two ways to detect and respond to an overflow error in a program:

(i) Place the jump on overflow (JO) instruction immediately after the arithmetic instruction. If the overflow flag is set due to the result of the arithmetic instruction, execution is transferred to the address specified in the JO instruction. At this address, an error routine that responds to the overflow in the required manner can be placed.

(ii) Place the interrupt on overflow (INTO) instruction immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, it is treated as a NOP (no operation) instruction. However, if the overflow flag is set, the 8086 generates a type 4 interrupt after executing the INTO instruction. Instructions in the ISR produce the desired response to the error condition. The advantage of using the INTO instruction is that the type 4 interrupt's ISR can be easily accessed from any program in a multitasking environment.

5.5 SOFTWARE INTERRUPTS—TYPES 00H–FFH

The INT instruction of the 8086 can be used to generate any one of the 256 possible interrupt types, which are called *software interrupts*. The desired interrupt type is specified as part of the INT instruction. For example, the INT 21H instruction causes the 8086 to generate an interrupt of the type 21H. The response of the 8086 to the software interrupt is same as that for any of the interrupt types described in Section 5.4.

In general, when the 8086 executes the INT n instruction where n is the

interrupt type, the 8086 pushes the content of the flag register, CS, and IP values into the stack register, and clears IF and TF. Then the 8086 goes to the memory address (given by $4 \times n$) to obtain the interrupt vector for the type n from the IVT and loads it in the IP and CS registers. This makes the 8086 execute the ISR for the interrupt type n. The IRET instruction at the end of the ISR makes the 8086 return to the main program to the instruction next to the INT n instruction, to continue the execution of the main program.

Software interrupts produced by the INT instruction have the following uses:

(i) Inserting break points in a program for debugging. The INT 03H instruction is used for this purpose.

(ii) Testing the function correctness of various ISRs. For example, the INT 02H instruction can be used to test the ISR for the NMI interrupt, without giving any input signal to the NMI pin of the 8086.

5.6 INTR INTERRUPTS—TYPES 00H–FFH

The 8086 INTR interrupt allows an external signal to interrupt the execution of a program. The INTR interrupt can be masked (disabled) so as to not cause an interrupt. If IF is set, the INTR interrupt is enabled and if IF is cleared, INTR is disabled. IF can be set and cleared at any time, using the STI and CLI instructions, respectively. After the 8086 is reset, IF is set using the STI instruction, if the user needs to use the INTR interrupt. The INTR interrupt is activated by a high level (i.e., logic 1) signal in the INTR pin. The minimum duration for which the INTR signal must be kept high to be recognized by the 8086 is equal to the execution time of the instruction that takes longest time for execution. This is because the 8086 tests the INTR signal during the last clock cycle of an instruction cycle.

If the INTR input is high and IF is set, the 8086 is interrupted. As part of the response to the interrupt, the 8086 automatically clears IF. This is done for the following two reasons:

(i) To prevent a signal on the INTR input from interrupting a higher priority ISR in progress. If needed, IF can be set at the beginning of the lower priority ISR, so that 8086 can be interrupted by an INTR interrupt while executing that ISR.

(ii) To make sure that a signal in the high state, existing for a sufficient duration (say, a few μs), on the INTR input, does not cause the 8086 to interrupt it again before completing the execution of its ISR.

The IRET instruction at the end of the ISR restores IF and TF to their original value. When the 8086 processes an INTR interrupt signal, its response is slightly different from its response to other interrupts. For an INTR interrupt, the interrupt type is sent to the 8086 from an external hardware device such as a programmable interrupt controller (the 8259) or a tri-state octal buffer (IC 74244) connected to an 8-bit DIP switch having the specific interrupt type.

Figure 5.3 shows the 8086 INTR interrupt's acknowledgement cycle.

Figure 5.4 shows the simplified diagram for interfacing the 8259 with the 8086.

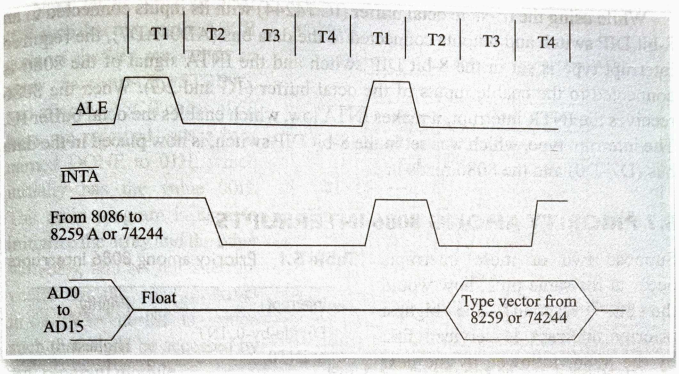


Fig. 5.3 8086 INTR interrupt's response

When the 8259 receives an interrupt signal on one of its IR inputs (IR0–IR7), it sends an interrupt signal (INT) to the INTR input of the 8086. If the INTR interrupt is enabled (in the 8086) by setting IF, the 8086 responds as shown in Fig. 5.3.

The 8086 does two interrupt acknowledge cycles when it receives the INTR

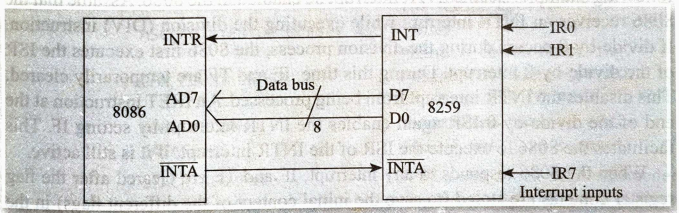


Fig. 5.4 Simplified diagram of interfacing the 8259 with the 8086

interrupt. During the first acknowledgement, the 8086 floats the data bus AD15–AD0 and sends out an Interrupt Acknowledgement ($\overline{\text{INTA}}$) pulse through its $\overline{\text{INTA}}$ pin. This pulse instructs the 8259 to perform certain internal operations to get the interrupt type related to the interrupt received by it. The interrupt type for the IR0 interrupt in the 8259 is pre-programmed in it during its initialization process. The interrupt type for successive interrupts in the 8259 (IR1, IR2,...IR7) is one greater than the interrupt type of the previous interrupt. For example, if the interrupt type assigned to IR0 is 50H, the interrupt type assigned to IR1 is 51H, that assigned to IR2 is 52H, and so on. During the second acknowledge cycle, the 8086 sends out another pulse on its $\overline{\text{INTA}}$ pin. In response to this second $\overline{\text{INTA}}$ pulse, the 8259 places the interrupt type on the lower eight lines of the data bus (AD7–AD0), which is read by the 8086. After receiving the interrupt type, the 8086 goes on to execute the ISR of the received interrupt type. The advantage of using the 8259 with the 8086 is the ability of the 8086 to handle multiple hardware interrupts and not merely two (INTR and NMI).

While using the tri-state octal buffer (IC 74244) with its inputs connected to an 8-bit DIP switch and outputs connected to the data bus (AD0–AD7), the required interrupt type is set in the 8-bit DIP switch and the $\overline{\text{INTA}}$ signal of the 8086 is connected to the enable inputs of the octal buffer ($\overline{1G}$ and $\overline{2G}$). When the 8086 receives the INTR interrupt, it makes $\overline{\text{INTA}}$ low, which enables the octal buffer IC. The interrupt type, which was set in the 8-bit DIP switch, is now placed in the data bus (D7–D0) and the 8086 reads it.

5.7 PRIORITY AMONG 8086 INTERRUPTS

Suppose two or more interrupts occur at the same time, how would the 8086 respond? The highest priority interrupt is serviced first by the 8086, followed by the next highest priority interrupt, and so on. Table 5.1 shows the priority assigned to the different interrupts in the 8086.

Table 5.1 Priority among 8086 interrupts

Interrupt	Priority
Divide-by-0, INT n, INT0	Highest
NMI	↓
INTR	↓
Single step or trap	Lowest

To explain the use of the assigning of priority among interrupts, consider the following example. Let the INTR interrupt be enabled in the 8086. Assume that the 8086 receives an INTR interrupt while executing the division (DIV) instruction. If divide-by-0 occurs during the division process, the 8086 first executes the ISR of the divide-by-0 interrupt. During this time, IF and TF are temporarily cleared. This disables the INTR interrupt from being processed. An IRET instruction at the end of the divide-by-0 ISR again enables the INTR interrupt by setting IF. This facilitates the 8086 to execute the ISR of the INTR interrupt, if it is still active.

When the 8086 responds to any interrupt, IF and TF are cleared after the flag register contents are stored (to save the initial content of the different flags) in the stack. If needed, IF, TF, or both the flags can be set at the beginning of the ISR of any interrupt, in case the user wants to enable them while executing the current ISR.

5.8 INTERRUPT SERVICE ROUTINES

While using an interrupt, the programmer must set its interrupt vector with the CS and IP addresses of the starting location of the ISR of that interrupt type, either through the program or externally. The method of defining the ISR for software and hardware interrupts is the same. This is explained with a few examples.

Example 5.1

Figure 5.5 shows the interfacing of an ASCII keyboard with the 8086 through a port in the 8255 having the address FFE0H. When a key is pressed on the keyboard, the ASCII code of that key is available on its data lines (D7–D0) and the KBINT pin is pulled low for some time. This causes the NMI input of the 8086 to go high, thereby interrupting the 8086. In the NMI interrupt's ISR, the ASCII code of the key pressed can be read through the 8255.

Now, let us write an NMI ISR such that it stores the ASCII code of the key pressed in an array named ASC_STRING, and after the ASCII codes of 50 keys are received, sets a byte named DONE to 01H, which initially has the value 00H. The main program is used to initialize the array and the other variables, and set the interrupt vector for the NMI interrupt in the IVT. The ISR is written such that it can be accessed by any program module.

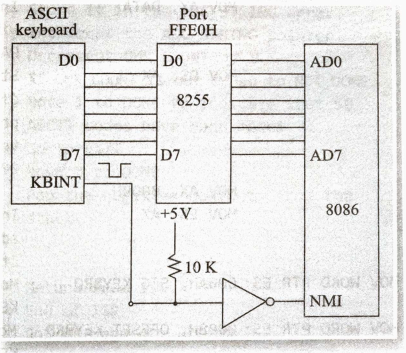


Fig. 5.5 Interfacing an ASCII keyboard with the 8086

Solution:

```
ASSUME CS:    CODE, DS: DATA, SS: STACK
              DATA SEGMENT WORD PUBLIC ; This data segment can be
                                         accessed by any other module.
              ASC_STRING DB 50 DUP (0) ; Reserve 50 bytes for storing
                                         the ASCII codes.
              ASC_POINTER DW OFFSET ASC_STRING
                                       ; Pointer to ASCII string.
              CHR_COUNT DB 50          ; Assign the number of ASCII
                                         codes to CHR_COUNT.
              DONE DB 00H              ; Initialize DONE to 00H.
              DATA ENDS                ; End of the data segment.
              STACK SEGMENT            ; Set up the stack segment needed
                                         for handling the interrupt.
              DW 100 DUP (0)           ; Reserve 100 words for the
                                         stack.
              STACK_TOP LABEL WORD     ; Assign the label STACK_TOP
                                         to the top of the stack.
              STACK ENDS               ; End of the stack segment.
              PUBLIC CHR_COUNT, DONE, ASC_POINTER
                                       ; Make the variables available
                                         to other modules.
EXTRN KEYBRD: FAR                      ; KEYBRD procedure (which is
                                         the NMI ISR) is present in
                                         another module.
              CODE SEGMENT WORD PUBLIC ; Code segment to initialize
                                         NMI IV starts.
START:        MOV AX, STACK            ; Initialize the SS register
                                         with the segment address of
                                         the STACK.
              MOV SS, AX
              MOV SP, OFFSET STACK_TOP ; Initialize the SP register.
```

```
                MOV AX, DATA            ; Initialize the DS register
                                          with the segment address of
                                          DATA.
                MOV DS, AX              ; Store the segment address and
                                          offset address of the KEYBRD
                                          procedure in the interrupt
                                          vector table in the addresses
                                          0008H-000BH.
                MOV AX, 0000H
                MOV ES, AX              ; Initialize ES with the segment
                                          address 0000H, as the IVT is
                                          stored in this segment.
MOV WORD PTR ES: 000AH, SEG KEYBRD      ; Move the segment address to
                                          KEYBRD to the IVT.
MOV WORD PTR ES: 0008H, OFFSET KEYBRD   ; Move the offset address of
                                          KEYBRD to the IVT.
HERE:           JMP HERE                ; Wait until a key is pressed in
                                          the keyboard.
                CODE ENDS
                END START

; The KEYBRD procedure (i.e., NMI ISR) follows.
ASSUME CS:      CODE, DS: DATA
                DATA SEGMENT WORD PUBLIC
                                        ; This segment can be accessed
                                          by any other module.
EXTRN CHR_COUNT: BYTE, DONE: BYTE, ASC_POINTER: WORD
                                        ; These variables are present in
                                          another module.
                DATA ENDS               ; End of data segment
                PUBLIC KEYBRD           ; The procedure KEYBRD can be
                                          accessed by some other module.
                CODE SEGMENT WORD PUBLIC
                                        ; Code segment having KEYBRD
                                          procedure starts
                KEYBRD PROC FAR         ; Beginning of the KEYBRD
                                          procedure
                PUSH AX                 ; Store the content of the AX,
                                          BX, and DX registers in the
                                          stack.
                PUSH BX
                PUSH DX
                CMP CHR_COUNT, 0        ; Check whether CHR_COUNT = 0.
                JZ EXIT                 ; If it is 0, go to EXIT.
                MOV BX, ASC_POINTER     ; Move the value in ASC_POINTER
                                          to BX.
                MOV DX, 0FFE0H          ; Store the address of the 8255
                                          port in DX.
                IN AL, DX               ; Get the ASCII code of the key
                                          from the keyboard.
```

```
        MOV [BX], AL          ; Store it in the ASC_STRING array.
        INC ASC_POINTER       ; Increment the ASC_STRING pointer.
        DEC CHR_COUNT         ; Decrement CHR_COUNT by 1.
        JNZ NOT_DONE          ; If CHR_COUNT is not 0, go to NOT_DONE.
        MOV DONE, 01          ; Move 1 to DONE to indicate that 50
                                ASCII codes have been received.
        JMP EXIT              ; Go to EXIT.
NOT_DONE: MOV DONE, 00        ; Move 0 to DONE.
EXIT:   POP DX                ; Pop the register contents from the
                                stack.
        POP BX
        POP AX
        IRET                  ; Return from ISR.
        KEYBRD ENDP           ; End of ISR
        CODE ENDS             ; End of segment
        END
```

Example 5.2

Write a program that displays the message 'IRQ2 IS WORKING', in the monitor of the personal computer (PC), if a hardware interrupt signal appears on the IRQ2 pin present in the I/O channel of the PC, and the message 'IRQ3 IS WORKING' if a hardware interrupt signal appears on the IRQ3 pin present in the I/O channel of the PC. Make use of the DOS (disk operating system) interrupt INT 21H.

Solution:

When a hardware interrupt signal appears on the IRQ2 pin present in the I/O channel of the PC, it activates the INTR pin of the CPU (8086). When the CPU sends the $\overline{\text{INTA}}$ pulse, the interrupt type 0AH is supplied to the CPU by the I/O channel of the PC. Hence, the effect of this action is the same as that of executing the software instruction INT 0AH. Similarly, when a hardware interrupt signal appears at the IRQ3 pin present in the I/O channel of the PC, it activates the INTR pin of the CPU (i.e., processor), and when the CPU sends the $\overline{\text{INTA}}$ pulse, the interrupt type 0BH is supplied to the CPU by the I/O channel of the PC. Hence, the effect of this action is the same as that of executing the software instruction INT 0BH.

The DOS interrupt or function call INT 21H, which comes along with the DOS program, is used for performing various functions in the PC such as accessing the printer, monitor, and keyboard, and creating files. Before using INT 21H for executing a specific instruction, the register AH, DX, or DS, or a combination of these registers has to be loaded with a specific value. Now the specified operation is carried out and a particular value is returned in specific registers or in flags, after the execution of the INT 21H instruction, to reflect the result of the operation.

Main program:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT
        MESSAGE1    DB      "IRQ2 IS WORKING", 0AH, 0DH, "$"
        MESSAGE2    DB      "IRQ3 IS WORKING", 0AH, 0DH, "$"
```

```
        DATA ENDS

        CODE SEGMENT
START:
        MOV AX, CODE
        MOV DS, AX              ; Set DS with the segment address of
                                  CODE, for setting the IVT.
        MOV DX, OFFSET IRQ2_ISR ; Set DX with the offset of IRQ2_ISR.
        MOV AX, 250AH           ; Set the IVT using the function
                                  value 250AH in AX (AH = 25H,
                                  AL = 0AH (interrupt type))
        INT 21H                 ; Call the DOS interrupt INT 21H to set
                                  the IVT.
        MOV DX, OFFSET IRQ3_ISR ; Set DX with the offset address of
                                  IRQ3_ISR.
        MOV AX, 250BH           ; Set IVT using the function value 250BH
                                  in AX (AH = 25H, AL = 0BH (interrupt
                                  type)).
        INT 21H                 ; Call the DOS interrupt INT 21H to
                                  set the IVT.
HERE:   JMP HERE
        IRQ2_ISR PROC NEAR
        MOV AX, DATA
        MOV DS, AX              ; Set DS with the segment address of
                                  DATA.
        MOV DX, OFFSET MESSAGE1 ; Set DX with the offset of MESSAGE1.
        MOV AH, 09H             ; Display MESSAGE1 in the monitor.
        INT 21H
        IRET                    ; Return from ISR.
        IRQ2_ISR ENDP
        IRQ3_ISR PROC NEAR
        MOV AX, DATA
        MOV DS, AX              ; Set DS with the segment address of
                                  DATA.
        MOV DX, OFFSET MESSAGE2 ; Set DX with the offset of MESSAGE2.
        MOV AH, 09H             ; Display MESSAGE2 in the monitor.
        INT 21H
        IRET                    ; Return from ISR.
        IRQ3_ISR ENDP
        CODE ENDS
        END START
```

In this program, a data segment is first defined with the messages to be displayed in the monitor of the PC when the interrupt signal is given in the I/O channel of the PC. Then, storing the segment address of CODE in the DS, the offset address

of the ISR (IRQ2_ISR) in the DX, the function value 250AH in AX (i.e., AH = 25H and AL = 0AH (interrupt type)), and by using the DOS interrupt INT 21H, the interrupt vector for the interrupt type 0AH is created in IVT. Similarly, the interrupt vector for the interrupt type 0BH is created in the IVT. In the IRQ2 ISR, the segment address of DATA is placed in DS, the offset address of MESSAGE1 is placed in DX, AH is loaded with the value 09H, and by calling the DOS interrupt INT 21H, MESSAGE1 is displayed in the monitor of the PC. A similar procedure is used in IRQ3_ISR as well.

The 0AH, 0DH, and $ characters given in MESSAGE1 and MESSAGE2 represent the ASCII code of line feed (LF), ASCII code of carriage return (CR), and end of string, respectively.

Example 5.3

Write a program to create a file named AGE in the PC and store 100 bytes of data in it, which have to be taken from the memory block starting at 3000H: 2000H, if the software instruction INT 0AH is executed by the PC. Make use of the DOS interrupt INT 21H.

Solution:

```
ASSUME CS: CODE, DS: DATA
        DATA SEGMENT
        FILENAME DB "AGE", "$"
        MESSAGE DB "FILE CREATION WAS NOT SUCCESSFUL", 0AH, ODH,"$"
        DATA ENDS

        CODE SEGMENT
START:
        MOV AX, CODE
        MOV DS, AX                  ; Set DS with the segment address
                                      of CODE, for setting the IVT.
        MOV DX, OFFSET ISR          ; Set DX with the offset address of
                                      ISR.
        MOV AX, 250AH               ; Set the IVT using the function
                                      value 250AH in AX.
        INT 21H                     ; Execute the DOS interrupt INT 21H
                                      to set the IVT.
        MOV DX, OFFSET FILENAME ; Set DX with the offset address of
                                  FILENAME.
        MOV AX, DATA
        MOV DS, AX                  ; Load the segment address of DATA
                                      in DS.
        MOV CX, 00H
        MOV AH, 3CH
        INT 21H                     ; Create a file with the file name
                                      'AGE', using INT 21H.
        JNC SUCCESS                 ; If there is no carry, the file
                                      creation operation was
                                      successful. So go to the
```

```
                                    location SUCCESS.
        MOV DX, OFFSET MESSAGE  ; If there is a carry, display the
                                    message using INT 21H.
        MOV AH, 09H
        INT 21H
        JMP END1
SUCCESS: INT 0AH                ; Execute the software INT 0AH
                                    instruction to write the data
                                    into the file.
END1:   MOV AH, 4CH             ; Return to DOS prompt.
        INT 21H                 ; ISR for interrupt type 0AH is as
                                    follows:
        ISR PROC NEAR
        MOV BX, AX              ; Take the file handle information
                                    in AX to BX.
        MOV CX, 100             ; Move the number of bytes to be
                                    transferred, to CX.
        MOV DX, 2000H           ; Store the offset address of the
                                    data to be moved into the file
                                    in DX.
        MOV AX, 3000H           ; Store the segment address of the
                                    data to be moved into the file
                                    in AX.
        MOV DS, AX              ; Move the segment address in AX
                                    to DS.
        MOV AH, 40H             ; Using INT 21H, write the data
                                    into the file.
        INT 21H
        IRET
        ISR ENDP
        CODE ENDS
        END START
```

In this program, a data segment is first defined with the file name to be assigned to the file and the message to be displayed in the monitor, if the file creation is not successful. Then, storing the segment address of CODE in DS, the offset address of the ISR in DX, and the function value 250AH in AX (i.e., AH = 25H and AL = 0AH (interrupt type)), and by using the DOS interrupt INT 21H, the interrupt vector for the interrupt type 0AH is created in the IVT. Next, storing the offset address of the file name in DX, the segment address of DATA in DS, 00H in CX, and 3CH in AX, and by using the DOS interrupt INT 21H, the file named AGE is created.

If the file creation operation is successful, the carry flag is cleared after the execution of INT 21H and the AX register is loaded with the file handle information. Otherwise, the carry flag is set. If the carry flag is cleared, the processor goes to the location named SUCCESS in the program and executes the INT 0AH instruction, which causes the execution of the ISR, to store 100 bytes of data taken from the memory block starting at 3000H: 2000H into the file. If the carry flag is set after

the execution of INT 21H, the processor executes INT 21H with DX having the offset address of the message and AH having the value 09H, to display the message 'FILE CREATION WAS NOT SUCCESSFUL' in the monitor of the PC.

In the ISR, the file handle information in AX is first transferred to BX, followed by the loading of CX with the number of data bytes to be stored into the file. Then DX and DS are loaded with the offset address and the segment address, respectively, of the memory block from where the data is to be taken. By loading AH with the value 40H and by using INT 21H, data is moved into the file.

5.9 BIOS INTERRUPTS OR FUNCTION CALLS

The BIOS (basic input/output system) is boot firmware, which is designed to be the first program run by a PC when powered on. The initial function of the BIOS is to identify, test, and initialize system devices such as the video display card, hard disk, floppy disk, and other hardware. The BIOS prepares the machine for a known state, so that the software stored on the compatible media can be loaded, executed, and given control of the PC. *BIOS function calls*, also known as BIOS interrupts, are stored in the system ROM and in the video BIOS ROM present in the PC. These BIOS function calls directly control the I/O devices with/without DOS loaded in the system. Some BIOS function calls that are used to control the monitor (video), disk, COM port, I/O devices, keyboard, and printer in the PC are briefly discussed in this section.

5.9.1 INT 10H

The INT 10H BIOS interrupt, which is also called video services interrupt, directly controls the video display in a system. INT 10H uses register AH to select the video service provided by this interrupt. The video BIOS ROM is located on the video board and varies from one video card to another used in the PC.

5.9.1.1 *Video Mode Selection*

The mode of operation for the video display is selected by placing 00H in AH, followed by one of the mode numbers in AL. Table 5.2 shows the mode of operation found in VGA (video graphics array) type video display systems using standard video modes.

Table 5.2 Video display modes

Mode	Columns	Rows	Type	Resolution	Colours
00H	40	25	Text	360 × 400	2
01H	40	25	Text	360 × 640	16
02H	80	25	Text	720 × 400	2
03H	80	25	Text	720 × 400	16
07H	80	25	Text	720 × 400	4
11H	80	30	Graphics	640 × 480	2
12H	80	30	Graphics	640 × 480	16
13H	40	25	Graphics	320 × 200	256

The set of instructions used to place the video display in mode 2 is as follows. After the instructions are executed in the PC, the mode of the display is changed and the screen is cleared.

```
MOV AH, 00H          ; Video mode service
MOV AL, 02H          ; Select mode 2.
INT 10H              ; Call BIOS interrupt.
```

To know the current video mode used in the display, AH is set to 0FH and INT 10H is executed. After execution, AL has the current video mode, AH has the number of character columns, and BH has the page number. The instructions are as follows:

```
MOV AH, 0FH          ; Select read video mode.
INT 10H              ; Call BIOS interrupt.
```

If an SVGA (super VGA) or an EVGA/XVGA (extended VGA) adapter is available, the SVGA mode is set by using the INT 10H interrupt with AX = 4F02H and BX = VGA mode. This conforms to the VESA (Video Electronics Standards Association) standard for VGA adapters. VESA is an international standards body for computer graphics, founded in 1989 by NEC Home Electronics and eight other video display adapter manufacturers. Table 5.3 shows the modes selected by the register BX for this INT 10H interrupt. Most video cards are equipped with the driver called VVESA.COM or VVESA.SYS, which ensures that the card conforms to the VESA standard functions.

Table 5.3 Extended VGA functions

BX	Extended mode
100H	640 × 400 with 256 colours
101H	640 × 480 with 256 colours
102H	800 × 600 with 16 colours
103H	800 × 600 with 256 colours
104H	1024 × 768 with 16 colours
105H	1024 × 768 with 256 colours
106H	1280 × 1024 with 16 colours
107H	1280 × 1024 with 256 colours
108H	80 × 60 in text mode
109H	132 × 25 in text mode
10AH	132 × 43 in text mode
10BH	132 × 50 in text mode
10CH	132 × 60 in text mode

5.9.1.2 *Cursor Control*

The INT 10H interrupt is also used for cursor control in the video display (i.e., monitor). Table 5.4 shows the function codes placed in AH, which are used to control the cursor on the video display. The code is shown in the Entry field and the result obtained after execution of INT 10H is shown in the Exit field. These cursor control functions work on a wide range of video displays—from the VGA display to the latest SVGA display.

Table 5.4 Functions provided by INT 10H for cursor control

Function	Entry	Exit
Select cursor type	AH = 01H CH = Starting line number CL = Ending line number	Cursor size changed
Select cursor position	AH = 02H BH = Page number (usually 0) DH = Row number (beginning with 0) DL = Column number (beginning with 0)	
Read cursor position	AH = 03H BH = Page number	CH = Starting line number (cursor size) CL = Ending line number (cursor size) DH = Current row DL = Current column
Read attribute/ character at current cursor position	AH = 08H BH = Page number	AL = ASCII character code AH = Character attribute (Note: This function does not advance the cursor.)
Write attribute/ character at current cursor position	AH = 09H AL = ASCII character code BH = Page number BL = Character attribute CX = Number of characters to write	(Note: This function does not advance the cursor.)
Write character at current cursor position	AH = 0AH AL = ASCII character code BH = Page number CX = Number of characters to write	(Note: This function does not advance the cursor.)

5.9.2 INT 11H

This interrupt is used to determine the type of equipment installed in the system. To use this interrupt, the AX register is loaded with FFFFH and then the INT 11H instruction is executed. In return, INT 11H provides information in the AX register, as given in Fig. 5.6.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
P1	P0		G	S2	S1	S0	D2	D1							

Fig. 5.6 Content of AX register after execution of INT 11H

P1 and P0 = Number of parallel ports
S2, S1, and S0 = Number of serial ports
G = 1, if game I/O is attached
D2 and D1 = Number of disk drives

5.9.3 INT 12H

The memory size present in the system is obtained by the INT 12H interrupt. After executing the INT 12H instruction, the AX register contains the number of 1 KB blocks of memory (conventional memory in the first 1 MB of address space) installed in the computer.

5.9.4 INT 13H

This interrupt controls diskettes that are within 5.25 or 3.5 inches in size and also hard disk drives attached to the system. Table 5.5 shows the functions available to this interrupt via register AH. The direct control of the hard disk drive by a programmer using INT 13H leads to problems, including the alteration or corruption of important programs such as operating system programs, compilers, and other software that are stored on the disk. This may result in system failure. Only upon reinstallation of the operating system programs in the hard disk will the PC function normally. This wastes a lot of time for the programmer. Therefore, the functions are listed without details about their usage. Before using these functions, the BIOS literature available from the company that produced the particular version of the BIOS ROM in the system should be referred to.

5.9.5 INT 14H

The INT 14H interrupt controls the serial COM (communication) ports attached to the computer. There are two COM ports—COM1 and COM2—in a computer system. In newer AT style machines, the number of COM ports is extended to four (including COM3 and COM4). Communication ports are normally controlled using software packages that allow programming of microcontrollers/digital signal processors (DSPs) serially, or by transmitting and receiving data through a modem and a telephone line. The INT 14H instruction is used to control these ports, as given in Table 5.6.

5.9.6 INT 15H

The INT 15H interrupt controls various I/O devices interfaced with the computer. It also allows access to protected mode operation and the extended memory system on an 80286, Pentium Pro, etc., but it is not recommended for use by the normal user; it is commonly used by programmers to develop OS-related programs. The functions provided by INT 15H are given in Table 5.7.

5.9.7 INT 16H

The INT 16H interrupt is used to control the keyboard in a system. This interrupt is usually accessed by the DOS interrupt INT 21H, but can also be accessed directly. Table 5.8 indicates the functions provided by INT 16H.

5.9.8 INT 17H

The INT 17H interrupt accesses the parallel printer port, called LPT1 in most systems. Table 5.9 shows the functions provided by INT 17H.

Table 5.5 Functions provided by INT 13H

AH	Function
00H	Reset the system disk
01H	Read disk status to AL
02H	Read sector
03H	Write sector
04H	Verify sector
05H	Format track
06H	Format bad track
07H	Format drive
08H	Get drive parameters
09H	Initialize fixed disk characteristics
0AH	Read long sector
0BH	Write long sector
0CH	Seek
0DH	Reset fixed disk system
0EH	Read sector buffer
0FH	Write sector buffer
10H	Get drive status
11H	Re-calibrate drive
12H	Controller RAM diagnostics
13H	Controller drive diagnostics
14H	Controller internal diagnostics
15H	Get disk type
16H	Get disk changed status
17H	Set disk type
18H	Set media type
19H	Park heads
1AH	Format ESDI drive

Table 5.7 Functions provided by INT 15H

AH	Function
00H	Cassette motor on
01H	Cassette motor off
02H	Read cassette
03H	Write cassette
0FH	Format ESDI periodic interrupt
21H	Keyboard intercept
80H	Device open
81H	Device closed
82H	Process termination
83H	Event wait
84H	Read joystick
85H	System request key
86H	Delay
87H	Move extended block of memory
88H	Get extended memory size
89H	Enter protected mode
90H	Device wait
91H	Device power on self test (POST)
C0H	Get system environment
C1H	Get address of extended BIOS data area
C2H	Mouse pointer
C3H	Set watchdog timer
C4H	Programmable opinon select

Table 5.6 Functions provided by INT 14H

AH	Function
00H	Initialize communications port
01H	Send character
02H	Receive character
03H	Get COM port status
04H	Extended initialize communications port
05H	Extended communications port control

Table 5.8 Functions provided by INT 16H

AH	Function
00H	Read keyboard character
01H	Get keyboard status
02H	Get keyboard flags
03H	Set repeat rate
04H	Set keyboard click
05H	Push character and scan code

Table 5.9 Functions provided by INT 17H

AH	Function
00H	Print character
01H	Initialize printer
02H	Get printer status

5.10 INTERRUPT HANDLERS

Any program, in general, requires to access hardware I/O devices for I/O operations such as reading data from the keyboard, displaying data on the CRT, reading/writing data on the disc, sending data to the printer or serial data transfer through EIA232C (485). The programming techniques used to handle I/O operations are as follows:

Direct access There are I/O instructions provided by the manufacturers of the processors. Using these instructions, program can be developed for interaction with peripherals. This method is time consuming since the user has to develop the code using basic I/O instructions.

High level language (HLL) support There are functions provided by the OS, such as library functions, to handle I/O operations. User can use these functions in the program which makes development faster and easier.

DOS services It is known that most of the peripherals or I/O devices are interfaced to the system employ interrupt driven data transfer. Disk operating system (DOS) has built-in routines for I/O operations. These routines, termed as DOS services, are included in the library as interrupt handlers. Application of these service routines makes program more legible and compact. DOS interrupts are used for specific purposes, such as file access, and so on. Thus, DOS interrupts facilitate the work with files, so that the user doesn't need to have a full knowledge of this file-system in order to create, read, and write a file. They can access basic input output services (BIOS) and DOS functions from their programs through software interrupts (INT instruction).

BIOS services System has built-in routines for I/O operations, which are resident in the ROM. They are called BIOS. Application of these service routines makes program more legible and compact. BIOS interrupts allow access to low-level system resources (hardware).

We will now study how to apply last two methods to the development of application programs.

5.11 DOS SERVICES: INT 21H

Invoking an interrupt can be done using the assembly language instruction INT XX. DOS services are used to accept data from the input devices and display data on video terminals. INT 21H is used for these I/O operations. We will discuss the uses of some of the basic functions provided in INT 21H. It is to be noted that sub-function code has to be loaded in AH register before issuing the command INT 21H.

Termination of a program There are two possible requirements for termination of a program—the user wants to go either to the OS or to the parent program. The sub-functions 00H and 4CH can be used as follows:

```
(a) MOV AH, 00H          ; Segment used to terminate program
    INT 21H
    END
(b) MOV AH, 4CH          ; Terminate the program but AL will decide
                           whether to return to OS or parent program.
    MOV AL, XX
    INT 21H
    END
```

Accepting input from a standard input device (keyboard) The data entered through the standard input device is either displayed on the default output device (terminal) for confirmation or not. The sub-functions [01], [07] and [08h] can be used as follows:

```
(a) MOV AH, 01H ; Accepts one character from default device
                ; with echo on output terminal
    INT 21H     ; (AL)–ASCII code of input character
    END
(b) MOV AH, 07H ; Accepts one character from default device without
                ; echo on output terminal
    INT 21H     ; (AL)–ASCII code of input character
    END         ; ^c will terminate the entry operation
(c) MOV AH, 08H ; Accepts one character from default device without
                ; echo on output terminal
    INT 21H     ; ^c will generate INT 23H to decide the next operation
    END
```

Display data on the standard output device (terminal) The two modes of data display functions are provided based on whether the program needs to display a single character or a character string. The sub-functions [02] and [09h] can be used as follows:

```
(a) MOV AL, XX  ; (al)–ASCII code of character to be displayed
    MOV AH, 01H ; Displays one character on the default device
    INT 21H
    END
```

```
(b) MOV DX, YYYYH       ; Offset of the string in memory to be displayed
    MOV AH, 09H         ; Displays string on the default device
    INT 21H
    END
(c) MOV AL, XX          ; (AL)–ASCII code of character to be displayed
    MOV AH, 01H         ; Displays one character on the default device
    INT 21H
    END
```

Example 5.4

Write a procedure to display a 4-digit value in the [AX] register.

Solution:

```
                DISPPROCNEAR
                PUSH CX
                MOV CL, 04H             ; Counter
SET_LOCATION:
                ROLAX, CL               ; Position digit
                PUSHAX
                ANDAL, OFH              ; Convert into ASCII code.
                ADDAL, 30
                CMPAL, '9'
                JBEDIS_SECT
                ADDAL, 7H
DIS_SECT:
                MOVAH, 02H
                MOVDL, AL
                INT21H
                POPAX
                DECCX
                JNZSET_LOCATION         ; Repeat for all four digits.
                POP CX
                RET
                DISPEDNP
```

Example 5.5

Write a program segment to display types of roots of quadratic equation $ax^2 + bx + c = 0$.

Solution:

```
    DISPLAY-MSG MACRO MSG
    MOV AH, 09H
    MOV DX, OFFSET MSG
    INT 21H
    ENDM
```

```
      DATA
      CR          DB0DH
      LF          DB0AH
      A           DD1.0
      B           DD3.0
      C           DD1.0
      FLAG              DD?
      CONTROL-WORD      DD?
      STATUS-WORD       DD?
      MSG-REAL                DB 'Roots are real', '$'
      MSG-EQUAL               DB 'Roots are equal'
      MSG-IMAG                DB 'Roots are imaginary', '$'
      CODE
      STARTUP
MAIN: MOVAX, @DATA
      MOVDS, AX
      FINIT
      MOV CONTROL-WORD, 03FFH
      FLDCWCONTROL-WORD
      FLD1                              ; 1.0
      FLADDST, ST                       ; 2.0
      FLADDST, ST                       ; 4.0
      FLDDWORDPTR a
      FLDDWORDPTR c
      FMUL                              ; a*c, 4
      FMUL                              ; 4*a*c
      FTST                              ; Tests status of ST
      FLD b                             ; b, 4*a*c
      FMULST, ST                        ; b*b, 4*a*c
      FSUBR                             ; b*b - 4*a*c
      FTST                              ; Tests status of ST(0)
      FSTSW  WORDPTR FLAG               ; Stores status in memory
      FWAIT
      MOV AX, WORD PTR FLAG
      SAHF
      JZ EQUAL
      JC IMAG
      DISPLAY-MSGMSG-REAL
      JMP OVER
EQUAL:
      DISPLAY-MSGMSG-EQUAL
      JMP OVER
IMAG:
      DISPLAY-MSGMSG-IMAG
```

```
OVER:
      MOV AH, 4CH
      MOV AL, 00H
      INT 21H
      ENDS
```

5.12 SYSTEM CALLS—BIOS SERVICES

The BIOS program is always located in a special reserved memory area, the upper 64 KB of the system area (addresses F0000H–FFFFFH). On system startup, the BIOS places addresses into the IVT. When DOS or an application wants to use a BIOS routine, it generates a software interrupt. On processing the interrupt, the IVT value in the table provides the jump address to the BIOS routine. BIOS functions contain two types of routines:

(i) Test (post) and initialization routines
(ii) Control routine for I/O operations

BIOS functions available to the user can be activated by the instruction INT XX, which generates a software interrupt of the type specified in the instruction that depends upon the desired I/O operation through low-level access to hardware resources. Table 5.10 depicts some of the interrupt services used normally.

Table 5.10 Normally used interrupt services

Int. No.	Function	Purpose
05H	Print screen	Print page (video)
10H	Video services	• Set/get mode • Read/write pixel; write string • Read page; set color • Write TTY mode
11H	M/C configuration	System information
12H	Memory information	Size (KB)
13H	Disk I/O	Function code: AH Drive code: DL
14H	Serial I/O	• Initialize • Send/receive or status
15H	APM	
16H	Keyboard services	• Read/status/flag INT
17H	Printer I/O	Printer status
19H	Warm reboot	• Avoid POST • Reset
1AH	Date/Time services	Time/date/day/alarm

Each interrupt is associated with number of sub-functions, which can be specified by loading its corresponding number in (AH) register. Depending on the complexity

of a function a series of parameters can be specified by following its pattern and placing values in GPR or data structures specified by the vendor. In this section, we will study some of the basic I/O operations using these BIOS calls.

5.12.1 Print Screen Service: INT 05H

This service is used to print all the printable characters present on the screen either in text or graphics mode. This system call does not return the status in any register but it is stored in the form of a code at the reserved location [0500:0000]. The content of this location may be [00], [01], or [FF], and indicates whether the print screen operation was successful, disabled, or has an error been encountered respectively.

Example 5.6

Write a program segment to perform print screen operation and display one of the following messages: 'success', 'disabled', 'error encountered during print screen operation' BIOS service INT 05H.

Solution:

```
; PROGRAM SEGMENT TO CHECK PRINT SCREEN STATUS
PRINT_SCREEN_MSGMACRO MSG
   MOV AH, 09H
   MOV DX, OFFSET MSG
   INT 21H
   ENDM
DATA_SEG
   CREQUODH
   LFEQUOAH
DOS_SEG_ADDREQU 0050H
MSG 0 DB 'PRINT SCREEN STATUS', '$'
   MSG 1 DBCR, LF, 'PRINT SCREEN OPERATION SUCCESSFUL' '$'
   MSG 2 DBCR, LF, 'PRINT SCREEN IS ALREADY IN PROGRESS'
        DB 'SCREEN IS DISABLED', '$'
   MSG 3 DBCR, LF, 'ERROR ENCOUNTERED DURING PRINT SCREEN, '$'
   PASSDB ?
   ENDS
CODE_SEG
START:
   MOV AX, _DATA_SEG
   MOV DS, AX
   MOV AX, DOS_SEG_ADDR
   MOVES, AX
   MOV SI, 0000H
   MOV AL, BYTE PTRES:[SI]
   MOV PASS, AL
```

```
CMP AL, 00H
JNENEXT_1
PRINT_SCREEN_MSGMSG1
JMP SKIP
 NEXT_1:
CMP AL, 01H
JNE NEXT-2
PRINT_SCREEN_MSGMSG2
JMP SKIP
 NEXT_2:
CMP AL, 0FFH
JNE NEXT_3
PRINT_SCREEN_MSGMSG3
 NEXT_3:
MOV AH, 4CH
MOV AL, 00H
INT 21H
ENDS
END          START
```

5.12.2 Video Services: INT 10H

A collection of video BIOS functions are termed as video services. They allow more control over the video display and lower execution time than DOS functions.

It is necessary to detect the cursor position before using the video screen so that it can be cleared and started at any desired location. The cursor position assumes that the left-hand page column is column 0 progressing across a line to column 79. The row number corresponds to the character line number on the screen. Row 0 is the uppermost line whereas row 24 is the last line on the screen. For text mode, the video adapter defines 80 characters per line by 25 lines.

Table 5.10 gives a summary of the function codes and their purposes. The page number is often ignored after a cursor read. Page zero is available in the color graphics adapter (CGA), enhanced graphics adapter (EGA), and variable graphics array (VGA) text modes of operation. These functions are provided for the use of the video terminal in either the character or the graphic mode. Normally, video RAM is used for displaying data in character or graphic modes. The use of interrupt helps programmers in avoiding calculation of video memory addresses at which every character has to be written. Video services can be called by loading the function code from Table 5.11 in AH register.

Table 5.11 Sub-function code for BIOS services—INT 10H

Function code	Description
00H	Set video mode
01H	Set cursor shape
02H	Set cursor position

(Contd)

Table 5.11 Sub-function code for BIOS services—INT 10H *(Contd)*

Function code	Description
03H	Get cursor position and shape
04H	Get light pen position
05H	Set display page
06H	Clear/scroll screen up
07H	Clear/scroll screen down
08H	Read character and attribute at cursor
09H	Write character and attribute at cursor
0AH	Write character at cursor
0BH	Set border color
0EH	Write character in TTY mode
0FH	Get video mode: Returns no. of characters/row in AH register number of bytes in a video page @word location [4CH]
13H	Write string

Example 5.7

Write a program segment to clear the screen using the BIOS service INT 10H.

Solution:

In the text mode, two bytes are stored in video RAM for one character as per the format shown in Fig. 5.7.

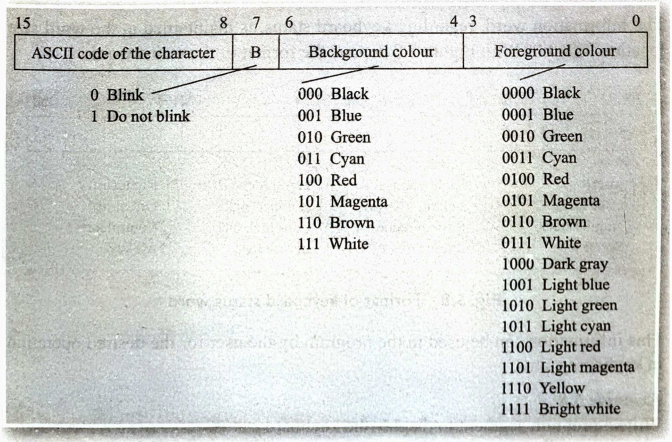


Fig. 5.7 Character storage format in text mode

Algorithm:

1. Divide the number of bytes by two to find the number of characters in the page.
2. Obtain the number of rows by dividing the number of characters in a page by the number of characters per row.
3. Set the cursor at the beginning of row 0.
   (function: 9: 10H → write a dummy character with black color on black background. This will make the row blank.)
4. Repeat the procedure for all the rows till the entire screen is blanked.

The code development based on this algorithm is left as an exercise for the reader.

5.12.3 Keyboard Services: INT 16H

The keyboard unit contains a processor, which is programmed to carry out housekeeping operation such as scan, key press, key release, and identification. It maintains a buffer, and transmits each keystroke serially to the PC's system unit. There are two bidirectional data lines in the cable connecting the keyboard unit to the system unit. Data are transmitted serially at 10KBPs along with the baud rate clock. In the system unit, the character received is reconverted to parallel format, gated into port 60H, and a interrupt is sent on IRQ1 to the interrupt controller, which triggers INT 9 for IRQ1.

KBD_INT, the name given to the default BIOS keyboard INT 9 handler, reads the scan code from port 60h and carries out following operations:

(i) Sends clear signal and re-enables the handshaking signal to the keyboard unit
(ii) Processes the scan code
(iii) Sends an end-of-interrupt (EOI) to the interrupt controller to port 20h
(iv) Returns from the interrupt

The information word, regarding keyboard status, is maintained at the word size location [0040:0017h]. Figure 5.8 depicts the format of the status word.

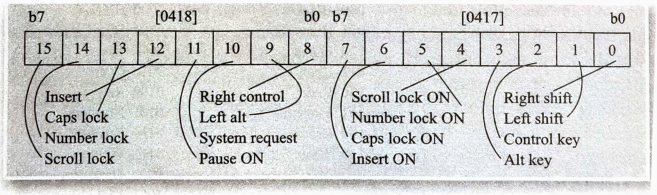


Fig. 5.8 Format of keyboard status word

This information can be used in the program by the user for the desired operation in keyboard status.

Example 5.8

Write a program segment to display the keyboard status.

Solution:

```
PRINT_STRING   MACRO  MSG
PUSH    AX
PUSH    DX
MOV     AH,     09H
MOV     DX,     OFFSET MSG
INT 21H
POP DX
POP AX
ENDM
DATA
CR      EQU     0DH
LF      QU      0AH
MSG     DB      'Keyboard status is as follows.'
MSGB0   DB      'Right shift key - pressed.'
MSGB1   DB      'Left shift key - pressed.'
MSGB2   DB      'Ctrl key - pressed.'
MSGB3   DB      'Alt key - pressed.'
MSGB4   DB      'Scroll Lock ON.'
MSGB5   DB      'Num Lock ON.'
MSGB6   DB      'Caps Lock ON.'
MSGB7   DB      'Insert ON.'
ends

CODE
STARTUP
MAIN:
MOV AX, _DATA
MOV DS, AX
PRINT_STRING MSG0
MOV AH, 02H             ; Gets keyboard flags
INT 16H                 ; Calls BIOS keyboard driver, ax-contains
                          keyboard status

TEST AL, 01H            ; Masks b0 of al
JZ SKIP1
PRINT_STRING MSGB0
SKIP1:
TEST AL, 02H            ; Checks Left Shift key
JZ SKIP2
PRINT_STRING MSGB1
SKIP2:
TEST AL, 04H            ; Checks Ctrl key status
JZ SKIP3
PRINT_STRING MSGB2
SKIP3:
```

```
        TEST AL, 08H            ; Checks Alt key status
        JZ SKIP4
        PRINTSTRING MSGB3
SKIP4:
        TEST AL, 10H            ; Checks Scroll key status
        JZ SKIP5
        PRINTSTRING MSGB4
SKIP5:
        TEST AL, 20H            ; Checks Num Lock key status
        JZ SKIP6
        PRINTSTRING MSGB5
SKIP6:
        TEST AL, 40H            ; Checks Caps Lock key status
        JZ SKIP7
        PRINTSTRING MSGB6
SKIP7:
        TEST AL, 80H            ; Checks Insert key status
        JZEXIT_PROGRAM
        PRINTSTRING MSGB7
EXIT_PROGRAM:
        MOV AH, 4CH
        MOV AL, 00H
        INT 21H
        ENDMAIN
```

Example 5.9

Write a program segment to enter a random number when any key is pressed.

Solution:

```
START:                  MOV AH, 1           ; Is a key available?
                        INT 16H             ; Issue interrupt command
                        JZ INITIALIZE_NUMBER
                        MOV AH, 0
                        INT 16H
INITIALIZE_NUMBER:      MOV CX, 0           ; Initialize "random" number
PROCESS:                INC CX
                        MOV AH,1            ; See if a key is available yet?
                        INT 16H
                        JZPROCESS
                        XOR CL, CH          ; Randomize the content
                        MOV AH,0            ; Read character from buffer
                        INT 16H
```

5.12.4 Printer Services: INT 17H

It is necessary to initialize the printer port, write characters, or read the status of the printer. The status of the printer is returned by the handler in the format shown in Fig. 5.9.

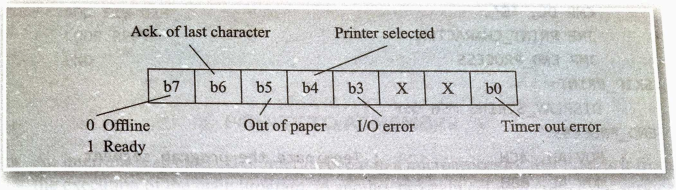


Fig. 5.9 Printer status format

The printer number [00–02] is loaded in DX and the sub-functions [00], [01], and [02h] can be used.

Example 5.10

Write a program segment to check whether the printer is online.

Solution:

```
        DISPLAY_STRING          MACRO   MSG
        MOV AH, 09H
        MOV DX, OFFSETMSG
        INT 21H
        ENDM
        DATA
        MSG_ON    DB      CR, LF,'ON_LINE'CR,LF,'$'
        MSG_OFF   DB      CR, LF,'OFF_LINE','$'
        STATUS_INFO       DB ?
        ENDS
        CODE
        STARTUP
MAIN:
        MOV AX, _DATA
        MOV DS, AXMSG 0
        MOV AH, 02H       ; Sub-function code for reading the status.
        MOV DX, LPT
        INT 17H           ; Issue BIOS command for printer services.
        MOV STATUS_INFO, AH
        MOV AL, STATUS_INFO
        AND AL, 10H
        CMP AL, 00
        JE SKIP_PRINT_CHARACTER
        DISPLAY_STRING  MSG_ON
        MOV SI, OFFSET  MSG_OFF
Print_character:
```

```
      CMP DL, '$'
      JNE PRINT_CHARACTER
      JMP END_PROCESS
SKIP_PRINT:
      DISPLAY_STRING MSG_OFF
END_PROCESS:
      MOV AH, 4CH              ; Terminate the program segment.
      MOV AL, 00H
      INT 21H
      ENDS
```

Example 5.11

Write a program segment to check and display the status of a printer.

Solution:

```
      MODEL SMALL
      DATA
      STATUS_MSG       DB 'PRINTER STATUS XX',0DH,0AH,'$'
      B0               DB 'TIME-OUT ERROR$'
      B1               DB 'RESERVED$'
      B2               DB 'RESERVED$'
      B3               DB 'I/O ERROR$'
      B4               DB 'PRINTER SELECTED$'
      B5               DB 'OUT OF PAPER$'
      B6               DB 'ACKNOWLEDGE$'
      B7               DB 'PRINTER NOT BUSY$'
      MESSAGE_STRING   DW B0, B1, B2, B3, B4, B5, B6, B7
      CODE
      STARTUP
      LEA DX, STATUS_MSG
      MOV AH, 09H
      INT 21H
      MOV AH, 02H
      MOV DX, 00H
      INT 17H
      MOV CX, 08H
      LEASI, MESSAGE_STRING
REPT:
      SHR AH, 01H
      PUSH AX
      JNC SKIP
      MOV DX, [SI]
      MOV AH, 09H
      INT 21H
SKIP:
      POP AX
```

```
ADD SI, 02H
LOOP REPT
END
```

POINTS TO REMEMBER

- An interrupt is an external or internal event in a microprocessor that diverts it from the execution of the main program, to another program called the interrupt service routine (ISR).
- The interrupt can be either a hardware interrupt or a software interrupt. The 8086 has two hardware interrupts—NMI and INTR. The software interrupt is created in the 8086 using the INT instruction.
- There are 256 interrupt types available in the 8086 and the interrupt vector for each type, which is four bytes long, is stored in an interrupt vector table (IVT) from address 00000H in the memory.
- Whenever an interrupt is received, the 8086 saves the current value of IP, CS, and the flag register in the stack, clears TF and IF, and loads CS and IP with the interrupt vector corresponding to the received interrupt type. This causes the 8086 to start the execution of the ISR.
- The IRET instruction at the end of the ISR makes the 8086 return to the main program.
- There exist different levels of priority among the interrupts, and if two interrupts appear simultaneously in the 8086, the interrupt having higher priority is serviced first.
- BIOS function calls (also called BIOS interrupts) are stored in the system ROM and the video BIOS ROM present in the PC. These BIOS function calls directly control the I/O devices with/without the DOS (disk operating system) loaded in the system.
- Interrupts can be invoked using the assembly language instruction INT XX. Each interrupt is associated with a number of sub-functions, which can be specified by loading the corresponding number in the AH register.
- Depending on the complexity of a function, a series of parameters can be specified by following its pattern and placing values in GPR/data structures specified by the vendor.
- BIOS functions contain two types of routines—test (post) initialization routines and control routine for I/O operations.
- DOS services are used to accept data from the input devices and display data on the video terminals. INT 21H is used for these I/O operations.
- BIOS program is always located in a special reserved memory area, the upper 64 KB of the system area (addresses F0000H–FFFFFH).
- The zero page is available in the colour graphics adapter (CGA), enhanced graphics adapter (EGA), and variable graphics array (VGA) text modes of operation.

KEY TERMS

Hardware interrupt It is an interrupt generated by activating the interrupt pin of the microprocessor.

Interrupt vector It is a four-byte entry in the IVT, which contains a 16-bit offset

part and a 16-bit segment part that are loaded in the IP and CS registers, respectively, when an interrupt is received.

Interrupt vector table (IVT) It is a table in the memory that contains the interrupt vectors of the different interrupts.

INTR It is a maskable hardware interrupt in the 8086 that can be enabled/disabled using the I flag.

Non-maskable interrupt (NMI) It is an interrupt that cannot be disabled by software.

Software interrupt It is an interrupt generated by the execution of the software interrupt instruction in the microprocessor.

Trap interrupt It is used for performing single-step operations in the 8086 and can be enabled/disabled using the T flag.

REVIEW QUESTIONS

1. What is the function of an interrupt in a microprocessor?
2. What is the difference between maskable and non-maskable interrupts?
3. What is the difference between hardware and software interrupts?
4. How many interrupt types are present in the 8086 and how they are classified?
5. Name the dedicated interrupts in the 8086 along with their functions.
6. What are the differences between INTR and NMI interrupts in the 8086?
7. How does the 8086 recognize an NMI interrupt?
8. What is the function of the T and I flags in the 8086 and how can they be set/reset?
9. Write the sequence of steps performed by the 8086 when it receives an interrupt other than INTR.
10. How does the 8086 return to the main program after completing the ISR of an interrupt?
11. What is an interrupt vector? What is the maximum number of interrupt vectors that can be stored in the IVT of the 8086?
12. How is a software interrupt generated in the 8086?
13. What is the function of the INTO instruction?
14. What are the advantages of software interrupts?
15. Write the priority among the interrupts in the 8086.
16. Explain the interrupt structure of the 8086 in detail.
17. With the necessary timing diagram, explain the processing of the INTR interrupt by the 8086.
18. Draw the diagram showing the supply of the interrupt type 80H through an 8-bit DIP switch and the 74244 IC, when the 8086 receives the INTR interrupt.
19. List the BIOS interrupts used to select the video mode and cursor control in the computer monitor.
20. What are the BIOS interrupts used to control the keyboard and the COM port?
21. Enlist the advantages of modular programming.
22. What is an interrupt handler?
23. Explain the term video services. Enlist and explain the video services provided by BIOS.
24. Differentiate between DOS and BIOS services.
25. Discuss techniques for developing programs to handle operations of I/O devices.

PROGRAMMING EXERCISES

1. Write an 8086 ISR to add the byte type data stored in an array starting at the address 2000H: 5000H in the memory with the corresponding data in another array stored in the memory starting at the address 3000H: 5000H and store the result in another array in the memory starting at the address 4000H: 5000H, when the NMI interrupt is given to the 8086. The number of byte type data in the array is 100. Assume that the result after addition of all the data in the array is an 8-bit data. The ISR must be accessible by any module.
2. Write an 8086 ISR to send the byte type data stored in the address 6000H: 5000H in the memory, to port A in the 8255, whose address is FF00H, when the IRQ2 interrupt in the I/O channel of the PC is activated.
3. Write an 8086 ISR to receive byte type data through port B of the 8255, whose address is FF01H. Store the data in the address 7000H: 5000H in the memory, when the software interrupt INT 0BH is executed by the PC.
4. Write a program to display the keyboard status.
5. Write a program to determine the status of a printer.
6. Write a program segment to accept a string consisting of digit and non-digit characters from the keyboard and display the sum of the digits present in the input string.

THINK AND ANSWER

1. For what purpose is the NMI interrupt commonly used in an 8086-based system?
2. What is the minimum duration for which the INTR signal must be kept high for being recognized by the 8086?
3. Is it possible to store the IVT starting from the address 20000H in the memory of the 8086? Why?
4. If the ISR of interrupt type 0 is stored from the memory address 2000: 3000H, what is the segment and offset part of the interrupt vector?
5. If the ISR of interrupt type 40H is stored from the memory address 8000: 4500H, what is the segment and offset part of the interrupt vector?
6. Is it possible to enable the INTR and the trap interrupts again when the 8086 starts executing the ISR of an interrupt? How?
7. How does the 8086 obtain the specific interrupt type when it receives the INTR interrupt?
8. If the interrupt type allotted for the interrupt IR0 is 70H in the 8259, what is the interrupt type allotted for IR2 and IR4?
9. How can the breakpoint technique for debugging a program be implemented in the 8086?
10. Is it possible to access the divide-by-0 ISR by using a software interrupt in the 8086? How?
11. If both INTR and NMI occur simultaneously in the 8086, which interrupt is processed first? Why?
12. 'It is necessary to initialize a stack before using procedures.' Comment on the validity of this statement.

Memory and I/O Interfacing

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Physical memory organization in the 8086
- Generation of separate address and data buses in the 8086
- Interfacing RAM and EPROM chips with the 8086
- Difference between I/O-mapped and memory-mapped I/O
- Interfacing I/O devices with the 8086

6.1 PHYSICAL MEMORY ORGANIZATION IN 8086

Since the 8086 has 20 address lines, it can access 1 MB (= 2^{20} bytes) of memory. The memory addresses in the 1 MB memory range from 00000H to FFFFFH. The memory is constructed using RAM and ROM/EPROM chips. The 1 MB memory in the 8086 is physically organized as an odd bank and an even bank, where each of the 512 KB (= 1 MB/2) is addressed in parallel by the 8086. Each memory location stores one byte of data. The byte data at an even memory address is transferred through the 8-bit data bus D7–D0, while the byte data at an odd memory address is transferred through the 8-bit data bus D15–D8. The 8086 provides two enable signals, $\overline{BHE}$ and A0, for the selection of odd banks and even banks, respectively. Figure 6.1 shows the physical memory organization in the 8086. The 8086 is a 16-bit processor and hence it can transfer two bytes of data in one memory read/write cycle (or I/O read/write cycle).

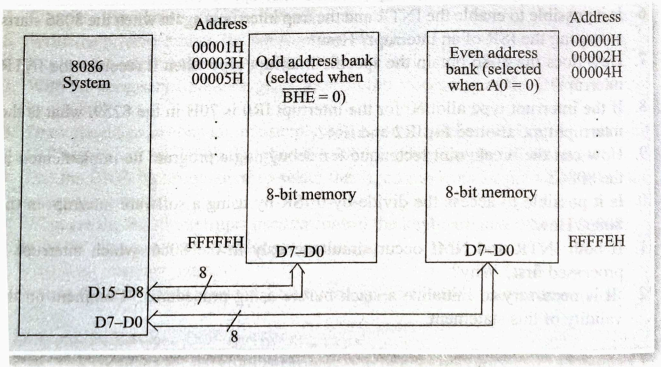


Fig. 6.1 Physical memory organization in the 8086

Two memory locations are needed to store a word in the memory in an 8086 system. While reading or writing word data (16 bits), the bus interface unit of the 8086 requires one or two memory cycles, depending upon whether the lower-order byte of the word is located at an even or odd memory address, respectively. It is better to store the word type data in the memory such that its lower-order byte is stored at an even memory address, since only one read cycle is required to read the data through the 16-bit data bus (D15–D0) of the 8086. If the lower-order byte of the word is located at an odd memory address, the first read cycle is required for accessing the lower-order byte of the word through the higher-order data bus (D15–D8), and the second is required for accessing the higher-order byte of the word through the lower-order data bus (D7–D0). Thus, two bus cycles are required to access a word whose lower-order byte is stored in an odd memory address in the memory. While initializing data structures such as an array of word type data or a stack, they should be initialized at an even address for efficient operation. This is also applicable to the memory write operation.

The use of the $\overline{BHE}$ and A0 signals to fetch data or instruction from the memory and to write data in the memory is given in Table 6.1.

Table 6.1 Function of $\overline{BHE}$ and A0 signals

$\overline{BHE}$	A0	Operation
0	0	16-bit data is read from or written into the memory.
0	1	8-bit data is read from or written into the odd memory bank.
1	0	8-bit data is read from or written into the even memory bank.
1	1	Memory is not accessed.

The $\overline{BHE}$ and A0 signals, along with a few higher-order address lines of the 8086, are used to generate the Chip Select ($\overline{CS}$) or Chip Enable ($\overline{CE}$) signal for different memory chips.

6.2 FORMATION OF SYSTEM BUS

The 8086 has a multiplexed 16-bit address/data bus (AD15–AD0) and a multiplexed 4-bit address/status bus (A19/S6–A16/S3). The multiplexed address bus can be split into a separate address bus and data bus/status bus, using the Address Latch Enable (ALE) signal of the 8086 and three external octal latches (IC 74373). Figure 6.2 shows the de-multiplexing of the address bus and the data bus using the 74373 ICs.

The data bus can be buffered using two bidirectional buffers (74245). Since the data can flow in either direction (i.e., from and into the microprocessor) while accessing the memory or I/O devices, the bidirectional buffers are used for deriving the data bus. The signals $\overline{DEN}$ and DT/$\overline{R}$ indicate the presence of data on the bus and the direction of the data (i.e., from/to the microprocessor), respectively. They are connected to the chip enable and direction pins of the buffers, as shown in Fig. 6.3.

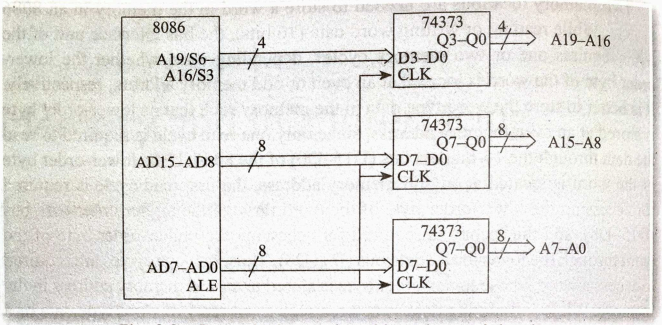


Fig. 6.2 De-multiplexing the address bus and data bus

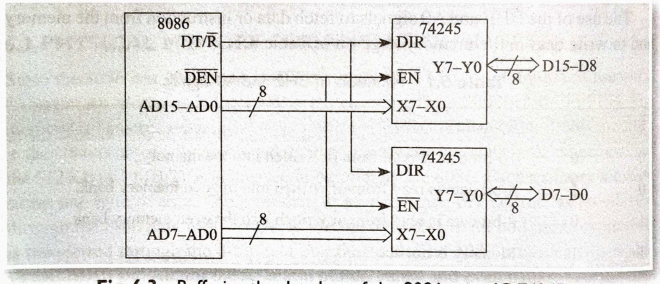


Fig. 6.3 Buffering the data bus of the 8086 using IC 74245

If $\overline{DEN}$ is low, it indicates that the data is available on the multiplexed address/data bus (AD0–AD15). Both the bidirectional buffers (74245s) are enabled to transfer that data since their enable inputs are activated at that time. When the DIR pin goes high, the data available at the X pins of the 74245 are transferred to the Y pins, i.e., data is transmitted from the 8086 to either the memory or the I/O device (write operation). If the DIR pin goes low, the data available at the Y pins of the 74245 are transferred to the X pins, i.e., data is received by the microprocessor from the memory or the I/O device (read operation). For generating the Memory Read ($\overline{MEMR}$) and Memory Write ($\overline{MEMW}$) control signals, the $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ signals of the 8086 are used along with the combinational circuit (as shown in Fig. 6.4) during the minimum mode operation of the 8086. In the case of maximum mode operation of the 8086, a bus controller chip (8288) derives all the memory control signals using the status signals $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$. Section 9.4 in chapter 9 gives the complete details of the 8086 bus timings, such as memory read/write operations, I/O read/write operations, etc., in minimum and maximum mode operation.

Certain locations in the memory are reserved for specific CPU operations. After resetting the 8086, CS and IP are initialized to FFFFH and 0000H,

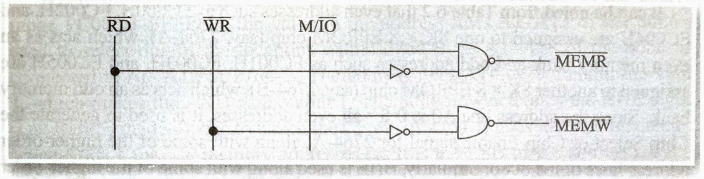


Fig. 6.4 Generation of control signals for the memory in the 8086

respectively; the first instruction for execution is taken from the address FFFF0H in the memory. Hence, the locations from FFFF0H to FFFFFH in the memory are reserved for storing instructions, execution of which causes the 8086 to jump to the initialization program of the system. The memory locations 00000H–003FFH are reserved for the interrupt vector table. These memory locations are assigned to the ROM/EPROM chips in an 8086-based system, so that the programs stored in them are permanent. The interrupt vector table may be located in the RAM chips in some systems. The memory chips can be interfaced with the 8086 using only logic gates, or using both logic gates and the decoder IC 74138. This is explained in Sections 6.3 and 6.4.

6.3 INTERFACING RAM AND EPROM CHIPS USING ONLY LOGIC GATES

When RAM and ROM/EPROM chips with the same or different storage capacities have to be interfaced with the 8086, it can be easily done using logic gates. The following example illustrates this concept.

Example 6.1

Interface two 8K × 8 EPROMs (2764) and two 8K × 8 RAM chips (6264) with the 8086 using logic gates, such that the memory address ranges assigned to them are FC000H–FFFFFH and 00000H–03FFFH, respectively.

Solution:

First, let us see the interfacing of the two 8K × 8 EPROM chips with the 8086, so that they have the address range FC000H–FFFFFH. The addresses FC000H–FFFFFH are given in binary form in Table 6.2.

Table 6.2 Memory addresses assigned to the EPROM chips

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Address
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FC000H
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	FC001H
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	FC002H
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	FC003H
.		.			.					.								.	.	
.		.			.					.								.	.	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH

It can be noted from Table 6.2 that even addresses such as FC000H, FC002H, and FC004H are assigned to one 8K × 8 EPROM chip (say, 2764-A), which acts as an even memory bank and odd addresses such as FC001H, FC003H, and FC005H are assigned to another 8K × 8 EPROM chip (say, 2764-B), which acts as an odd memory bank. Since the address line A0 is 0 for all even addresses, it is used to generate the Chip Select or Chip Enable signal for 2764-A, along with some of the higher-order address lines of the 8086. Similarly, $\overline{BHE}$ is used along with some of the higher-order address lines of the 8086 to select the odd memory bank formed by 2764-B.

First, the number of address lines in the 8K × 8 EPROM chip is noted, which is 13 (A12–A0) since $2^{13} = 8K$. The address lines A1–A13 of the 8086 are connected to the address lines A0–A12 of 2764-A and 2764-B, since the address line A0 of the 8086 is used for selecting the even memory bank. The remaining address lines A19–A14 of the 8086 are used for address decoding. Figure 6.5 shows the interfacing of two EPROM chips with the 8086.

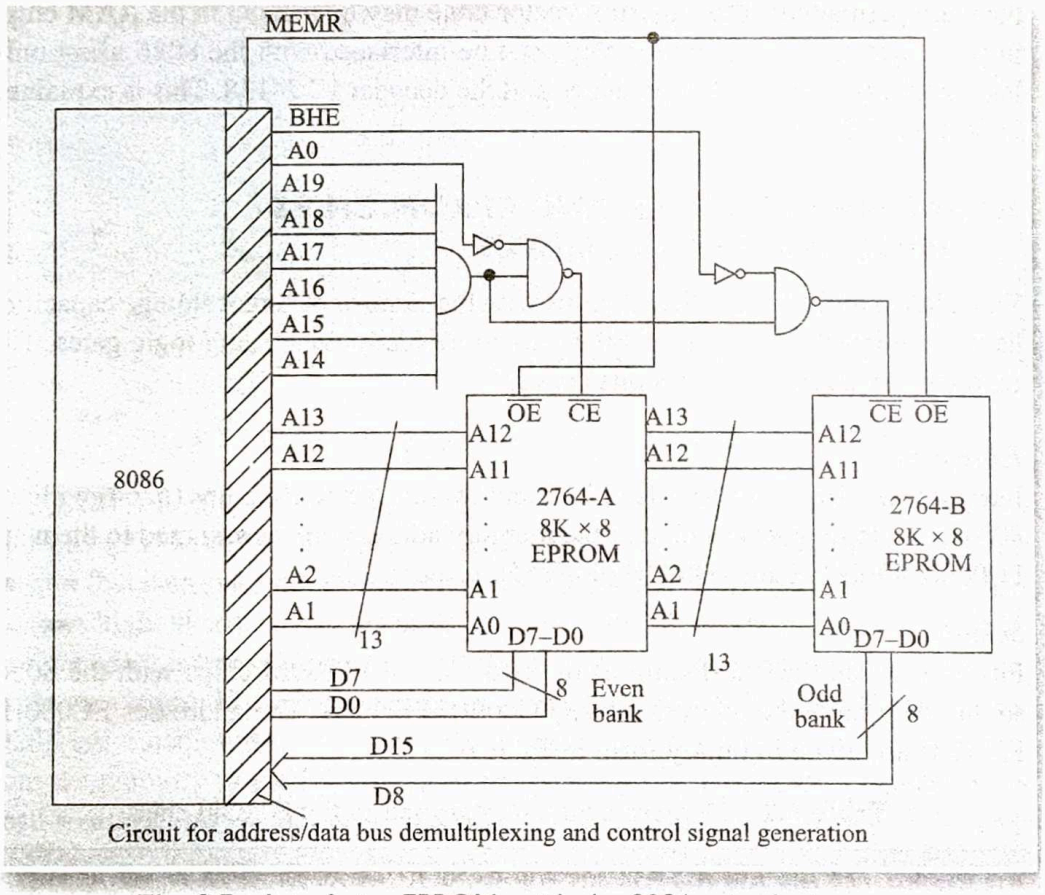


Fig. 6.5 Interfacing EPROMs with the 8086 using logic gates

Since all the address lines A14–A19 are 1 for the addresses FC000H–FFFFFH, these address lines are directly connected to an AND gate to produce the output '1'. The AND gate output and the inverted A0 signal are given to a NAND gate and the output of this NAND gate is connected to the chip enable pin of 2764-A, which is the even memory bank. Similarly, the same AND gate output and the inverted $\overline{BHE}$ signal are given to another NAND gate, whose output is used to select the 2764-B chip, which is the odd memory bank.

When the 8086 wants to access a byte from any odd address in the address range FC000H–FFFFFH, the value in the address lines A1–A13 of the 8086 is used to select one of the locations within 2764-B, as A1–A13 of the 8086 are connected to A0–A12 of 2764-B. The address lines A14–A19 contain the value 1, which makes the AND gate output 1. The 8086 now activates the $\overline{BHE}$ signal (i.e., $\overline{BHE}$ is made 0), due to which the $\overline{CE}$ pin of 2764-B goes low and is selected. Since A0 = 1 for odd memory addresses (as it is the LSB of the address), $\overline{CE}$ of 2764-A is high and is not selected.

When the 8086 wants to access a byte from any even address in the address range FC000H–FFFFFH, the values in the address lines A1–A13 and A14–A19 are used for the purposes we have just discussed. Now, the address line A0 is 0 while the 8086 sends out an even address. $\overline{BHE}$ is made 1 by the 8086. Due to this, $\overline{CE}$ of 2764-A is made low and is selected. Since $\overline{BHE}$ = 1, $\overline{CE}$ of 2764-B is in high state and is not selected. While accessing a byte from either an odd or an even memory bank, the 8086 activates $\overline{MEMR}$ after sending the memory address to get the data.

When a word (16-bit data) whose lower-order byte is stored in an even address is accessed by the 8086, both A0 and $\overline{BHE}$ are made 0, due to which both the chips are selected. One byte from each memory bank is placed in the data bus (D15–D0) when the 8086 activates the $\overline{MEMR}$ signal. The 8086 processor then reads the entire word in the data bus. For example, if the 8086 wants to read the word whose lower-order byte is stored in the address FFFFEH, all the address lines (A1–A19) of the 8086 contain 1. A0 and $\overline{BHE}$ are made 0. This makes the A0–A12 lines of both the memory chips 1 and the $\overline{CE}$ input to both the chips 0. Due to this, the data in the last memory location in both the chips are placed in their data buses, when the $\overline{MEMR}$ signal is activated by the 8086.

Now, let us discuss the interfacing of the two 8K × 8 RAM chips with the 8086, so that they have the address range 00000H–03FFFH. The addresses 00000H–03FFFH are given in binary form in Table 6.3.

Table 6.3 Memory addresses assigned to the RAM chips

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Address
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001H
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	00002H
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	00003H
.			.			.					.							.	.	.
.			.			.					.							.	.	.
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	03FFEH
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	03FFFH

The even addresses such as 00000H and 00002H are assigned to one 8K × 8 RAM chip (say, 6264-A), which acts as an even memory bank and the odd addresses

such as 00001H and 00003H are assigned to another 8K × 8 RAM chip (say, 6264-B), which acts as an odd memory bank.

As discussed in the interfacing of the EPROM chip, the inverted A0 line and the output of the address decoder formed using the AND gate are given to a NAND gate, and the NAND gate's output is connected to the $\overline{CE}$ input of 6264-A. The inverted $\overline{BHE}$ line and the same address decoder output are given to another NAND gate, and the output of that NAND gate is connected to the $\overline{CE}$ input of 6264-B. The interfacing of the RAM chips with the 8086 is shown in Fig. 6.6.

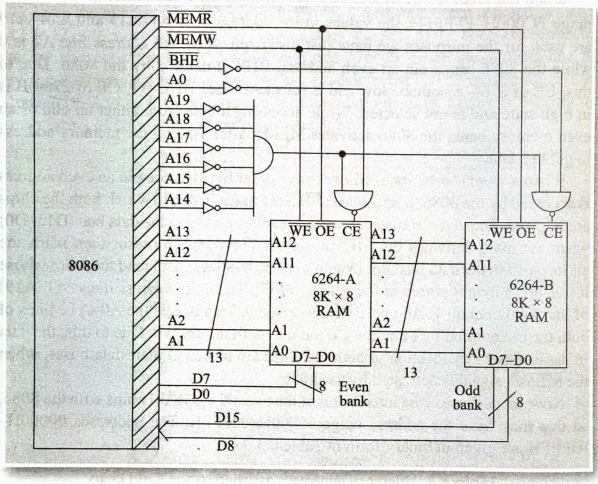


Fig. 6.6 Interfacing RAM chips with the 8086 using logic gates

Since the address lines A14–A19 contain 0 for the addresses 00000H–03FFFH, the signals in these lines are inverted and then given to the AND gate, so that they produce an output of 1 for the same addresses. This AND gate output and the inverted A0 signal through the NAND gate activate the $\overline{CE}$ input of the even memory bank. The same AND gate output and the inverted $\overline{BHE}$ signal activate the $\overline{CE}$ input of the odd memory bank. The $\overline{CE}$ input of both the memory chips are activated when the 8086 wants to access a word whose lower-order byte is stored in the even memory bank.

The $\overline{MEMR}$ signal of the 8086 is connected with the $\overline{OE}$ (Output Enable) input. The 8086 activates the $\overline{MEMR}$ signal while reading a byte or word from the RAM, after sending the address through the address bus. The $\overline{MEMW}$ signal of the 8086 is connected with the $\overline{WE}$ (Write Enable) input. The 8086 activates the $\overline{MEMW}$ signal while writing a byte or word in the RAM after sending the address through the address bus and placing the data in the data bus.

6.4 INTERFACING RAM/EPROM CHIPS USING DECODER IC AND LOGIC GATES

When RAM/EPROM chips with the same storage capacity have to be interfaced with the 8086, the interfacing can be easily done using a decoder IC and logic gates. The following examples illustrate this concept.

Example 6.2

Interface two 8K × 8 EPROM chips with the 8086, such that the memory address range assigned to them is FC000H–FFFFFH, using an address decoder made up of the 74138 IC and logic gates.

Solution:

The 13 address lines A0–A12 in the 2764 are connected to the address lines A1–A13 of the 8086. For the entire address range FC000H–FFFFFH, the value in the address lines A19–A14 is equal to 1. The address lines A19–A15 are used to enable the 74138 decoder IC, and the address lines A14, A0, and $\overline{BHE}$ are connected to the selection lines of the 74138 IC.

Figure 6.7 shows the interfacing of the EPROM chips with the 8086 chips using the 74138 decoder. For simplification, only the decoder and EPROM chips are shown in the figure. The connection of the EPROM chips with the 8086 is the same as in Example 6.1.

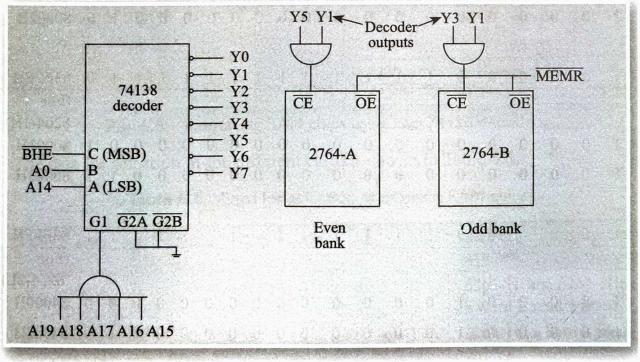


Fig. 6.7 Interfacing EPROM chips with the 8086 using 74138 decoder

When the address lines A19–A14 are 1, the decoder is enabled. The selection of a particular EPROM chip under that condition is explained in Table 6.4.

When we want to interface more RAM and EPROM chips of the same capacity with the 8086, we can use two separate decoders (74138), one for accessing the lower bank and the other for accessing the upper bank. A0 and $\overline{BHE}$ are used to enable the two decoders.

Table 6.4 Selection of EPROM chips

$\overline{BHE}$	A0	A14	Y5	Y3	Y1	$\overline{CE}$ OF 2764-A	$\overline{CE}$ OF 2764-B	Operation
0	0	1	1	1	0	0	0	A word is read from the memory.
0	1	1	1	0	1	1	0	A byte is read from the odd memory bank.
1	0	1	0	1	1	0	1	A byte is read from the even memory bank.

Example 6.3

Interface four 8K × 8 RAM chips (6264) with the 8086, to assign the address range 80000H–87FFFH using two 74138 ICs.

Solution:

The addresses assigned to various memory chips are written in binary form as shown in Table 6.5.

Table 6.5 Addresses assigned to various memory chips

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Address in hex
																				(For 6264-1L)
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80000H
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	80002H
...			...			...				...				...			...		...	...
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	83FFEH
																				(For 6264-1H)
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	80001H
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	80003H
...			...			...				...				...			...		...	...
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	83FFFH
																				(For 6264-2L)
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	84000H
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	84002H
...			...			...				...				...			...		...	...
1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	87FFEH
																				(For 6264-2H)
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	84001H
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	84003H
...			...			...				...				...			...		...	...
1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	87FFFH

Here, 6264-1L and 6264-2L are the RAM chips forming the lower banks. 6264-1H and 6264-2H are the 6264 RAM chips forming the higher banks. For simplification, only the decoder and RAM chip connections are shown in Fig. 6.8. The connection of the RAM chips with the 8086 is as explained in Example 6.1.

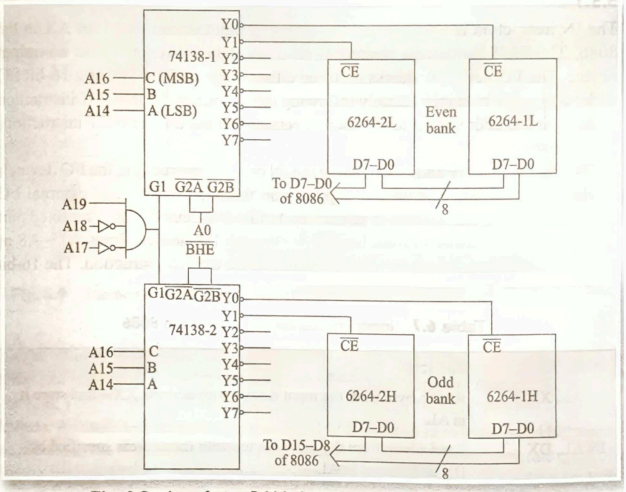


Fig. 6.8 Interfacing RAM chips using two 74138 decoders

The data for selection of the different chips is shown in Table 6.6.

Table 6.6 Data for selection of different RAM chips

A19	A18	A17	A16	A15	A14	A0	$\overline{BHE}$	RAM chips and byte/word selected
1	0	0	0	0	0	0	0	6264-1L and 6264-1H; a word is read/written
1	0	0	0	0	0	0	1	6264-1L; a byte is read/written
1	0	0	0	0	0	1	0	6264-1H; a byte is read/written
1	0	0	0	0	1	0	0	6264-2L and 6264-2H; a word is read/written
1	0	0	0	0	1	0	1	6264-2L; a byte is read/written
1	0	0	0	0	1	1	0	6264-2H; a byte is read/written

6.5 I/O INTERFACING

In this section, the operation of I/O instructions (IN and OUT), the concept of I/O-mapped I/O and memory-mapped I/O, and the interfacing of simple I/O devices such as DIP switches and LEDs with the 8086 are discussed.

6.5.1 I/O Instructions in 8086

The IN instruction is used to read data from an input device to AL or AX in the 8086. The OUT instruction is used to send the data in AL or AX to an output device. The I/O device's address is stored either in the register DX as a 16-bit I/O address or in the byte immediately following the opcode of the IN/OUT instruction as an 8-bit I/O address. Table 6.7 lists all versions of the IN and OUT instructions in the 8086.

Whenever data are transferred using the IN or OUT instruction, the I/O device's address, often called *port number*, appears on the address bus. The external I/O interface decodes this address to select a particular I/O device. The 8-bit fixed port number appears on the address lines A7–A0, with the address lines A15–A8 as 00H. The address lines A15–A19 are undefined for an I/O instruction. The 16-bit port number in DX appears on the address lines A15–A0.

Table 6.7 Input/output instructions in the 8086

Instruction	Operation
IN AL, XXH	Read a byte from the input device with address XXH and store it in AL.
IN AL, DX	Read a byte from the input device with the address specified by DX and store it in AL.
IN AX, XXH	Read a word from the input device with the address XXH and store it in AX.
IN AX, DX	Read a word from the input device with the address specified by DX and store it in AX.
OUT XXH, AL	Send a byte from AL to the output device with the address XXH.
OUT DX, AL	Send a byte from AL to the output device with the address specified by DX.
OUT XXH, AX	Send a word from AX to the output device with the address XXH.
OUT DX, AX	Send a word from AX to the output device with the address specified by DX.

6.5.2 I/O-mapped and Memory-mapped I/O

Similar to the 8085, there are two methods for interfacing I/O devices with the 8086—I/O-mapped I/O and memory-mapped I/O schemes. In I/O-mapped I/O scheme, the IN and OUT instructions are used to transfer data between the microprocessor and the I/O devices. In memory-mapped I/O, any instruction that references the memory can be used to transfer data.

6.5.2.1 I/O-mapped I/O

The most common I/O data transfer technique used in the Intel microprocessor-based system is I/O-mapped I/O; it is also called isolated I/O scheme. The term *isolated* indicates that the I/O locations are isolated from the memory system in a separate I/O address space. Figures 6.9 (a) and 6.9 (b) show both the isolated I/O and memory-mapped I/O address spaces for the 8086.

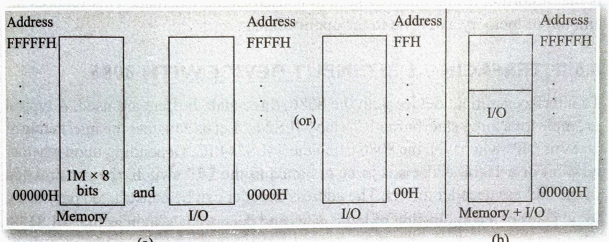


Fig. 6.9 Memory and I/O maps for the 8086 (a) I/O-mapped I/O (b) memory-mapped I/O

The address for isolated I/O devices, called *ports*, is separate from the memory in the isolated I/O scheme. As a result, the user can expand the memory to its full size (i.e., 1 MB) without using any of its address space (00000H–FFFFFH) for I/O devices. A disadvantage of I/O-mapped I/O is that the data is transferred between the 8086 and the I/O devices only by the IN and OUT instructions. Separate control signals for the I/O devices are generated, which indicate an I/O read or an I/O write operation. The generation of the $\overline{IOR}$ and $\overline{IOW}$ signals in the minimum mode operation of the 8086 is shown in Fig. 6.10. In the maximum mode operation of the 8086, the $\overline{IOWC}$ and $\overline{IORC}$ signals generated by the 8288 bus controller are used to interface the I/O devices with the 8086.

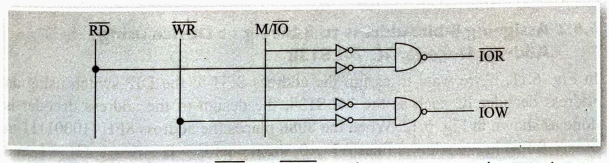


Fig. 6.10 Generation of $\overline{IOR}$ and $\overline{IOW}$ signals in minimum mode operation of the 8086

6.5.2.2 Memory-mapped I/O

The memory-mapped I/O scheme does not use the IN and OUT instructions. Any instruction that transfers data between the microprocessor and the memory can be used for transferring data between the 8086 and the I/O devices. The main

advantage of this scheme is that there are many memory transfer instructions in the 8086 and all of them can be used to access the I/O device. The same control signals used for accessing the memory ($\overline{MEMR}$ and $\overline{MEMW}$ in the minimum mode and $\overline{MRDC}$ and $\overline{MWTC}$ from the 8288 in the maximum mode) are used for accessing the I/O devices. This reduces the additional circuitry needed to generate the control signals. The main disadvantage of the memory-mapped I/O scheme is that a portion of the memory system is used as the I/O map. This reduces the amount of memory available to the applications.

6.6 INTERFACING 8-BIT INPUT DEVICE WITH 8086

To interface an input device with the 8086, three-state buffers are used. A typical example for a three-state buffer IC is the 74LS244. Let us consider the interfacing of an 8-bit DIP switch with the 8086 using the 74LS244 IC. Depending upon whether an 8-bit or a 16-bit address is to be assigned to the DIP switch, the construction of the address decoder differs. The address decoder can be constructed only using logic gates or a combination of logic gates and decoder ICs such as the 74LS138.

6.6.1 Assigning 8-bit Address to 8-bit Input Device using Address Decoder having only Logic Gates

Let us interface an 8-bit DIP switch with the 8086 operating in the minimum mode, such that the address assigned to it is 8FH, using an address decoder having only logic gates. Figure 6.11 shows the required interfacing circuitry. When the 8086 has to read the data from the 8-bit DIP switch, the instruction IN AL, 8FH or IN AL, DX with DX already loaded with the value 008FH has to be executed by it. During the execution of any one of these instructions, the address lines A7–A0 contain 8FH and the $\overline{IOR}$ signal is made low for some duration (a few μs) by the 8086. As a result, the enable inputs ($\overline{1G}$ and $\overline{2G}$) of the 74LS244 are activated (i.e., made low), and the data from the DIP switch is placed on the data bus (D15–D8). The 8086 reads that data and places it in the AL register. The data bus D7–D0 of the 8086 is used if the I/O device address is an even number. The reason for this is explained in Section 6.9.

6.6.2 Assigning 8-bit Address to 8-bit Input Device using Address Decoder IC 74LS138

In Fig. 6.11, if we want to assign the address 8FH to the DIP switch using an address decoder IC such as the 74LS138, the design of the address decoder is done as shown in Fig. 6.12. When the 8086 places the address 8FH (10001111 in binary form) in the address lines A7–A0, the inputs C = B = A = 1, G1 = 1, and $\overline{G2A} = \overline{G2B} = 0$ in the 74LS138 IC, due to which the decoder IC is enabled, its Y7 output goes low, and other outputs remain high. This Y7 output of the decoder IC along with the $\overline{IOR}$ signal of the 8086 is used to enable the 74LS244 IC, thereby transferring data from the DIP switch to the AL register of the 8086 when the instruction IN AL, 8FH is executed. The same decoder IC's other outputs (i.e., Y0–Y6) can be used to assign the addresses 88H–8EH to other I/O devices.

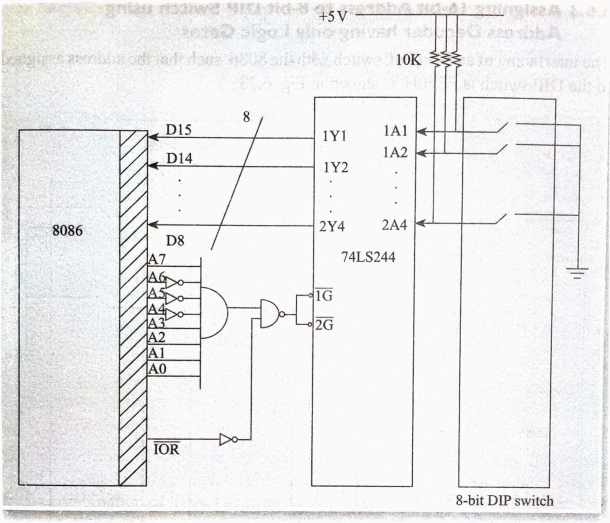


Fig. 6.11 Interfacing an 8-bit DIP switch with the 8086 (8-bit address)

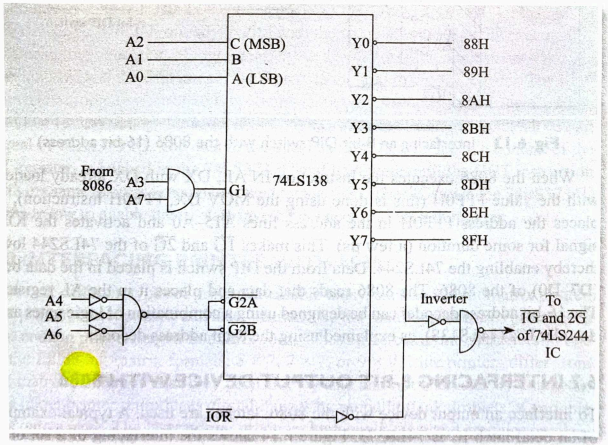


Fig. 6.12 Address decoder using 74LS138 IC

6.6.3 Assigning 16-bit Address to 8-bit DIP Switch using Address Decoder having only Logic Gates

The interfacing of an 8-bit DIP switch with the 8086, such that the address assigned to the DIP switch is FFF0H, is shown in Fig. 6.13.

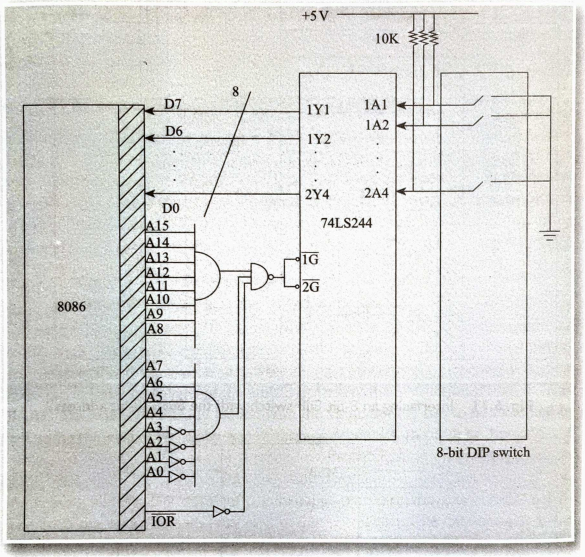


Fig. 6.13 Interfacing an 8-bit DIP switch with the 8086 (16-bit address)

When the 8086 executes the instruction IN AL, DX with DX already loaded with the value FFF0H (this is done using the MOV DX, FFF0H instruction), it places the address FFF0H in the address lines A15–A0 and activates the $\overline{IOR}$ signal for some duration (a few μs). This makes $\overline{1G}$ and $\overline{2G}$ of the 74LS244 low, thereby enabling the 74LS244. Data from the DIP switch is placed in the data bus (D7–D0) of the 8086. The 8086 reads that data and places it in the AL register. The 16-bit address decoder can be designed using a combination of logic gates and decoder ICs (74LS138), as explained using the 8-bit address decoder.

6.7 INTERFACING 8-BIT OUTPUT DEVICE WITH 8086

To interface an output device with the 8086, latches are used. A typical example of an octal latch IC is 74LS373. Figure 6.14 shows the interfacing of a set of 8 LEDs with the 8086 using the 74LS373 IC. Either an 8-bit or 16-bit address can be

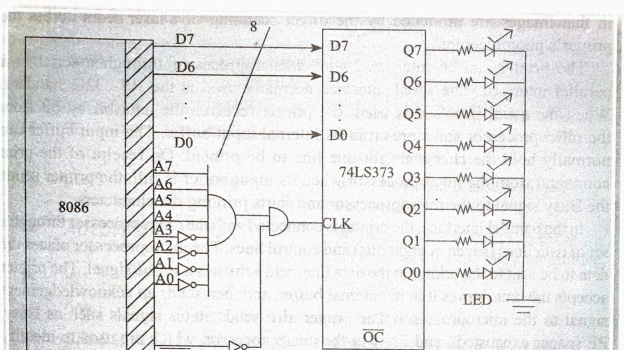


Fig. 6.14 Interfacing eight LEDs with the 8086 (8-bit address)

assigned to the set of LEDs, as explained in the interfacing of input devices with the 8086. The address decoder can be constructed either using only logic gates or using a combination of logic gates and decoder ICs such as the 74LS138.

Let us discuss the interfacing of an 8-bit output device having an 8-bit address with the 8086. In Fig. 6.14, the address assigned to the LEDs is F0H. When the 8086 has to send the data in the AL register to the LEDs, either OUT F0H, AL or OUT DX, AL with DX already loaded with the value 00F0H has to be executed by it. During the execution of any one of these instructions, the address lines A7–A0 contain F0H and the data lines D7–D0 contain the data in the AL register. The $\overline{IOW}$ signal (assuming that the 8086 is operating in minimum mode) is made low for some duration (a few μs) by the 8086. This activates (i.e., makes high) the clock (CLK) signal of 74LS373 IC. The data in the data bus D7–D0, which is the content of the AL register, is latched in the 74LS373 IC and held there until the OUT instruction with the same address is again executed by the 8086. The $\overline{OC}$ pin in the 74LS373 IC is made low to enable the tri-state inverter connected to each output pin.

6.8 INTERFACING PRINTER WITH 8086

There are different types of printers available today, such as the dot matrix printer, line printer, inkjet printer, and laser printer. The dot matrix printer uses print heads that contain pins arranged in matrix form. These pins can print characters in one of the following matrix formats: 5×7, 7×7, or 9×9. Line printers differ from dot matrix printers in that they print line by line and not character by character. The inkjet printer reproduces digital images by propelling ink droplets of variable size onto a page. The laser printer produces high quality text and graphics on plain paper, at a very high speed. Like digital photocopiers, laser printers employ a xerographic printing process. However, they differ from analog photocopiers

in that images are produced by the direct scanning of a laser beam across the printer's photoreceptor.

The printer can be interfaced with a microprocessor through a serial or a parallel interface. The serial interface normally used is the RS-232C standard. When the serial interface is used, the printer receives the data bit by bit from the microprocessor and stores it in its internal input buffer. The input buffer can normally hold the characters for one line to be printed. On receipt of the print command from the microprocessor when its input buffer is full, the printer sends the Busy signal to the microprocessor and starts printing the characters.

In the parallel interface, the printer is connected with the microprocessor through a set of data lines (seven or eight bits) and control lines. The microprocessor places the data to be sent to the printer in the data lines and activates a strobe signal. The printer accepts the data, stores it in its internal buffer, and then sends an acknowledgement signal to the microprocessor. The printer also sends status signals such as Busy, PE (paper exhausted), and Error to the microprocessor, which are used to identify the readiness of the printer for data transfer. Most of the printers accept the data in ASCII (American Standard Code for Information Interchange) form. Data transfer through a parallel interface is faster and simpler than that through a serial interface. The Centronics interface is a popular parallel interface that is used for interfacing the printer with the microprocessor and is named after the manufacturer of the Centronics printer, who introduced it. Table 6.8 shows the pin number and the description of various signals in the Centronics printer connector.

Table 6.8 Pin connections and signals in the Centronics interface

Pin no.	Signal	Direction of signal (with respect to the printer)	Description of signal
1	$\overline{\text{STROBE}}$	Input	When the $\overline{\text{STROBE}}$ signal goes low, the printer reads the data in the data lines and stores them in its internal buffer. The minimum low state (i.e., 0) duration of the $\overline{\text{STROBE}}$ signal must be 0.5 μs.
2–9	DATA1–DATA8	Input	This represents the data (eight bits) to be printed in the printer.
10	$\overline{\text{ACK}}$	Output	When the printer has received data and is ready for next data, it makes the $\overline{\text{ACK}}$ signal low (i.e., 0) for a minimum period of 5 μs.
11	Busy	Output	The Busy signal is high (i.e., 1) when the printer is unable to receive data. It is high during data entry into the printer, during printing operation, when the printer is in offline state, or when it is in error state.

(Contd)

Table 6.8 Pin connections and signals in the Centronics interface *(Contd)*

Pin no.	Signal	Direction of signal (with respect to the printer)	Description of signal
12	PE	Output	If the printer is out of paper, the PE signal goes high.
13	SLCT	Output	When the printer is in selected state, SLCT is high.
14	$\overline{\text{AUTOFEEDXT}}$	Input	When this signal is low, the paper is automatically fed one line after printing.
15	NC	–	Not used
16	0 V	–	Logic ground level
17	CHASSISGND	–	Printer chassis ground. In the printer, the chassis ground and logic ground are isolated from each other.
18	NC	–	Not used
19–30	GND	–	This is the twisted pair return (ground) signal for the $\overline{\text{STROBE}}$, DATA, $\overline{\text{ACK}}$, Busy, and PE signals.
31	$\overline{\text{INIT}}$	Input	When the $\overline{\text{INIT}}$ signal is made low (for more than 50 µs), the printer controller is reset to its initial state and the printer buffer is cleared.
32	$\overline{\text{ERROR}}$	Output	This signal goes low when the printer is in 'offline state', 'paper end state', or 'error state'.
33	GND	–	Ground
34	NC	–	Not used
35	+5 V	Output	This signal is pulled up to +5 V through a 4.7 kΩ resistor.
36	$\overline{\text{SLCTIN}}$	Input	Data entry to the printer is possible only when this signal is low.

Figure 6.15 shows the timing diagram of the important signals involved in interfacing of the Centronics printer with the microprocessor. When the Busy signal is 0, which means that the printer is ready for accepting the character from the microprocessor, the microprocessor places the ASCII code of a character or the code of a special command in the data lines (DATA1–DATA8). After a minimum time of 0.5 µs, it activates the $\overline{\text{STROBE}}$ signal (i.e., makes it 0) for a minimum period of 0.5 µs. The data in the data lines is kept at the same value for a minimum period of 0.5 µs after the $\overline{\text{STROBE}}$ signal is deactivated (i.e., made 1). When the $\overline{\text{STROBE}}$ signal is activated, the Busy signal from the printer immediately goes

high (i.e., becomes 1). It remains high until the printer sends the $\overline{ACK}$ signal, as shown in Fig. 6.15. This is done because the microprocessor should not send another data to the printer before the first data is processed in the printer. During the rising edge of the $\overline{ACK}$ signal, the Busy signal goes low (as shown in Fig. 6.15) and now, another data can be sent to the printer.

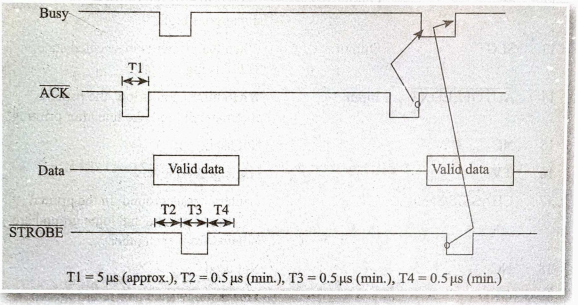


Fig. 6.15 Timing diagram of important signals in the Centronics printer interface

Table 6.9 indicates the signals that are mainly required for interfacing the Centronics printer with the microprocessor. There are totally ten output signals that have to be sent from the microprocessor to the printer and four input signals that have to be received by the microprocessor from the printer. Microprocessors such as the 8085 and the 8086 can use one 8255 IC (programmable peripheral interface) to interface a Centronics printer.

Table 6.9 Signals needed to interface the Centronics printer with the microprocessor

Signal description	Signal name	Number of signals	Input/output (with respect to the microprocessor)
Data lines	DATA1–DATA8	8	Output
Strobe	$\overline{STROBE}$	1	Output
Acknowledge	$\overline{ACK}$	1	Input
Busy	Busy	1	Input
Error	$\overline{ERROR}$	1	Input
Paper exhausted	PE	1	Input
Initialize	$\overline{INIT}$	1	Output

Figure 6.16 shows the interfacing of 8086 microprocessor with the Centronics printer using one 8255 IC. It shows the main signals involved in the data transfer. In Fig. 6.16, port A is used to send the data (eight bits) to the printer and hence it should be configured as an output port. Port B is used to send the $\overline{INIT}$ and

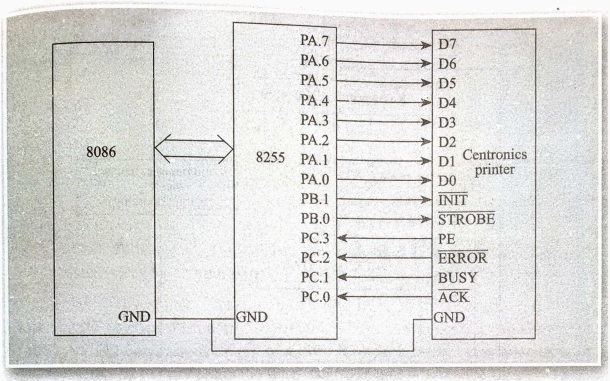


Fig. 6.16 Interfacing the Centronics printer with the 8086 using the 8255

$\overline{\text{STROBE}}$ signals to the printer and hence it should be configured as an output port. Port C is used to receive status signals such as $\overline{\text{ACK}}$, Busy, $\overline{\text{ERROR}}$, and PE from the printer and hence it should be configured as an input port.

The complete sequence of steps to be carried out in software, for the Centronics printer to print a message having several lines, is given in the flowchart shown in Fig. 6.17. The ASCII code of various characters in the message to be printed is first stored in some portion of the RAM in the microprocessor system. The microprocessor has to send the ASCII code of characters in the RAM to the printer one by one, with the line feed and carriage return characters (0AH and 0DH, respectively) as the last code. Printers are often capable of executing commands that are sent through the data lines by the microprocessor. The difference between the data and the command is achieved by means of escape (ESC) codes. Whenever the microprocessor sends an ESC code, the printer interprets the following code as a command. Such commands are needed to specify the desired font, the size of the margin, the line spacing, etc., in the message that is printed.

6.9 INTERFACING 8-BIT AND 16-BIT I/O DEVICES OR PORTS WITH 8086

Let us see how data are transferred between the 8086 and 8- or 16-bit I/O devices. Data transferred to an 8-bit I/O device or port exists in one of the I/O banks of the 8086. The I/O system contains two 8-bit I/O banks, just like the memory system of the 8086. This is shown in Fig. 6.18, which indicates the separate I/O banks for a 16-bit system. When an 8-bit address is used for I/O devices, the even bank contains even addresses such as 00H, 02H, and 04H and the odd bank contains odd addresses such as 01H, 03H, and 05H. When a 16-bit address is used for I/O devices, the even bank contains even addresses such as 0000H, 0002H, and 0004H and the odd bank contains odd addresses such as 0001H, 0003H, and 0005H.

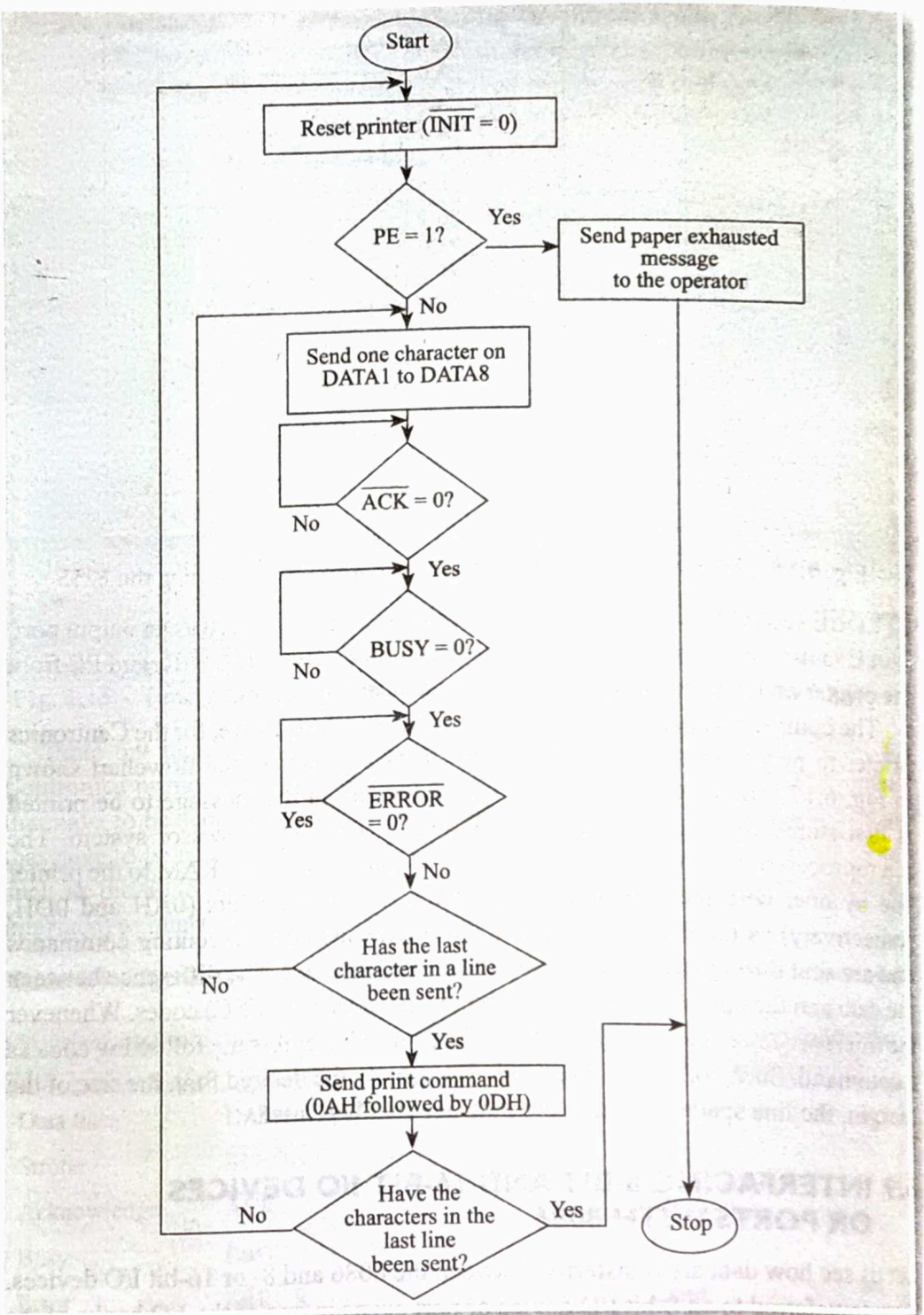


Fig. 6.17 Software sequence for interfacing the Centronics printer with the microprocessor

An 8-bit I/O device having an even address is connected to the data bus D7–D0 of the 8086, and a device having an odd address is connected to D15–D8 of the 8086. A 16-bit I/O device is connected to the data bus D15–D0 of the 8086. When address line A0 is 0, the even I/O bank is accessed and when $\overline{BHE}$ is 0, the odd I/O bank is accessed.

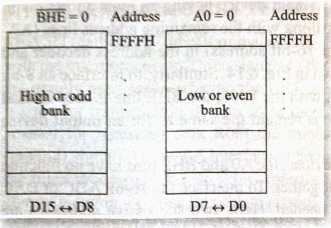


Fig. 6.18 I/O banks in an 8086-based system with 16-bit addresses

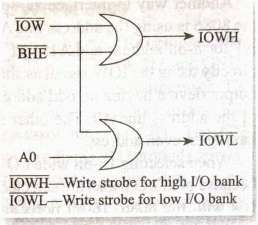


Fig. 6.19 Generation of write strobes for I/O banks

Since two I/O banks exist, any 8-bit I/O write operation requires separate write strobes to function correctly. These are generated as shown in Fig. 6.19. I/O read operations do not require separate read strobes because as with the memory, the 8086 only reads the byte it expects and ignores the other byte.

Figure 6.20 shows a system that contains two different 8-bit output devices located at the 8-bit I/O addresses F0H and F1H. Since these are 8-bit devices and appear in different I/O banks, separate I/O write signals are needed. In Fig. 6.20, the connections of only the address decoder and the 74LS373 ICs are shown. The remaining connections to the 8086 are the same as in Fig. 6.14.

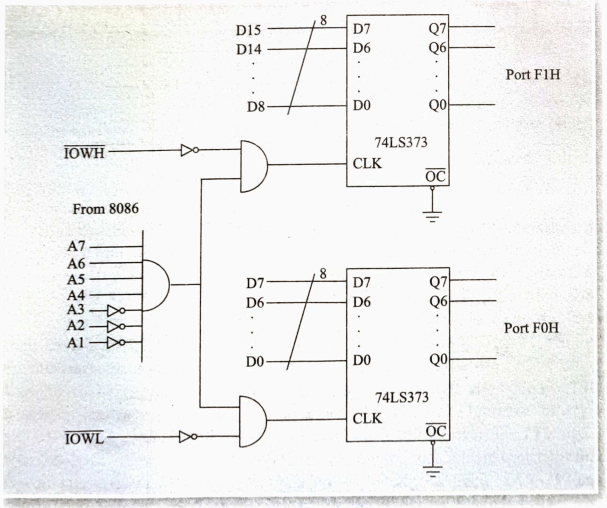


Fig. 6.20 I/O port decoder to select 8-bit output ports F0H and F1H

Another way to interface an 8-bit output device having an even address with the 8086 is using the address line A0 along with the remaining address lines (A7–A1 for 8-bit address and A15–A1 for 16-bit address) in the address decoder and directly using the $\overline{IOW}$ signal as shown in Fig. 6.14. Similarly, to interface an 8-bit output device having an odd address with the 8086, the $\overline{BHE}$ line is used instead of the address line A0. The other steps remain the same as for an output device having an even address.

When selecting 16-bit wide I/O devices, the A0 and $\overline{BHE}$ pins have no function because both I/O banks are selected together. To interface the 16-bit ADC or DAC ICs with the 8086, 16-bit ports are needed. Here, two successive addresses are assigned for the same I/O device. One address is an even number such as 00H (for 8-bit address) or 0000H (for 16-bit address), where the lower-order byte of the 16-bit data is present. The other address is an odd number such as 01H (for 8-bit address) or 0001H (for 16-bit address), where the higher-order byte of the 16-bit data is present. In the IN or OUT instruction, only the address of the lower-order byte of the 16-bit data is specified either directly or implicitly through DX. Figure 6.21 shows the interfacing of a 16-bit input device connected to function at the 8-bit I/O addresses F4H and F5H. In the figure, only the connections for the address decoder and the 74LS244 ICs are shown. The remaining connections to the 8086 are as shown in Fig. 6.11. Using the instructions IN AX, F4H or IN AX, DX with DX already loaded with the value 00F4H, the data from the 16-bit input port can be read and placed in AX.

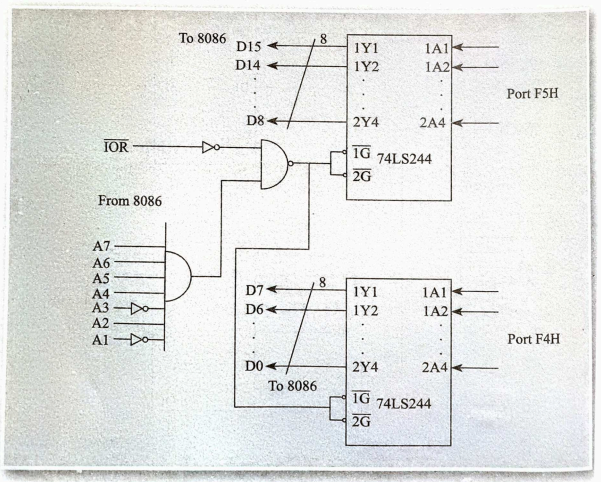


Fig. 6.21 16-bit input port decoded at I/O addresses F4H and F5H

6.10 INTERFACING CRT TERMINAL WITH 8086

The CRT (cathode ray tube) terminal uses the RS-232C interface for communication with the microprocessor. Three signals in the RS-232C—TXD, RXD, and GND—are mainly used for interfacing the CRT terminal with the microprocessor. TXD is used for transmission of data from the CRT to the microprocessor and RXD is used for receiving data from the microprocessor into the CRT. The GND (ground) signal in the CRT interface is connected to the GND (ground) signal in the microprocessor. The RS-232C interface transmits or receives data by serial communication, i.e., one bit of data is transmitted or received at a time. Each byte of data transmitted or received by the RS-232C interface is enclosed by one start bit and 1, 1.5, or 2 stop bits. Figure 6.22 shows the RS-232C format for transmission or reception of one byte of data, 4DH (which is equal to 01001101 in binary form), with one start bit and two stop bits. When no data is transmitted or received, the TXD and RXD lines remain high. In the RS-232C standard, any voltage between +3 V and +12 V in the data lines (TXD and RXD) is used to represent binary 0 and any voltage between −3 V and −12 V is used to represent binary 1. Due to this reason, the RS-232C standard is said to be using negative true logic.

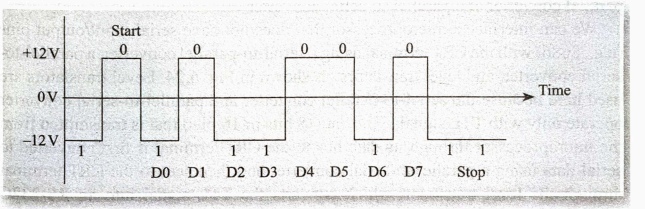


Fig. 6.22 RS-232C format for transmission or reception of a byte of data (4DH)

There are three methods by which a CRT terminal can be interfaced with the microprocessor:

(i) Direct connection of the microprocessor with the CRT terminal

A microprocessor (e.g., 8085) that has facilities for serial input/output (through its SID/SOD pins), can be directly connected to the CRT terminal through level translators. The SID pin of the 8085 is connected to the TXD pin of the CRT terminal; the SOD pin of the 8085 is connected to the RXD pin of the 8085 through level translators, as shown in Fig. 6.23. The reason for using the level translators is the mismatch in the voltage levels for representing binary 1 and 0 in the microprocessor and the CRT terminal. We already know that the CRT terminal uses the RS-232C interface. Microprocessors such as the 8085 and 8086 use TTL (transistor–transistor logic) standard, in which +5 V is used to represent binary 1 and 0 V is used to represent binary 0. The level translators convert the TTL signal to an RS-232C signal and vice versa. One example for such a level translator that is available in an integrated circuit (IC) form is MAX-232. Each MAX-232 can convert two TTL signals to the corresponding RS-232C signals and two RS-232C signals to the corresponding TTL signals.

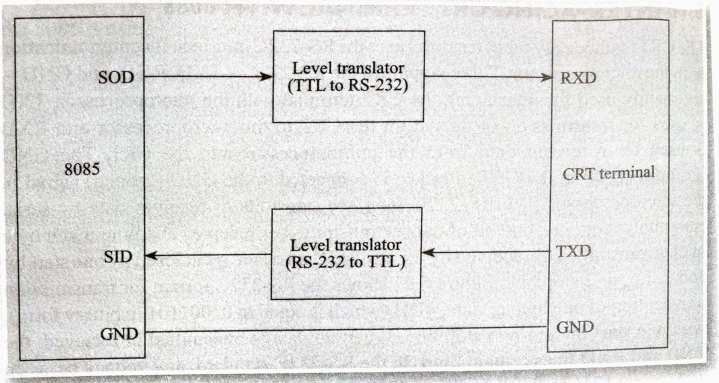
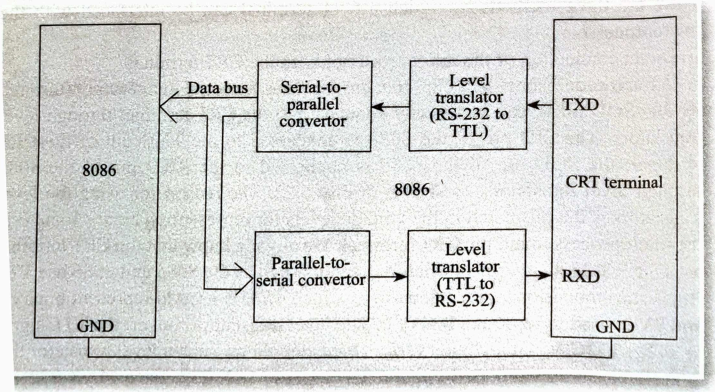


Fig. 6.23 Direct connection of the microprocessor with the CRT terminal

(ii) Connection of the microprocessor with the CRT terminal through serial-to-parallel converter and parallel-to-serial converter

We can interface a microprocessor that does not have serial input/output pins (e.g., 8086) with the CRT terminal using a serial-to-parallel converter, a parallel-to-serial converter, and level translators, as shown in Fig. 6.24. Level translators are used here because the serial-to-parallel converter and parallel-to-serial converter operate only with TTL signals. The data (8 bits or 16 bits) that is transmitted from the microprocessor through its data bus to the CRT terminal is first converted to serial data using a parallel-to-serial converter and then sent to the CRT terminal through the level translator, which converts the TTL signal into an RS-232C signal. Similarly, the serial data that is transmitted from the CRT terminal to the



ig. 6.24 Connection of the microprocessor with the CRT terminal using serial-to-parallel converter and parallel-to-serial converter

microprocessor, which is in RS-232C format, is first converted into a TTL signal using a level translator, then converted to parallel data using a serial-to-parallel converter, and sent to the microprocessor through its data bus.

(iii) Connection of the microprocessor with the CRT terminal through USART (universal synchronous asynchronous receiver–transmitter)

There exists a special IC chip such as USART (IC 8251), which has a built-in parallel-to-serial converter (eight bits) and a built-in serial-to-parallel converter (eight bits). Figure 6.25 shows the connection of the 8086 microprocessor with the CRT terminal through USART and level translators. The level translators are used here because the USART operates only with TTL signals.

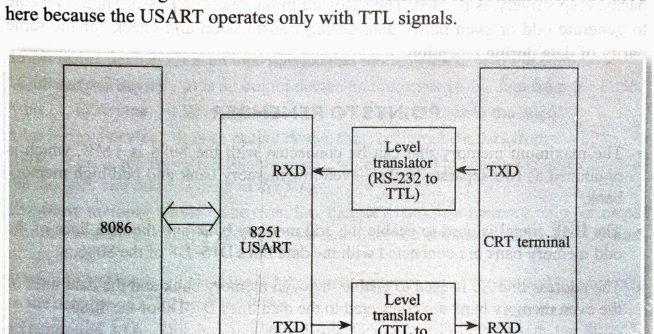


Fig. 6.25 Connection of the microprocessor with the CRT terminal using USART

The CRT terminal transmits or receives data at a fixed baud rate. Baud rate represents the number of bits transmitted or received per second. There are some standard values for baud rate, such as 600, 1200, 2400, 4800, and 9600. One of these speeds can be selected in the CRT terminal by properly configuring certain switches present in it. The microprocessor must also be programmed to the same baud rate as the CRT terminal, for proper data transfer between them. The time between transmitting or receiving two consecutive bits is known as bit time in serial communication and it is the reciprocal of the baud rate. The required bit time can be obtained using a delay program in the microprocessor. Based on the baud rate input given to the microprocessor, the delay count used in the delay program can be found using look-up table technique.

The microprocessor software that controls the data transfer between the microprocessor and the CRT terminal does the following operations sequentially:

(i) During the transmission of data from the microprocessor to the CRT terminal, the microprocessor first sends the start bit, then the data bits one by one, and finally, the stop bit(s).

(ii) During the reception of data from the CRT terminal, the microprocessor first checks whether start bit has occurred (i.e., whether RXD is made 0). If start bit is received, then the microprocessor receives the data bits one by one. Then it checks for the reception of stop bit(s).

The CRT terminal uses the parity bit along with the data, to ensure that the transmission or reception of data does not involve any error. Some terminals use odd parity and some use even parity. The number of 1s in the data is made odd or even using the seventh bit of the data, depending upon whether odd or even parity, respectively, is needed. The software in the microprocessor should be able to generate odd or even parity data during transmission and check for the same parity of data during reception.

POINTS TO REMEMBER

- The maximum memory that can be connected with the 8086 is 1 MB, which is organized as two separate banks—even/low memory bank and odd/high memory bank.
- The $\overline{\text{BHE}}$ signal is used to enable the odd memory bank and the data lines of the odd memory bank are connected with the data lines D15–D8 of the 8086.
- The address line A0 is used to enable the even memory bank and the data lines of the even memory bank are connected to the data lines D7–D0 of the 8086.
- When the lower-order byte of a word is stored in the even memory bank, the 8086 can access both bytes of that word in a single memory read cycle. Otherwise, it takes two memory read cycles to read the same word. Therefore, while storing an array of word type data in the memory or while initializing the stack, the lower-order bytes of the words are stored in the even addresses.
- There are two methods that can be used to interface I/O devices with the 8086—memory-mapped I/O and I/O-mapped I/O.
- In the memory-mapped I/O method, the I/O device is treated as if a memory location and the instructions used for transferring data between the memory and the 8086 can be used for data transfer between the 8086 and the I/O devices. The $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ signals are used to activate the input device and output device, respectively. The I/O devices have a 20-bit address in memory-mapped I/O and the design of the address decoder is same as that of the memory address decoder.
- The I/O-mapped I/O scheme is commonly used to interface I/O devices with the 8086. Here, there are two methods of addressing I/O devices—fixed port addressing (in which the 8-bit address of an I/O device is specified in the IN or OUT instruction directly) and variable port addressing (in which the 16-bit address of an I/O device is specified in the IN or OUT instruction implicitly through the DX register). In I/O-mapped I/O, only the IN and OUT instructions are used to communicate with the I/O devices. The advantage of this method is that the user can fully utilize the 1 MB memory space, which is not possible in memory-mapped I/O.
- The 8086 can be interfaced with either an 8-bit or a 16-bit I/O port. The I/O space in the 8086 is also organized as two separate I/O banks—odd and even I/O bank,

which is the same as the memory organization in the 8086. The odd I/O bank contains odd I/O addresses and the data lines of the odd I/O bank are connected to the D15–D8 lines of the 8086. The even I/O bank contains even I/O addresses and the data lines of the even I/O bank are connected to the D7–D0 lines of the 8086. The $\overline{BHE}$ signal is used to enable the odd I/O bank and A0 is used to enable the even I/O bank, which is the same as the process for enabling the memory in the 8086. The $\overline{IOR}$ and $\overline{IOW}$ signals are used to activate the input and output devices, respectively, in the I/O-mapped I/O scheme.

KEY TERMS

16-bit input device It is an input device that sends 16-bit data to the 8086.

16-bit output device It is an output device that receives 16-bit data from the 8086.

8-bit input device It is an input device that sends 8-bit data to the 8086.

8-bit output device It is an output device that receives 8-bit data from the 8086.

$\overline{BHE}$ This is the Bus High Enable signal, which is used to enable the upper bank of the memory and odd I/O bank in the 8086.

Even/low memory bank The even/low memory bank is a memory chip (or chips) that contains even memory addresses; its data lines are connected to the D7–D0 lines of the 8086.

High/odd I/O bank This is the I/O bank that contains odd addresses and is connected to the data lines D15–D8 of the 8086.

I/O-mapped I/O This is a method of interfacing an I/O device with the 8086, in which an I/O device is treated differently from the memory.

IN and OUT instructions These are the instructions used for transfer data between the accumulator and the I/O devices in I/O-mapped I/O.

$\overline{IOR}$ This is the I/O read control signal that is activated during the I/O read operation.

$\overline{IOW}$ This is the I/O write control signal that is activated during the I/O write operation.

Latch The latch is used for interfacing output device with microprocessor.

Low/even I/O bank This is the I/O bank that contains even addresses and is connected to the data lines D7–D0 of the 8086.

Memory address space or memory map The memory addresses that can be generated by the 8086 (00000H–FFFFFH) together constitute the memory map.

Memory-mapped I/O This is a method of interfacing an I/O device with the 8086, in which an I/O device is treated as if a memory location.

$\overline{MEMR}$ This is the Memory Read control signal that is activated during the memory read operation.

$\overline{MEMW}$ This is the Memory Write control signal that is activated during the memory write operation.

Odd/high memory bank The odd/high memory bank is a memory chip (or chips) that contains odd memory addresses; its data lines are connected to the D15–D8 lines of the 8086.

Physical memory address The memory address in the physical memory such as the RAM or EPROM chip is called physical memory address.

Tri-state buffer The tri-state buffer is used for interfacing the input device with the microprocessor.

REVIEW QUESTIONS

1. What is the maximum memory, in terms of bytes, that can be interfaced with the 8086? Why?
2. What is the memory address space in the 8086?
3. How is the physical memory organized in the 8086?
4. How are the A0 and $\overline{BHE}$ signals in the 8086 used in the selection of memory banks?
5. Why should the data structures such as array of word type data or stack be stored from an even address in the memory?
6. How is the multiplexed address bus in the 8086 separated into address bus and data bus? Draw the diagram for the same.
7. What are the functions of IC 74244 and IC 74245?
8. How are the Memory Read and Memory Write control signals generated in the minimum mode of operation of the 8086?
9. What is the importance of the memory address ranges 00000H–003FFH and FFFF0H–FFFFFH in the 8086?
10. What are the differences between memory-mapped I/O and I/O-mapped I/O?
11. Write the different forms of the IN instruction in the 8086.
12. Write the different forms of the OUT instruction in the 8086.
13. What is meant by fixed port addressing in the 8086 and how many I/O devices can be connected to the 8086 by this method?
14. What is meant by variable port addressing in the 8086 and how many I/O devices can be connected to the 8086 by this method?
15. Draw a diagram showing the memory and I/O map when memory-mapped I/O and I/O-mapped I/O schemes are used.
16. Draw a circuit showing the generation of I/O read and write control signals in the minimum mode operation of the 8086.

NUMERICAL/DESIGN-BASED EXERCISES

1. Interface two 16K × 8 EPROM chips with the 8086, such that the memory address range assigned to the EPROM chips is F8000H–FFFFFH, using an address decoder having only logic gates.
2. Interface two 16K × 8 RAM chips with the 8086, such that the memory address range assigned to the RAM chips is 00000H–07FFFH, using an address decoder having only logic gates.
3. Interface two 8K × 8 EPROM chips with the 8086, such that the memory address range assigned to the EPROM chips is F0000H–F3FFFH, using an address decoder that employs the 74138 IC and logic gates.

4. Interface two 8K × 8 RAM chips with the 8086, such that the memory address range assigned to the RAM chips is 20000H–23FFFH, using an address decoder that employs the 74138 IC and logic gates.
5. Interface four 16K × 8 EPROM chips with the 8086, such that the memory address range assigned to the EPROM chips is 90000H–9FFFFH, using an address decoder that employs two 74138 ICs and logic gates.
6. Interface four 16K × 8 RAM chips with the 8086, such that the memory address range assigned to the RAM chips is A0000H–AFFFFH, using an address decoder that employs two 74138 ICs and logic gates.
7. Interface an 8-bit DIP switch with the 8086 operating in minimum mode, such that the address assigned to it is F0H, using an address decoder having only logic gates. Write the instructions needed to read the data from the DIP switch into AL, in fixed port and variable port addressing.
8. Interface an 8-bit DIP switch with the 8086 operating in minimum mode, such that the address assigned to it is F0H, using an address decoder that employs the 74138 decoder and logic gates.
9. Interface a seven-segment LED in common cathode connection with the 8086 operating in minimum mode, such that the address assigned to it is 7FH, using an address decoder having only logic gates. Write the instructions needed to display the number 5 in the LED, using fixed port and variable port addressing.
10. Interface a seven-segment LED in common cathode connection with the 8086 operating in minimum mode, such that the address assigned to it is 3FH, using an address decoder that employs the 74138 decoder and logic gates. Write the instructions needed to display the number 7 in the LED, using fixed port and variable port addressing.
11. Interface an 8-bit DIP switch with the 8086 operating in minimum mode, such that the address assigned to it is FF80H, using an address decoder having only logic gates. Write the instructions needed to read the data from the DIP switch into AL.
12. Interface a seven-segment LED in common anode connection with the 8086 operating in minimum mode, such that the address assigned to it is 7FFFH, using an address decoder having only logic gates. Write the instructions needed to display the number 5 in the LED.
13. Interface a 16-bit DIP switch with the 8086 operating in minimum mode, such that the addresses assigned to it are 80H and 81H, using an address decoder having only logic gates. Write the instructions needed to read the data from the DIP switch into AX, in fixed port and variable port addressing.
14. Interface two seven-segment LEDs with common cathode connection with the 8086 operating in minimum mode, such that the addresses assigned to them are 70H and 71H, using an address decoder having only logic gates. Write the instructions needed to display the number F5 in the LEDs, using fixed port and variable port addressing.

7

Features and Interfacing of Programmable Devices for 8086-based Systems

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Architecture, need, features, and operation of the IC 8255
- Applications of the 8255 including interfacing of switches, seven-segment displays, A/D converter, D/A converter, stepper motor, and intelligent liquid crystal display (LCD) systems
- Features of keyboard/display interface of the IC 8279, interfacing of matrix keyboard, and multiplexed LED display
- Architecture, details, interfacing, and programming of the 8253 timer
- Serial port basics and definitions
- Features, details, interfacing, and programming of USART 8251
- Architecture and details of the programmable interrupt controller 8259
- Features and operation of the DMA controller 8237

7.1 INTEL 8255 PROGRAMMABLE PERIPHERAL INTERFACE

Intel microprocessors can transfer data between external devices such as input and output devices through ports. Normally, a register can act as an I/O port. However, having a separate register and configuring it for input and output operation becomes difficult and tedious. Hence, Intel has designed a separate IC 8255 with the objective of interfacing input and output devices with Intel microprocessors. The 8255 is used on a range of several I/O cards that plug into available slots in the personal computer (PC).

The 8255 programmable peripheral interface (PPI) is a very popular and versatile input/output chip that can be easily programmed to function in several different configurations. This chip can perform both digital input and output (DIO) operations from the processor in a preprogrammed manner.

The common applications of the 8255, include turning on or off an electronic switch such as a bipolar junction transistor (BJT), a metal oxide semiconductor field effect transistor (MOSFET), or an insulated gate bipolar transistor (IGBT), controlling movement by use of DC/AC/stepper motors, detecting the position using proximity sensors and interfacing different sensors (temperature, flow, pressure or level, etc.) through an analog to digital converter (ADC), etc.

7.1.1 Features of 8255

Each 8255 has three 8-bit TTL-compatible registers or ports, which allow programmers to control digital outputs, inputs, or a combination of both. The common features of the Intel 8255 IC are as follows:

(i) Three 8-bit ports named as A, B, and C are present.
(ii) Port C has been divided to two groups of 4 bits each as port C upper (PCU) and port C lower (PCL). Each of them can be programmed independently for input and output operation.
(iii) All the ports can be programmed for simple I/O or handshake I/O for the data transfer in I/O modes.
(iv) Each port C bit can be set/reset individually in bit set/reset (BSR) mode.
(v) Port A bits and PCU bits are grouped as group A (GA).
(vi) Port B bits and PCL bits are grouped as group B (GB).

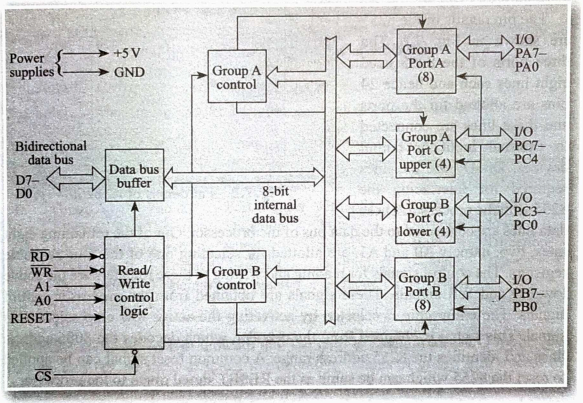


Fig. 7.1 Internal block diagram of 8255 PPI

7.1.2 Block Diagram of Intel 8255

The internal block diagram of the 8255 PPI is shown in Fig. 7.1.

As shown in Fig. 7.1, the block diagram of the 8255 has three basic registers called ports A, B, and C, each containing 8 bits. Port A and the upper 4 bits of port C are grouped together as group A. Similarly, port B and the lower 4 bits of port C are together known as group B. In addition to the three registers A, B, and

Table 7.1 Address lines and register selection of 8255

A1	A0	Register selected
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

C, there is another register called the control register. The contents written into the control register decide the operating modes of the three parallel ports. In order to identify the four registers, the 8255 uses two address lines A0 and A1. These lines get their signals from the processor address bus. The identification of the registers based on A0 and A1 is given in Table 7.1.

The pin details of the 8255 are shown in Fig. 7.2. The three ports of the 8255 need eight lines each and hence 24 pins are allotted for the ports and these lines are connected to external input or output devices. D0–D7 are the lines required for interfacing the 8255 with the processor. These data lines are connected to the data bus of the processor. Out of the remaining eight lines, two, namely A0 and A1, are allotted for selecting one of the four available registers. The control signals for reading and writing to these registers are the active low $\overline{RD}$ and $\overline{WR}$ signals. These signals are obtained from the processor control signals. The entire chip is selected by activating the active low chip select ($\overline{CS}$) signal. This signal is obtained from the decoder, which decodes the 8086 address lines and identifies the 8255 address range. A common reset signal can be applied to reset the 8255 which can be same as the RESET signal given to the processor.

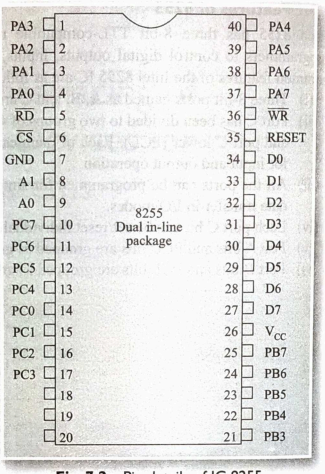


Fig. 7.2 Pin details of IC 8255

7.1.3 Operating Modes and Control Words of 8255

The function of each port in the 8255 is software-programmed by the programmer. The programming of the 8255 is done by writing a control word (CW) to the control register of the 8255. The control word contains information such as mode, bit set, bit reset, etc., that initializes the functional configuration of the 8255.

The basic operating modes of the 8255 are shown in Fig. 7.3. There are two different configurations of the 8255 namely input/output mode (I/O mode) and BSR mode. I/O mode consists of three different modes for the ports. The programmer can select a particular operating mode using commands and control words. The three ports of the 8255 are grouped as groups A and B which accept commands from the read/write (R/W) control logic and receive control words from the internal data bus, and issue proper commands to the associated ports.

The chip has to be programmed to configure its operation, before using it. The configuration is done by the control word (CW) which determines whether the ports are input, output, bidirectional, or strobed.

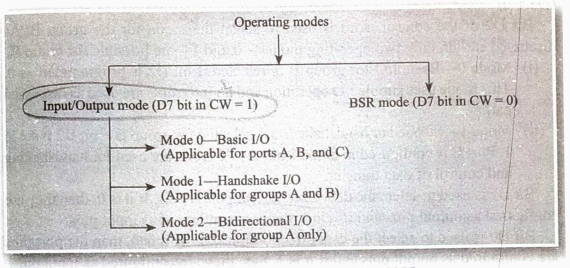


Fig. 7.3 Operating modes of the 8255

7.1.3.1 *I/O Control Word Format*

The control word format for the I/O configuration is given in Table 7.2.

Table 7.2 I/O control word format of the 8255

D7	D6 D5	D4	D3	D2	D1	D0
1 (1 = I/O)	Group A Mode select 00—mode 0 01—mode 1 1X—mode 2	Port A Direction select 1—input 0—output	Port C upper Direction select 1—input 0—output	Group B Mode select 0—mode 0 1—mode 1	Port B Direction select 1—input 0—output	Port C lower Direction select 1—input 0—output

The MSB D7 is set to 1 to indicate that the chip is configured in I/O mode. The bits D6 and D5 are used to select the operating modes of group A to one of the following three basic modes:

(i) Mode 0—Basic I/O (bits D6 and D5 are both 0)—Ports A, B, and C can be operated as inputs or outputs. This mode uses simple I/O operation and no interrupts are used. The outputs written to the ports are latched and available at any time. Inputs available at the port pins are buffered through port latches.

(ii) Mode 1—Strobed or handshake I/O (bits D5 and D6 are 0 and 1, respectively)—Port A is configured in mode 1 but upper port C is used for handshaking and control of data transfer in port A. Input and output data are latched.

(iii) Mode 2—Bidirectional bus (bits D5 and D6 are 1 and X, respectively)—Port A is bidirectional (both input and output) and port C is used for handshaking. Port B cannot be programmed to this mode.

Bit D4 is used to select the direction of data flow in the port A bits, that is, it decides whether the pins of port A are input (D4 = 1) or output pins (D4 = 0). Bit D3 is used to decide whether the PCU pins are used for input (D3 = 1) or output (D3 = 0).

Bit D2 of the control word is used to select the mode for the group B. As discussed earlier, only two operating modes—0 and 1—are possible for group B.

(i) Mode 0—Basic I/O for group B is selected if bit D2 is programmed as 0. This mode uses simple I/O operation and no interrupts are used as discussed earlier.

(ii) Mode 1—Strobed or handshake I/O is selected for group B if bit D2 is set to 1. Port B is configured in mode 1 but the PCL bits are used for handshaking and control of data transfer.

Bit D1 is used to select the data direction for port B pins. If it is 0, then they are configured as output pins and if it is 1, they are configured as input pins.

Bit D0 is used to select the data direction for PCL. If it is 0, then the port pins are configured as output pins and if it is 1, then they are configured as input pins.

7.1.3.2 *BSR Mode Control Word Format*

The control word format for the BSR configuration is given in Table 7.3.

Table 7.3 BSR control word format of the 8255

D7	D6	D5	D4	D3	D2	D1	D0
0 (0 = BSR mode)	X (Don't care)	X (Don't care)	X (Don't care)	B2	B1	B0	Bit set/reset 1 = set 0 = reset
				Bit Select bits—select one of 8 bits of port C			

In BSR mode, any of the eight bits of port C can be set or reset using a single control word written to the control register. This feature helps the programmer to control the port C pin outputs individually. This feature is also used in the mode 1 and mode 2 I/O operations wherein the individual ports of port C can be controlled by the programmer to indicate the status and control.

7.1.3.3 *I/O Mode 1 Operation*

Mode 1 configuration of the 8255 provides a means for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. In mode 1, ports A and B use the lines on port C to generate or accept these handshaking signals. The ports are divided into two groups—A and B. Each group contains one 8-bit port and one 4-bit control/data port. The 8-bit data port is either port A or port B and can be either an input or output port. Both inputs and outputs are latched. The 4-bit control port—either PCU or PCL is used to control and decide the status of the 8-bit ports A and B.

The operation of handshake signals for the input operation in mode 1 of the 8255 is explained with the help of Fig. 7.4.

The sequence of operations for the data input operation from an input device to a microprocessor through the 8255 is listed as follows.

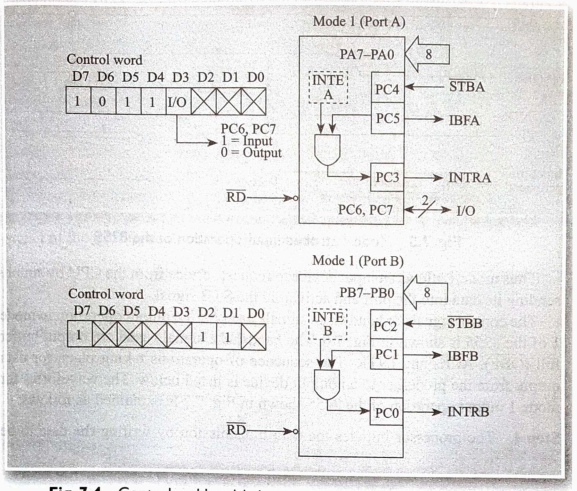


Fig. 7.4 Control and handshake signal for input operation in mode I

Step 1 The input device places data in the data lines of port A or port B. This is communicated to the 8255 by making the strobe input pin ($\overline{STB}$) low. $\overline{STB}$ is an active low signal applied through PC4 and PC2, for ports A and B respectively.

Step 2 The 8255 acknowledges the receipt of the data to the input by making input buffer full pin (IBF) high. This also indicates that the data has been latched into the input port.

Step 3 The 8255 then makes interrupt request line (INTR) high and applies an interrupt to the processor. This signal is applied only when the interrupt enable signal (INTE) is high. The INTE signal for port A is controlled by set/reset of PC4 and the INTE signal for port B is controlled by set/reset of PC2. PC2 and PC4 can be controlled using BSR mode.

Step 4 In the interrupt service routine, the processor reads the data from the corresponding input port. Reading from the port is done by selecting the 8255 port and applying $\overline{RD}$ active low signal.

Step 5 During read operation, the $\overline{RD}$ signal is low. When the $\overline{RD}$ signal goes low, the INTR signal is reset. The IBF is reset by the rising edge of the $\overline{RD}$ input.

Figure 7.5 shows the waveforms for mode 1 input operation of the 8255.

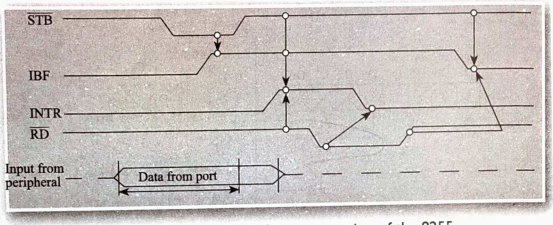


Fig. 7.5 Mode 1 strobed input operation of the 8255

Thus mode 1 allows an input device to request service from the CPU by simply sending its data into the port and activating the $\overline{\text{STB}}$ signal.

The control signals or handshake signals used for the output operation in mode 1 of the 8255 is shown in Fig. 7.6. The handshake signals used are output buffer full ($\overline{\text{OBF}}$), $\overline{\text{ACK}}$, and INTR. The sequence of operations taking place for data output from the processor to an output device is listed below. The waveforms for mode 1 output operation of the 8255 shown in Fig. 7.7 is explained as follows:

Step 1 The processor initiates the data transmission by writing the data to be

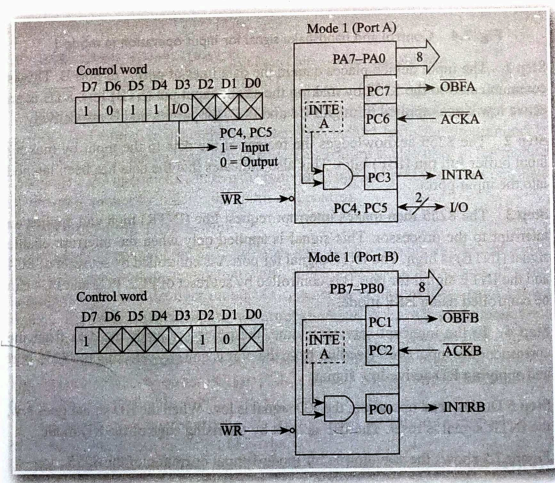


Fig. 7.6 Control and handshake signals for output operation in mode 1

transmitted to the output device, to the corresponding port of the 8255. This is done by sending the port address to the 8255, placing the data on the data lines and then activating the active low $\overline{WR}$ signal.

Step 2 To transfer the data to the output device, the 8255 makes the $\overline{OBF}$ low, to indicate that the CPU has written data to the specified port. The $\overline{OBF}$ signal is reset by the rising edge of the $\overline{WR}$ input.

Step 3 The data available on the output port pins is then read by the output device. After receiving data from the port pins, the output device acknowledges the receipt by making $\overline{ACK}$ low. $\overline{ACK}$ is an active low input signal to the 8255 from the peripheral device indicating that it has accepted a data. The $\overline{OBF}$ output signal of the 8255 is set by the $\overline{ACK}$ input going low.

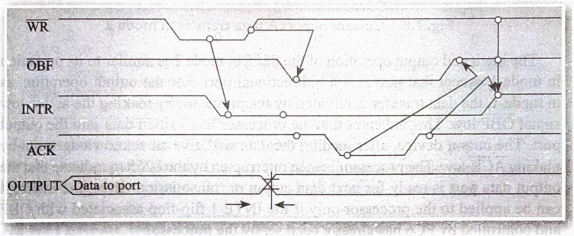


Fig. 7.7 Mode I strobed output operation of the 8255

Step 4 The 8255 now informs the processor that data has been transferred to the output device by making the INTR line high. A high on this output can be used to interrupt the CPU when an output device has accepted the data transmitted by the CPU. INTR is set when ACK, OBF, and INTE are all 1. INTE for port A is controlled by the set/reset of PC6 and INTE for port B is controlled by the set/reset of PC2. PC2 and PC6 can be controlled using BSR mode.

Step 5 In the interrupt service routine, the processor writes the next data to be transmitted to the output device to the output port of the 8255. The INTR signal is reset by the falling edge of the $\overline{WR}$ signal.

7.1.3.4 I/O Mode 2 Operation

In mode 2, data is transmitted and received via port A pins (bidirectional I/O) with handshaking capability. Only port A can be configured in mode 2 and is used as a bidirectional port, while port C is used for handshaking signals. Interrupt generation and enable/disable functions are also available through port C pins. Port B can be configured to be in mode 0 or 1 but not in mode 2. Both inputs and outputs are latched. The 5-bit control port (port C) is used for control and status for the 8-bit, bidirectional port (port A). The basic control signal transmission and operation of the data transfer in mode 2 is shown in Fig. 7.8.

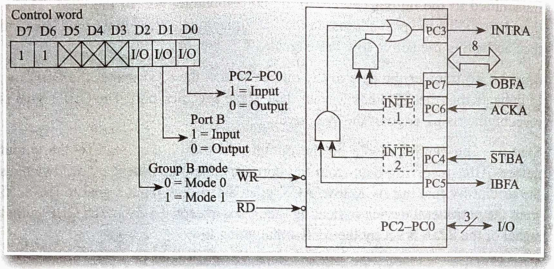


Fig. 7.8 Control of port A data transfer in mode 2

The input and output operation of the 8255 in mode 2 is similar to its operation in mode 1 except that port A is a bidirectional port. For the output operation, as in mode 1, the data transfer is initiated by the processor by making the active low signal $\overline{OBF}$ low. This indicates that the processor has written data into the output port. The output device, after reading the data will give an acknowledgement by making $\overline{ACK}$ low. The processor is then interrupted by the 8255 to indicate that the output data port is ready for next data output or transmission. Here, the interrupt can be applied to the processor only if the INTE 1 flip-flop associated with $\overline{OBF}$ and controlled by PC6 has already been set by the processor.

The input operation is also similar to mode 1 operation. Here, the data transfer is initiated by the input device by placing the data on the port pins. Then an active low control signal $\overline{STB}$ is given to the 8255 by the input device. The 8255 now latches up the data to its port and then gives an active high signal IBF to the input device. The 8255 then issues an interrupt signal to the processor to indicate that data is readily available for read operation. Here, the interrupt can be applied to the processor only if INTE 2 flip-flop associated with IBF and controlled by PC4 has already been set by the processor.

7.1.4 Programming Examples

Example 7.1

Configure the ports of the 8255 (PPI) as follows: port A = input, port B = output, PCU = output, PCL = input. Assume that the control register's address in the 8255 PPI is 46H. Configure the ports in simple I/O mode.

Solution:

The control word format for the given conditions is given in Table 7.4.

Table 7.4 Control word bit pattern (Example 7.1)

D7	D6	D5	D4	D3	D2	D1	D0
1	Group A		Port A	PCU	Group B	Port B	PCL
(1 = I/O)	mode—00		input—1	output—0	mode—0	output—0	input—1

The control word from this table is 10010001B, that is, 91H. The following program instructions will configure the control word of the 8255.

```
MOV AL, 91H     ; Load control word in the accumulator.
OUT 46H, AL     ; Transfer it to the control register of the 8255.
```

Example 7.2

Find the data direction and the modes of operation of ports of the 8255, if the control word written into it is A0H.

Solution:

The control word bit pattern is given in Table 7.5.

Table 7.5 Control word bit pattern (Example 7.2)

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	0	0	0	0	0
1 (1 = I/O)	Group A mode—1		Port A direction 0—output	Port C upper direction 0—output	Group B mode—0	Port B direction 0—output	Port C lower direction 0—output

The direction and modes of all ports are as follows:

Port A—output port in mode 1
Port B—output port in mode 0
Port C upper—output port
Port C lower—output port

7.2 INTERFACING SWITCHES AND LEDS

In this section, we discuss the interfacing of four switches and four LEDs with the 8086 through the 8255. Data is obtained from the switches and displayed using the LEDs.

The 8255 is interfaced with the 8086, with the 8255 ports connected to the switches and LEDs. A latch is used to demultiplex the lower address bus and the data bus (AD7–AD0). In the 8086-based system, either 8-bit or 16-bit addresses are used for the I/O devices. When 8-bit addresses are used, the address of the I/O device appears in the lines AD7–AD0 when the 8086 executes the IN/OUT instructions. When 16-bit addresses are used, the address of the I/O device appears in the lines AD15–AD0 when the 8086 executes the IN/OUT instructions. In this example, it is assumed that 8-bit addresses are used for the different ports and control register of the 8255. So, the lower-order address bus alone is enough for addressing the 8255 and the address decoder uses the address bus A7–A0. If 16-bit addresses are used for the different ports and control register of the 8255 then the higher-order address bus is required and the address decoder uses the address bus A15–A0. The signals M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ are also used in decoding and selecting the 8255.

The 8255 needs two address lines A0 and A1 in order to select one of its four registers, as detailed in Table 7.1 and they are respectively connected to the lines A1 and A2 of the 8086. This is because, there are two I/O banks in the 8086, namely odd and even banks. The odd bank contains the I/O devices that have

only odd addresses and their data lines are connected to the data bus D15–D8 of the 8086. The even bank contains the I/O devices that have only even addresses and their data lines are connected to the data bus D7–D0 of the 8086. Since the 8255 has only 8-bit data bus namely D7–D0, it can be connected to either D7–D0 or D15–D8 of the 8086. If the data bus of the 8255 is connected to D7–D0 of the 8086, only even addresses can be assigned to the ports and the control register of the 8255; if it is connected to D15–D8 bus of the 8086, only odd addresses can be assigned to the ports and control register of the 8255.

Let us assume that the data bus of the 8255 is connected to D7–D0 of the 8086, so that only even addresses can be assigned to the ports and control register of the 8255 throughout the discussion of the 8255 interfacing in this chapter. If address line A0 of the 8086 is connected to line A0 in the 8255 then there occurs a problem in selecting the port B and control register in the 8255 using even addresses. The reason being, for all even addresses, the address line A0 is always zero and hence it is not possible to select port B and the control register of the 8255, for the selection of which line A0 must be 1. Hence, the lines A0 and A1 of the 8255 are connected to lines A1 and A2 of the 8086. The $\overline{IOR}$ and $\overline{IOW}$ signals from the 8086 are connected to the $\overline{RD}$ and $\overline{WR}$ control signals of the 8255, respectively.

Figure 7.9 shows the interfacing of the switches and LEDs with the 8255 through the 8086. The four switches in Fig. 7.9 are connected to the lower-order four bits of port A of the 8255. The switch connection is such that when it is open, it connects logic 0, that is, 0 volts to the port and when it is closed, it connects logic 1, that is, 5 volts to the port pins. These connections ensure that the port is not damaged and also not sourcing over current. This ensures safe operation of the ports and switches. The interfacing of four LEDs through an inverter (which acts as a driver) to the ports is shown in Fig. 7.9. When logic 1 is given on the port pin, it will be inverted by the inverter and will connect ground (logic 0) to the cathode of the LED. This will forward bias the LED and light will be emitted by the LED. This connection ensures that the port pin is not sourcing enormous current and also the current required for the LED illumination is from the supply and the driver IC.

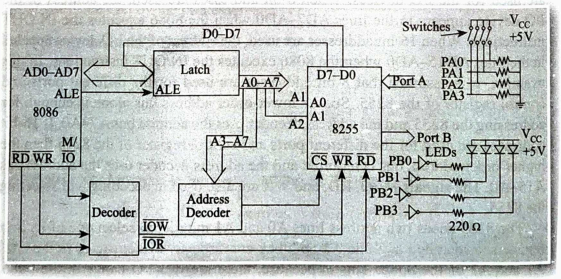


Fig. 7.9 Interfacing keys and LEDs with the 8086 through the 8255

The design of the address decoder for the 8255 is explained as follows with an example. Let us assume that we want to assign the addresses 40H, 42H, 44H, and 46H to port A, port B, port C, and the control register of the 8255, respectively. In Table 7.6, these addresses are given in binary and hexadecimal form. The address decoder is shown in Fig. 7.10.

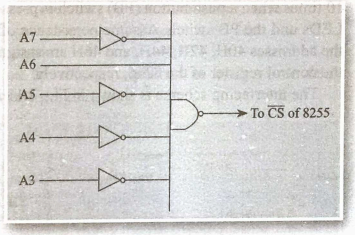


Fig. 7.10 Address decoder for the 8255 (shown in Fig. 7.9)

Table 7.6 Addresses assigned for 8255 (shown in Fig. 7.9)

Addresses in hexadecimal form	Addresses in binary form A7	A6	A5	A4	A3	A2	A1	A0	Register selected in the 8255
40H	0	1	0	0	0	0	0	0	Port A
42H	0	1	0	0	0	0	1	0	Port B
44H	0	1	0	0	0	1	0	0	Port C
46H	0	1	0	0	0	1	1	0	Control register

The software part consists of initializing the 8255 for port A input and port B output operation. All the ports are initialized in mode 0. Hence, the control word shown in Table 7.7 is used and the control word is 90H.

Table 7.7 Control word bit pattern for interfacing LEDs and switches

D7	D6	D5	D4	D3	D2	D1	D0
1 (1 = I/O)	Group A mode—00		Port A input—1	Port C upper output—0	Group B mode—0	Port B output—0	Port C lower output—0

The program for initializing the 8255 and transferring the data available in port A to port B is as follows:

```
MOV AL, 90H     ; Move control word to accumulator.
OUT 46H, AL     ; Send control word in AL to control register of 8255.
IN AL, 40H      ; Get the data from port A in accumulator.
OUT 42H, AL     ; Send the data in AL to port B.
HLT             ; Terminate program execution.
```

Example 7.3

Design a system (both software and hardware) that will cause four LEDs to flash

10 times when a push button (PB) switch is pressed. Use the 8255 to interface the LEDs and the PB switch. Assume persistence of vision to be 0.1 s. Assume that the addresses 40H, 42H, 44H, and 46H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

The interfacing scheme is shown in Fig. 7.11 in simplified form.

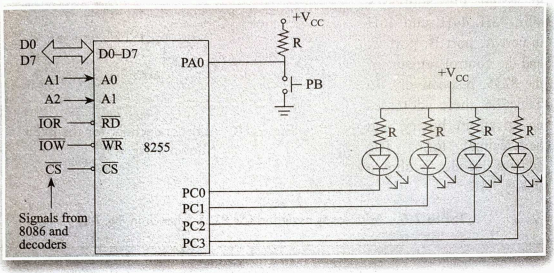


Fig. 7.11 Interfacing LEDs with the 8086 through the 8255

Program:

```
        MOV AL, 90H     ; Move the control word to configure port A as
                          input port and port C as output port in AL.
        OUT 46H, AL     ; Move data in AL to control register.
        MOV BL, 0AH     ; Move count of 10 decimal (=0AH) in BL.
CHECK:  IN AL, 40H      ; Input data from Port A into AL (i.e., PA0).
        RCR AL, 1       ; Rotate content of AL right by 1 bit
                          through carry to check LSB in AL.
        JC CHECK        ; If carry = 1, PB switch is not
                          pressed, so go to CHECK.
REP:    MOV AL, 00H     ; Turn on all LEDs by sending 00H to port C.
        OUT 44H, AL     ; Move data in AL to port C.
        CALL DELAY      ; Call delay program of 0.1 second delay.
        MOV AL, 0FH     ; Turn off all LEDs by sending 0FH to port C.
        OUT 44H, AL     ; Move data in AL to port C.
        CALL DELAY      ; Call delay program of 0.1 second delay.
        DEC BL          ; Decrement BL.
        JNZ REP         ; If BL is not 0, go to REP to turn on and turn
                          off LEDs again.
        JMP CHECK       ; If BL is 0, go to CHECK, to check status of PB
                          switch.
DELAY:  MOV CX, COUNT   ; Load COUNT in CX.
L1:     NOP             ; No operation
        NOP             ; No operation
        DEC CX          ; Decrement value in CX.
```

```
JNZ L1          ; Execute loop L1, until CX becomes zero.
RET             ; If time delay over, then return from
                  subroutine.
```

COUNT is calculated based on the concept explained in writing time delay programs in 8086 assembly language programming.

7.2.1 Debouncing of Keys

A key, in general, is a type of push-button switch, toggle switch, or electromechanical relay, having spring contacts. Metal contacts make and break the circuit and carry the current in switches and relays. These contacts have mass and contain springs to control the movement. Since the moving contacts have mass and springs, with low damping they will be 'bouncy' as they make and break. When a normally open (NO) pair of contacts is closed, the contacts will come together and bounce off each other several times before finally coming to rest in a closed position. The effect is called *contact bounce* or, in a switch, it is called *switch bounce*. The waveform of a switch with contact bouncing, from position 1 to 0, is shown in Fig. 7.12.

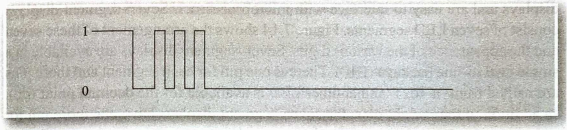


Fig. 7.12 Contact bounce waveform

If such a switch is used for sensing by input ports of a microprocessor, then there is a chance that the microprocessor will respond several times, that is, input will be sensed repeatedly even though the key is pressed only once. In general, the bouncing of the switch may last for several milliseconds. Since the microprocessor works at a speed of a few microseconds, it senses the input several times.

The simplest hardware solution uses a resistor capacitor (RC) time constant to suppress the bounce and the circuit for this is shown in Fig. 7.13. The RC time constant has to be larger than the switch bounce and is generally around 0.1 s. The capacitor takes at least twice the time constant to change from one position to the other. During this time, any change in the switch position is not transmitted beyond the buffer. The buffer, after the switch is used to make the transition from high-to-low or low-to-high sharp.

The key bouncing problem can be solved by software methods also. The easiest software method is to make the processor wait until the bouncing oscillation settles down. This wait-and-see technique is implemented using software time delays. When the voltage from the switch changes, an appropriate delay routine is executed and the value of the voltage on the switch line is checked again to make sure that the line has stopped bouncing. The delay is normally 10 ms as in most of the switches, the oscillations settle within that period.

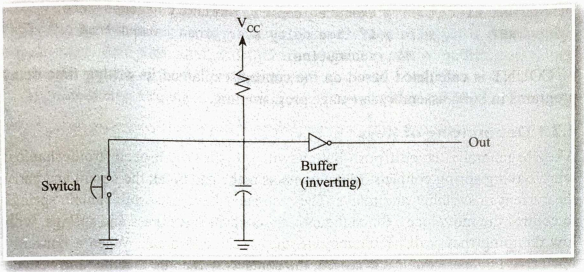


Fig. 7.13 Hardware solution for debouncing of keys

7.3 INTERFACING SEVEN-SEGMENT DISPLAYS

Seven-segment light emitting diode displays are the most commonly used low-cost displays and are easy to interface with microprocessors. Seven-segment displays consist of seven LED segments. Figure 7.14 shows the arrangement of these seven and the appearance of the various digits. Seven-segment displays are available in a single dual in-line package (DIP). There is one pin for each segment and these pins are named from 'a' to 'f' and another LED is available for the decimal point (dp). In addition to these eight pins, the seven-segment displays have one more pin for power supply. Seven-segment displays come in two types—common anode and common cathode.

In common anode display, the anodes of all segment LEDs are connected together. So, to illuminate a segment, the common anode is connected to the supply and then the segment input, that is, 'a' to 'f' is connected to a low-level voltage or logic 0.

In common cathode display, the cathodes of all the LEDs are connected together. So, to illuminate a segment, the corresponding segment input is connected to the high-level voltage or logic 1 and the common cathode is connected to the ground. This forward biases the LEDs and illuminates them.

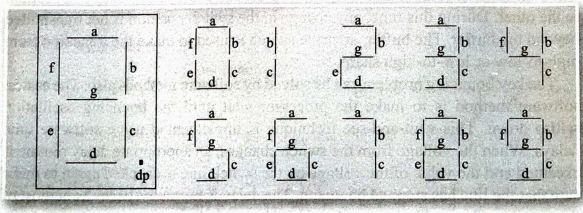


Fig. 7.14 Arrangement of LEDs and appearance of digits in seven-segment displays

The circuit required to drive a single seven-segment LED display from a 4-bit BCD output is shown in Fig. 7.15.

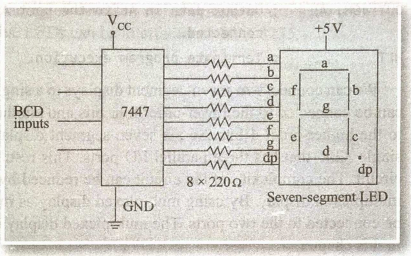


Fig. 7.15 Driver circuit for single seven-segment display

The BCD to the seven-segment display decoder IC 7447 converts the 4-bit BCD code applied at its input into the patterns required to display the BCD number in a seven-segment LED. The patterns generated are active low outputs, meaning that logic 0 is given as output when a particular segment is to be illuminated. So, the common anode display is suitable for use with the 7447.

The complete circuit diagram for interfacing the seven-segment display with the 8086 through the 8255 and the 7447 is shown in Fig. 7.16.

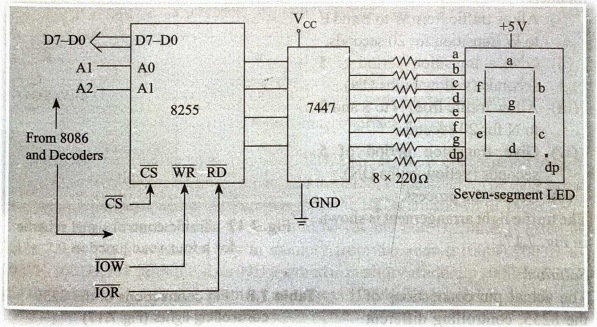


Fig. 7.16 Circuit for interfacing single seven-segment display

Let us assume that the addresses 40H, 42H, 44H, and 46H are assigned to port A, port B, port C, and the control register of the 8255, respectively. The following instructions can be used to display the data '7' in the seven-segment display.

Program:

```
MOV AL, 80H     ; Load the control word 80H in AL to configure port A as
                  output port.
OUT 46H, AL     ; Output it to the control register of the 8255.
MOV AL, 07H     ; Load accumulator with data to be displayed in the
                  lower nibble as PA3-PA0 is connected to the 7447.
```

```
OUT 40H, AL     ; Output data in AL to the port A, where display is
                  connected.
HLT             ; Terminate program execution.
```

We can connect two seven-segment displays to a single 8-bit port. One 7447 IC can be connected to the lower-order four bits and another 7447 can be connected to the higher-order 4 bits. So six seven-segment displays can be connected to a single 8255 that has three parallel I/O ports. This results in a more complicated circuit. The complexity of the circuit can be reduced by using a technique called multiplexed display. By using multiplexed display as many as eight displays can be connected to the two ports. The multiplexed display concept is discussed later in this chapter.

7.4 TRAFFIC LIGHT CONTROL

Example 7.4

Design a microprocessor system to control traffic lights. The traffic should be controlled in the following manner:

(i) Allow traffic from W to E and E to W transition for 20 seconds.
(ii) Give a transition period of 5 seconds (yellow bulbs ON).
(iii) Allow traffic from N to S and S to N for 20 seconds.
(iv) Give transition period of 5 seconds (yellow bulbs ON).
(v) Repeat the process.

The traffic light arrangement is shown in Fig. 7.17.

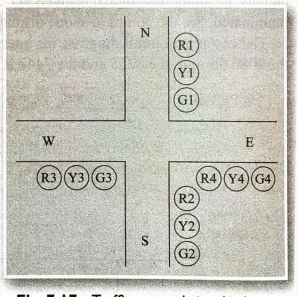


Fig. 7.17 Traffic control signal scheme for a four-road junction

Solution:

The actual pin connections of the 8255, controlling different lights are listed in Table 7.8. The interfacing diagram to control 12 electric bulbs is shown in Fig. 7.18. Port A is used to control the lights on N–S road and port B is used to control lights on W–E road.

Table 7.8 Pin connections of the 8255 controlling lights (Fig. 7.17)

Pins	Light	Pins	Light
PA0	R1	PB0	R3
PA1	Y1	PB1	Y3
PA2	G1	PB2	G3
PA3	R2	PB3	R4
PA4	Y2	PB4	Y4
PA5	G2	PB5	G4

The electric bulbs (i.e., lights) 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. The interfacing of the 8255 to the system is also shown in Fig. 7.18. Instead of 230V bulbs, LEDs can be interfaced with the 8255.

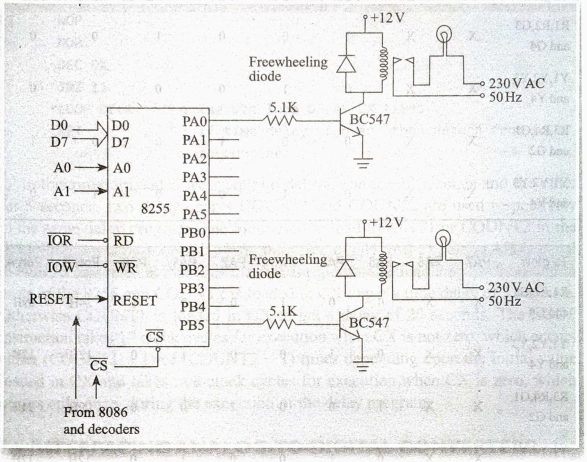


Fig. 7.18 Traffic light control interface diagram

Let us assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively. The data bytes to be sent to the ports of the 8255 to glow specific LEDs are shown in Table 7.9.

Let us assume that the data to be sent to port A and port B, which is listed in Table 7.9 is stored consecutively in memory from the address1000H:2000H to 1000H:2007H. For example, data 09H is stored in memory at address 1000H:2000H and the data 24H is stored in memory at address 1000H:2001H and so on.

Program:

```
START: MOV AL, 80H    ; Move the control word 80H in AL to configure
                        port A and port B as output ports.
       OUT 86H, AL    ; Move the control word in AL to the control
                        register of the 8255.
       MOV BX, 1000H  ; Move the segment address of data (=1000H) to
                        BX.
       MOV DS, BX     ; Move the segment address in BX to DS.
START: MOV AH, 04H    ; Move the number of data sets (8/2 = 4) to AH.
       MOV BX, 2000H  ; Move the offset address of data (=2000H) to
                        BX.
```

Table 7.9 Traffic signal subsequences for a four-road junction

To glow	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
R1,R2,G3 and G4	X	X	1	0	0	1	0	0
Y1,Y2,Y3 and Y4	X	X	0	1	0	0	1	0
R3,R4,G1 and G2	X	X	0	0	1	0	0	1
Y1,Y2,Y3 and Y4	X	X	0	1	0	0	1	0

To glow	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port B	Port A
R1,R2,G3 and G4	X	X	0	0	1	0	0	1	24H	09H
Y1,Y2,Y3 and Y4	X	X	0	1	0	0	1	0	12H	12H
R3,R4,G1 and G2	X	X	1	0	0	1	0	0	09H	24H
Y1,Y2,Y3 and Y4	X	X	0	1	0	0	1	0	12H	12H

```
REP:    MOV CX, COUNT1 ; Load count1 corresponding to 20 seconds delay
                         in CX.
        MOV AL, [BX]   ; Move the data at [BX] in memory into AL.
        OUT 80H, AL    ; Send the data in AL to Port A.
        INC BX         ; Increment BX.
        MOV AL, [BX]   ; Move the data at [BX] in memory into AL.
        OUT 82H, AL    ; Send the data in AL to Port B.
        CMP AH, 03H    ; Check whether AH = 3.
        JNZ CHECK      ; Otherwise check whether AH = 1.
        JMP LOAD       ; If AH = 03, go to LOAD.
CHECK:  CMP AH, 01H    ; Check whether AH = 1.
        JNZ NEXT       ; If AH is not equal to 1, go to NEXT.
LOAD:   MOV CX, COUNT2 ; Load COUNT2 corresponding to 5 seconds delay
                         in CX.
NEXT:   CALL DELAY     ; Call DELAY program to wait for 20 or 5
                         seconds.
        INC BX         ; Increment BX.
        DEC AH         ; Decrement AH.
        JNZ REP        ; If AH is not zero, go to REP to send next set
                         of data to ports.
```

```
        JMP START       ; Jump to START to start from the first set of
                          data.
DELAY:  MOV DX, 0FFFFH
L1:     NOP
        NOP
        DEC DX
        JNZ L1
        LOOP DELAY      ; Execute loop DELAY CX times.
        RET             ; If time delay is over, then return from
                          subroutine.
```

In this program, since we want two delays, one for 20 seconds and the other for 5 seconds, two delay counts COUNT1 and COUNT2 are used respectively in the same delay program. The logic used to load COUNT1 or COUNT2 in the delay program is based on checking the value of AH. If the value in AH is equal to either 3 or 1, then at that time the data for glowing yellow lamps will be sent to ports of the 8255 and COUNT2 is loaded in CX to get a time delay of 5 seconds. Otherwise COUNT1 is loaded in CX to get a delay of 20 seconds. The LOOP instruction takes 17 clock cycles for execution when CX is not zero, which occurs either (COUNT1 − 1) or (COUNT2 − 1) times depending upon the initial value loaded in CX and takes five clock cycles for execution when CX is zero, which occurs only once, during the execution of the delay program.

7.5 INTERFACING ANALOG-TO-DIGITAL CONVERTERS

The basic function of the analog-to-digital converter (ADC) is to convert the input analog voltage levels into corresponding discrete digital signals. An ADC is essential in a microprocessor-based system as the microprocessor can only handle digital data, though the real-world signals are all in analog form only.

There are many types of ADC. The major ones are counter ramp type ADC, dual slope ADC, flash type ADC, and successive approximation type ADC. Each type of ADC has its own advantages and disadvantages. Successive approximation type ADC is a commonly available ADC. This ADC has fixed conversion time for any analog input voltage level.

The specifications of the ADC are the range of analog input voltage, number of digital bits at the output, resolution, the conversion time, and the number of analog input channels. The analog input voltage can be either unipolar or bipolar. Unipolar means that the input voltage can have only one polarity such as 0 to +5V or 0 to +10 V. Bipolar means that the input voltage can range from one polarity to the other such as −5 to +5 V or −10 to +10 V. Most of the ADC chips come with an option of selecting one of these voltage ranges using the V_{ref} input pins. The ADC chips are available for different number of output binary bits. ADCs are available with 8-, 10-,12-, or 16-bit digital outputs. The number of bits will decide the number of voltage levels sensed. For example, an 8-bit ADC will have 2^8 possible levels, that is, 256 levels. The number of bits and the input voltage range will decide the resolution. The resolution of an ADC is defined as the smallest change

in the input voltage that can be sensed or detected at the output. The resolution can be mathematically defined as the range of input voltage divided by the number of levels at the output. For example, an ADC with the input voltage range of 0 to +5V with eight bits at the output will have a resolution of 5/256, that is, approximately 19.5 mV. The conversion time of the ADCs will be decided by the type of the ADC, and the clock frequency used in the converter circuits.

Some ADC chips come with an option of having more than one analog input. One of these analog input channels are selected using select lines and an analog multiplexer circuit. The ADC chips also have a sample and hold circuit. The sample and hold circuit is used to maintain the analog input voltage constant, when the conversion is in progress.

7.5.1 ADC Chips and Interfacing to Microprocessor

The single chip ADCs available in the markets have many options. The commonly available ADC chip family is ADC 080X from National Semiconductor. ADC 0800, ADC 0804, ADC 0808, and ADC 0816 are the common chips available in this family. ADC 0804 has one analog input channel with an 8-bit output. ADC 0808/0809 has eight analog inputs with 3-bit channel select lines and an 8-bit output. ADC 0816 has 16 analog input channels with four select lines and 8-bit outputs.

This section discusses the operation and interfacing of ADC 0816 with the 8086 microprocessor through the 8255 PPI. ADC 0816 is an 8-bit successive approximation type ADC chip with an in-built analog multiplexer, which can select one of 16 analog inputs for conversion into digital format. One of 16 analog inputs IN0–IN15 in the ADC0816 chip can be selected by the select lines A, B, C, and D. The analog to digital conversion can be started by using the active high control signal Start Conversion (SC). The conversion of the analog voltage on the input channel selected, will then take place based on the clock signal applied to the ADC chip. After the conversion is over, the ADC chip will issue an active high 'end of conversion' (EOC) signal on the EOC line. The digital output can then be read from the data lines after issuing an active high Output Enable (OE) signal on the OE line in the ADC chip.

The interfacing of ADC 0816 with the 8255 is shown in Fig. 7.19. The 8255 PPI is in turn interfaced with the 8086 as shown in Fig. 7.9. In the interfacing diagram shown in Fig. 7.19, it can be noted that the port A of the 8255 is used to output or send the channel select lines and the related control signals. The port B lines are used to get or input the digital result data from the ADC chip. The LSB of port C (i.e., PC0) is used to check the end of conversion signal. With this hardware arrangement, the ADC chip can only be interfaced with software polling method. For interrupt driven interface, the EOC signal can be connected to any interrupt input. Analog inputs can be applied to the analog input pins of the ADC 0816.

The software interfacing procedure follows the flowchart shown in Fig. 7.20. The ADC conversion process can be started after applying the analog input to any of the channels. The conversion process is started by initializing the 8255 with the proper control word. The control word format for the hardware interface in

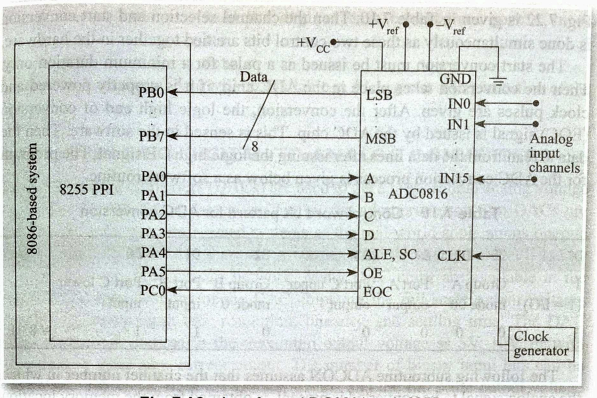


Fig. 7.19 Interfacing ADC0816 with 8255

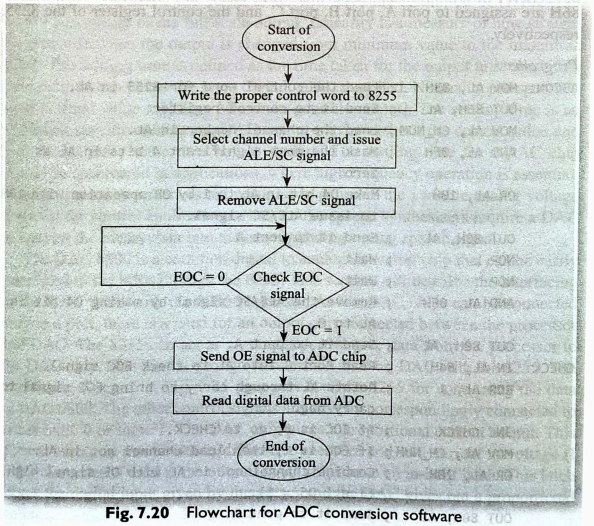


Fig. 7.20 Flowchart for ADC conversion software

Fig. 7.22 is given in Table 7.10. Then the channel selection and start conversion is done simultaneously as these two control bits are tied together in the hardware.

The start conversion must be issued as a pulse for a minimum duration only. Then the conversion takes place in the ADC chip, if it is properly powered and clock pulses are given. After the conversion, the logic high end of conversion (EOC) signal is issued by the ADC chip. This is sensed in the software. Then the data is read from the data lines after issuing the logic high OE signal. The program for the ADC conversion process is given below as a software routine.

Table 7.10 Control word bit pattern for ADC conversion

D7	D6	D5	D4	D3	D2	D1	D0	
1 (1 = I/O)	Group A mode 00		Port A output	Port C upper output	Group B mode 0	Port B input	Port C lower input	
1	0	0	0	0	0	1	1	= 83H

The following subroutine ADCON assumes that the channel number in which the analog signal to be converted is present, is obtained from the memory location named CH_NUM (i.e., if the analog data in the 5th channel has to be converted then CH_NUM will have the data 05H) and the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Program:

```
ADCON: MOV AL, 83H     ; Load the control word for 8255 in AL.
       OUT 86H, AL     ; Send it to control register.
       MOV AL, CH_NUM  ; Load the channel number in AL.
       AND AL, 0FH     ; Mask the most significant 4 bits in AL as
                         zero.
       OR AL, 10H      ; Make D4 bit in AL to 1 by OR operation with 10H
                         to issue ALE/SC signal.
       OUT 80H, AL     ; Send it to port A.
       NOP             ; Wait.
       NOP             ; Wait.
       AND AL, 0FH     ; Remove the ALE/SC signal by making D4 bit in
                         AL to 0.
       OUT 80H, AL     ; Send it to port A.
CHECK: IN AL, 84H      ; Read Port C into AL to check EOC signal.
       RCR AL, 1       ; Rotate AL through carry to bring EOC signal to
                         carry flag.
       JNC CHECK       ; If EOC is 0, go to CHECK.
       MOV AL, CH_NUM  ; If EOC is 1, then load channel no. in AL.
       OR AL, 20H      ; Combine channel no. in AL with OE signal high
                         (i.e., D5 = 1).
       OUT 80H, AL     ; Send it to port A.
```

```
IN AL, 82H      ; Read the digital data result from port B into
                  AL.
RET             ; Return.
```

The resultant digital data will be available in AL register after execution of the subroutine.

7.6 INTERFACING DIGITAL-TO-ANALOG CONVERTERS

Digital-to-analog converters (DACs) are used to get a proportional analog voltage or current for the digital data given out by the microprocessor. The DACs are essential in microprocessor-based systems as the real-world applications operate with analog data. Basically, there are two types of DACs. They are R–2R ladder network and weighted resistor network. Many DAC chips are available in the market. The specifications of the DAC chips are the full scale output voltage, number of binary input bits, resolution, linearity, and settling time. The DAC chips come with choices in the maximum output voltage as 5V, 10V, or with a predefined maximum current output. The number of binary input bits can be four, eight, 10, or 12. Both the number of bits and the full scale output voltage will determine the resolution. For example, an 8-bit DAC can have 256 input combinations and so has a resolution of (1/256) or 0.39 percentage of the full scale output. Similarly, the 10-bit DAC will have the resolution of (1/1024) or 0.0977 percentage of the full scale output. Linearity is a measure of how straight the output is when the output is changed from minimum value to the maximum value. The settling time is defined as the time taken for the output to settle within pre-specified band after the input digital value is applied. Normally, pre-specified band is [final value ± (1/2) × Minimum possible output]. The settling time is an important specification as the DAC output may overshoot the correct value and may oscillate for some time before settling. The settling time for a DAC chip should be considered in applications, where high frequency operation is essential. Digital to analog converters are required to generate the variable analog voltage essential for control applications. Most of the speech synthesizers require a DAC to convert the binary data into the corresponding analog speech signal.

The DAC 0800 is a common digital to analog converter chip that can be easily interfaced to the 8086 through the 8255. This section will describe the interfacing of the DAC 0800 with the 8086 processor. As the DAC chips can be connected only to a port, there is a need for an output port connected between the processor and DAC. The 8255 can act as an output port to give data from the processor to the DAC chip. One port is enough to interface an 8-bit DAC with the 8255. The interfacing diagram in Fig. 7.21 uses port A of the 8255 for connecting the data to DAC 0800. The other control signals are directly correspondingly connected to either logic 0 or logic 1. The DAC chip gives a proportional current output. This current output in most cases is difficult to measure and so a current to voltage (I to V) converter is used at the output. DAC chips have an in-built latch. This latch stores the digital input given by the port A and the DAC gives out a proportional voltage.

This section explains the software for common application examples involving the DAC interfaced with the 8255 as shown in Fig. 7.21. Four common applications such as square wave generation, ramp wave generation, staircase wave generation, and sine wave generation are discussed using examples.

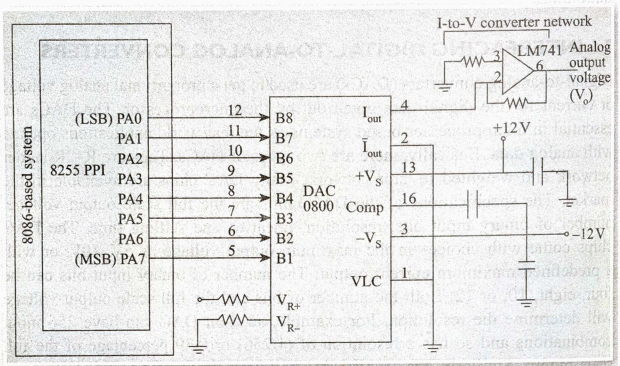


Fig. 7.21 Interfacing DAC 0800 with the 8086

7.6.1 Square Wave Generation

Example 7.5

Write a program to generate a square waveform using a DAC chip. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Solution:

The DAC chip interfaced to the 8255 and connected to the processor 8086 shown in Fig. 7.21 is considered for all the examples. In this example, a square waveform is generated using the DAC chip. The square waveform has 0 V output for one half period and then a voltage of amplitude V_1 volts for the other half period.

The software part of the program consists of initializing the 8255 for making port A as an output port. Then the binary data for 0 V is given to port A and a delay routine is called to wait for half the time period. After this, the data on the port A is made equivalent to that of V_1 volts. The delay routine is called once again.

The digital data for any particular voltage V_1 is calculated using the following formula:

Digital value for V_1 voltage output = [$(2^n - 1)$/Maximum output voltage] × V_1, where n is the number of binary bits in the input of the DAC.

For example, for a 3 V output in a DAC with an 8-bit binary input and a maximum of 5 V output, the equivalent digital value for 3 V is $[(2^8 - 1)/ 5] \times 3 = 51 \times 3 = 153$ in decimal form. The same value in hexadecimal will be 99H.

The following program can generate a square of amplitude 3 V and 0 V with

a predefined delay and so a predefined frequency. The time delay produced by the delay routine is explained in the programming section of the 8086. Using the required time delay and the clock frequency of the system, the delay count can be calculated. The square waveform generated is shown in Fig. 7.22.

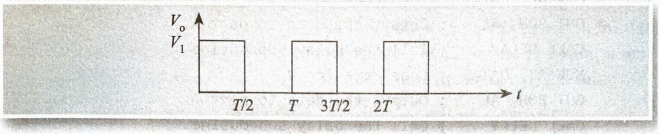


Fig. 7.22 Square waveform generated

Program:

```
START: MOV AL, 80H     ; Load the control word for 8255.
       OUT 86H, AL     ; Send it to control register.
REP:   MOV AL, 00H     ; Load initial data for DAC.
       OUT 80H, AL     ; Output the data to port A, where the DAC is
                         interfaced.
       CALL DELAY      ; Call the delay subroutine.
       MOV AL, 99H     ; Load the data corresponding to V1 in AL.
       OUT 80H, AL     ; Send it to port A.
       CALL DELAY      ; Call the delay subroutine.
       JMP REP         ; Loop again to get continuous waveform.
DELAY: MOV CX, COUNT   ; Load CX register with a count value.
L1:    NOP             ; No operation
       NOP             ; No operation
       LOOP L1         ; Execute loop L1 CX times.
       RET             ; If time delay over, then return from
                         subroutine.
```

7.6.2 Staircase Waveform Generation

Example 7.6

Write a program to generate a staircase waveform using a DAC chip. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Solution:

The waveform to be generated is shown in Fig. 7.23 and the hardware is assumed to be the same as shown in Fig. 7.21. Three levels V_1, V_2, V_3 are assumed in the output voltage waveform. The hexadecimal output to be given to the port is calculated using the formula given in Example 7.5. The following program assumes the three digital values corresponding to V_1, V_2, and V_3 are DATA1, DATA2, and DATA3 respectively. The fixed time delay is used in all the three levels.

Program:

```
START: MOV AL, 80 H    ; Load the control word for the 8255.
       OUT 86H, AL     ; Send it to control register.
```

```
L1:     MOV AL, DATA1  ; Load data for V1.
        OUT 80H, AL    ; Output the data to port A, where the DAC is
                         interfaced.
        CALL DELAY     ; Call the delay subroutine.
        MOV AL, DATA2  ; Load data for V2.
        OUT 80H, AL    ; Output the data to port A.
        CALL DELAY     ; Call the delay subroutine.
        MOV AL, DATA3  ; Load data for V3.
        OUT 80H, AL    ; Output the data to port A.
        CALL DELAY     ; Call the delay subroutine.
        JMP L1         ; Loop again to get continuous waveform.
DELAY:  MOV CX, COUNT  ; Load CX register with a count value.
L2:     NOP            ; No operation
        NOP            ; No operation
        LOOP L2        ; Execute loop L2, CX times.
        RET            ; If time delay is over, then return from
                         subroutine.
```

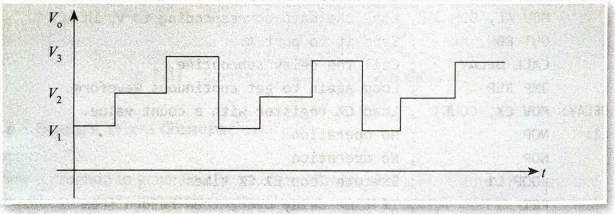


Fig. 7.23 Staircase waveform generated

7.6.3 Ramp Waveform Generation

Example 7.7

Write a program to generate a ramp waveform using a DAC chip. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Solution:

The 8255 port A is assigned values starting from 0 and increasing gradually. The hardware details are assumed to be the same as that in Fig. 7.21. The following program generates the ramp waveform shown in Fig. 7.24 with $V_1 = 5$ V.

The delay calculation is slightly different from the previous examples. Here, the voltage levels are increased

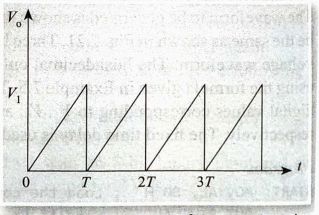


Fig. 7.24 Ramp waveform generated

from 0 to FFH, that is, 0–255 in decimal. So, within T seconds, there are 255 levels. Hence, the delay for each level will be $T/255$. The delay time should be as small as possible for ramp generation. Otherwise, the waveform will look like a staircase waveform of 255 levels. The frequency of the wave is the reciprocal of time t.

The software part consists of incrementing the value given to port A from 00H to FFH with a time delay routine called at each level. In the program, after the content of AL reaches the value FFH and when it is incremented, the value 00H is automatically obtained in AL.

Program:

```
        MOV AL, 80H     ; Load the control word for the 8255.
        OUT 86H, AL     ; Send it to control register.
        MOV AL, 00H     ; Load initial data for DAC.
L1:     OUT 80H, AL     ; Output the data to port A, where the DAC is
                          interfaced.
        CALL DELAY      ; Call the delay subroutine.
        INC AL          ; Increment the data in AL to get ramp wave.
        JMP L1          ; Loop again to send the data to DAC.
DELAY:  MOV CX, COUNT   ; Load CX register with a count value.
L2:     NOP             ; No operation
        NOP             ; No operation
        LOOP L2         ; Execute loop L2 CX times.
        RET             ; If time delay is over, then return from
                          subroutine.
```

7.6.4 Waveform Generation using Stored Data

Example 7.8

Write a program to give out a set of digital data stored in the memory locations from the address 2000H:1000H to 2000H:1050H to the DAC chip connected to the port A of the 8255 repeatedly. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Solution:

In this example, the voltage data to be given to the port A is stored in memory locations starting from the address 2000H:1000H. These data can be anything to generate any type of waveform. It can be any predefined waveform such as a sine wave. In such cases, the sine wave is divided into many levels and each level is converted into the corresponding digital equivalent data and the equivalent data is stored in the memory locations. The time delay between the levels is fixed and is generated in the delay routines. It determines the frequency of the wave.

In the software part, the segment register DS and offset register BX are initialized with the stating address of the memory location. The AH register is initialized with a count value equal to number of bytes to be sent to the port A. The data from the memory is taken out and sent to the port A. The offset register BX is

incremented and then counter register is decremented. If the counter register has not become 0, then the looping is done to send the next data to the port A. This is done after the required time delay.

Program:

```
        MOV AL, 80H     ; Load the control word for the 8255.
        OUT 86H, AL     ; Send it to control register of the 8255.
        MOV BX, 2000H   ; Initialize BX with the segment address
                          2000H.
        MOV DS, BX      ; Move the segment address to DS.
L1:     MOV AH, 51H     ; Initialize AH register with the count value
                          (i.e., no. of data).
        MOV BX, 1000H   ; Initialize BX with the offset address 1000H.
L2:     MOV AL, [BX]    ; Load the data at [BX] in memory into AL.
        OUT 80H, AL     ; Output the data to Port A, where DAC is
                          interfaced.
        CALL DELAY      ; Call the delay subroutine.
        INC BX          ; Point to next location.
        DEC AH          ; Decrement count value in AH.
        JNZ L2          ; If the count is not zero then loop back.
        JMP L1          ; Jump to L1 to generate the next cycle.
DELAY:  MOV CX, COUNT   ; Delay subroutine starts
L2:     NOP             ; No operation.
        NOP             ; No operation.
        LOOP L2         ; Execute L2 CX times.
        RET             ; Return.
```

7.7 INTERFACING STEPPER MOTORS

A stepper motor is a special motor that rotates in incremental steps, unlike other motors that run continuously. They find application in printers, plotters, robots, etc. Stepper motors are excited by pulses to get incremental displacements. The common step size of stepper motors ranges from 0.9° to 30°. Stepper motors are made of permanent magnet rotors with stator field excitation. Two-phase excitation and four-phase excitation are common. A four-phase stepper motor has four stator poles, which are excited by pulses. Each pole winding can be excited such that the pole can be made either a north pole or a south pole. The number of teeth or the number of poles in the rotor will decide the minimum incremental step angle when a particular phase is excited. In this arrangement, the poles should be properly excited in a particular sequence so that the rotor rotates in a particular direction. If the excitation sequence is reversed, the rotor rotates in the reverse direction. A typical stepper motor has a step angle of 1.8°. This motor has 50 teeth on the rotor and eight poles on the stator.

The interfacing of four-phase stepper motor to the 8086 through the 8255 is given in Fig. 7.25.

The stepper motor has six terminals—four terminals A, B, C, and D for

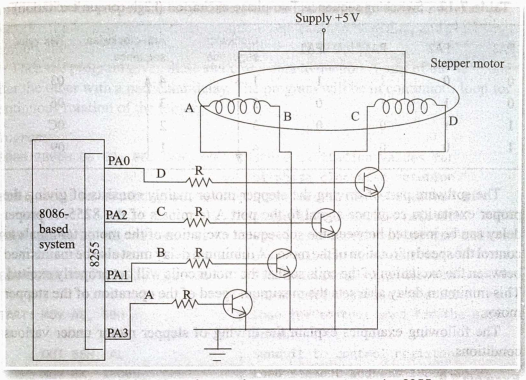


Fig. 7.25 Interfacing of stepper motor using the 8255

excitation and two more terminals for power supply. Figure 7.25 shows the four terminals A, B, C, and D connected to the 8255 ports through the transistor drivers. The transistor drivers or buffers are essential as the port pins cannot directly source the current required for the motor drive. As explained earlier, the motor terminals have to be excited in a proper sequence, so that rotor will have continuous rotation in one direction. Two types of excitation are possible with a four-phase motor—one-phase excitation and two-phase excitation. In one-phase excitation, only one phase of the stepper motor is excited at a time and in two-phase excitation, two phases are excited at a time. The exciting sequence is fixed for a rotation in a particular direction. The excitation sequence for the interface diagram in Fig. 7.25 is given in Tables 7.11 and 7.12. The single phase excitation results in low current through the motor windings and it is also called *wave mode*. In two-phase excitation, the excitation current through the motor winding is high and so it is called *high-torque excitation*. Tables 7.11 and 7.12 also show the corresponding hexadecimal bytes value to be given to the port A assuming that the higher-order four bits of the data are zero.

Table 7.11 Switching sequence: One-phase excitation (Wave mode)

PA3	PA2	PA1	PA0	Clockwise sequence	Anti-clockwise sequence	Hex value
0	0	0	1	1	4	01
0	0	1	0	2	3	02
0	1	0	0	3	2	04
1	0	0	0	4	1	08

Table 7.12 Switching sequence: Two-phase excitation (High-torque excitation)

PA3	PA2	PA1	PA0	Clockwise sequence	Anti-clockwise sequence	Hex value
0	0	1	1	1 ↓	4 ↑	03
0	1	1	0	2 ↓	3 ↑	06
1	1	0	0	3 ↓	2 ↑	0C
1	0	0	1	4 ↓	1 ↑	09

The software part of driving the stepper motor mainly consists of giving the proper excitation sequence signal to the port A terminals of the 8255. A proper delay can be inserted between the subsequent excitation of the motor terminals to control the speed of rotation of the motor. A minimum delay must also be maintained between the excitation of the coils so that the motor coils will get properly excited. This minimum delay also sets the maximum speed of the operation of the stepper motor.

The following examples explain the driving of stepper motor under various conditions.

Example 7.9

Write a program to drive the stepper motor continuously at 60 rpm using the interface diagram in Fig. 7.25. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Solution:

It is assumed that a stepper motor with a step angle of 1.8° is interfaced in Fig. 7.25. Each step rotation is 1.8°. If we control the time delay for each step, then the speed of the motor can be controlled. Let us assume that the required speed is N rpm. Then the speed in revolution per second (rps) is $N/60$. Hence, the time taken for one revolution will be $60/N$ seconds. Hence, for 360° rotation, the time taken is $60/N$ seconds. Therefore, the time taken for 1.8° rotation is $60 \times 1.8/(N \times 360)$, which is equivalent to $0.3/N$ seconds. If the time delay introduced for each 1.8° rotation is $0.3/N$ seconds, then for continuous rotation the speed will be N rpm. Here the required speed N is 60 rpm. Hence, the time delay required is 0.3/60, that is, 5 ms.

The following program is used for the continuous rotation of the stepper motor. The count in the delay routine must be calculated to produce the required time delay. The program first initializes the control word for the IC 8255. Then a counter of four is set up to indicate that the switching sequence needs four steps, which is to be repeated continuously. The segment and offset registers are then initialized to load the switching or excitation data to be given to port A of the 8255. The switching data is stored initially in the memory locations as a table. The following program shows four different sets of switching tables stored in different locations starting from the address 1000H:2000H in memory. The offset register has to be

initialized to the proper memory address to run in a particular mode. For example, to excite the stepper motor coils for one-phase excitation method and clockwise rotation, the offset register must be initialized with 2008H.

Then the program gives these switching data to the port A pins of the 8255 one after the other with a particular delay. The program will be in continuous loop for continuous rotation of the motor.

Program:

```
1000H:2000H DB 03, 06, 0CH, 09   ; Store excitation values for
                                   bi-phase clockwise rotation.
1000H:2004H DB 09, 0CH, 06, 03   ; Store excitation values for bi-
                                   phase anticlockwise rotation.
1000H:2008H DB 01, 02, 04, 08    ; Store excitation values for one-
                                   phase clockwise rotation.
1000H:200CH DB 08, 04, 02, 01    ; Store excitation values for one-
                                   phase anticlockwise rotation.
START: MOV AL, 80H               ; Load the control word for the 8255
                                   in AL.
       OUT 86H, AL               ; Send it to control register.
       MOV BX, 1000H             ; Load segment address 1000H in BX.
       MOV DS, BX                ; Load segment address 1000H in DS.
L1:    MOV DH, 04 H              ; Initialize the counter DH with 4
                                   for four excitation sequences to be
                                   sent.
       MOV BX, 2008H             ; Load offset address in BX for one
                                   phase excitation and clockwise
                                   rotation.
RPT:   MOV AL, [BX]              ; Get one excitation data from the
                                   memory.
       OUT 80H, AL               ; Send it to port A.
       CALL DELAY                ; Call the delay routine.
       INC BX                    ; Point to next memory location.
       DEC DH                    ; Decrement counter DH.
       JNZ RPT                   ; If it is not zero, then get next
                                   data by going to RPT.
       JMP L1                    ; Jump to L1 to start from the
                                   beginning.
DELAY: MOV CX, COUNT             ; Load count in CX.
L2:    NOP                       ; No operation.
       NOP                       ; No operation.
       LOOP L2                   ; Execute loop L2 CX times.
       RET                       ; Return to main program.
```

Example 7.10

Write a program to rotate a stepper motor by 180° using the interface diagram in Fig. 7.25. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port

A, port B, port C, and the control register of the 8255, respectively. The motor coils have to be excited by two-phase excitation method and for anti-clockwise rotation.

Solution:

In most of the stepper motor applications, the stepper motor will not be rotated continuously. The angular rotation of the stepper motor needs to be controlled. In this example, it is assumed that the stepper motor shaft has to be rotated by exactly 180°. This can be done by stopping the motor excitation when the motor shaft has rotated 180°. This is possible because each switching state of the motor coils rotates the shaft by exactly the step angle, that is, 1.8°. For 180° rotation, the number of switching steps required is 180/1.8 = 100. Out of this 100 switching steps, four switching sequences are repeated. So, we need a count of 100/4 = 25. A count of 25 is used for exciting the motor coils with four complete step sequences, then excitation is stopped and the motor rotation is halted. The following program implements this by adding another loop with DL register as a counter. The program uses the same logic and the instructions used in Example 7.9.

Program:

```
1000H:2000H DB 03, 06, 0CH, 09   ; Store excitation values for bi-
                                   phase-clockwise rotation.
1000H:2004H DB 09, 0CH, 06, 03   ; Store excitation values for bi-
                                   phase-anticlockwise rotation.
1000H:2008H DB 01, 02, 04, 08    ; Store excitation values for one-
                                   phase clockwise rotation.
1000H:200CH DB 08, 04, 02, 01    ; Store excitation values for one-
                                   phase anticlockwise rotation.
START: MOV AL, 80H               ; Load the control word for 8255 in
                                   AL.
       OUT 86H, AL               ; Send it to control register.
       MOV DL, 19H               ; Initialize the counter DL with 19H
                                   or 25D for 180º rotation.
L1:    MOV DH, 04H               ; Initialize the counter DH with 4
                                   for four excitation sequences.
       MOV BX, 1000H             ; Load segment address 1000H in BX.
       MOV DS, BX                ; Load segment address 1000H in DS.
       MOV BX, 2004H             ; Load offset address in BX for two-
                                   phase excitation and anticlockwise
                                   rotation.
L2:    MOV AL, [BX]              ; Get one excitation data from the
                                   memory.
       OUT 80H, AL               ; Send it to port A.
       CALL DELAY                ; Call the delay routine.
       INC BX                    ; Point to next memory location.
       DCR DH                    ; Decrement counter DH.
```

```
        JNZ L2                  ; If DH is not 0, go to L2 to send
                                  next excitation data.
        DCR DL                  ; Decrement counter DL.
        JNZ L1                  ; If DL is not 0, go to L1 to repeat
                                  the sequence.
        HLT                     ; Halt
DELAY:  MOV CX, COUNT           ; Load count in CX.
L3:     NOP                     ; No operation.
        NOP                     ; No operation.
        LOOP L3                 ; Execute loop L3 CX times.
        RET                     ; Return to main program.
```

7.8 INTERFACING INTELLIGENT LCDs

Many alphanumeric liquid crystal displays (LCDs) are available in the market. These displays have an in-built controller IC and a display section. These displays can be easily interfaced to any microprocessor or microcontroller. The data displayed in LCDs can be easily controlled and changed.

Liquid crystal displays are created by placing a thin layer of liquid crystal fluid between two glass plates. Transparent electrically conductive films are pasted on the front and back glass plates in the shape of the character to be displayed. When a voltage is applied between these two films, the electric field changes the behaviour of the liquid crystal and hence the light is transmitted through or reflected by it. Hence, the required display becomes visible.

Modern LCDs come with a controller IC and related control inputs. They get the American Standard Code for Information Interchange (ASCII) code of the data to be displayed and display the character in the exact location. The LCDs come with many options such as 8–80 characters display, single line, two–line, or four-line display, etc. Almost all these devices have 14 pins for interfacing with microprocessor or microcontroller. The functions of these 14 pins are listed in Table 7.13. Three pins E, R/$\overline{W}$, and RS are used for the control and handshake signals. Eight pins are used for transferring data to the display and can be connected to the data bus of the system. Two pins are allotted for supply and ground and one pin is used for adjusting the contrast of the display. The voltage applied to this pin can be varied to adjust the contrast.

The LCD consists of an internal RAM for storing the data to be displayed. The control signal, register select (RS) given as input to the LCD indicates whether the data available on the data lines is a command or a display data. RS is made zero to indicate that a command to the LCD is being sent on the data lines. The RS line is made high to indicate that the data lines contain display data that is to be read or written. Read/write signal (R/$\overline{W}$) is also an input control signal given to the LCD to indicate the direction of data transfer. Enable signal is the control input to the LCD and must be pulsed to perform the read or write operation with the LCD. A 1 to 0 transition on this line enables the corresponding operation decided by other control inputs.

Table 7.13 LCD pin configuration

Pin no.	Symbol	Level	I/O	Function
1	V_{SS}	–	–	Power supply (GND)
2	V_{CC}	–	–	Power supply (+5 V)
3	V_{ee}	–	–	Contrast adjust (V_0)
4	RS	0/1	I	Command or data register select line 0 = Command or instruction 1 = Data input
5	R/$\overline{W}$	0/1	I	0 = Write to LCD module 1 = Read from LCD module
6	E	1 to 0	I	Enable signal
7	DB0	0/1	I/O	Data bus line 0 (LSB)
8	DB1	0/1	I/O	Data bus line 1
9	DB2	0/1	I/O	Data bus line 2
10	DB3	0/1	I/O	Data bus line 3
11	DB4	0/1	I/O	Data bus line 4
12	DB5	0/1	I/O	Data bus line 5
13	DB6	0/1	I/O	Data bus line 6
14	DB7	0/1	I/O	Data bus line 7 (MSB)

The common commands used in the LCD units are given in Table 7.14. The display can be cleared by issuing a command with LSB alone as 1. Similarly cursor control, display position control, and display method control can be done using appropriate control words.

Table 7.14 LCD command words

Command/ Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	Clears display and returns cursor to the home position (address 0)
Cursor home	0	0	0	0	0	0	0	0	1	x	Returns cursor to home position (address 0). Display RAM contents remain unchanged
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction (I/D) and specifies whether to shift the display (S). These operations are performed during data read/write. For I/D, 1 = increment; 0 = decrement

(Contd)

Table 7.14 LCD command words *(Contd)*

Command/ Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets on/off position of all displays (D) and cursors (C) and blink of character at cursor position (B)
Cursor/ display shift	0	0	0	0	0	1	S/C	R/L	x	x	Sets move cursor—0 or shift display—1(S/C), shifts direction (R/L) left—0 or right—1 DDRAM contents remain unchanged.
Function set	0	0	0	0	1	DL	N	F	x	x	Sets interface data length (DL)—1 for 8-bit data and 0 for 4-bit data, number of display lines (N)—0 for one-line and 1 for two-line display, and character font (F)—1 for 5 × 10 dot and 0 for 5 × 7 dot font
Set CGRAM address	0	0	0	1	CGRAM address						Sets the CGRAM address; CGRAM data is sent and received after this setting.
Set DDRAM address	0	0	1	DDRAM address							Sets the DDRAM address; DDRAM data is sent and received after this setting.
Read busy flag and address counter	0	1	BF	CGRAM/DDRAM address							Reads busy flag (BF), which indicates that internal operation is being performed and reads CGRAM or DDRAM address counter contents (depending on previous instruction).
Write to CGRAM or DDRAM	1	0	Write data								Writes data into CGRAM or DDRAM
Read from CGRAM or DDRAM	1	1	Read data								Reads data from CGRAM or DDRAM

Note: x denotes don't care condition.

The interfacing of an LCD to the 8255 is shown in Fig. 7.26 and the characters displayed for different ASCII data is shown in Fig. 7.27. The 8255 in turn is interfaced to the 8086 in the same way as shown in Fig. 7.9. Port A is connected to

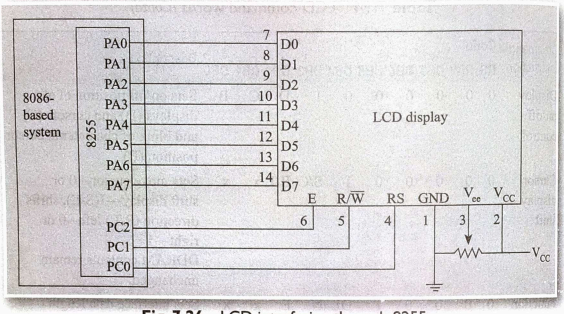


Fig. 7.26 LCD interfacing through 8255

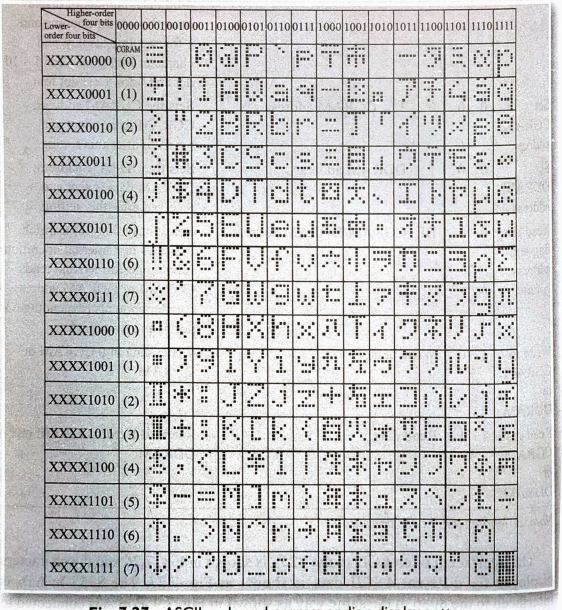


Fig. 7.27 ASCII code and corresponding display pattern

the data lines of the LCD display and port C is connected to the control lines. PC2 is connected to the enable line (E) of the LCD. The RS signal is connected to the port pin PC0. Port pin PC1 is connected to the R/$\overline{W}$ control signal.

The programming part of the LCD interface is as follows. The LCD display works with its own internal clock pulses. So, any command or data written to the LCD display must be enabled with enable (E) signal. This signal must be applied for predefined time duration. Each command and data requires a minimum of 40 microseconds to about 1.6 milliseconds depending upon the type of the LCD and its clock frequency. So, a separate subroutine is written to give proper control signals for the predefined delay time. Here, two subroutines COMMAND and DISP are written to write a command word and a data for display respectively. These two subroutines use a delay routine commonly.

The first step in the program is to clear the display and then to set the cursor to home position and start displaying data from there. The following program displays an array of characters stored in memory locations starting from the address 1000H: 2000H. The number of characters displayed is 16 and it is initialized as a count in the AH register. All the characters are displayed continuously. Assume that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

Program:

```
START:  MOV AL, 01H      ; Load the command word in AL to clear display
                           in LCD.
        CALL COMMAND     ; Call the subroutine to issue this command
                           to LCD.
        MOV AL, 02H      ; Load the command word to initialize the
                           cursor to home position and start displaying
                           data from there.
        CALL COMMAND     ; Call the subroutine to issue this command
                           to LCD.
        MOV BX, 1000H    ; Initialize the segment register DS to
                           1000H.
        MOV DS, BX
        MOV BX, 2000H    ; Initialize the offset register BX to
                           2000H.
        MOV AH, 16       ; Initialize the counter AH for number of
                           characters to be displayed.
NEXT:   MOV AL, [BX]     ; Get the display data in Accumulator (AL).
        CALL DISP        ; Call the subroutine to issue this data to
                           LCD.
        INC BX           ; Increment memory pointer to send next data
                           for display.
        DEC AH           ; Decrement the counter AH.
        JNZ NEXT         ; If AH is not zero, loop again.
        HLT              ; Otherwise halt.
```

```
COMMAND: OUT 80H, AL      ; Send the command word to the data lines of
                            LCD.
         MOV AL, 04H      ; Load data with E = 1, R/W = 0 (write) and
                            RS = 0 (command) in AL.
         OUT 84H, AL      ; Send data in AL to port C pins PC2, PC1,
                            and PC0.
         MOV AL, 00H      ; Load data in AL to make the E signal (PC2)
                            zero.
         OUT 84H, AL      ; Output this data to port C.
         CALL DELAY       ; Wait for predefined time delay.
         RET              ; Return.
DISP:    OUT 80H, AL      ; Send the data in AL to the data lines of
                            LCD.
         MOV AL, 05       ; Make data with E = 1, R/W = 0 (write) and
                            RS = 1 (data).
         OUT 84H, AL      ; Give data in AL to Port C pins PC2, PC1,
                            and PC0.
         MOV AL, 00H      ; Make the E signal (PC2) zero.
         OUT 84H, AL      ; Output this data to Port C.
         CALL DELAY       ; Wait for predefined time delay.
         RET              ; Return.
DELAY:   MOV CX, COUNT    ; Move count to CX.
L1:      NOP              ; No operation.
         NOP              ; No operation.
         LOOP L1          ; Execute loop L1 CX times.
         RET              ; Return.
```

7.9 KEYBOARD AND DISPLAY INTERFACE IC 8279

The interfacing of keys and displays with the 8086 processor has been discussed in Sections 7.2 and 7.3. It is seen that for displaying data, an external IC such as the 8255 is necessary. Similarly, for key interfacing, a port is necessary and the processor needs to check the inputs from the keys for identifying whether any key is pressed or not. Moreover, these simple interface circuits become more complicated when more number of display units and switches are interfaced to the processor. To reduce the hardware for large keyboard interfacing, the matrix keyboard concept is used. To interface more number of displays, the multiplexed display concept is used. This section introduces the matrix keyboard and the multiplexed display concepts. These circuits require a large amount of processing time. To reduce the processor involvement in the matrix keyboard scanning and multiplexed display, Intel has produced a dedicated IC 8279. This IC will relieve the processor from keyboard and multiplexed display scanning. The hardware details about this IC and the programming of this IC are discussed in this section.

7.9.1 Matrix Keyboard

The interface of a single key switch to the 8086 through the 8255 has been

discussed earlier. The limitation of this interface is that each switch must be allotted a separate port pin. The hardware complexity increases as more number of switches is interfaced. The solution to this problem is using a matrix keyboard. The matrix keypad has more number of switches organized or connected in matrix form and is interfaced with the 8086 using the 8255 as shown in Fig. 7.28. This figure shows a 4 × 4 key matrix. The rows of the keyboard matrix are connected to the four output port lines. The column lines are connected to the four port lines of an input port.

When no key is pressed, the column lines are connected to the (+5V) Vcc line and hence the data at all the input port pins will be 1. If a low or logic 0 is output on a single output port bit and if any switch in that row is pressed, then the corresponding column data bit will become logic 0, while other column bits will be at logic 1. For example, let us assume that D0 of the output port is made 0 while other bits of output port are at logic 1. If key 3 is pressed, then the input to D3 bit of the input port alone becomes 0 while D0–D2 input bits are at logic 1. So, by making D0 output line to 0, we can detect which key is pressed in the corresponding row.

To check the entire keyboard for a key press, the following technique can be used. One of the rows is made logic 0 and the input lines along the columns are

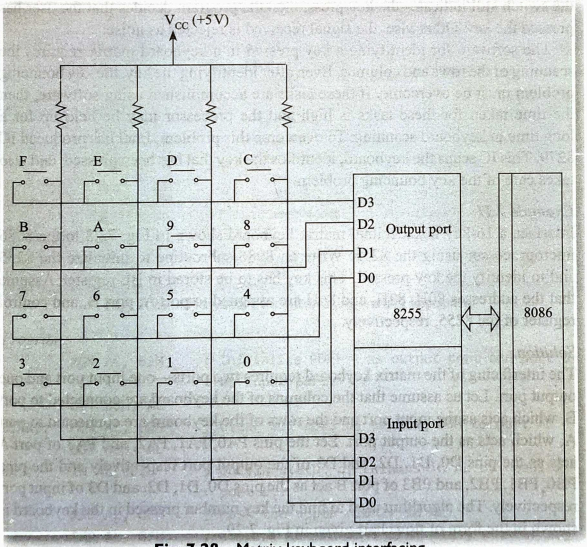


Fig. 7.28 Matrix keyboard interfacing

checked for occurrence of 0s. If there is none, the next row is made logic 0 and the procedure is repeated, until the key pressed is identified.

Once a key is pressed, then steps must be taken to remove the contact bounce problem. Pressing a mechanical switch must produce a single pulse output. Practically, instead of producing a single clean pulse output, the switches generate a series of pulses because the switch contacts do not come to rest immediately. As the microprocessor is faster than manual pressing of keys, the single key pressed will be registered as multiple key presses. This is the main disadvantage of key bouncing. The signals from keys fall and rise a few times within a period of about 5 ms as the contact bounces. Hence, the signal from the key must be made free from key bouncing transients. This technique is called debouncing of keys.

Key board debouncing can be accomplished using hardware or software. The bouncing of the key signal occurs within 5 ms. Since a human can not press and release a switch in less than 20 ms, the debouncing logic will check the signal after 20 ms and then recognize whether a key is pressed or not. This logic can be implemented in hardware or software. The hardware techniques employ set–reset flip-flops, non-inverting CMOS gates, or integrating debouncer. The software technique uses the wait-and-see method. When the signal from a switch is sensed, the program waits for 10 ms and look at the same key again. If the signal from the switch still indicates the key press, then the program decides that the user has pressed the key. Otherwise, the signal received is rejected as noise.

The software for identifying a key pressed in a keyboard matrix requires the scanning of the rows and columns. Even after identifying the key, the key bouncing problem must be overcome. If these tasks are accomplished using software, then the time taken for these tasks is high and the processor may be held up for a long time in keyboard scanning. To overcome this problem, Intel has produced IC 8279. This IC scans the keyboard, identifies the key that has been pressed, and also takes care of the key bouncing problem.

Example 7.11

Interface a 16-key hexadecimal matrix keyboard shown in Fig. 7.28 to the 8086 microprocessor using the 8255. Write an 8086 subroutine to initialize the 8255 and to identify the key pressed. This key has to be stored in BL register. Assume that the addresses 80H, 82H, and 86H are assigned to port A, port B, and control register of the 8255, respectively.

Solution:

The interfacing of the matrix keyboard requires two ports—one input port and one output port. Let us assume that the columns of the keyboard are connected to port B, which acts as the input port and the rows of the keyboard are connected to port A, which acts as the output port. Let the pins PA0, PA1, PA2, and PA3 of port A acts as the pins D0, D1, D2, and D3 of the output port respectively and the pins PB0, PB1, PB2, and PB3 of port B act as the pins D0, D1, D2, and D3 of input port respectively. The algorithm used to find the key number pressed in the keyboard is shown in the form of flowchart given in Fig. 7.29.

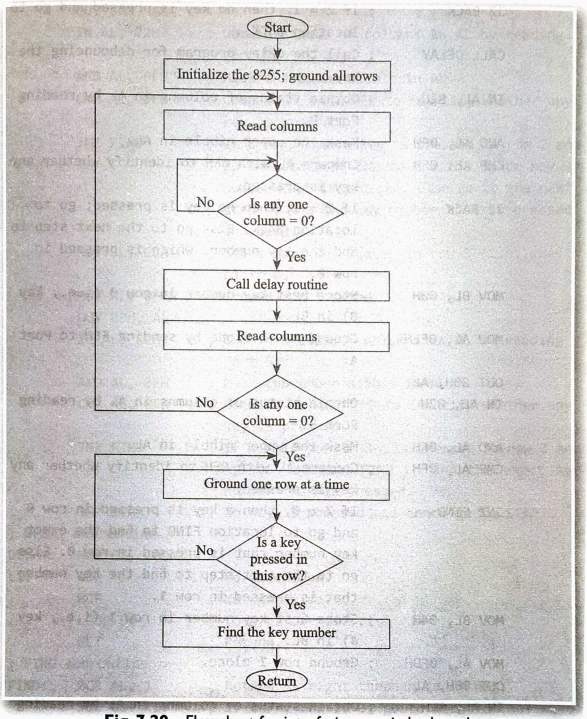


Fig. 7.29 Flowchart for interfacing matrix keyboard

Program:

```
        MOV AL, 82H     ; Initialize PORT A as output port and port B
                          as input port.
        OUT 86H, AL
START:  MOV AL, 00H     ; Clear all rows by sending 00H to port A.
        OUT 80H, AL
BACK:   IN AL, 82H      ; Obtain status of columns in AL by reading
                          port B.
        AND AL, 0FH     ; Mask the upper nibble in AL.
        CMP AL, 0FH     ; Compare AL with 0FH to identify whether any
                          key is pressed.
```

```
JZ BACK          ; If Z = 1, then no key is pressed and go to
                   location BACK.
CALL DELAY       ; Call the delay program for debouncing the
                   key.
IN AL, 82H       ; Obtain status of columns in AL by reading
                   Port B.
AND AL, 0FH      ; Mask the upper nibble in AL.
CMP AL, 0FH      ; Compare AL with 0FH to identify whether any
                   key is pressed.
JZ BACK          ; If Z = 1, then no key is pressed; go to
                   location BACK. Else go to the next step to
                   find the key number, which is pressed in
                   row 0.
MOV BL, 00H      ; Store first key number in row 0 (i.e., key
                   0) in BL.
MOV AL, 0FEH     ; Ground row 0 alone by sending FEH to Port
                   A.
OUT 80H, AL
IN AL, 82H       ; Obtain status of columns in AL by reading
                   Port B.
AND AL, 0FH      ; Mask the upper nibble in AL.
CMP AL, 0FH      ; Compare AL with 0FH to identify whether any
                   key is pressed.
JNZ FIND         ; If Z = 0, then a key is pressed in row 0
                   and go to location FIND to find the exact
                   key number that is pressed in row 0. Else
                   go to the next step to find the key number
                   that is pressed in row 1.
MOV BL, 04H      ; Store first key number in row 1 (i.e., key
                   4) in BL.
MOV AL, 0FDH     ; Ground row 1 alone.
OUT 80H, AL
IN AL, 82H       ; Obtain status of columns in AL by reading
                   port B.
AND AL, 0FH      ; Mask the upper nibble in AL.
CMP AL, 0FH      ; Compare AL with 0FH to identify whether any
                   key is pressed.
JNZ FIND         ; If Z = 0, then a key is pressed in row 1 and
                   go to location FIND to find the exact key
                   number that is pressed. Else go to the next
                   step to find the key number that is pressed
                   in row 2.
MOV BL, 08H      ; Store first key number in row 2 (i.e., key
                   8) in BL.
MOV AL, 0FBH     ; Ground row 2 alone.
```

```
        OUT 80H, AL
        IN AL, 82H      ; Obtain status of columns in AL by reading
                          Port B.
        AND AL, 0FH     ; Mask the upper nibble in AL.
        CMP AL, 0FH     ; Compare AL with 0FH to identify whether any
                          key is pressed.
        JNZ FIND        ; If Z = 0, then a key is pressed in row 2 and
                          go to location FIND to find the exact key
                          number that is pressed. Else go to the next
                          step to find the key number that is pressed
                          in row 3.
        MOV BL, 0CH     ; Store first key number in row 3 (i.e., key
                          C) in BL.
        MOV AL, 0F7H    ; Ground row 3 alone.
        OUT 80H, AL
        IN AL, 82H      ; Obtain status of columns in AL by reading
                          Port B.
        AND AL, 0FH     ; Mask the upper nibble in AL.
        CMP AL, 0FH     ; Compare AL with 0FH to identify whether any
                          key is pressed.
        JNZ FIND        ; If Z = 0, then a key is pressed in row 3 and
                          go to location FIND to find the exact key
                          number that is pressed.
        JMP START       ; No key is pressed and hence go to START.
; DELAY subroutine
DELAY:  MOV CX, COUNT
L1:     NOP             ; No operation.
        NOP             ; No operation.
        LOOP L1         ; Loop L1 CX times.
        RET             ; Return.
; FIND subroutine
FIND:   RCR AL, 1       ; Rotate AL right through carry flag.
        JNC GOT_KEY     ; If carry flag is 0, go to location GOT_KEY.
        INC BL          ; Increment BL to hold the next key number in
                          that row.
        JMP FIND        ; Go to location FIND.
GOT_KEY: RET            ; Return from subroutine with key number in
                          BL.
```

7.9.2 Multiplexed Display

The seven-segment display discussed in Section 7.3 uses individual port pins for each display. This requires one port for each seven-segment display. In order to reduce the hardware complexity for more number of display devices, matrix display method is used. Here with two ports, as many as eight display units can be interfaced. Such an arrangement where four display units (of common anode type) are connected to the 8086 through the 8255 is shown in Fig. 7.30.

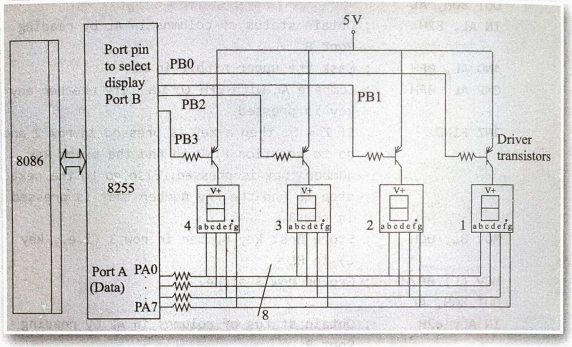


Fig. 7.30 Multiplexed display arrangements

The pins of each segment for all the display devices are tied together and connected to any one data port pin. One display unit alone is selected using another port pin through the driver transistor. This selection is done by using an active low pin output in the corresponding port. However, the user must see the output on all the display units simultaneously. This is done by displaying the data in quick succession in all the display units. Due to the persistence of vision, the human eye holds the display image in it and the user sees all the display units illuminated simultaneously. As long as the displays are turned on and off fast enough, the eye will perceive them as being illuminated all at the same time.

The software part for the matrix display consists of selecting one display unit and repeating the same procedure for all the display units at a faster rate. The timer can be used to control the rate of display and refreshing the display. This takes a lot of processor time. To overcome this problem, IC 8279 is used.

Example 7.12

Interface a 4-digit multiplexed seven-segment LED display using the 8255 to the 8086 microprocessor system and write an 8086 assembly language routine to display message on the display. The different addresses assigned for various ports in the 8255 are: port A—00H, port B—02H, port C—04H, and control register—06H. The seven-segment code of the message to be displayed and the data to be sent to the digit driver port (to select a particular seven-segment LED display) are alternatively stored in eight memory locations (i.e., four bytes for storing four seven-segment codes and four bytes for storing 4-digit driver codes) starting from the address 1000H:2000H.

Solution:

The multiplexed seven-segment display connected in the 8086 system using the 8255 is shown in Fig. 7.30. Hence, two ports are used for interfacing four seven-

segment LED displays. Let us assume that port A provides the segment data inputs (i.e., seven-segment code) to the display and port B provides a digit driver code for selecting a particular seven-segment display. In this example, ports A and B are used as simple latched output ports. In this type of display system, only one of the four display position is on at any given instant. Only one digit of the display is selected at a time by giving a low signal on the corresponding port B line. Let the pins PA0, PA1, PA2, ..., PA7 act as the pins D0, D1, D2, ..., D7 of the data port, respectively and the pins PB0, PB1, PB2, and PB3 act as the pins D0, D1, D2, and D3 of the display select port, respectively.

For the 8255, ports A and B are used as output ports. The control word format of the 8255 according to the assumed hardware connections is 80H.

Program:

```
        MOV AL, 80H     ; Load control word in AL.
        OUT 06H, AL     ; Send control word to control register of the
                          8255.
        MOV BX, 1000H   ; Load segment address 1000H in DS.
        MOV DS, BX
DISP:   MOV BX, 2000H   ; Load offset address 2000H in BX.
        MOV CX, 04H     ; Load count of 4 in CX due to the presence of 4
                          displays.
L1:     MOV AL, [BX]    ; Move one seven-segment code from memory into AL.
        OUT 00H, AL     ; Send data in AL to Port A.
        INC BX          ; Increment BX.
        MOV AL, [BX]    ; Get data to be sent to digit driver port from
                          memory.
        OUT 02H, AL     ; Send data in AL to Port B to select a particular
                          display.
        CALL DELAY      ; Wait for some time.
        INC BX          ; Increment BX to point next memory location.
        LOOP L1         ; Execute loop L1 four times.
        JMP DISP        ; Go to DISP to repeat the steps from
                          beginning.
DELAY:  MOV DX, COUNT   ; Delay subroutine.
L2:     NOP             ; No operation.
        NOP             ; No operation.
        DEC DX          ; Decrement DX.
        JNZ L2          ; If not zero, jump to L2.
        RET             ; If zero, return.
```

7.9.3 Features, Block Diagram, and Pin Details of 8279

The IC 8279 is a programmable keyboard and display interface controller IC, designed by Intel for use with Intel microprocessors. The major features of this IC are as follows:

(i) Supports keyboard of size up to 64-key matrix with 2-key lockout or *N*-key rollover options

(ii) Supports a display interface of up to 16 digits with many options
(iii) Simultaneous keyboard and display operations
(iv) 8-character first in first out (FIFO) memory to store codes of keys pressed
(v) 16-byte display RAM corresponding to 16 digits of display

The block diagram of the 8279 is shown in Fig. 7.31.

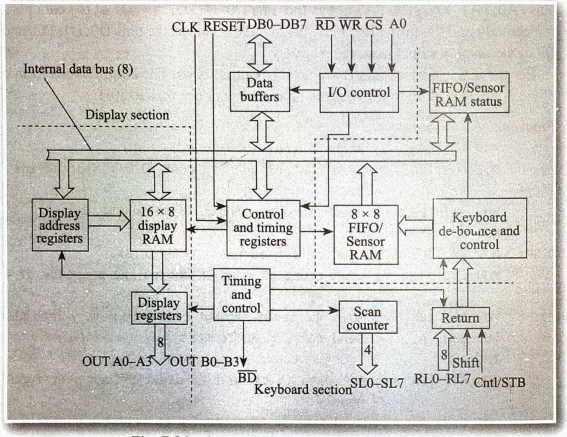


Fig. 7.31 Internal block diagram of IC 8279

IC 8279 has the following three sections:
(i) Display section with its own display RAM.
(ii) Keyboard scan section with FIFO registers.
(iii) Control logic with signals for interfacing with the processor.

The control section consists of a data bus buffer for interfacing to the processor. This I/O section uses the control signals such as A0, $\overline{CS}$, $\overline{RD}$, and $\overline{WR}$. The active low control signal $\overline{CS}$ is used to select the IC. Similarly, $\overline{RD}$ and $\overline{WR}$ are active low control signals for indication of direction of data transfer on the data bus, DB0–DB7. The signal A0 is used to select a data or control register. A logic 1 on the A0 line means that the data bus content is a command or status. A logic 0 means that the data bus content is the data for the IC 8279. The control and timing registers store the keyboard and display modes and other operating conditions.

Even though there are many control and data registers, the 8279 uses only two addresses—one with A0 = 0 and other with A0 = 1. This is done using a unique control word for each operation. For example, two different control words are available for accessing display RAM and keyboard FIFO. For every operation, the

corresponding control word is written, the necessary register is accessed, and then the operation is carried out.

SL0–SL3 are the four scan lines of the 8279. There are two programmable options for the scan lines—encoded mode and decoded mode. In encoded mode, the SL0–SL3 lines are binary counter outputs and need to be decoded externally for scanning keyboards and displays. In the decoded mode, the SL0–SL3 outputs are decoded; one of the four lines has an active low output. The scan lines SL0–SL3 are common to both keyboards and displays. RL7–RL0 are the eight return lines and are used as inputs to sense a key press in the keyboard matrix.

The other signals available in the 8279 are as follows;

(i) $\overline{BD}$—output signal that blanks the displays
(ii) CLK—clock input used internally for timing, whose maximum clock frequency is 3MHz
(iii) CNTL/$\overline{STB}$—control or strobe signal, connected to the control key on the keyboard
(iv) Shift—connects to the shift key on the keyboard
(v) IRQ—interrupt request, becomes 1 when a key is pressed and data is available
(vi) OUT A3–A0/B3–B0—outputs that send data to the most significant/least significant nibble of display
(vii) RESET—connects to system RESET

The complete pin diagram of the 8279 is shown in Fig. 7.32.

7.9.4 Programming of 8279

IC 8279 can be programmed to select the number of displays, the type of key scan, the memory to write the display data into, a blank display, and the key code read option, and to control the interrupt request signal. All these operations or commands are written into the 8279 through the data bus with logic 1 on A0 line. The most significant three bits of the control word differentiate the operations. The first three bits of the byte sent to the control port selects one of eight control words which are listed in Table 7.15.

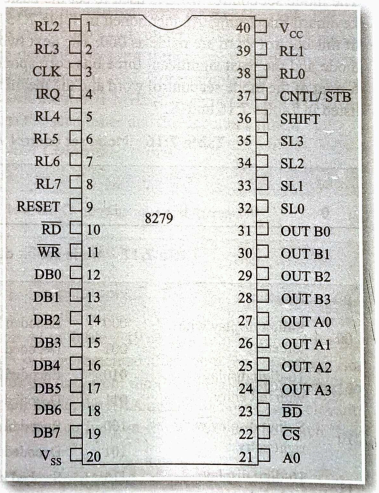


Fig. 7.32 Pin details of IC 8279

Table 7.15 Control word selection using the most significant three bits

D7	D6	D5	Function	Purpose
0	0	0	Mode set	Control word to select the number of displays, display position, and type of key scan
0	0	1	Clock	Control word to program the internal clock and set the scan and de-bounce times
0	1	0	Read FIFO	Control word to be written before reading the key code from FIFO
0	1	1	Read display	Control word written before reading the display RAM
1	0	0	Write display	Control word written in the control register before writing data to the display RAM
1	0	1	Display write inhibit	Control word to blank half-bytes.
1	1	0	Clear	Control word to clear the display and FIFO
1	1	1	End interrupt	Control word to clear the IRQ signal to the microprocessor

7.9.4.1 *Keyboard/Display Mode Set Control Word*

The mode set control word is used to set the basic control modes for display and keyboard interfacing. As mentioned in Table 7.15, the most significant three bits of this control word are made as 000. The next two bits correspond to the display mode and the least significant three bits correspond to the keyboard control. The format of the mode set control word and the meaning of the corresponding bits are listed in Tables 7.16 and 7.17.

Table 7.16 Mode set control word format

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	D	D	K	K	K

Table 7.17 Mode set bit definitions

DD	Function
00	8-digit display with left entry
01	16-digit display with left entry
10	8-digit display with right entry
11	16-digit display with right entry

KKK	Function
000	Encoded keyboard with 2-key lockout
001	Decoded keyboard with 2-key lockout
010	Encoded keyboard with *N*-key rollover
011	Decoded keyboard with *N*-key rollover
100	Encoded sensor matrix
101	Decoded sensor matrix
110	Strobed keyboard, encoded display scan
111	Strobed keyboard, decoded display scan

The display control word bits DD produce four options—8 bits or 16 bits, with calculator-like right entry or typewriter-like left entry.

The lines SL0–SL3 provide encoded and decoded output options for the keyboard interface. For encoded keyboard option, SL0–SL3 outputs are active-high in binary form. For decoded keyboard option, they are active-low with only one low output at any time. In strobed keyboard mode, an active high pulse on the CN/ST input pin strobes data from the RL pins into an internal FIFO. In 2-key lockout mode, the 8279 detects only one key pressed. If two keys are pressed simultaneously, then the key which is released last is considered. All the keys are debounced using the internal hardware delay for debouncing. In the case of *N*-key rollover, if two or more keys are pressed simultaneously, all the keys are sensed and stored in the FIFO according to the sequence in which the keys are recognized by the logic. In sensor matrix mode, the debounce logic is suppressed and any key sensed in the matrix is directly stored into the sensor RAM.

7.9.4.2 *Clock Signal Programming Command Word*

The clock signal programming command word format is given in Table 7.18.

Table 7.18 Clock signal programming command word format

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	P	P	P	P	P

The clock command word programs the internal clock driver. The code PPPPP shown in Table 7.18 corresponds to the binary code by which the input clock signal must be divided to achieve the desired operating frequency. With the five bits, division is possible by any number from two to 31. For example, for an operating frequency of 100 KHz and a clock input of 1 MHz, the count should be 01010B (10D). This control word decides the scan times and the debounce times.

7.9.4.3 *Read FIFO Sensor RAM Command Word*

The read FIFO command word format is given in Table 7.19.

Table 7.19 Read FIFO command word format

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	AI	0	A	A	A

The read FIFO control word selects the address (AAA) of a keystroke from the FIFO buffer (000 to 111). The bit AI in Table 7.19 selects auto-increment for the address. If AI is set to 1, the address is incremented after every read operation. So, continuous read operation fetches the data from the FIFO, one after the other, to the processor. In the scan keyboard mode, the AAA and AI bits become irrelevant. All data from the FIFO are read consecutively in the same order in which they were entered into the FIFO.

7.9.4.4 *Write Display RAM Command Word*

The display RAM command word format is given in Table 7.20.

Table 7.20 Display RAM command word format

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	AI	A	A	A	A
			Set to 1 for auto-increment option	Address of the 16-byte display RAM in 4 bits			

Writing this command to the command register sets the 8279 to get and store the data to be displayed in the display RAM. If AI is set to 1, the auto-increment option is implemented by which the address of the RAM is incremented automatically after every write operation. Data written with 0 in the address line A0 are written into subsequent RAM addresses, automatically incrementing the addresses.

7.9.4.5 *Other Command Words*

Other commands such as reading the display RAM, blanking display, clearing the display or FIFO, clearing the IRQ signal to the microprocessor may not be necessary for basic interfacing of a keyboard/display. Hence, a summary of these commands are provided in this section. More details of these commands can be obtained from the datasheet of the 8279. The read display RAM command word format is given in Table 7.21.

Table 7.21 Read display RAM command word

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	AI	A	A	A	A

The display RAM read control word selects the address of one of the display RAM positions and a subsequent read operation using A0 = 0 reads the data in that display RAM address.

The display write inhibit command word format is given in Table 7.22.

Table 7.22 Display inhibit/mask command word

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	0	I	I	M	M

The display write inhibit control word is used in applications where separate 4-bit display ports are used. One example using 4-bit data is a display which uses a BCD decoder. This control word is used to inhibit either the leftmost 4 bits of the display or the rightmost 4 bits using the I-I bits. Using this inhibit, it is possible to write a nibble into the display RAM without affecting the other digits being displayed. The masking of the bits MM is similar in operation to inhibit, but these bits selectively blank either the left-most or the right-most display.

The clear display command word format is given in Table 7.23.

Table 7.23 Clear display command word

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	1	CD	CD	CF	CA

The clear control word can clear the display RAM using CD bits. The setting of the D4 bit enables clearing of the display. This command has the option of making the display RAM all 0s by using 00 in D3 and D2 bits or all 1s by using 11 in D3 and D2. The setting of the CF bit clears the keyboard FIFO RAM and the CA bit clears both the display RAM and FIFO RAM.

The end interrupt command word format is given in Table 7.24.

Table 7.24 End interrupt command word

D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	E	0	0	0	0

End of interrupt command word is issued to clear the IRQ pin in sensor matrix mode.

7.9.4.6 *Keyboard Status Word Format*

To determine if a character has been typed, the FIFO status register is checked. The keyboard status word contains the status of FIFO, error, and display availability. This status word can be read from the 8279 when A0 is high. The status word format is given in Table 7.25. The least significant three bits are used to indicate the number of keys pressed and stored in the FIFO. The next bit F is used to indicate that the FIFO is full. Underrun error bit U is used to indicate a read attempt from the empty FIFO. Overrun error bit O is used when the entry into a full FIFO is attempted. S/E is used to indicate a multiple key press. The bit D is used to indicate the unavailability of the display.

Table 7.25 Keyboard status word format

D7	D6	D5	D4	D3	D2	D1	D0
D	S/E	O	U	F	N	N	N
					Number of keys pressed		

7.9.4.7 *Keyboard Code Word Format*

In the scanned keyboard mode, the character entered into the FIFO corresponds to the position of the switch in the key matrix and the status of the control and shift keys. The data read from the FIFO RAM has the format as given in Table 7.26. The MSB corresponds to the status of the control key while the next bit corresponds to the shift key. The next three bits are from the scan counter and indicate the row in which the key press was identified. The least significant three bits correspond to the column lines in which the key press was identified.

Table 7.26 Keyboard code word format

D7	D6	D5	D4	D3	D2	D1	D0
CTRL	SHIFT	S_2	S_1	S_0	R_2	R_1	R_0
		Encoded row position			Encoded column position		

7.9.5 Display Interface using 8279

The IC 8279 can be used to interface a maximum of 16 characters. The lines A0–A3 and B0–B3 are used to give the display data to the devices. The scan lines are used to select any one of the display devices.

The interfacing of seven-segment LEDs with the processor 8086 through the 8279 is shown in Fig. 7.33.

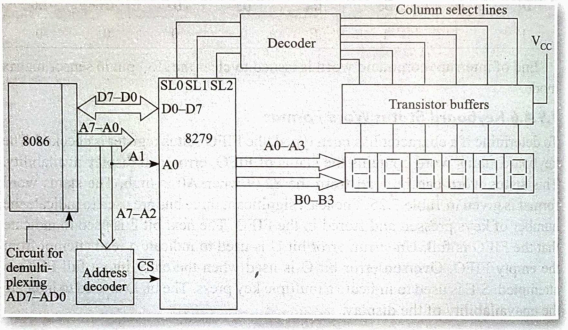


Fig. 7.33 Interfacing seven-segment display devices using IC 8279

The number of display devices used in this scheme is six. The seven-segment displays are all common anode type and a transistor driver is used with each display device. A PNP transistor drive is used to switch between the common anode and +5V supply similar to the drivers used in Fig. 7.30. A logic low is required to turn on the transistor driver and the same is generated using the decoder IC. Common decoder ICs such as IC 74138 can be used, as they can give an active low signal on any of their outputs. The segments of the display devices are all connected together on a common bus and connected to A3–A0 and B3–B0 outputs of the 8279. As the displays are all common anodes, the data output for illuminating an LED must be logic low. This means that the logic 1 on all the data lines A3–A0 and B3–B0 will blank the display and logic 0 in all these lines will display all the segments.

The software part for the display interface consists of initializing the 8279 for the encoded output and for 8-digit display. The writing of data to the display RAM

of the 8279 is enough to display data. The 8279 automatically scans and refreshes the display. The following program assumes that a 6-digit display is interfaced through the 8279 as shown in Fig. 7.33. Let us assume that the address of the command or status register is 42H and that of data register is 40H in the 8279.

Program:

```
MOV AL, 00H     ; Move mode set command word to AL.
OUT 42H, AL     ; Send out the same to command port.
MOV AL, 0C0H    ; Move clear display command to AL.
OUT 42H, AL     ; Send out the same to command port.
MOV AL, 90H     ; Move the write display RAM command word to AL.
OUT 42H, AL     ; Send out the same to command port.
MOV AL, DATA1   ; Move seven-segment display code for first digit
                  (DATA1)in AL.
OUT 40H, AL     ; Send out the same to data port.
MOV AL, DATA2   ; Move seven-segment display code for second digit
                  (DATA2) in AL.
OUT 40H, AL     ; Send out the same to data port.
MOV AL, DATA3   ; Move seven-segment display code for third digit
                  (DATA3) in AL.
OUT 40H, AL     ; Send out the same to data port.
MOV AL, DATA4   ; Move seven-segment display code for fourth digit
                  (DATA4) in AL.
OUT 40H, AL     ; Send out the same to data port.
MOV AL, DATA5   ; Move seven-segment display code for fifth digit
                  (DATA5) in AL.
OUT 40H, AL     ; Send out the same to data port.
MOVAL, DATA6    ; Move seven-segment display code for sixth digit
                  (DATA6) in AL.
OUT 40H, AL     ; Send out the same to data port.
HLT             ; Terminate program execution.
```

7.9.6 Keyboard Interface using 8279

The keyboard matrix that can be interfaced using the 8279 can be of any size from 2 × 2 to 8 × 8. Pins SL0–SL3 sequentially scan each column through a counting operation. The 74LS138 decoder IC can drive 0s on one line at a time from the SL0–SL3 lines. The 8279 scans the RL pins synchronously with the scan. The RL pins have internal pull-up resistors, and hence do not need external resistors. The 8279 does the three tasks of key board scan—placing a low in a scan line, checking for a low on the return lines, and detecting and debouncing the key pressed .

A key board matrix connected to the 8279 is shown in Fig. 7.34. Three column select lines SL0–SL2 are used to apply a low on any one column line and do consecutive scanning on all column lines. Any key pressed can be sensed by a low on the return lines. The 8279 does this scanning automatically and stores the key code into the FIFO RAM. In this example, the CNTL and SHIFT lines are not used and are connected to logic low.

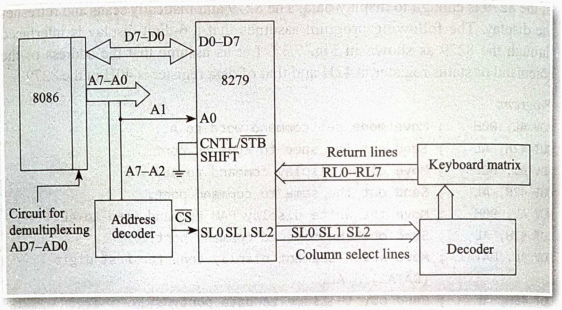


Fig. 7.34 Interfacing matrix keyboard using IC 8279

The software program using the 8086 mnemonics is as follows. Initially, the mode set command word is written into the command port. Here, the encoded scan keyboard with 2-key lock out mode is used. After writing the command word, IC 8279 starts scanning the key presses. Any key press can be sensed by reading the status word from the 8279 and then checking the least significant 3 bits. The following program checks for single key press and reads the key code from FIFO. The key code will be then stored in the memory location named KEY_CODE in the data segment. Let us assume that the address of the command or status register is 42H and that of the data register is 40H in the 8279.

Program:

```
    MOV AL, 00H          ; Move mode set command word to AL.
    OUT 42H, AL          ; Send out the same to command port.
L1: IN AL, 42H           ; Read the status word from the 8279.
    AND AL, 07H          ; Mask the higher-order bits (D3-D7) in AL.
    JZ L1                ; Check number of keys pressed; if no key is
                           pressed, then go to L1.
    MOV AL, 50H          ; If a key is pressed, then move the read FIFO
                           RAM command word to AL.
    OUT 42H, AL          ; Send out the same to the command port.
    IN AL, 40H           ; Read the data from the data port of the 8279.
    MOV KEY_CODE, AL     ; Store key code in memory.
    HLT                  ; Terminate program execution.
```

This program uses the polled method of data transfer from the 8279 FIFO to the processor by reading the status word. In order to save the processor time and to avoid the reading and checking of status register, the IRQ line of the 8279 can be used to interrupt the processor. The IRQ signal is activated by the IC 8279 whenever a key press is sensed and its code is loaded into the FIFO RAM. This interrupt request line can be tied to any one of the interrupt signals of the processor

and the corresponding interrupt service routine can be used to read the key code from FIFO RAM.

7.10 INTEL TIMER IC 8253

In software programming of the 8086, it has been shown that a delay subroutine can be programmed to introduce a predefined time delay. The delay is achieved by decrementing a count value in a register using instructions. The disadvantage of this software approach is that the processor is locked in the delay loop and precious processor time is lost in counting. This can be overcome by using hardware timer and interrupts. IC 555 can be used to generate the timing signals, but only at a fixed time interval. This cannot be easily interfaced with the microprocessor. So, Intel has produced programmable timer devices namely IC 8253 and IC 8254. These devices can be programmed to generate different types of delay signals and also count external signals. Other counter/timer functions that are also common to be implemented with the 8253 are programmable frequency square wave generator, event counter, real-time clock, digital one-shot, and complex motor controller.

7.10.1 Features of IC 8253

Timer ICs 8253 and 8254 are manufactured by Intel with similar operating functions. The 8254 can be operated at frequency of up to 8 MHz whereas the 8253 can be operated only up to a maximum frequency of 2.6 MHz. Following is the list of major features of the IC 8253.

(i) Generation of accurate time delay
(ii) Three independent 16-bit down counters called channels
(iii) Six different programmable operating modes
(iv) Timer or counter operation
(v) Counting in binary or BCD
(vi) Capability to interrupt the processor.
(vii) Single +5 V supply
(viii) Can operate in DC and AC up to 2.6 MHz

7.10.2 Block Diagram of IC 8253 and Pin Details

The internal block diagram of IC 8253 is shown in Fig. 7.35. The figure shows the three independent 16-bit timers—counters 0, 1, and 2. These counters are programmer-controlled and the programmer can initialize the count value and start the counting process. The initialization is done through the data bus of the system. Counting can be started and stopped using software instructions written into the control register. The count value of the counter can be read by the programmer at any time through the proper command and through the data bus of the system.

The pins of IC 8253 are shown in Fig. 7.36. Each counter has two input pins—CLK (clock input) and GATE, and one pin, OUT, for data output. The control input line GATE is used to start or stop the counting operation. The OUT signal from each counter can be used to indicate the completion of required counting or timing operation and also to interrupt the processor.

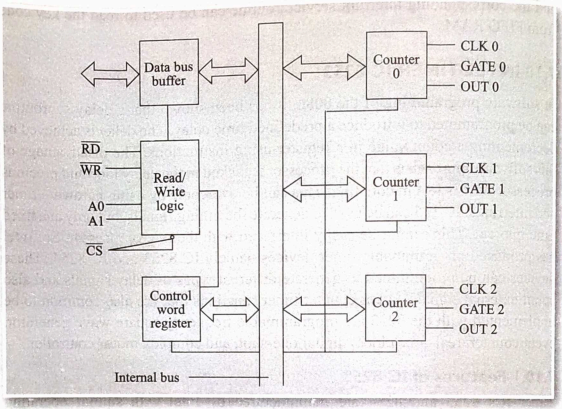


Fig. 7.35 Internal block diagram of IC 8253

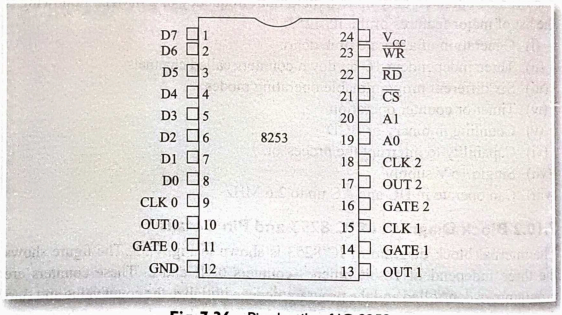


Fig. 7.36 Pin details of IC 8253

An 8-bit data bus is available on the 8253 pins to interface the IC with the microprocessor. A $\overline{CS}$ control signal is used to select the chip. This active low signal can be activated using the 8086 address lines and the decoder. In addition, the 8253 requires two address lines A_0 and A_1 to be issued from the 8086 hardware. These address lines are used to select one of four registers in the 8253—three counters and one control register. The $\overline{RD}$ and $\overline{WR}$ control signals are issued by the processor to indicate whether it is reading from or writing to the 8253 registers. The control signals applied to the pins of the 8253 for various operations are given in Table 7.27.

Table 7.27 Control signals and operation in IC 8253

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A1	A0	Operation
0	1	0	0	0	Load count value in counter 0
0	1	0	0	1	Load count value in counter 1
0	1	0	1	0	Load count value in counter 2
0	1	0	1	1	Write control word in control register
0	0	1	0	0	Read counter value from counter 0
0	0	1	0	1	Read counter value from counter 1
0	0	1	1	0	Read counter value from counter 2
0	0	1	1	1	No operation
0	1	1	X	X	No operation
1	X	X	X	X	Disable chip

7.10.3 Operating Modes and Control Word of IC 8253

The complete operation of the 8253 is programmed by the system software or the programmer. The programmer configures the 8253 to match his/her requirements. A set of control words must be sent out by the programmer to initialize each counter of the 8253. These control words program the mode, loading sequence, and selection of binary or BCD counting. Then the programmer initializes one of the counters of the 8253 with the desired quantity. Then upon proper command or control word, the 8253 will count-out the delay and interrupt the CPU when it has completed its tasks. It is easy to see that the software overhead is minimal and that multiple delays can easily be maintained by assignment of interrupt priority levels.

The normal procedure for software control of the 8253 has the following four main steps:

(i) Write the proper control word to the control register of the 8253 for each counter used.
(ii) Write the initial count value into the counter register.
(iii) Apply clock pulses to the counter.
(iv) Check for the desired count value, check for the interrupt signal from the counter, or check for the hardware signal OUT from the counter. After checking for the required time delay, the next operation can be carried out.

Each counter of the 8253 is individually programmed by writing a control word into the control word register. The control word format is given in Table 7.28. The LSB D0 is used to select it's the counting mode—binary or BCD. The next three bits M0 to M2 decide one of six operating modes for the counter selected. RL0 and RL1 bits decide whether read or load operation is to be performed on the counter. The counter can be made to count continuously, but the programmer can read the count value at any time by writing a control word to latch the count value and then read it. Similarly, initial setup can define whether an 8-bit or 16-bit value is loaded into the counter. The most significant two bits SC0 and SC1 decide the counter for mode and operation setting.

Table 7.28 Control word format of 8253

Bit position	D7	D6	D5	D4	D3	D2	D1	D0
Name	SC1	SC0	RL1	RL0	M2	M1	M0	Binary/BCD
Explanation	Select counter 00—Counter 0 01—Counter 1 10—Counter 2 11—Illegal		Read/Load option 00—Latch count 01—Read/Load LS byte only 10—Read/Load MS byte only 11—Read/Load LSB and then MSB		Mode selection bits 000—Mode 0 001—Mode 1 X10—Mode 2 X11—Mode 3 100—Mode 4 101—Mode 5			0—Binary counter 1—BCD counter

The six operating modes of the timer IC 8253 along with the function of GATE are listed in Table 7.29.

Table 7.29 Operating modes of 8253

M2	M1	M0	Operating modes		GATE control	Reloading of Count value
0	0	0	Mode 0	Interrupt on terminal count	0—Disables counting 1—Enables counting	No
0	0	1	Mode 1	Programmable one-shot	0 to 1 transition initiates counting	Yes, if triggered
X	1	0	Mode 2	Rate generator. Divide-by-n counter	0—Disables counting 1—Enables counting 0 to 1 transition reloads counter and initiates counting	Yes
X	1	1	Mode 3	Square wave rate generator	0—Disables counting 1—Enables counting 0 to 1 transition reloads counter and initiates counting	Yes
1	0	0	Mode 4	Software-triggered strobe	0—Disables counting 1—Enables counting	No
1	0	1	Mode 5	Hardware-triggered strobe	0 to 1 transition initiates counting	Yes, if gate input goes from 0 to 1

7.10.3.1 *Mode 0: Interrupt on Terminal Count*

In mode 0, the counter is used to issue an output after counting up to zero from the pre-initialized value. The initial count value is loaded into the counter and gets decremented for every clock pulse. The GATE signal input also controls the

counting operation. The output line OUT is made high from low, when the count value becomes 0. The OUT signal becomes low, when the counter is loaded with the next count value.

This mode 0 operation is a one-time operation and the OUT signal indicates the terminal condition of the required count operation. The GATE input signal of the corresponding counter either enables or disables the counting operation. The waveforms for counter operation in mode 0 are shown in Fig. 7.37.

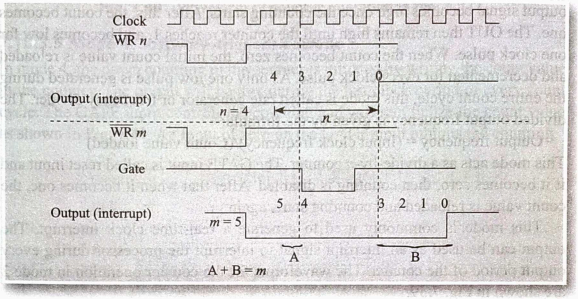


Fig. 7.37 Waveform of counter operation in mode 0

7.10.3.2 *Mode 1: Hardware-triggered One-shot*

Mode 1 is similar to mode 0, but has a minor difference. The similarity between the modes is that the output becomes low while counting down and becomes high once the count value reaches zero. The first difference between the modes 0 and 1 is that in mode 1, the counting is started or triggered whenever the GATE input becomes high. The second difference is that in mode 1, the count value is reloaded if in the middle of the count, the GATE goes low and becomes high again. Thus, the operation of the counter is similar to that of a monostable multivibrator, with

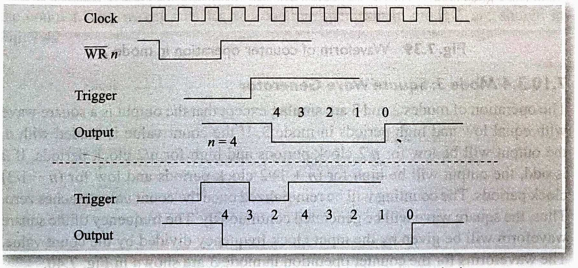


Fig. 7.38 Waveform of counter operation in mode 1

the GATE input signal acting as the trigger input to the multivibrator. The duration of the low output is the quasi-stable state of the multivibrator and is decided by the count value loaded initially.

The waveforms for counter operation in mode 1 are shown in Fig. 7.38.

7.10.3.3 *Mode 2: Rate Generator*

In mode 2, the counter generates continuous signals in the output line. The down counting starts once the counter is loaded with the count value. In this mode, the output signal becomes high once the counting starts. After this, the count becomes one. The OUT then remains high until the counter reaches 1, and becomes low for one clock pulse. When the count becomes zero, the initial count value is reloaded and decremented for every clock pulse. As only one low pulse is generated during the entire count cycle, this mode is called rate generator or frequency divider. The divided output frequency is given by the formula:

Output frequency = (Input clock frequency)/(Count value loaded)

This mode acts as a divide-by-*n* counter. The GATE input is called reset input and if it becomes zero, then counting is disabled. After that when it becomes one, the count value is reloaded and counting starts again.

This mode is commonly used to generate a real-time clock interrupt. The output can be used as an interrupt signal to interrupt the processor during every output period of the counter. The waveforms for the counter operation in mode 2 are shown in Fig. 7.39.

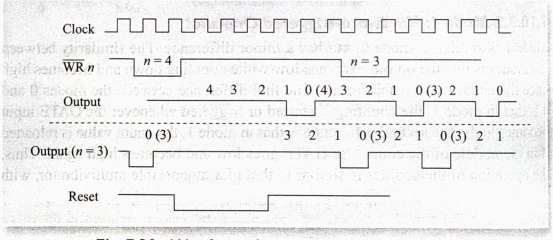


Fig. 7.39 Waveform of counter operation in mode 2

7.10.3.4 *Mode 3: Square Wave Generator*

The operation of modes 2 and 3 are similar, except that the output is a square wave with equal low and high periods in mode 3. If the count value is loaded with *n*, the output will be low for *n*/2 clock periods and high for n/2 clock periods. If *n* is odd, the output will be high for $(n + 1)/2$ clock periods and low for $(n - 1/2)$ clock periods. The counting will be reinitialized once the count value reaches zero. Thus, the square wave will be generated continuously. The frequency of the square waveform will be given by the input clock frequency divided by the count value. The waveforms for the counter operation in mode 3 are shown in Fig. 7.40.

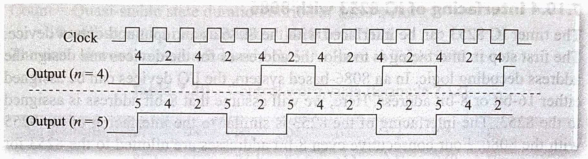


Fig. 7.40 Waveform of counter operation in mode 3

7.10.3.5 *Mode 4: Software-triggered Strobe*

In mode 4, the down counting in the counter is initiated by writing the count value in the counter. The output will be low during the last clock period of every count cycle. The GATE input controls the counting and will reinitialize the count value as shown in Fig. 7.41. An input of zero on the GATE input inhibits the counting.

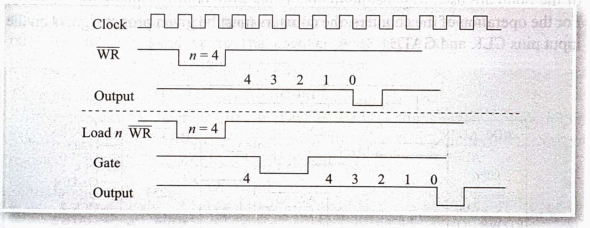


Fig. 7.41 Waveform of counter operation in mode 4

7.10.3.6 *Mode 5: Hardware-triggered Strobe*

Mode 5 of the timer IC 8253 is similar to mode 4, with the difference that the counting is triggered by the GATE input and not by writing of the count value by the software. The output is high once the count value is loaded. Counting starts with the rising edge of the gate pulse. The output is low during the last count of the counter. The waveforms for the counter operation in mode 5 are shown in Fig. 7.42.

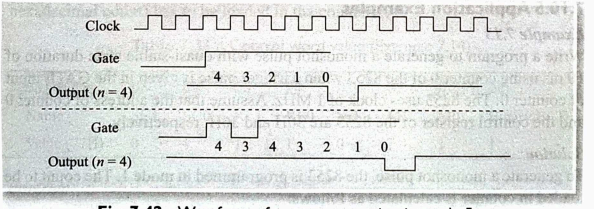


Fig. 7.42 Waveform of counter operation in mode 5

7.10.4 Interfacing of IC 8253 with 8086

The timer IC 8253 can be interfaced with the 8086 as an input and output device. The first step in interfacing is to allot the addresses for the devices and design the address decoding logic. In an 8086-based system, the I/O devices can be assigned either 16-bit or 8-bit address. Here, we will assume that 8-bit address is assigned to the 8253. The interfacing of the 8253 is similar to the interfacing of the 8255 with the 8086. Four consecutive even 8-bit addresses are allotted to the 8253 for accessing the three counters and one control register, if the data bus of the 8253 is connected with the data bus D7–D0 of the 8086. The A0 and A1 address lines can be used to select one of three counters and the control register. As usual, the lower address bus and the data bus must be demultiplexed in any 8086 system using a latch and ALE signal. The 8086 processor places the 8-bit I/O address on the lower-order address bus. Hence, the lower-order address lines A7–A3 are used for address decoding purpose as shown in Fig. 7.43. The M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals of the 8086 are used to generate the read and write control signals for the 8253. For the operation of the counters, the counters must be given proper signals on the input pins CLK and GATE.

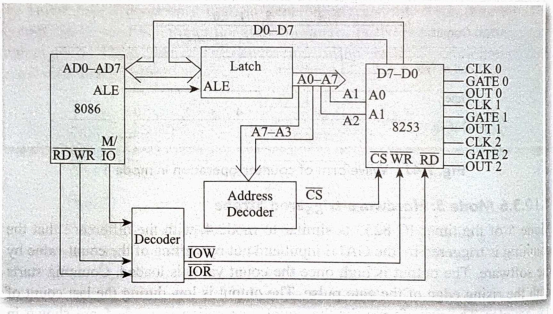


Fig. 7.43 Interfacing the 8253 with the 8086

7.10.5 Application Examples

Example 7.13

Write a program to generate a monoshot pulse with quasi-stable state duration of 10 ms using counter 0 of the 8253 when a trigger pulse is given in the GATE input of counter 0. The 8253 uses clock of 1 MHz. Assume that the address of counter 0 and the control register of the 8253 are 30H and 36H, respectively.

Solution:

To generate a monoshot pulse, the 8253 is programmed in mode 1. The count to be loaded in counter is calculated as follows:

Period of the 8253's clock = 1/(1 × 10^6) = 1 microsecond

Count = Quasi-stable state duration/Period of 8253's clock
= 10 millisecond/1 microsecond = 10000 = 2710H
The control word to be loaded in the 8253 is given in Table 7.30.

Table 7.30 Control word value (Example 7.13)

Bit position	D7	D6	D5	D4	D3	D2	D1	D0
Name	SC1	SC0	RL1	RL0	M2	M1	M0	Binary/BCD
Value (32H)	0	0	1	1	0	0	1	0

Program:

```
MOV AL, 32H    ; Load the control word for the timer IC 8253 in AL.
OUT 36H, AL    ; Send it to the control register of the 8253.
MOV AL, 10H    ; Load the lower-order count value in AL.
OUT 30H, AL    ; Send it to the counter 0 of the 8253.
MOV AL, 27H    ; Load the higher-order count value into AL.
OUT 30H, AL    ; Send it to the counter 0 of the 8253.
HLT            ; Terminate program execution.
```

Example 7.14

Write a program to generate a square wave of 1 KHz using counter 1 of the 8253. The 8253 uses clock of 1 MHz. Assume that the address of counter 1 and control register of the 8253 are 32H and 36H respectively.

Solution:

To generate a square wave, the 8253 is programmed in mode 3. The count to be loaded in the counter is calculated as follows:
Period of 8253's clock = 1/(1 × 10^6) = 1 microsecond
Count = Period of the square wave/Period of 8253's clock
= 1 millisecond/1 microsecond = 1000 = 03E8H
The control word to be loaded in the 8253 is given in Table 7.31.

In this example, the control word is found such that BCD value will be loaded in the counter 1 and hence the BCD value of count has to be loaded in the counter 1. However, the control word can also be found such that the binary value of count will be loaded in the counter 1 and at that time binary value of count (i.e., hexadecimal count) has to be loaded in the counter 1.

Table 7.31 Control word value (Example 7.14)

Bit position	D7	D6	D5	D4	D3	D2	D1	D0
Name	SC1	SC0	RL1	RL0	M2	M1	M0	Binary/BCD
Value (77H)	0	1	1	1	0	1	1	1

Program:

```
MOV AL, 77H    ; Load the control word for the timer IC 8253.
OUT 36H, AL    ; Send it to the control register of the 8253.
```

```
MOV AL, 00H    ; Load the lower-order BCD count value in AL.
OUT 32H, AL    ; Send it to the counter 1 of the 8253.
MOV AL, 10H    ; Load the higher-order BCD count value into AL.
OUT 32H, AL    ; Send it to the counter 1 of 8253
HLT            ; Terminate program execution.
```

Example 7.15

Timer interface using polling method

Assume that a timer IC 8253 is interfaced to the 8086 such that the addresses assigned to the 8253 are 30H, 32H, 34H, and 36H. The 8086-based system has another IC 8255 interfaced to it at the addresses 40H, 42H, 44H, and 46H. The two seven-segment displays are interfaced to port A of the 8255 using two 7447 decoders (BCD to seven-segment code converter). Write a program such that the seven-segment displays in the system display the count in decimal from 00 to 99 with 1 second delay between each count. (This type of interface can be developed as a stopwatch by adding a set of switches on a port.)

Solution:

The 8253 has been interfaced to the 8086 at the address 30H–36H. So, the address of the control word is 36H. Here, the software polling method is used. So, the counter must be run and then the program has to check whether the count is completed for the predetermined period—here one second. As the display has to be incremented for every second, we have to select a counter mode, which will have auto reload of the count value. So, mode 2 or 3 can be used for this application. Counter 0 is selected and the 16-bit count value has to be loaded for the binary counter. The mode 2 control word for the above configuration is given in Table 7.32.

Table 7.32 Control word value (Example 7.15)

Bit position	D7	D6	D5	D4	D3	D2	D1	D0
Name	SC1	SC0	RL1	RL0	M2	M1	M0	Binary/BCD
Value (34H)	0	0	1	1	0	1	0	0

The next step is to find the count value. The count value should be such that the counter becomes zero after counting the predetermined count value in one second. So, if the input clock frequency for the counter operation is selected as 1 KHz, then the counter will be decremented after every clock period, that is, after every 1 ms. So, the count value of 1000 decimal will result in a delay of one second, when the counter becomes zero after 1000 counts. If the counter is designed to count in binary, then the count value 1000 decimal must be converted to binary and loaded into the counter as 03E8 (in hexadecimal form). If the counter is designed to count in BCD, then the count value can be loaded in BCD format itself as 1000.

The program consists of three parts. The first part is initializing the counter and the count value. The clock signal must be applied to the selected counter's clock input pin. Here, the counter is operated in mode 2 and hence the count value need not be loaded repeatedly after the count is over. The count value is reloaded automatically after it becomes zero. The second part is to check whether the

counter value has become zero using software polling technique. In the software polling method, the counter is first latched with a latch counter command control word. Then the count value is read from the counter. After the 16-bit count value is loaded to the processor registers, it is checked for zero value. For this the two bytes are ORed. The third part is to increment the value sent to the seven-segment displays connected at port A of the 8255.

Program:

```
        MOV AL, 80H     ; Load the control word for the 8255.
        OUT 46H, AL     ; Send it to control register of 8255.
        MOV AL, 00H     ; Load the initial BCD number to be sent to
                          displays in AL.
        OUT 40H, AL     ; Send it to the port A of the 8255.
        MOV BL, AL      ; Store the BCD number currently displayed in
                          BL.
        MOV AL, 34H     ; Load the control word for the timer IC 8253.
        OUT 36H, AL     ; Send it to the control register of the 8253.
        MOV AL, 0E8H    ; Load the lower-order count value in AL.
        OUT 30H, AL     ; Send it to counter 0 of the 8253.
        MOV AL, 03H     ; Load the higher-order count value in AL.
        OUT 30H, AL     ; Send it to counter 0 of the 8253.
CHECK:  MOV AL, 04H     ; Load the control word to latch the count value
                          in counter 0 in AL.
        OUT 36H, AL     ; Send it to the control register of the 8253.
        IN AL, 30H      ; Read the lower-order count value from Counter
                          0 into AL.
        MOV CL, AL      ; Store it in CL.
        IN AL, 30H      ; Read the higher-order count value from Counter
                          0 into AL.
        MOV CH, AL      ; Store it in CH.
        CMP CX, 0       ; Compare CX value with 0.
        JNZ CHECK       ; If Z = 0 then 1 second is not over, so go to
                          location CHECK.
        INC BL          ; Increment BL to hold next BCD number to be sent
                          to displays.
        MOV AL, BL      ; Move the value in BL to AL.
        DAA             ; Convert the result in AL to BCD form.
        OUT 40H, AL     ; Send it to the Port A of the 8255
        JMP CHECK       ; Jump to CHECK.
```

Example 7.16

Timer interface using interrupt method

Implement the same application illustrated in Example 7.15 with an interrupt method.

Solution:

Here, the counter is initialized and starts counting. In mode 2, the counter gives a

logic 0 pulse for a clock period after the count is over. This clock pulse is used as an interrupt signal to the 8086 processor.

This interface uses the interrupt feature of the 8086 to increment the count value in the display at port A of the 8255. The 8253 timer IC is programmed to generate an interrupt signal at every second. To achieve this, one of the counters in the 8253 is programmed in mode 2 and it generates an active low pulse for one clock period in its OUT pin after every second. This OUT signal is given to the non-maskable *interrupt* (NMI) interrupt pin of the 8086 through an inverter. The NMI requires an active high signal. However, the 8253 gives out an active low signal whenever the counting is over in mode 2. So, an inverter is connected between OUT0 and NMI pins. Then in the ISR, the value sent to the display is incremented. In this example, the OUT0 signal from the 8253 is connected to the NMI interrupt line of the 8086 as shown in Fig. 7.44. The clock frequency applied at CLK0 is selected as 1 KHz signal and GATE0 is connected to logic 1. The program for the same is as follows:

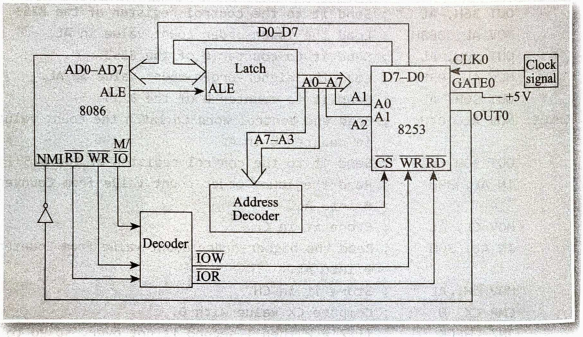


Fig. 7.44 Interfacing the 8253 timer 0 in interrupt driven mode

Let us assume that BL register is used for storing the BCD data sent to the display and is not used by the main program for other purposes.

Main program:

```
START:  MOV AL, 80H     ; Load the control word for the 8255.
        OUT 46H, AL     ; Send it to control register of 8255.
        MOV AL, 00H     ; Load the initial BCD number to be sent to
                          displays in AL.
        OUT 40H, AL     ; Send it to the port A of 8255.
        MOV BL, AL      ; Store the BCD number currently displayed in
                          BL.
        MOV AL, 34H     ; Load the control word for the timer IC 8253.
        OUT 36H, AL     ; Send it to the control register of the 8253.
```

```
        MOV AL, 0E8H     ; Load the lower-order count value in AL.
        OUT 30H, AL      ; Send it to counter 0 of the 8253.
        MOV AL, 03H      ; Load the higher-order count value in AL.
        OUT 30H, AL      ; Send it to the Counter 0 of the 8253.
        …                ; Main program continues
        …
```

NMI interrupt service routine:

```
NMI_ISR: INC BL          ; Increment BL.
         MOV AL, BL      ; Move BL content to AL.
         DAA             ; Convert the result to BCD form.
         OUT 40H, AL     ; Send the same to port A of the 8255.
         IRET            ; Return from the interrupt service routine.
```

7.11 INTRODUCTION TO SERIAL COMMUNICATION

Serial communication is sending and receiving information bit by bit. For short range communication, parallel data transfer is preferred as it is the fastest means. While transferring data over long distances, parallel communication requires numerous wires and complex error handling/data recovery mechanisms. Moreover, for parallel data transmission of eight bits at a time, both the receiver and the transmitter side equipments need eight differential amplifiers and related hardware. This results in complex circuitry and becomes costlier for long distance transmission. Thus, serial communication is preferred for long range communication and it can be easily implemented using a single wire or a pair of wires.

As the microcomputer uses parallel data, it is converted into serial form and then transmitted. On receiving the serial data, it is converted into parallel form and then transferred to the microcomputer.

The terms mainly used in serial data systems are simplex, half-duplex, and full-duplex. In *simplex* data transmission, data can be transferred only in one direction. Examples for this type of systems are radio, television, etc. In *half-duplex* transmission, the communication can take place in either direction between two systems but only in one direction at a time. An example of half-duplex transmission is a two way radio system, where one user always listens while the other talks. This is possible by turning off the receiver circuitry during transmission. In *full-duplex* communication, both the receiver and the transmitter can send and receive data at the same time. A normal telephone conversation is an example of a full-duplex system.

Serial data can be sent either in synchronous or asynchronous modes. For *synchronous transmission*, data is sent in blocks at a constant rate. The constant rate means the frequency of transmission and reception are the same and both transmission and reception take place simultaneously. The start and end of a block are identified with specific bytes or bit patterns. In general, the synchronous transmission is used for high transmission speeds of more than 20 kb/second. For *asynchronous transmission*, each data character has a bit to identify its start and one or two bits to identify its end. So, the characters can be sent at any time

randomly without checking the receiver. The reception and transmission are not synchronized.

The bit format used for transmitting asynchronous serial data is shown in Fig. 7.45. This format is also called *frame*.

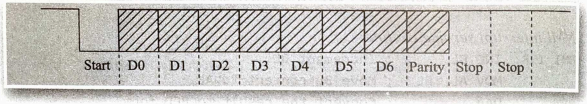


Fig. 7.45 Asynchronous serial data transfer format

When no data is being sent, the signal line is in a constant high level. The starting data character is indicated by the line going low for one bit duration and is usually called *start bit*. The data bits are then sent out on the line one followed by the other. Here, the LSB is sent out first. The data bit is followed by a parity bit, which is used to check for errors in received data but it is optional. After the data bits and the parity bit, the signal bit is made high for at least one bit duration to identify the end of character and is referred to as *stop bit*. Some systems may also use two stop bits.

Baud rate is the rate at which serial data is being transferred and in general, it is measured in bits/second. Baud rate = 1/(Time between signal transitions). If the signal is changing every 1.67 ms, then baud rate is $1/(1.67 \times 10^{-3})$, or 600 Bd. Common baud rates are 300, 600, 1,200, 2,400, 4,800, 9,600, and 19,200.

RS-232C is a standard that describes the function of the signal and handshake pins for serial data transfer. A major problem with RS-232C is that it can only transmit data reliably for about 50 ft (16.4 m) at its maximum rate of 20,000 Bd. If longer lines are used, the transmission rate has to be drastically reduced. This limitation is caused by the open signal lines with a single common ground that are used in RS-232C.

The Electronics Industries Association (EIA) has a standard named *RS-423A* which is an improvement over RS-232C. This standard specifies a low-impedance single-ended signal, which can be sent over a 50 Ω coaxial cable. Logic high in this standard is represented by the signal line being between 4V and 6V negative with respect to ground and logic low is represented by the signal line being to 4V to 6V positive with respect to ground. The RS-423 standard allows a maximum data range of 100,000 Bd over a 40 ft line or a maximum baud rate of 1,000 Bd on a 4,000 ft line.

RS-422A is a newer standard for serial data transfer which specifies that each signal will be sent differentially over two adjacent wires in a ribbon cable or a twisted pair of wires. The term differential used in this standard means that the signal voltage is developed between the two signal lines rather than between the signal line and ground as in RS-232C and RS-423A. In RS-422A, a logic high is transmitted by making the 'b' line more positive than the 'a' line. A logic low is transmitted by making the 'a' line more positive than the 'b' line. The voltage difference between the two lines must be greater than 0.4V but less than 12V.

Modem is a modulator and demodulator that sends digital 1s and 0s over standard phone lines as modulated tones. A modem is essential in communication whenever the signal has to be transmitted over long distances. In the United States, modem standards are handled by the Telecommunications Industry Association (TIA), which works closely with the Comite Consultatif International Telegraphique et Telephonique (CCITT), which is part of the International Telecommunications Union (ITU). The CCITT standards, which relate to modems start with a 'V'. Examples are the V.22bis, which is a 2,400 bit/s modem standard, and the V.29bis, which is a 9,600 bit/s modem standard. The major modulation techniques used in modems are amplitude modulation, frequency shift keying, phase shift keying, and multiple carrier modulation. Modems can be directly connected to the microcomputer buses for establishing serial communication between two systems.

The serial port transmission has a lot of technical terms and protocols involved. This section focusses on the basic serial port IC Intel 8251 that can be interfaced with any processor for data transmission and reception in a serial manner.

7.11.1 Features and Details of 8251 USART

The 8251 is a universal synchronous asynchronous receiver transmitter (USART) used for serial data communication. As a peripheral device of a microcomputer system, the 8251 receives parallel data from the CPU and transmits the same in a serial form. This device also receives serial data from outside, converts them into parallel data, and sends them to the CPU. The 8251 can support both synchronous and asynchronous transmission formats and is programmable. It supports full-duplex serial transmission and reception and variable baud rates.

The internal block diagram of the 8251 is shown in Fig. 7.46. Basically, it consists of a parallel-to-serial shift register for transmitting over transmit data (TXD) line from a buffer and a serial-to-parallel converter for data received on the receive data (RXD) line. A separate control unit is available to determine the operation of the IC according to the control word written into it. A modem control unit is present for interfacing the modem with the 8251. In addition to these units, IC 8251 has an input and output port that can be used for interfacing with any processor along with its read and write control logic. The 8251 requires clock and reset signals for working in a synchronized manner with the processor. It has a 16-bit control register and can be programmed using this control register. The status of operation of the 8251 can be read from the status register. These two registers can be accessed by the processor by making the $C/\overline{D}$ pin of the 8251 logic 1. Another register called the data register can be accessed by making the $C/\overline{D}$ logic 0. The read operation is used to read the serial data received and the write operation is used to write the data to be transmitted. These basic operations of the 8251 are shown in the Table 7.33. The address line A1 from the 8086 is connected to the $C/\overline{D}$ signal, when the data bus is connected to the D7–D0 bus of the 8086. Hence, two successive even addresses are allotted to the 8251, one for control/status register, which is selected when A0 = 1 and another for data register, which is selected when A0 = 0.

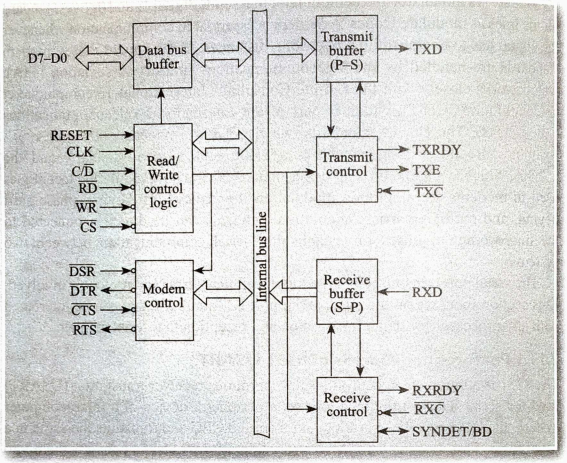


Fig. 7.46 Block diagram of the 8251 USART

Table 7.33 Basic operations of 8251 and related control signals

$\overline{CS}$	C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	Function
1	X	X	X	Chip not selected; data bus in high impedance state
0	X	1	1	Data bus in high impedance state
0	1	0	1	Status word read by CPU from status register
0	1	1	0	Control word written into control register by CPU
0	0	0	1	Data read by CPU from data register
0	0	1	0	Data written into data register by CPU

The 8251 has 28 pins. The details and functions of these pins are listed as follows:

(i) Data bus (D0–D7): A group of bi-directional lines that are used for data and control word transfer between the CPU and the 8251.

(ii) Reset: An active high signal applied on this pin brings the 8251 into reset state. The device after reset waits for the writing of the mode instruction. The time duration required for the reset signal is six clock pulses.

(iii) CLK: A clock signal is used to generate internal device timing. It is independent of receive clock ($\overline{RXC}$) or transmit clock ($\overline{TXC}$). In general, the CLK frequency must be much higher than the $\overline{RXC}$ and $\overline{TXC}$ frequencies.

(iv) Write data/command ($\overline{WR}$): It is an active low input signal for writing data and control words from the CPU into the 8251.

(v) Read data ($\overline{RD}$): It is an active low input signal for reading data and status words from the 8251.

(vi) Control/data (C/$\overline{D}$): It is an input signal for selecting data or command/status words when the 8251 is accessed by the CPU. If the C/$\overline{D}$ = 0, data are accessed. If the C/$\overline{D}$ = 1, command word or status word are accessed.

(vii) Chip select ($\overline{CS}$): It is an active low input signal, which selects the 8251 for CPU accesses.

(viii) Transmit data line (TXD): It is an output signal for transmitting converted serial data from the 8251. The device is in mark status (high-level) after resetting or during a status, when transmit is disabled.

(ix) Transmitter ready (TXRDY): It is an output signal, which indicates that the 8251 is ready to accept a data character for transmission. However, the terminal is always at a low level if clear to send ($\overline{CTS}$) = 1 or the device is set in transmitter disable status by a command.

(x) Transmitter empty (TXEMPTY): It is an output signal that indicates that the 8251 has transmitted all the characters and had no data character for transmission.

(xi) Transmitter clock ($\overline{TXC}$): This is a clock input signal that determines the transfer speed of the transmitted data or in other words, the baud rate for transmission. In synchronous mode, the baud rate will be the same as the frequency of the $\overline{TXC}$. In asynchronous mode, it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16, or 1/64 of the $\overline{TXC}$.

(xii) Receive data (RXD): A signal line that receives serial data.

(xiii) Receiver ready (RXRDY): It is a signal that indicates that the 8251 contains a character that is ready to be read and the CPU can read the data. If the CPU reads a data character, the RXRDY will be reset by the leading edge of the $\overline{RD}$ 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.

(xiv) Receiver clock ($\overline{RXC}$): This is a clock input signal that determines the transfer speed of received data or the baud rate of reception. In synchronous mode, the baud rate is the same as the frequency of the $\overline{RXC}$. In asynchronous mode, it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16, 1/64 of the $\overline{RXC}$.

(xv) Sync detect/break detect (SYNDET/BD): It is an active high output signal. In asynchronous mode, it is used to indicate a data break. In synchronous mode, it is used to indicate the correct receipt of synchronous characters and the next data is to be received.

The following signals are used with a modem for handshaking and establishing connection.

(i) Data set ready ($\overline{DSR}$): This is an input signal to the 8251 from the modem interface. The input status of the signal can be recognized by the CPU by reading status words. $\overline{DSR}$ indicates that the modem is powered up.

(ii) Data terminal ready ($\overline{DTR}$): This is an output signal from the 8251 for the modem interface. It is possible to set the status of the $\overline{DTR}$ by a command. $\overline{DTR}$ indicates that the 8251 is powered up.

(iii) Clear to send data ($\overline{CTS}$): This is an input signal to the 8251 from the modem interface, which is used for controlling the 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.

(iv) Request to send ($\overline{RTS}$): Active low output signal sent to the modem by the 8251, indicating that it is ready to send data.

7.11.2 Control Words

The 8251 operations should be initialized after reset and before using it. To initialize it, the programmer must send the mode word and then the command word to the control register address. There are two types of control words—one is the mode instruction (setting of function) and the other is command (setting of operation).

7.11.2.1 *Mode Command Word*

Mode instruction is used for setting the function of the 8251. The writing of a control word after resetting will be recognized as a mode instruction. The functions set by mode instruction are as follows:

(i) Selecting synchronous or asynchronous modes
(ii) Stop bit length (asynchronous mode)
(iii) Character length
(iv) Parity bit
(v) Baud rate factor (asynchronous mode)
(vi) Internal/external synchronization (synchronous mode)
(vii) Number of synchronous characters (synchronous mode)

The bit configuration of the asynchronous mode instruction is given in Table 7.34 and that of the synchronous mode instruction is given in Table 7.35. In the case of synchronous mode, it is necessary to write 1- or 2-byte sync characters. The writing of sync characters constitutes part of mode instruction. The mode command word given in Fig. 7.34 is applicable if D1 and D0 are not 0. For D1 and D0 being 0, the mode command format in Table 7.35 is used.

Table 7.34 Mode instruction (asynchronous) bit configuration

D7 D6	D5 D4	D3 D2	D1 D0
S1 S0	EP PEN	L1 L0	B1 B0
Frame control stop bit length 00—Inhibit 01—1 stop bit 10—1.5 stop bits 11—2 bits	Parity check X0—Disable 01—Odd parity 11—Even parity	Character length 00—5 bits 01—6 bits 10—7 bits 11—8 bits	Baud rate select bits 00—SYN mode 01—1X clock 10—16X clock 11—64X clock

Table 7.35 Mode instruction (synchronous) bit configuration

D7	D6	D5	D4	D3	D2	D1	D0
SCS	ESD	EP	PEN	L1	L0	0	0
Number of sync characters 0—2 characters 1—1 character	Synchronous mode 0—Internal synchronization 1—External synchronization	Parity check X0—Disable 01—Odd parity 11—Even parity		Character length 00—5 bits 01—6 bits 10—7 bits 11—8 bits			

7.11.2.2 *Serial Command Word*

Serial port 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 functions set by command are as follows:

(i) Transmit enable/disable
(ii) Receive enable/disable
(iii) Data terminal ready (DTR), ready to send (RTS) output of data.
(iv) Resetting of error flag
(v) Sending break characters

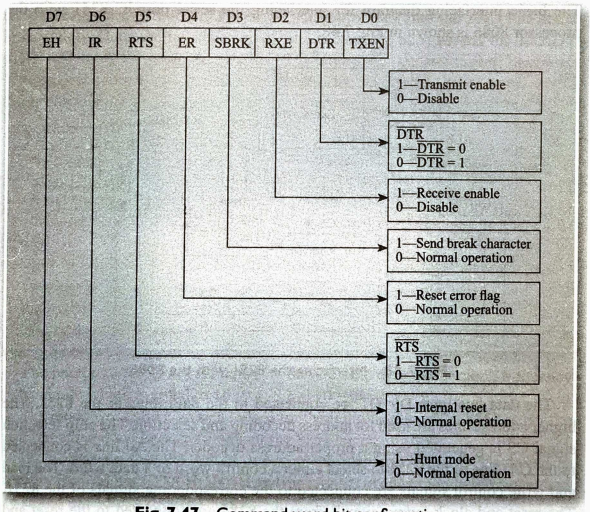


Fig. 7.47 Command word bit configuration

(vi) Internal resetting
(vii) Hunt mode (synchronous mode)

The various bit configurations for the command word written into the 8251 is shown in Fig. 7.47.

7.11.2.3 *Status Word*

It is possible to see the internal status of the 8251 by reading a status word. The bit configuration of the status word is given in Table 7.36. The programmer can read the status information such as parity error, overrun error, framing error, and the signal on the selected pins.

Table 7.36 Status word bit configuration

D7	D6	D5	D4	D3	D2	D1	D0
DSR	SYNDET/BD	FE	OE	PE	TXEMPTY	RXRDY	TXRDY
Data set ready 0—$\overline{DSR}$ = 1 1—$\overline{DSR}$ = 0	Same as in I/O pin	1—Framing error	1—Overrun error	1— Parity error	Same as in I/O pin	Same as in I/O pin	Same as in I/O pin

7.11.3 Interfacing 8251 with 8086

The basic interfacing diagram for interfacing the serial port IC 8251 with the processor 8086 is shown in Fig. 7.48.

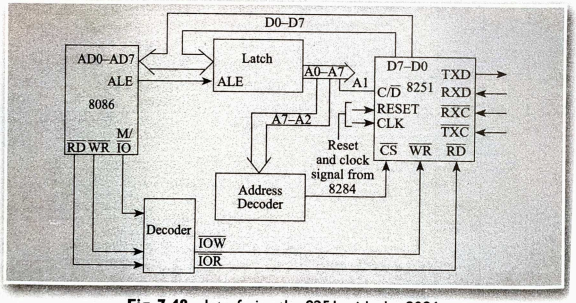


Fig. 7.48 Interfacing the 8251 with the 8086

The data bus lines D0–D7 are connected to the data lines of the 8251. The higher address lines are used for address decoding and selection. The chip selection signal $\overline{CS}$ is generated using a proper address decoder. The A0 line is connected to the C/$\overline{D}$ line of the 8251 to select either a control word or a data word. The read and write control signals are connected to the corresponding signals of the 8251. The reset and clock output signals from the 8284 are connected to the reset and

clock inputs of the 8251. In addition, the 8251 needs separate clock signals for transmission and reception. These $\overline{RXC}$ and $\overline{TXC}$ clock signals can be obtained by dividing the clock output from the 8284 or 8253. This is not shown in Fig. 7.48.

Normally, two computer systems can be interconnected using the serial port. In such communication, the TXD signal of one system is connected to the RXD line of another system and vice versa. Care must be taken to ensure that the transmit clock of the transmitting computer is the same as the receive clock of the receiving computer.

The software part of programming the 8251 consists of initializing the 8251 and then using it for data transmission and reception. Initialization of the 8251 consists of writing proper mode command word immediately after reset. The mode control word for synchronous operation must be followed by the corresponding sync characters. Then the command word for setting the parameters of the serial port is written into the control register. Once the initialization is over, the 8251 is ready for transmission and reception of data if proper clock signals are applied to it.

The serial data received is stored in the serial data buffer and the reception of data is informed to the processor by using the RXRDY signal. This signal may be connected to an interrupt request in the 8086 and the corresponding ISR can read the received data from the 8251. The programmer can also use the status word read from the 8251 IC for checking whether a data has been received or not.

The serial transmission is started by writing the data to be transmitted into the data register of the 8251. The serial shifting of data into the TXD line starts immediately. Once the transmission of data is over, the 8251 asserts the TXRDY signal informing the processor that the 8251 is ready for transmission of next data. The programmer can also read the status word for checking whether the next data can be written in the data register for transmission.

Example 7.17

Write an 8086 program to initialize the 8251 to transmit a message from an 8086-based single board microcomputer to a CRT terminal which is interfaced by RS-232 standard. The message 'HELLO!' has to be displayed in the CRT terminal. The baud rate should be 9,600 and the number of stop bits, parity bit, and character bits should be 2, no parity, and 7 respectively. The frequency of the transmitter clock is 153.6 KHz. Assume that the addresses of the control/status register and data register are 82H and 80H, respectively.

The flowchart for the program is shown in Fig. 7.49.

The mode word format, command word format, and status word format for this example problem are given in Tables 7.37, 7.38, and 7.39 respectively.

The baud rate factor is calculated by dividing the transmitter clock frequency by baud rate which is equal to 16 (= 153.6 × 10^3/9,600).

The mode word and command word for the initialization of the 8251 as per the requirement and to check the status of the 8251 are as follows:

Let us assume that the ASCII codes of the message 'HELLO!' is stored in memory from address 2000H:1001H and the number of characters to be transmitted are stored in address 2000H:1000H.

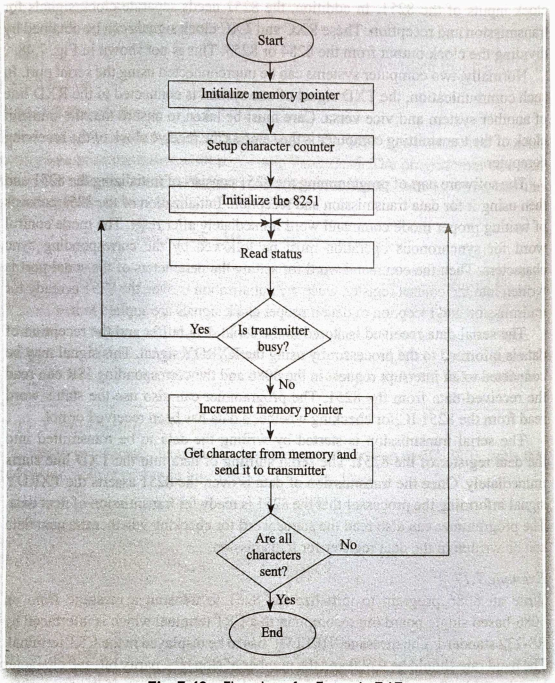


Fig. 7.49 Flowchart for Example 7.17

Table 7.37 Mode word format (Example 7.17)

D7	D6	D5	D4	D3	D2	D1	D0	
1	1	0	0	1	0	1	0	= CAH
Two stop bits		No parity		7-bit character		Baud rate = $\overline{\text{TXC}}$/16		

Table 7.38 Command word format (Example 7.17)

D7	D6	D5	D4	D3	D2	D1	D0	
X	0	X	1	X	0	X	1	= 11H
			↓		↓		↓	
			Error reset		Receive disable		Transmit enable	

Table 7.39 Status word format (Example 7.17)

D7	D6	D5	D4	D3	D2	D1	D0	
X	X	X	X	X	X	X	1	= 01H

↓ Transmitter ready

Program:

```
        MOV AX, 2000H ; Load segment address 2000H in AX.
        MOV DS, AX    ; Move the same to DS.
        MOV BX, 1000H ; Move offset address 1000H in BX.
        MOV CX, [BX]  ; Set up CX as character counter.
        MOV AL, 00H   ; Move dummy mode word (= 00H) in AL.
        OUT 82H, AL   ; Write dummy mode word in control register.
        OUT 82H, AL
        OUT 82H, AL
        MOV AL, 40H   ; Move internal reset (IR) command word.
                      ;(D6 = 1 and other bits = 0) to AL.
        OUT 82H, AL   ; Send it to control register to reset the 8251.
        MOV AL, CAH   ; Initialize the 8251 by sending mode word.
        OUT 82H, AL
        MOV AL, 11H   ; Send command word to the 8251.
        OUT 82H, AL
STATUS: IN AL, 82H    ; Get Status from the 8251.
        AND AL, 01H   ; Check TX Ready.
        JZ STATUS     ; If TX RDY = 0, wait.
        INC BX        ; Point to next character.
        MOV AL, [BX]  ; Get Character in AL.
        OUT 80H, AL   ; Send character to transmitter.
        LOOP STATUS   ; Repeat the procedure CX times.
        HLT           ; Terminate program execution.
; MESSAGE-HELLO! is stored in memory as follows:
; Memory Address              Data      Comment
2000H: 1000H                  DB   08   ; No of characters
2000H: 1001H                  DB   48   ; Letter 'H'
2000H: 1002H                  DB   45   ; Letter 'E'
2000H: 1003H                  DB   4C   ; Letter 'L'
2000H: 1004H                  DB   4C   ; Letter 'L'
2000H: 1005H                  DB   4F   ; Letter 'O'
2000H: 1006H                  DB   21   ; Letter '!'
2000H: 1007H                  DB   0D   ; Carriage return
2000H: 1008H                  DB   0A   ; Line feed
```

7.12 8259 PROGRAMMABLE INTERRUPT CONTROLLER

Interrupts are used in a system to handle routines such as reading ASCII

characters from a keyboard, detecting and performing an emergency operations such as sounding a fire alarm, and so on. For this, the processor's maskable or non-maskable interrupts are used. However, the processor has limited number of hardware interrupts. For applications that use interrupts from multiple sources, the processor can use external device called programmable interrupt controller (PIC) or priority interrupt controller.

The Intel 8259 is a PIC designed and developed for use with the Intel 8086 and the 8085 microprocessors. The family originally consisted of the 8259, 8259A, and 8259B PICs, though a number of manufacturers make a wide range of compatible chips.

7.12.1 Features and Architecture of 8259

The basic operation of the interrupt mechanism lies in calling a subroutine, whenever a hardware interrupt signal is activated. When more number of interrupt sources is present, the process of calling an ISR involves priority resolving and checking mask for interrupts. The main purpose of using the 8259 interrupt controller is also to do the same task of calling the ISR based on the interrupt priority and masks. The 8259 acts as a multiplexer, combining multiple interrupt input sources into a single interrupt request to the processor. The main features of the 8259 are the following:

(i) It supports eight interrupt inputs from the peripherals and issues a single interrupt signal to the processor.
(ii) It supports cascading of eight 8259As and multiplexes 64 interrupt sources into one.
(iii) It can set priorities for the interrupts, mask the interrupt sources, and provide different interrupt vector addresses.

The interrupts in the controller are individually maskable. The modes and masks can be changed dynamically. It accepts interrupt requests from external devices, determines priority, and checks whether the incoming priority is greater than the current level being serviced and issues the interrupt signal and the corresponding vector address to the processor. In the 8085-based systems, it is provided by a three-byte CALL instruction. In the 8086-based systems, it is provided by an 8-bit interrupt vector number. It can be operated in polled and vectored modes. The starting address of the ISR or the vector number is programmable. No clock is required for the IC.

Using the read/write logic, the 8259 is interfaced with the processor. The data bus lines D0–D7 of the 8259 are connected to the data lines of the processor. The 8259 chip is selected using the $\overline{CS}$ line. Address line A0 is used to select the control word or the data word. If A0 is low, then the controller selects writing a command word/reading a status. If A0 is high, then the controller selects another register for writing the initialization words.

The 8259 control logic has the INT and $\overline{INTA}$ signals. The INT output pin of the 8259 is used to interrupt the CPU. The 8259 receives the interrupt acknowledge pulse from the CPU through its $\overline{INTA}$ input. The 8259 can receive interrupt signals from eight different sources on the lines IR0–IR7. When these lines go high, the

requests are stored in the interrupt request register (IRR). The interrupt service register (ISR) stores all the interrupt levels that are currently being serviced. The interrupt mask register (IMR) stores the masking bits of the interrupt lines to be masked. The priority resolver examines the interrupt registers and determines whether the INT should be sent to the microprocessor or not when an interrupt is received. The cascade buffer or comparator is used to expand the number of interrupt levels by cascading two or more 8259s. The internal block diagram of PIC 8259 is shown in Fig. 7.50.

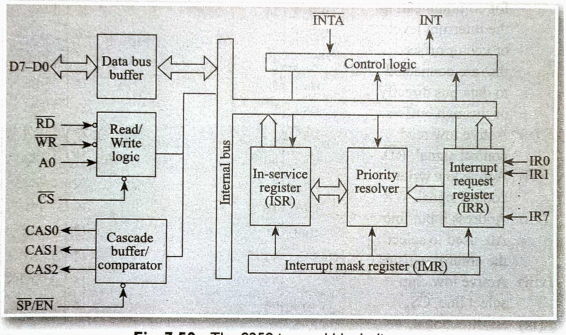


Fig. 7.50 The 8259 internal block diagram

The 8259 can be used in cascaded mode. Up to eight slave 8259s may be cascaded to a master 8259 to provide up to 64 IRQs (interrupt requests). The 8259s are cascaded by connecting the INT line of one slave 8259 to one IRQ line of the master 8259.

The three registers of the 8259 are as follows:

(i) Interrupt mask register (IMR)
(ii) Interrupt request register (IRR)
(iii) In-service register (ISR)

The *IRR* maintains a list of the current interrupts that are pending acknowledgement, the *ISR* maintains a list of the interrupts that are pending to be sent an end of interrupt (EOI), and the *IMR* maintains a mask of interrupts to enable or disable specified interrupts.

The EOI operations support specific EOI, non-specific EOI, and auto-EOI. A specific EOI specifies the IRQ level to be reset in the ISR. A non-specific EOI resets the IRQ level that is currently being serviced in the ISR. Auto-EOI resets the IRQ level in the ISR immediately after the interrupt is acknowledged. Edge and level interrupt trigger modes are supported. Fixed priority and rotating priority modes are supported. The 8259 may be configured to work with an 8086 or an 8085.

7.12.2 Pin Diagram and Details of 8259

The pin details of the 8259 are shown in Fig. 7.51.

The main signals on an 8259 are as follows:

(i) Eight interrupt input request lines named IRQ0–IRQ7

(ii) An interrupt request output line named INT

(iii) Interrupt acknowledgment line named $\overline{\text{INTA}}$

(iv) Bi-directional data bus lines D0–D7 for communicating the interrupt level or vector offset, that are connected to data bus directly or through buffers.

(v) Active low read control signal, $\overline{\text{RD}}$.

(vi) Active low write control signal, $\overline{\text{WR}}$.

(vii) Address input line A0, used to select the control register.

(viii) Active low chip select line, $\overline{\text{CS}}$.

(ix) Bi-directional, 3-bit cascade lines, CAS0–CAS2. In master mode, the PIC places the slave ID number on these lines. In slave mode, the PIC reads the slave ID number from the master on these lines. It may be regarded as slave-select.

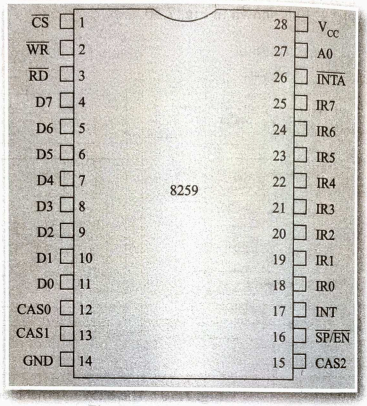


Fig. 7.51 8259 pin diagram

(x) Slave program/enable $\overline{\text{SP}}/\overline{\text{EN}}$: In non-buffered mode, it is an input line, used to distinguish between the master or slave PIC. In buffered mode, it is an output line used to enable buffers.

(xi) Interrupt line, INT, connected to INTR of microprocessor.

(xii) Interrupt acknowledgement, $\overline{\text{INTA}}$, active low signal received from microprocessor.

(xiii) Asynchronous IRQ input lines, IR0–IR7, generated by peripherals.

7.12.3 Initialization of 8259

To service the interrupt requests, the interrupt controller should be initialized by writing control words in the control register. It requires two types of control words—*initialization command words* (ICWs) and *operational command words* (OCWs). The ICWs are used to set up initial conditions and specify the restart vector location. The OCWs are used for masking interrupts, setting up status read operations, etc. The 8259 can be initialized with four ICWs of which the

first two are essential and other two are optional, based on the modes being used. These words must be issued in a given sequence. Once initialized the interrupt controller can be set up to operate in various modes using three different OCWs. The sequence by which the IC8259 must be initialized is shown in Fig. 7.52. Operation command words can be written into the 8259 at any time to perform the specific functions.

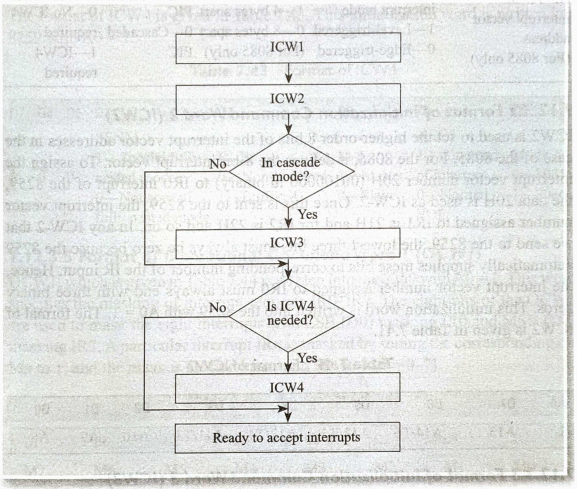


Fig. 7.52 Sequence for 8259 initialization

7.12.3.1 *Format of Initialization Command Word 1 (ICW1)*

ICW1 must be written first into the 8259 with A0 = 0. This should be followed by ICW2. The 8-bit data has the format as given in Table 7.40. The bit definitions are self explanatory. The D2 bit (ADI) is used to set the address interval in the interrupt vector table. It is used only in the 8085, and not in the 8086. If it is 1, then ISR addresses are 4 bytes apart (0200, 0204, etc.) or if it is 0, then the ISR addresses are 8 bytes apart (0200, 0208, etc.). The D3 bit (LTIM) is used to indicate the details about the hardware signal used in IRQ lines. It is used to select whether the signal is level-triggered or edge-triggered. D4–D7 bits are used specify the higher-order bits of the lower byte of the ISR vector address. The higher-order bits are A7, A6, A5 if ADI = 1, and A7, A6 only if ADI = 0. The remaining bits A4–A0 (or A5–A0) are set by the 8259. This is applicable only for the 8085.

Table 7.40 Format of ICW1

D7	D6	D5	D4	D3	D2	D1	D0
A7	A6	A5	1	LTIM	ADI	SNGL	ICW4
Address lines A5–A7 of interrupt vector address (For 8085 only)				Level-triggered interrupt mode 1—Level-triggered 0—Edge-triggered	Address interval 1—4 bytes apart 0—8 bytes apart (For 8085 only)	1—Single PIC 0—Cascaded PIC	Requirement for ICW4 0—No ICW4 required 1—ICW4 required

7.12.3.2 *Format of Initialization Command Word 2 (ICW2)*

ICW2 is used to set the higher-order 8 bits of the interrupt vector addresses in the case of the 8085. For the 8086, it defines the 8-bit interrupt vector. To assign the interrupt vector number 20H (00100000 in binary) to IR0 interrupt of the 8259, the data 20H is used as ICW-2. Once this is sent to the 8259, the interrupt vector number assigned to IR1 is 21H and for IR2 is 22H and so on. In any ICW-2 that we send to the 8259, the lowest three bits must always be zero because the 8259 automatically supplies these bits to corresponding number of the IR input. Hence, the interrupt vector number assigned to IR0 must always end with three binary zeros. This initialization word is written into the 8259 with A0 = 1. The format of ICW2 is given in Table 7.41.

Table 7.41 Format of ICW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

7.12.3.3 *Format of Initialization Command Word 3 (ICW3)*

ICW3 given in Table 7.42 is required only when the PIC 8259 is connected in cascaded mode, that is, more than one 8259 is connected in a system. This initialization word is written into the 8259 with A0 = 1.There are two different formats of ICW3—one for the master and the other for the slave.

Table 7.42 Format of ICW3

	D7	D6	D5	D4	D3	D2	D1	D0
Master	S7	S6	S5	S4	S3	S2	S1	S0
Slave	0	0	0	0	0	ID2	ID1	ID0

(i) For the master 8259, the ICW3 is used to indicate whether a slave 8259 is connected in the interrupt request line IRQ or not. If a bit is 1, it indicates that a slave is present on that interrupt request line. A 0 indicates that it is a direct interrupt request from an external device.

(ii) For the slave 8259, ICW3 assigns the slave with a specific ID number. So

the least significant 3 bits are used for that purpose. ID2–ID0 is the slave ID number. For example, slave 4 has ICW3 = 04H (0000 0100).

7.12.3.4 *Format of Initialization Command Word 4 (ICW4)*

ICW4 is necessary only when it is clearly specified in the ICW-1. It is used to indicate whether the 8085 or the 8086 is used in the system. It specifies the end of interrupt mode, buffered or non-buffered mode, and the special fully nested mode. The format of ICW4 is given in Table 7.43. This initialization word is written into 8259 with A0 = 1.

Table 7.43 Format of ICW4

D7	D6	D5	D4	D3	D2	D1	D0
			SFNM			AEOI	
			1—Special fully	BUF	M/S	1—Auto end	Mode
0	0	0	nested mode	0—Non-buffered	1—Master	of interrupt	0—8085
			0—Not special	1—Buffered	0—Slave	0—Normal	1—8086
			fully nested mode			EOI	

7.12.3.5 *Format of Operational Command Word 1 (OCW1)*

OCW1, which is given in Table 7.44 is written into the 8259 with A0 = 1. This word specifies the masking of interrupt requests IR0–IR7. The eight bits of the OCW1 are used to mask the eight interrupts with LSB (D0) masking IR0 and MSB (D7) masking IR7. A particular interrupt IR*n* is masked by setting the corresponding bit M*n* to 1; and the mask is cleared by setting M*n* to 0 (n = 0–7).

Table 7.44 Format of OCW1

D7	D6	D5	D4	D3	D2	D1	D0
M7	M6	M5	M4	M3	M2	M1	M0

7.12.3.6 *Format of Operational Command Word 2 (OCW2)*

OCW2 is written with the A0 = 0 into the 8259. This word is used to specify the priorities of interrupts and issue of end of interrupt commands. It is usually written to reset a bit in the in-service register. Normally, a bit is set in the ISR whenever the corresponding interrupt is serviced. This is generally written at the end of the interrupt service routine. OCW2 can be programmed for non-specific end of interrupt mode with the data (0010 0000) to automatically reset the ISR bit. The programmer can also use OCW2 to reset a specific ISR bit. OCW2 can also be used to rotate the priorities of the interrupts. The bit format of the OCW2 is given in Table 7.45.

Table 7.45 Format of OCW2

D7	D6	D5	D4	D3	D2	D1	D0
R	SL	EOI	0	0	L3	L2	L1
					IR level to be acted upon		
					000–IR0, 001–IR1, ..., 111–IR7		

(Contd)

Table 7.45 Format of OCW2 *(Contd)*

D7–D5	R	SL	EOI	Action
EOI	0	0	1	Non-specific EOI (L3L2L1 = 000); reset all bits of the in-service register
	0	1	1	Specific EOI command—Clear the bits encoded by L3, L2, and L1 in the in-service register
Auto rotation of priorities (L3L2L1 = 000)	1	0	1	Rotate priorities on non-specific EOI command
	1	0	0	Set—Rotate priorities in auto EOI mode set
	0	0	0	Clear—Rotate priorities in auto EOI mode
Specific rotation of priorities (lowest priority ISR = L3L2L1)	1	1	1	Rotate priority on specific EOI command (reset current ISR bit)
	1	1	0	Set priority (does not reset current ISR bit)
	0	1	0	No operation

7.12.3.7 *Format of Operational Command Word 3 (OCW3)*

OCW3 is used to specify special masking of interrupts. More information on this command can be obtained from the Intel datasheet. The format of OCW3 is given in Table 7.46.

Table 7.46 Format of OCW3

D7	D6	D5	D4	D3	D2	D1	D0
0	ESMM	SMM	0	1	P	RR	RIS
	0X—No effect 10—Reset special mask 11—Set special mask				0—Polling 1—No Polling	0X—No effect 10—Read IR register on next read 11—Read IS register on next read	

7.12.4 Operation of 8259

The following steps show how interrupt handling is done when an external device places an interrupt request on the IR lines of the 8259. It is assumed that the system has a single 8259 chip.

(i) One or more of the IR lines in the 8259 may go high.
(ii) Corresponding IRR bit/bits is/are set in the 8259.
(iii) The 8259 evaluates the interrupt request based on masking and priority.
(iv) If no higher priority interrupt other than the currently received interrupt/interrupts is currently processed then the 8259 sends out interrupt request (INT) signal to the 8086.
(v) The 8086 microprocessor sends out two $\overline{\text{INTA}}$ pulses in response to the INT signal from the 8259.
(vi) The bit corresponding to the highest priority interrupt among the currently

received interrupts in the ISR is set and its corresponding bit in IRR is reset in the 8259.

(vii) After receiving the first $\overline{\text{INTA}}$ pulse from the 8086, the 8259 does the above internal operation and after receiving the second $\overline{\text{INTA}}$ pulse from the 8086, the 8259 sends the desired interrupt type or interrupt vector number of the interrupt currently to be processed by the 8086 through the data bus D7–D0 of the 8259.

(viii) The interrupt service routine (ISR) is executed in the 8086 processor with the following steps:
   (a) The flags are pushed onto the stack.
   (b) The interrupt and trap flags of the processor are cleared.
   (c) The return address is pushed onto the stack.
   (d) The starting address of the interrupt service routine obtained from the interrupt vector table corresponding to the currently received interrupt type or vector number in the program counter and instruction pointer is loaded.
   (e) The ISR is executed.

(ix) At the end of the ISR, instructions are written for sending a command word to reset the bit corresponding to the currently processed interrupt in the ISR of the 8259 so that lower priority interrupts can be processed.

7.12.5 Interfacing of 8259 to 8086

The connection diagram for interfacing the 8259 with the 8-bit processor 8086 is shown in Fig. 7.53.

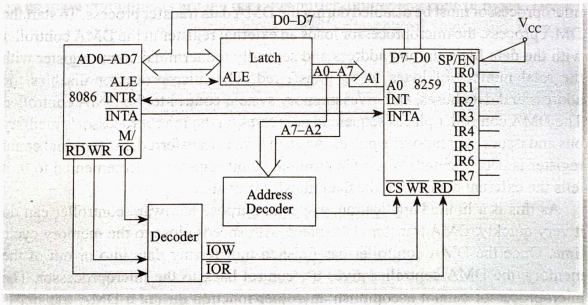


Fig. 7.53 Interfacing of 8259 to 8086

The PIC 8259 requires two addresses with A0 being 0 and 1. The A1 line from the address bus is connected to the A0 line in the 8259. The higher-order address bus is used to select the particular chip through a proper address decoder. Read and write control signals of the 8086 are connected to the corresponding signals of the 8259. The data lines of IC 8259 are connected to the lower-order address and

data bus of the 8086. The multipurpose SP/EN pin is tied to logic high because only one 8259 is used in the system. The interrupt request line, INT of the 8259 is connected to the 8086's interrupt line INTR. The $\overline{\text{INTA}}$ of the 8086 is connected to the $\overline{\text{INTA}}$ of the 8259. When only one 8259 is used in a system, the cascade lines (CAS0, CAS1, CAS2) can be left open. The eight IR inputs of the 8259 can be connected to the interrupt sources from different external devices such as A/D converter, keyboard, and printer. Unused IR inputs must be tied to ground in order to avoid noise being recognized as interrupt signal.

The software part of the 8259 initializing involves writing initialization command words in proper sequence as shown in Fig. 7.52. After initialization, the proper operation command words can be written as and when required.

7.13 8237 DMA CONTROLLER

The programmed data transfers move data from memory into the accumulator, and then from the accumulator into the output ports. A program has to be written to transfer data from a device to the memory in programmed data transfers. Thus, the programmed data transfer is a slow process. This causes a problem only while transferring large amounts of data.

DMA stands for direct memory access. It is one of the ways to accomplish high-speed data transfers, directly between memory and peripheral devices. The DMA is a method of data transfer between memory and I/O devices without the intervention of microprocessor. This method is often used when large block of data is to be transferred.

DMA data transfer is controlled by using a separate DMA controller. The microprocessor must be disabled during the DMA data transfer process. To start the DMA process, the microprocessor loads an external register in the DMA controller with the data file's starting address and secondly the terminal count register with the total number of bytes to be transferred. The microprocessor disables the address and data buses, and gives memory system control to the DMA controller. The DMA controller places sequential addresses on the microprocessor's memory bus and issues the read-write pulses. As each byte is transferred, the terminal count register is decremented. When the terminal count register is decremented to 0, it tells the external device that the data transfer is complete.

As this is a limited application, a special purpose hardware controller can do it very quickly. DMA transfers take place with speeds close to the memory cycle time. Once the DMA controller has finished transferring data into or out of the memory, the DMA controller gives the control back to the microprocessor. The microprocessor cannot accomplish any other function during a DMA transfer is taking place. This is caused due to two reasons. First, the microprocessor's memory is being used for a data transfer. It is not available to supply program instructions or receive the results of computations. Secondly, the typical DMA process requires that the microprocessor place its memory address bus and data bus in a high impedance condition. This high impedance condition allows the DMA controller and the memory system to control the bus but prevents the microprocessor from providing any bus control.

Thus, a dedicated hardware device called direct memory access controller or DMA controller manages the data transfer. The DMA controller temporarily borrows the address bus, data bus, and control bus from the microprocessor and transfers the data bytes directly from the external peripheral devices to a series of memory locations. Because the data transfer is handled totally in hardware, it is much faster than it would be if done by program instructions.

7.13.1 Features, Pin Details, and Architecture of 8237

The DMA controller 8237 is designed to improve the data transfer rate in systems that transfer data from an I/O device to memory, or move a block of memory to an I/O device. It also performs memory-to-memory block moves, or fills a block of memory with data from a single location. Operating modes are provided to handle single byte transfers as well as discontinuous data streams. The DMA controller permits data to be transferred directly from an I/O device to memory or vice versa without the need for a temporary register. This increases the data transfer rate for sequential operations, compared with processor moves or repeated string instructions. The main features of the 8237 are as follows:

(i) Four independent DMA channels
(ii) Enable and disable control of individual requests
(iii) Possibility for memory-to-memory transfer
(iv) Address increment or decrement
(v) Cascading and expandable to any number of DMA channels

The block diagram of the 8237 is shown in Fig. 7.54. The pin details of the 8237 are shown in Fig. 7.55.

The main components are the data bus buffer, timing and control block, DMA channels, corresponding priority block, read/write control logic, and internal

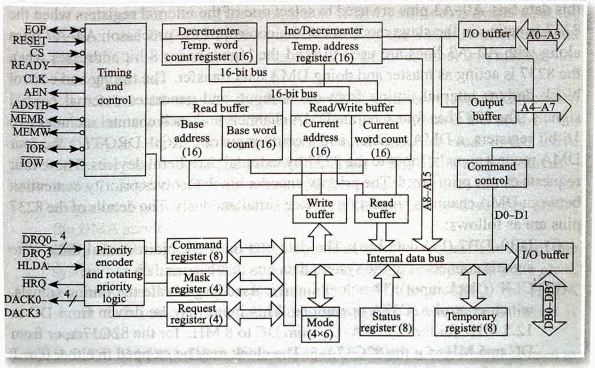


Fig. 7.54 Block diagram of the 8237

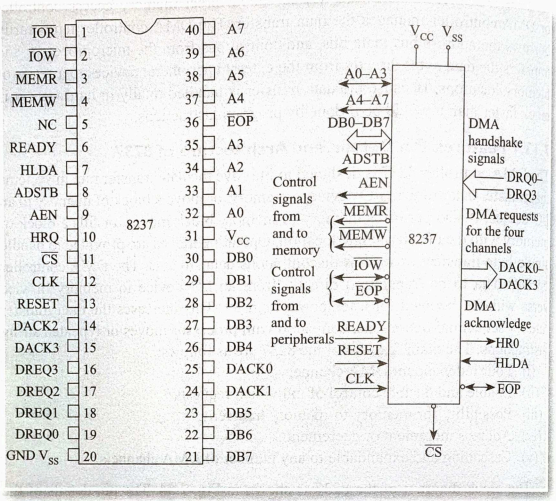


Fig. 7.55 Pin details of the 8237

registers. The data bus consists of 8-bit tri-state pins DB0–DB7. These pins are connected to the system data bus. The programming of the 8237 is done through this data bus. A0–A3 pins are used to select one of the internal registers when the 8237 is acting in the slave mode under the control of the processor. A4–A7 lines along with A0–A3 lines are used to send the higher-order 8-bit addresses when the 8237 is acting as master and doing DMA data transfer. The timing and control block derives internal timing from clock input, and generates external control signals. The 8237 has four separate DMA channels and each channel includes two 16-bit registers, a DMA register, and a count register. DRQ0–DRQ3 are the four DMA request signals, input to the 8237 by external peripheral devices. These four requests can be prioritized. The priority encoder block resolves priority contention between DMA channels requesting service simultaneously. The details of the 8237 pins are as follows:

(i) DB0–DB7 (I/O data bus): The data bus lines are bidirectional three-state signals connected to the system data bus that carries data.

(ii) CLK (Clock input): The clock input is used to generate the timing signals, which control 82C37A operations. This input may be driven from DC to 12.5 MHz for the 82C37A-12, from DC to 8 MHz for the 82C37A, or from DC to 5 MHz for the 82C37A-5. The clock may be stopped in either 0 or 1 states for standby operation.

(iii) CS (Chip select): Chip select is an active low input used to enable the controller.

(iv) Reset: This is an active high input, which clears the command, status, request, and temporary registers, the first/last flip-flop, and the mode register counter. The mask register is set to ignore requests. Following a reset, the controller is in an idle cycle.

(v) Ready: This signal can be used to extend the memory read and write pulses from the 82C37 to accommodate slow memories or I/O devices.

(vi) HLDA (Hold acknowledge): The active high hold acknowledge, from the CPU indicates that it has hand over control of the system busses.

(vii) DREQ0–DREQ3 (DMA request): The DMA request (DREQ) lines are individual asynchronous channel request inputs used by peripheral circuits to obtain DMA service. In fixed priority mode, DREQ0 has the highest priority and DREQ3 has the lowest priority. A request can be generated by activating the DREQ line of a channel. Polarity of DREQ is programmable. RESET initializes these lines to an active high. DREQ must be maintained until the corresponding DACK goes active. DREQ will not be recognized while the clock is stopped.

(viii) $\overline{\text{IOR}}$ (I/O read): It is a bidirectional active low three-state line. In the idle or slave mode, it is an input control signal used by the CPU to read the control registers. In the active or master mode, it is an output control signal used by the 82C37 to access data from the peripheral during a DMA write transfer.

(ix) $\overline{\text{IOW}}$ (I/O write): It 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 82C37A. In the active cycle, it is an output control signal used by the 82C37A to load data to the peripheral during a DMA read transfer.

(x) EOP (End of process): The EOP is an active low bidirectional signal. Information concerning the completion of DMA services is available at the bidirectional EOP pin. A pulse is generated by the 82C37A when terminal count (TC) for any channel is reached, except for channel 0 in memory-to-memory mode.

(xi) A0–A3 (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 82C37A to address the control register to be loaded or read. In the active cycle, they are outputs and provide the lower 4 bits of the output address.

(xii) A4–A7 (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.

(xiii) HRQ (Hold request): The HRQ output is used to request control of the system bus. When a DREQ occurs and the corresponding mask bit is clear, or a software DMA request is made, the 82C37A issues HRQ. The HLDA signal then informs the controller when access to the system busses is permitted.

(xiv) DACK0–DACK3 (DMA acknowledge): The DMA acknowledge is used to notify the individual peripherals when one has been granted a DMA cycle.

DACK acknowledges the recognition of a DREQ signal.

(xv) AEN (Address enable): Address enable signal is an active high signal used to indicate the availability of higher-order 8-bit address and can be used by the latch to store the same. The AEN can also be used to disable other system bus drivers during DMA transfers.

(xvi) ADSTB (Address strobe): This is an active high signal used to control latching of the upper address byte.

(xvii) $\overline{\text{MEMR}}$ (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.

(xviii) $\overline{\text{MEMW}}$ (Memory write): The memory write signal is an active low three-state output used to write data to the selected memory location during a DMA write or a memory-to-memory transfer.

7.13.1.1 *Register Description*

The name and size of the internal registers are listed in Table 7.47. The details of these registers are explained in this section.

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 by one after each transfer and the 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.

Table 7.47 Internal registers in the 8237

Name	Size
Base address registers	16 bits
Base word count registers	16 bits
Current address registers	16 bits
Current word count registers	16 bits
Temporary address register	16 bits
Temporary word count register	16 bits
Status register	8 bits
Command register	8 bits
Temporary register	8 bits
Mode registers	6 bits
Mask register	4 bits
Request register	4 bits

Current word count 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 50 will result in 51 transfers). The word count is decremented after each 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.

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. These registers cannot be read by the microprocessor.

Command register This 8-bit register controls the operation of the 8237. It is programmed by the microprocessor and is cleared by RESET or a master clear instruction. The function of the command register bits is shown in Fig. 7.56.

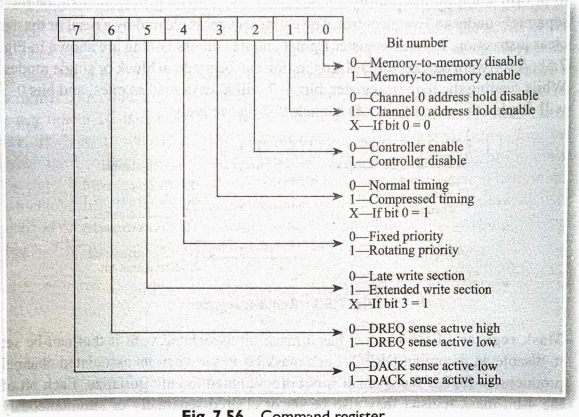


Fig. 7.56 Command register

Mode register Each channel has a mode register associated with it. When the register is being written into by the microprocessor in the program condition, least significant bits 0 and 1 determine, which channel is chosen. The details of the mode register bits are shown in Fig. 7.57.

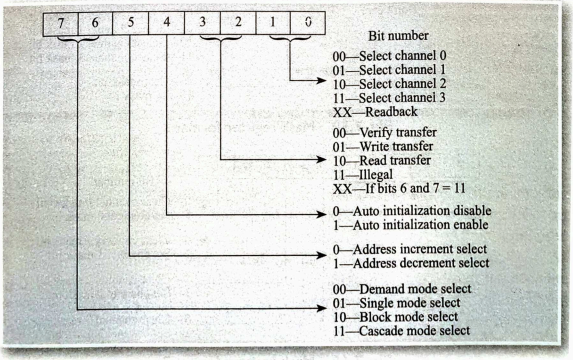


Fig. 7.57 Mode register

Request register The 8237 can respond to requests for DMA service, which are initiated by software or by DREQ input. 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. The entire register is cleared by a reset or master clear instruction. Request register format and its address coding are shown in Fig. 7.58. A software request for DMA operation can be made in block or single modes. While reading the request register, bits 4–7 will always read as ones, and bits 0–3 will display the request bits of channels 0–3 respectively.

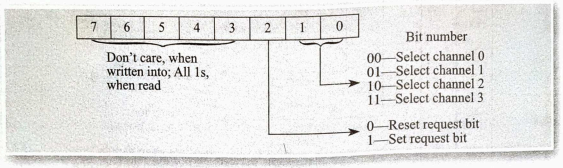


Fig. 7.58 Request register

Mask register Each channel has a mask bit associated with it that can be set to disable an incoming DREQ. Each mask bit is set when its associated channel produces an EOP if the channel is not programmed to auto initialize. Each bit of the 4-bit mask register may also be set or cleared separately or simultaneously under software control. The entire register is also set by a reset or master clear. This disables all hardware DMA requests until a clear mask register instruction allows them to occur. The mask register formats are shown in Figs 7.59 and.7.60.

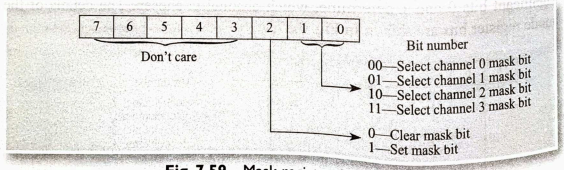


Fig. 7.59 Mask register format 1

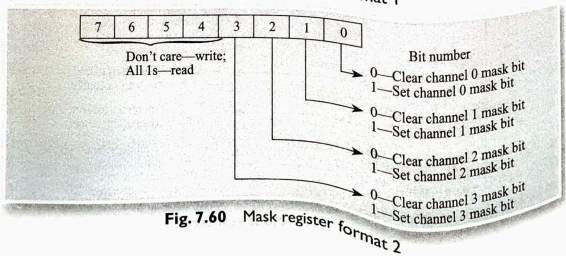


Fig. 7.60 Mask register format 2

While reading the mask register, bits 4–7 will always read as logical ones, and bits 0–3 will display the mask bits of channels 0–3, respectively. The four bits of the mask register can be cleared simultaneously by using the clear mask register command and they can also be written with a single command.

Status register The status register contains information about the status of the devices at any time to be read by the processor. The format of the status register is shown in Fig. 7.61. 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, master clear, and on each status read. Bits 4–7 are set when their corresponding channel is requesting service irrespective of the mask bit state. If the mask bits are set, software can poll the status register to determine, which channels have DREQs, and selectively clear a mask bit, thus allowing user defined service priority. Status bits 4–7 are updated while the clock is high. Status bits 4–7 are cleared upon RESET or master clear.

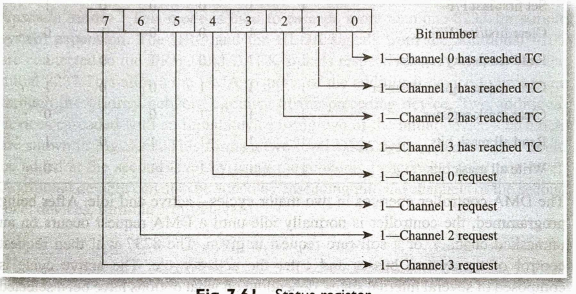


Fig. 7.61 Status register

Temporary register The temporary register is used to hold data during memory-to-memory transfers. The temporary register always contains the last byte transferred in the previous memory to memory operation, if not cleared by a reset or master clear.

7.13.2 DMA Initialization and Operation

Initializing the 8237 requires a large number of bytes to be written into the registers as discussed in Section 7.13.1.1. The 8237 is connected as an I/O port with the processor. In the idle cycle or idle mode, the 8237 A3–A0 lines are used to program the internal registers and operation of the DMA controller 8237. As discussed earlier, the 8237 has four channels of DMA request. So, it has separate registers to hold the base memory address, current memory address, and the count register in each channel. In general A1 and A2 are used to select one of the four DMA channel registers and A0 is used to select the memory address or count

register. The DMA channel registers are accessed when A3 = 0. If A3 = 1, then the other control registers are accessed according to Table 7.48.

Table 7.48 Selection of control registers using control signals

Operation	A3	A2	A1	A0	IOR	IOW
Read status register	1	0	0	0	0	1
Write command register	1	0	0	0	1	0
Read request register	1	0	0	1	0	1
Write request register	1	0	0	1	1	0
Read command register	1	0	1	0	0	1
Write single mask bit	1	0	1	0	1	0
Read mode register	1	0	1	1	0	1
Write mode register	1	0	1	1	1	0
Set first/last F/F	1	1	0	0	0	1
Clear first/last F/F	1	1	0	0	1	0
Read temporary register	1	1	0	1	0	1
Clear mode register counter	1	1	1	0	0	1
Clear mask register	1	1	1	0	1	0
Read all mask bits	1	1	1	1	0	1
Write all mask bits	1	1	1	1	1	0

The DMA controller operates in two major cycles—active and idle. After being programmed, the controller is normally idle until a DMA request occurs on an unmasked channel, or a software request is given. The 8237 will then request control of the system busses and enter the active cycle. The active cycle is composed of several internal states, depending on what options have been selected and what type of operation has been requested.

7.13.2.1 *Idle Cycle*

When no channel is requesting service, the 8237 enters the idle cycle. In this cycle, the 8237 samples the DREQ lines on the falling edge of every clock cycle to determine if any channel is requesting a DMA service.

7.13.2.2 *Active Cycle*

When the 8237 is in the idle cycle, and a software request or an unmasked channel requests a DMA service, the device issues a HRQ to the microprocessor and enters the active cycle. It is in this cycle that the DMA service takes place in one of the following four modes:

Single transfer mode In single transfer mode, the device is programmed to make one transfer only. The word count is decremented and the address is decremented or incremented following each transfer. When the word count rolls over from zero

to FFFFH, a terminal count bit in the status register is set, and an EOP pulse is generated. DREQ must be held active until DACK becomes active. If DREQ is held active throughout the single transfer, HRQ will go inactive and releases the bus to the system. It becomes active again and upon receipt of a new HLDA, another single transfer is performed. The exception for this occurs when a higher priority channel takes over.

Block transfer mode In block transfer mode, the device is activated by DREQ or software request continues making transfers until a TC, caused by word count going to FFFFH, or an external End of Process (EOP) is encountered. The DREQ need only be held active until the DACK becomes active.

Demand transfer mode In demand transfer mode the device continues making transfers until a TC or external EOP is encountered, or until DREQ goes inactive. The data transfer continues until the I/O device has exhausted its data capacity. Higher priority channels may intervene in the demand process, once DREQ has gone inactive. The EOP is generated either by TC or by an external signal.

Cascade mode This mode is used to cascade more than one 8237 for simple system expansion. The HRQ and the HLDA signals from the additional 8237 are connected to the DREQ and DACK signals respectively of a channel for the initial 8237.This allows the DMA requests of the additional device to propagate through the priority network circuitry of the preceding device. Two additional devices cascaded with an initial device using two of the initial device's channels are shown in Fig. 7.62. This forms a two-level DMA system. More 8237s 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 devices, forming a third level.

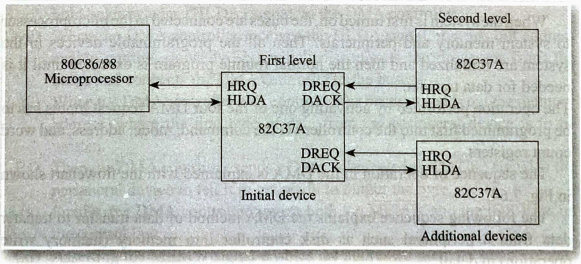


Fig. 7.62 Cascaded 8237s

7.13.3 Operation of 8237 with 8086

The block diagram in Fig. 7.63 shows, how a DMA transfer takes place between a memory and an I/O device with the help of a DMA controller. Here, the microprocessor and the DMA controller timeshare the use of address, data, and control buses. The 8237 address, control outputs, and data bus pins are connected

in parallel with the system busses. An external latch is required for the upper address byte. While inactive, the controller's outputs are in a high impedance state. When activated by a DMA request and bus control is surrendered by the host, the 8237 drives the buses and generates the control signals to perform the data transfer.

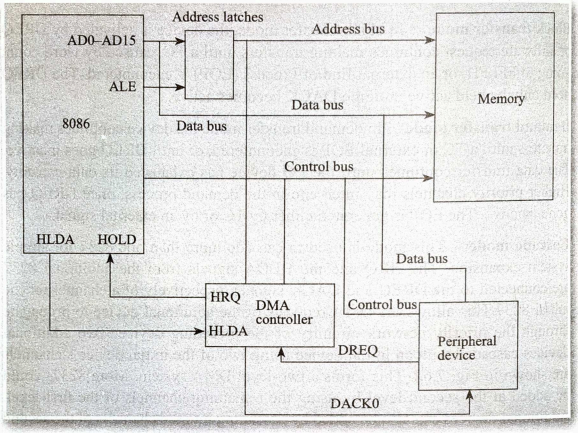


Fig. 7.63 Interfacing DMA controller with the processor

When the system is first turned on, the buses are connected to the microprocessor to system memory and peripherals. Then all the programmable devices in the system are initialized and then the normal routine program is executed until it is needed for data transfer.

The operation performed by activating one of the four DMA request inputs has to be programmed first into the controller via the command, mode, address, and word count registers.

The sequence of operation in the DMA is explained with the flowchart shown in Fig. 7.64.

The following sequence explains the DMA method of data transfer to transfer data from a peripheral such as disk controller into memory (memory write operation) in detail.

(i) The starting memory address, where the data is stored is loaded into the 8237 address registers for a particular channel, and the length of the block is loaded into the channel's word count register.

(ii) The corresponding mode register is programmed for I/O to memory operation (write transfer), and various options are selected by the command register and the other mode register bits.

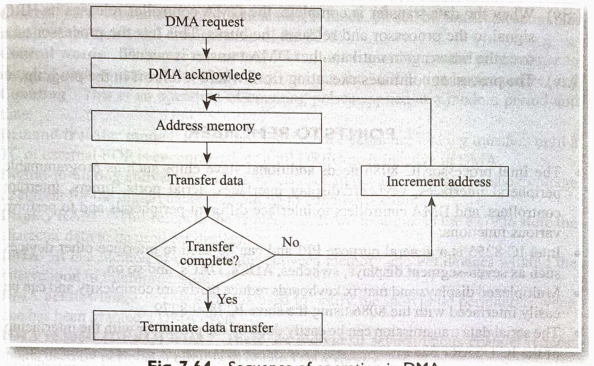


Fig. 7.64 Sequence of operation in DMA

(iii) The channel's mask bit is cleared to enable recognition of a DMA request (DREQ). The DREQ can either be a hardware signal or a software command.

(iv) When the peripheral device has the first byte of data ready, it sends a DREQ signal to the DMA controller.

(v) If the input (channel) of the DMA controller is unmasked, the DMA controller sends a hold request (HRQ) signal to the microprocessor HOLD input.

(vi) The microprocessor responds to this input by floating its buses and sends a hold-acknowledge (HLDA) signal, to the DMA controller.

(vii) When the DMA controller receives the HLDA signal, it sends out address enable control signal (AEN) which disconnects the processor from the buses and connects the DMA controller to the buses.

(viii) When the DMA controller gets control of the buses, it sends out the memory address, where the first byte of data from the peripheral device is to be written.

(ix) Then the DMA controller sends a DMA acknowledge (DACK) signal to the peripheral device to tell it to get ready to output the byte.

(x) Finally, the DMA controller asserts both the $\overline{\text{MEMW}}$ and the $\overline{\text{IOR}}$ lines on the control bus.

(xi) Asserting the $\overline{\text{MEMW}}$ signal enables the addressed memory to accept data written into it.

(xii) Asserting the $\overline{\text{IOR}}$ signal enables the disk controller to output the byte of data from the disk on the data bus.

(xiii) Then the byte of data is transferred directly from the peripheral device to the memory location without passing through the CPU or the DMA controller. The DMA transfers may be done a byte at a time or in blocks.

(xiv) When the data transfer is complete, the DMA controller releases its HRQ signal to the processor and releases the buses. This lets the processor take over the buses again until another DMA transfer is needed.

(xv) The processor continues executing from where it left off in the program.

POINTS TO REMEMBER

- The Intel processor IC 8086 needs additional slave chips such as programmable peripheral interfaces, keyboard/display interfaces, serial ports, timers, interrupt controllers, and DMA controllers to interface different peripherals and to perform various functions.
- Intel IC 8255 is a general purpose PPI and can be used to interface other devices such as seven-segment displays, switches, ADCs, DACs, and so on.
- Multiplexed displays and matrix keyboards reduce hardware complexity and can be easily interfaced with the 8086 using the slave IC Intel 8279.
- The serial data transmission can be easily done by the processor with the interfacing of the IC USART 8251.
- The timing of various events can be controlled by the programmer by interfacing the timer IC such as 8253 to the processor and connecting a clock to it.
- There is a need of programmable interrupt controller if the number of peripherals interfaced using the interrupt driven I/O method is higher than the interrupt capability of the processor. Intel provides the programmable interrupt controller IC 8259 for such applications.
- The high speed data transfer between I/O devices and the processor can be achieved by using a technique called direct memory access. The DMA controller IC 8237 can be interfaced with the processor to achieve the process of direct memory access of memory by I/O devices.

KEY TERMS

Active cycle This is the cycle during which DMA service takes place.

Analog–to-digital converter The ADC converts the input analog voltage levels into corresponding discrete digital signals.

Asynchronous transmission This is the method of serial data transfer without a common clock but at a common baud rate and it is character oriented.

Baud rate This is the rate at which serial data is being transferred.

Bit set–reset mode The BSR mode is applicable to port C of the 8255 for setting and resetting individual port C bits.

Block transfer mode It is the mode in which the device that is activated by DREQ or software request continues making transfers during the service until a TC caused by word count going to FFFFH, or an external end of process (EOP) is encountered in DMA.

Cascade mode In this mode, the system is constructed using more than one 8237s cascaded for simple system expansion.

Cascading It is the method of connecting more than one 8259s in a microcomputer system in order to increase the number of interrupt sources.

Command instruction It is used for setting the operation features of the 8251.

Control word It contains information such as mode, bit set, bit reset, and so on that initializes the functional configuration of the 8255.

Control words These are commands that must be sent out by the programmer to initialize each counter of the 8253. They program the MODE, loading sequence, and

Counting This is an operation of counting pulses applied at a random period and time.

Demand transfer mode In this mode, the device continues making transfers until a TC or external EOP is encountered, or until DREQ goes inactive in DMA.

Digital-to-analog converter The DAC is used to get a proportional analog voltage or current for the digital data given out by the microprocessor.

Display RAM This refers to the sequence of RAM locations in the 8279 to store the character data to be used for display.

DMA It is a method of data transfer between memory and I/O devices without the intervention of microprocessor.

DMA acknowledge This signal is used to notify the individual peripherals when one has been granted a DMA cycle.

DMA request (DREQ) lines These are individual asynchronous channel request inputs used by peripheral circuits to obtain DMA service.

FIFO RAM It is the sequence of RAM locations in the 8279 to store the key code pressed in a matrix keyboard interfaced.

Idle cycle This is the state of the system when no DMA channel is requesting service.

Initialization command words These are used to set up proper conditions and specify restart vector location in the 8259.

Input/output mode The I/O mode is applicable to ports A, B, and C of the 8255 for programming the data transfer and direction of data transfer.

Interrupt mask register This register stores the masking bits of the interrupt lines to be masked in the 8259.

Interrupt service register This register stores all the levels that are currently being serviced in the 8259.

Key board debouncing It is a process of removing switch transient voltages and detecting an actual key press.

Matrix keyboard It is an arrangement of switches in matrix wiring so that it can be interfaced with the processor with minimum hardware and scanning technique.

Mode instruction It is used for setting the function of the 8251.

Modem It is a modulator or demodulator, which send digital 1s and 0s over standard phone lines as modulated tones.

Multiplexed display This is a method of interfacing many display devices to a processor and using scanning method to display digits with one digit being displayed at a time.

Operational command words These are used for masking interrupts, setting up status read operations, etc. in the 8259.

Priority resolver It examines the interrupt registers and determines whether the INT should be sent to the microprocessor or not in the 8259.

Programmable timer A device in which the initial count value can be loaded using the data from the data bus, and counting can be started and stopped using software instructions written to the control register.

the type of counter as binary or BCD counting.

Rate generator The frequency output of this mode will be equal to the input frequency divided by *N*.

Serial communication This refers to sending and receiving information bit by bit.
Single transfer mode It is the mode in which the device is programmed to make one transfer only in DMA.
Synchronous transfer This is the method of serial transfer by which the transmission and reception of data is done with a common clock and simultaneously.
Timing This is an operation of counting, using a precise clock pulse at fixed frequency.

REVIEW QUESTIONS

1. Name the registers available in the 8255.
2. Give the control word format for I/O mode operation in the 8255.
3. Write a brief note on the different I/O modes in the 8255.
4. Write the BSR mode control word format in the 8255.
5. List the components needed to interface seven-segment displays to the 8086.
6. Find the data direction and the modes of operation of ports of the 8255 if the control word written into is 80H.
7. Specify the handshaking signals and their functions if port A of the 8255 is setup as input port in mode 1.
8. Describe the function of EOC and SC signals in ADC interface to the 8086.
9. How can the frequency of the waveform generated using DAC be changed?
10. Why do you need a driver circuit for interfacing a LED to port pin?
11. With encoded scan keyboard mode, the total number of keys that can be connected to the 8279 is 128. Justify this statement.
12. Describe the block diagram of the 8279 keyboard/displace interface.
13. What are the functions performed by the 8279?
14. Describe the different modes of operation of the keyboard interface to the 8279.
15. What are the different formats of display possible with the 8279?
16. What are the different control words of the 8279? Explain the function of each command.
17. Name the applications of the 8253.
18. What is the difference between the 8253 and the 8254?
19. Explain how to configure a timer/counter through software.
20. List the various operating modes of the 8253.
21. How will you determine the output frequency of the 8253 in rate generator mode?
22. What is the difference between hardware-triggered strobe and software-triggered strobe?
23. Compare serial and parallel communication.
24. Compare simplex and duplex transmission.
25. What is the difference between synchronous and asynchronous serial data transfer?
26. What is a modem?
27. Compare RS 232, RS 422, and RS 432 standards.
28. Draw the block diagram and explain the operations of the 8251 serial communication interface.
29. Write and explain the mode word, command word, and status word formats of the 8251.

30. List the features modified by the mode instruction of the 8251.
31. Name the features modified by the command instruction of the 8251.
32. Synchronous mode of the 8251 is used for very high rate of data transfer. Is this statement true or false? Justify your answer.
33. The order of instructions used to initialize the 8279 is important. Is this statement true or false? Justify your answer.
34. Explain how data can be transferred using the 8251 USART at different baud rates.
35. What is the need for an interrupt controller?
36. Compare maskable and non-maskable interrupts.
37. What is a priority resolver?
38. List the internal registers in the 8259.
39. What is EOI in the 8259?
40. Explain the initialization process of the 8259.
41. Explain how the 8259 communicates with the 8086. Explain the different functions available in the priority interrupt controller.
42. Draw the block diagram of the 8259 and explain how it can be used for increasing the interrupting capabilities of the 8086.
43. How is DMA better than programmed data transfer?
44. Give examples of I/O devices that can be interfaced with DMA.
45. Give the sequence of operation carried out in DMA.
46. List the internal registers in the 8237.
47. Explain how data is transferred between the RAM and an I/O device using DMA.
48. Discuss the different modes of operation in the 8237.
49. Write a note on the cascaded mode of operation of the 8237.
50. Describe in detail how the 8237 can be interfaced with the 8086 processor.

NUMERICAL/DESIGN-BASED EXERCISES

1. Find the BSR control words for setting PC4 pin and resetting PC2 pin in the 8255.
2. Configure the ports of the 8255 (PPI) as follows: port A as output, port B as input, port C higher as output, port C lower as input. (Assume that the control register of the 8255 PPI is located at the address 26H.)
3. Design an interface using the 8279 for interfacing six seven-segment displays and a matrix hexadecimal keypad to work with the 8086 processor and explain the software needed to find the key pressed in the keyboard and display the same in the display.
4. Write an 8086 program to set up the 8253 as a square wave generator with a 10 millisecond period (Assume input clock frequency to the 8253 is 1 MHz).
5. Write an 8086 program using the 8253 to generate a PWM signal whose frequency and pulse width can be changed. (Hint: You can use two timers—one in programmable one-shot mode to generate variable pulse width and the other in rate generator mode to trigger the one-shot mode counter at desired frequency.)
6. Write an 8086 assembly language program (ALP) to initialize the 8251 USART

and receive 10 bytes of data on polled basis and store them in memory from address 3000H: 1000H with the following parameters: baud rate factor = 64, character length = 8 bits, no parity check, and 1 stop bit. Assume port address 50H for data and 52H for control/status.

PROGRAMMING EXERCISES

1. Draw and explain a typical stepper motor interface. Further, write an 8086 ALP to rotate the shaft of a 4-phase stepper motor five times in clockwise direction.
2. Write an 8086 ALP to a generate square wave of 1 KHz using the 8255, which is interfaced with the 8086. The clock frequency of the 8086 is 5 MHz.
3. Show how you would interface a 3 × 3 matrix keyboard having keys 1–9 with an 8086 processor using the 8255. Write an 8086 ALP to find the ASCII code of key pressed and store it in DL register.
4. Interface a set of eight simple switches and eight simple LEDS with the 8086 using an 8255 PPI chip. The 8255 should be selected for the following memory addresses: port A = 1740H, port B = 1742H, port C = 1744H, and CWR = 1746H. Write an 8086 ALP to indicate the status of the switches on the LEDs.
5. Assume that a key matrix with the keys 0–9 and with *, −, /, + keys and an eight digit display unit are interfaced with the 8279. Develop an 8086 ALP for using the display and the keyboard as a calculator.

8

Multiprocessor Configuration

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Necessity and advantages of a multiprocessor system
- Difference between closely-coupled and loosely-coupled multiprocessor systems
- Interconnection topologies between processors and memories in a multiprocessor system
- Physical interconnections between processors in a multiprocessor system
- Multiprocessor system containing 8086 and 8087 (numeric coprocessor)
- Multiprocessor system containing 8086 and 8089 (I/O processor)

8.1 INTRODUCTION

The speed of any microprocessor-based system depends upon the clock frequency at which it is operating, amongst other factors such as the presence of a pipeline execution unit and the microprocessor's on-chip cache. For example, when bulk I/O data transfer is done under the control of the microprocessor alone, the processor has to spend most of its time idle due to the slow operating speed of the peripherals. A single processor system has an upper limit on its processing capability. For further enhancement of the speed of operation, an appropriate system involving several connected processors using a certain topology may provide the solution. Such a system is called *multiprocessor system*. If a system having a single processor takes a particular duration to complete a task, a system having more than one processor may require lesser time.

The simplest type of multiprocessor system consists of a CPU (such as the 8086) and a numeric data processor (NDP) (also called *numeric coprocessor*) or an input/output processor (IOP). The NDP is an independent processing unit that is capable of performing complicated numeric calculations in comparatively lesser time than the microprocessor. The NDP works in coherence with the microprocessor. The I/O operations in a microprocessor-based system are slow due to the low operating speed of the I/O devices. An IOP takes care of the I/O activities of the 8086-based system and thus saves the time of the main processor (the 8086). The NDP and IOP work in synchronism with the main processor to complete specific tasks and are called *coprocessors*. Coprocessors do not work independently, as they cannot fetch code from the memory. They work under the control of the main processor. Additional hardware circuits such as bus arbiter and bus controller may be needed to coordinate the activities of all the processors working at a time in the system.

8.2 MULTIPROCESSOR SYSTEM—NEED AND ADVANTAGES

By using a DMA controller with a CPU (such as the 8086), the system throughput can be improved by concurrently performing I/O data transfer while the CPU continues its processing. This is possible because the CPU does not utilize all bus cycles. Depending on the application, the 8086 typically uses only 50% to 80% of the available bus time. A DMA controller can steal bus cycles to transfer data between the memory and the I/O devices, while affecting the CPU processing minimally. It releases the CPU from performing the relatively slow I/O data transfer operations. Such a system, having more than one processor such as the 8086 and the DMA controller (8257 or 8237), with both of them operating in parallel to improve the system performance, is an example of a multiprocessor system.

In general, if a system includes two or more processors that can execute instructions simultaneously, it is called a *multiprocessor system*. The additional processors could be general-purpose processors or special-purpose processors that are specifically designed to perform certain tasks efficiently. For example, due to the 8086's limited data bus width (16 bits) and its lack of floating-point arithmetic instructions, it requires many instructions to perform a single floating-point operation. For a system requiring several floating-point computations, it is desirable to perform such computations with a supporting numeric coprocessor such as the 8087, which is specifically designed to quickly operate on floating-point numbers and numbers having larger size, such as 32 bits, 64 bits, and 80 bits. Sometimes, it is advantageous to include in a system, an I/O processor such as the 8089, which has greater capabilities than a DMA controller, since the 8089 can perform string manipulations, character searching, and bit testing as well as the normal DMA operations. This permits the 8086 CPU to concentrate on higher-level functions.

As the ratio of cost to performance of a single-chip microprocessor reduces day by day, it becomes more cost effective to use multiple processors than to use a single complex processor. In addition to improving the overall cost to performance ratio of a system, a multiprocessor configuration offers several desirable features that are not found in a single complex processor design. Some of these features are listed here:

(i) Several processors may be combined to fit the needs of an application, while avoiding the expense of the unnecessary capabilities of a single complex processor.

(ii) The modularity of a multiprocessor system provides means for expansion because it is easy to add more processors as the need arises.

(iii) In a multiprocessor system, tasks are divided among the processors. If a failure occurs, it is easier and cheaper to find and replace the malfunctioning processor than it is to find and replace the failing part in a complex processor.

Two problems—bus contention and inter-processor communication—must be considered while designing a multiprocessor system. Since more than one processor shares the system memory and the I/O devices through a common

system bus, extra logic must be included to ensure that only one processor has access to the system bus at a time. For one processor to send a task or return a result to another processor, an unambiguous way must be provided for the two processors to interact. The connections between the processors are dictated by how the bus contention and processor communication problems are resolved.

8.3 DIFFERENT CONFIGURATIONS OF MULTIPROCESSOR SYSTEM

The maximum mode operation of the 8086 is specifically designed to implement multiprocessor systems. Multiprocessing features are provided in the maximum mode operation of the 8086, to accommodate three basic configurations—the coprocessor, the closely-coupled, and the loosely-coupled configurations.

8.3.1 Coprocessor and Closely-coupled Configurations

The coprocessor and the closely-coupled configurations are similar, as both the CPU (i.e., 8086) and the external/supporting processor share not only the entire memory and the I/O subsystem, but also the same bus control logic and clock generator, as shown in Fig. 8.1. In both these configurations, the 8086 is the master or the host, and the supporting processor is the slave. Since the bus access control is provided by the 8086, the bus request signal from the supporting processor is connected to the 8086. In the closely-coupled configuration, the supporting processor may act independent of the CPU, but in the coprocessor design, it is dependent on the CPU and must interact directly with the CPU. Since the 8086 always acts as the host in the coprocessor and closely-coupled designs, two 8086 processors cannot appear in these configurations. In a coprocessor arrangement, there are more direct connections between the processing elements.

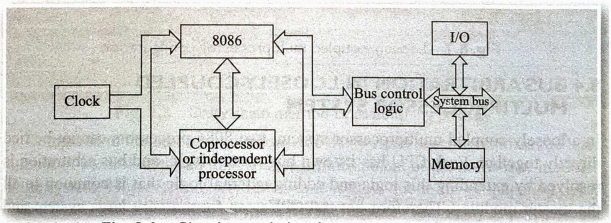


Fig. 8.1 Closely-coupled multiprocessor configuration

8.3.2 Loosely-coupled Configuration

The loosely-coupled configuration is used for medium to large size systems. This configuration is shown in Fig. 8.2. Each module in a loosely-coupled system may act as the system bus master, and may consist of an 8086 or another processor capable of being a bus master, a coprocessor, or a closely-coupled configuration. Several modules may share the system resources and the system bus control logic must resolve the bus contention problem. Each potential bus master runs independently

and there are no direct connections between them. Inter-processor communication is made possible through the shared resources. In addition to the shared resources, each module may include its own memory and I/O devices. The processors in the separate modules can simultaneously access their private subsystems through their local buses and perform their local data references and instruction fetches independently, thus improving the degree of concurrent processing.

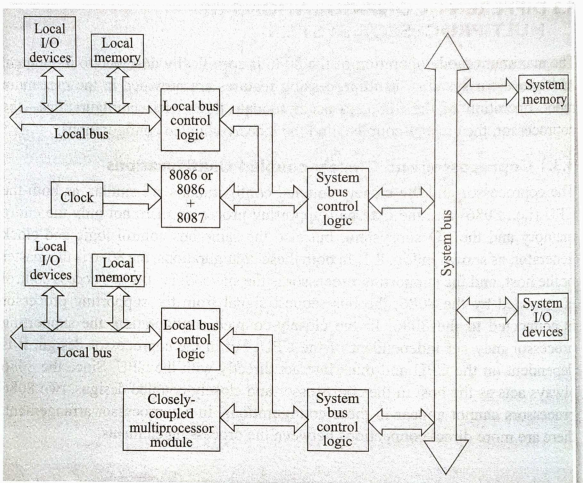


Fig. 8.2 Loosely-coupled multiprocessor configuration

8.4 BUS ARBITRATION IN LOOSELY-COUPLED MULTIPROCESSOR SYSTEM

In a loosely-coupled multiprocessor system, two 8086 processors cannot be tied directly together. Each CPU has its own bus control logic, and bus arbitration is resolved by extending this logic and adding external logic that is common to all the master modules. Therefore, several CPUs can form a very large system and each CPU may have independent processors and/or a coprocessor attached to it. A loosely-coupled configuration provides the following advantages:

(i) The system can be expanded in a modular form. Each bus master module is an independent unit and normally resides on a separate PC board and hence, a bus master module can be added or removed without affecting the other modules in the system.

(ii) High system throughput can be achieved by having more than one CPU.

(iii) A failure in one module does not cause a breakdown of the entire system; the faulty module can be easily detected and replaced.

(iv) Each bus master may have a local bus to access dedicated memory or I/O devices, so that a greater degree of parallel processing can be achieved.

In a loosely-coupled multiprocessor system, more than one bus master module may have access to the shared system bus. Since each master is running independently, extra bus control logic must be provided to resolve the bus arbitration (i.e., allotment of system bus to a particular requesting master) problem. This extra logic is called *bus access logic* and its responsibility is to make sure that only one bus master at a time has control of the bus. Simultaneous bus requests are resolved on a priority basis. There are three schemes for establishing priority—daisy chaining, polling, and independent requesting. The three schemes are discussed in Sections 8.4.1–8.4.3.

8.4.1 Daisy Chaining

Figure 8.3 shows the daisy chaining scheme of establishing priority. The daisy chain method is characterized by its simplicity and low cost. All the masters use the same line for making bus requests. To respond to a Bus Request (BR) signal, the controller sends a Bus Grant (BG) signal if the Bus Busy (BB) signal is inactive. The grant signal serially propagates through each master, until it encounters the first one that is requesting access to the bus. This module blocks the propagation of the Bus Grant signal, activates the Bus Busy line, and gains control of the bus. Any other requesting module present after the master will not receive the grant signal. Therefore, the priority is determined by the physical location of the modules. The requesting module located closest to the controller has the highest priority.

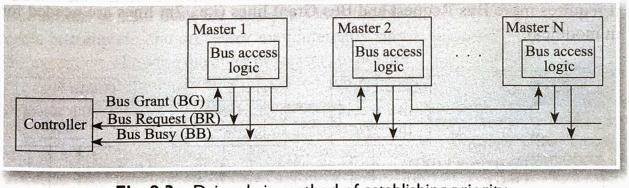


Fig. 8.3 Daisy chain method of establishing priority

Compared to the other two methods, the daisy chain scheme requires the least number of control lines and this number is independent of the number of modules in the system. However, the arbitration time is slow due to the propagation delay of the Bus Grant signal through the different masters. This delay is proportional to the number of modules and therefore, a daisy chain-based system is limited to multiprocessor systems having only a few modules. Further, the priority of each module is fixed by its physical location and the failure of even one module in the system causes the whole system to fail.

8.4.2 Polling

The polling scheme, which is shown in Fig. 8.4, uses a set of lines sufficient to address each module. In response to a bus request, the controller generates and sends out a sequence of module addresses to the requesting modules. When a

requesting module recognizes its address, it activates the Busy line and begins to use the bus. The major advantage of polling is that the priority can be dynamically changed by altering the polling sequence (i.e., the order in which the module addresses are sent) stored in the controller.

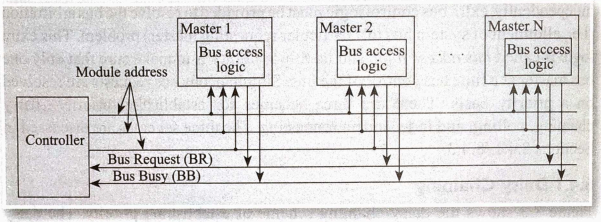


Fig. 8.4 Polling method of establishing priority

8.4.3 Independent Requesting

The independent requests scheme, which is shown in Fig. 8.5, resolves the priority in a parallel fashion. Each module has a separate pair of Bus Request (BR) and Bus Grant (BG) lines, and each pair has a priority assigned to it. The controller includes a priority decoder, which selects the request with the highest priority, and activates the corresponding Bus Grant signal. Arbitration is fast and is independent of the number of modules in the system. Compared to the other two methods, the independent requests design is the fastest method. However, it requires more Bus Request and Bus Grant lines (i.e., 2m lines are needed for m modules).

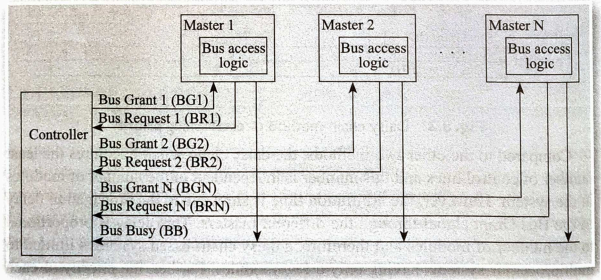


Fig. 8.5 Independent requests method of establishing priority

A module's host 8086 lacks the capability of requesting bus access and recognizing bus grants. Therefore, it is necessary for each module containing a bus master to have extra logic for sending and receiving the bus access signals. The Intel bus arbiter (8289) is specifically designed to provide the necessary bus access handshaking. The 8289 operates in conjunction with the bus controller (8288) and

controls the access of its associated master to the bus by using either the daisy chain or the independent requests scheme.

8.5 INTERCONNECTION TOPOLOGIES IN A MULTIPROCESSOR SYSTEM

A microprocessor with its external bus connections needs memory to form a minimum workable processing system. In a multiprocessor system, a number of microprocessors are connected with each other using a single bus. The bus is also used to address a multi-port memory or a shared single I/O port. In both the cases, the memory serves the following purposes:

(i) It stores the local (individual) instructions and data for all the processors.
(ii) It stores the common (global) instructions and/or data for all the processors.
(iii) It acts as a temporary storage for the instructions, data, and other parameters that are transferred between the processors.

Based on the method of communication among the microprocessors in a multiprocessor system, we have some interconnection topologies discussed in Sections 8.5.1–8.5.4.

8.5.1 Shared Bus Architecture

The shared bus architecture uses a common memory, which may be partitioned into local memory banks for different processors. This is shown in Fig. 8.6. At a time, only one processor performs a bus cycle to fetch instructions or data from the memory. Once the bus cycle is complete, the processor may internally start the execution, allowing the other processors to use the bus. Additional hardware is required for controlling the access of the bus by different processors. All the processors in Fig. 8.6 share a common memory, but they can also have a local memory individually to store the local instructions or data.

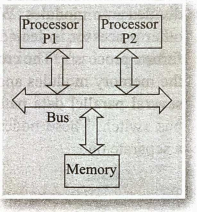


Fig. 8.6 Shared bus architecture

8.5.2 Multi-port Memory

In the multi-port memory configuration shown in Fig. 8.7, the processors P1 and P2 address a multi-port memory that can be accessed at a time by both the processors. They also have local memories, which are used by them to store individual instructions and data. Each processor uses its local memory for the execution of its individual tasks. The multi-port memory may be used for storing the instructions, data, and the results to be shared by more than one processor.

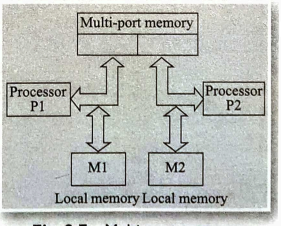


Fig. 8.7 Multi-port memory configuration

8.5.3 Linked Input/ Output

The linked input/output interconnection utilizes the I/O capabilities of a microprocessor-based system to communicate with other systems, as shown in Fig. 8.8. Parallel or serial I/O may be used to establish communication with other processors. The direct access of common instructions and data that are available in a local system memory is not possible in this method.

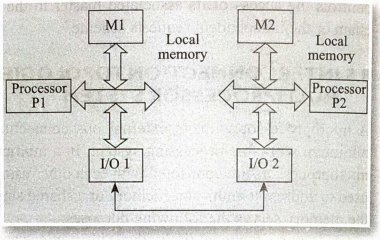


Fig. 8.8 Linked input/output interconnection

8.5.4 Crossbar Switching

The crossbar switching interconnection is shown in Fig. 8.9. It uses an extension of the concept of shared memory for a number of processors. In this method, more than one processor can have simultaneous access to the different memory modules to be shared individually, as long as there is no conflict. The memory is divided into modules. While one processor is accessing a memory module, the other processor is denied access to the same module till it is relinquished by the former processor. The crossbar switch provides the interconnection paths between the memory modules and the processors. In the crossbar switch interconnection, several parallel data paths are possible. Each node of the crossbar represents a bus switch. These nodes may be controlled by one of these processors or by a separate one.

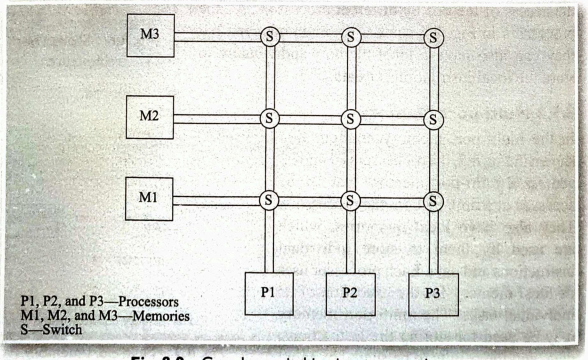


Fig. 8.9 Crossbar switching interconnection

8.6 PHYSICAL INTERCONNECTIONS BETWEEN PROCESSORS IN A MULTIPROCESSOR SYSTEM

The interconnections discussed in the previous sections are based on the communication methods between the microprocessors in a multiprocessor system. Besides those interconnections, we have star configuration, loop configuration, complete interconnection, regular topologies, and irregular topologies. These are based on the physical interconnections between the processors in a multiprocessor system.

8.6.1 Star Configuration

In this configuration, all the processors are connected to a central switching element via dedicated paths, as shown in Fig. 8.10. The central switching element may be an independent processor. The switching element controls the interconnections between the processing elements.

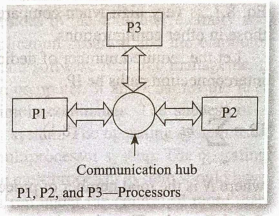


Fig. 8.10 Star configuration

8.6.2 Ring or Loop Configuration

The ring or loop configuration is shown in Fig. 8.11. The processors are arranged in a loop and each processor can communicate with the rest through intermediate processors in the path. The number of intermediate processors depends upon the position of the sender and the receiver in the loop. The direction of the data transfer along the loop may be unidirectional or bidirectional.

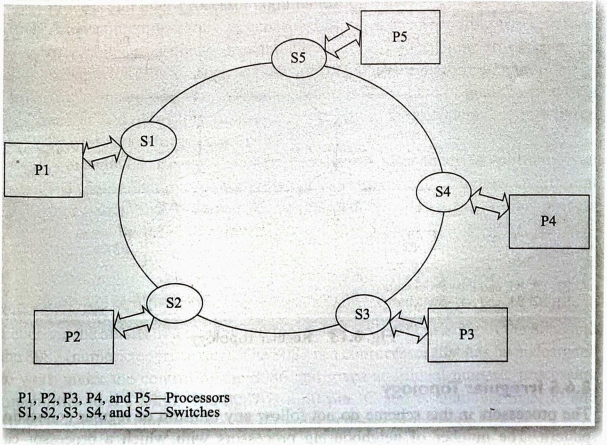


Fig. 8.11 Ring or loop configuration

8.6.3 Completely-connected Configuration

In the complete interconnection scheme, every processing element can directly communicate with another processor, one at a time, as shown in Fig. 8.12. The main drawback of this configuration is that the required number of dedicated interconnection paths, which is given by Eq. 8.1, is very high when compared to those in other configurations.

Let the required number of dedicated interconnection paths be IP.

$$\text{IP} = \sum_{m=1}^{N-1} m \qquad (8.1)$$

where N is the total number of processors.

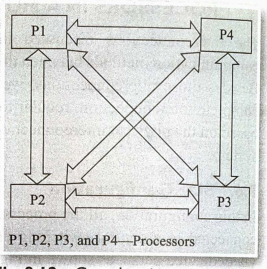


Fig. 8.12 Complete interconnection

8.6.4 Regular Topology

In this configuration, the processors are arranged in a regular fashion. The processors can be arranged in any of the regular structures such as linear array, square, hexagonal, or cubical configurations. Each processor (node) has a local memory to be accessed only by that processor. Each processor can communicate with a fixed number of neighbours in the specific regular structure. One of the regular topologies is shown in Fig. 8.13.

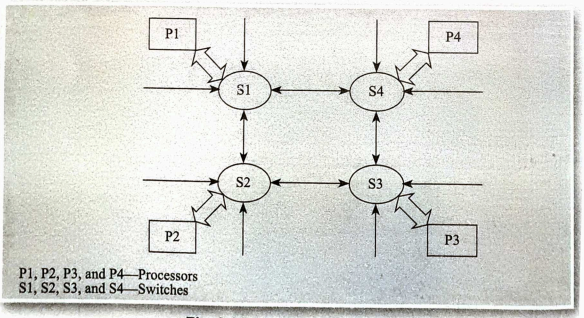


Fig. 8.13 Regular topology

8.6.5 Irregular Topology

The processors in this scheme do not follow any uniform or regular connection pattern. The number of neighbouring processors with which a processor can communicate is not fixed and may even be programmable.

8.7 OPERATING SYSTEM USED IN A MULTIPROCESSOR SYSTEM

All interconnection topologies are implemented using the microprocessor as a node. The microprocessors used as nodes may also work as standalone processors or sub-processing units under the control of other microprocessors. Once the microprocessors are arranged in a particular topology, an appropriate operating system and system software are required, that will be able to work in coordination with the new system resources.

An *operating system* is a program that resides in the computer memory and acts as an interface between the user or application program and the computer resources. It provides a means of hardware and software resource management, including memory and I/O management in a computer. It also enables the user to communicate with the hardware using simple commands. The success of a multiprocessor system relies on the operating system. The operating system used for a single processor cannot be used for a multiprocessor system. The operating system and the system software needed for the multiprocessor system should have the flexibility and the ability to work with or run under the control of more than one processor at a time. Distributed operating systems and the related system software are the solution to this.

Distributed operating systems are designed to run parallel processes. Hence it is essential that a proper environment exists for concurrent processes to communicate and cooperate, to complete the allotted task. The features expected from a distributed operating system used in a multiprocessor system are as follows:

(i) A distributed operating system should provide a mechanism for inter-process and inter-processor communication.

(ii) A distributed operating system must be capable of handling the structural or architectural changes in the system, which occur due to expected or unexpected reasons such as faults or modifications in the configuration.

(iii) A distributed operating system should also take care of unauthorized data access and data protection, as the data sets in these systems are referred to by more than one processor.

(iv) A distributed operating system must have a mechanism to split the given tasks into concurrent subtasks, which can be executed in parallel on different processors, and to collect the results of the subtasks and further process these to obtain the final result.

8.8 TYPICAL MULTIPROCESSOR SYSTEM HAVING 8086 AND 8087

Let us see the details of a typical multiprocessor system consisting of the 8086 and the 8087 (numeric coprocessor). The 8087 is a coprocessor that has been designed to work under the control of the 8086 and gives additional numeric processing capabilities to the 8086. The 8087 is a 40 pin IC and is available in 5, 8, and 10 MHz versions, compatible with different versions of the 8086.

When the 8086 is interfaced with the 8087, the instructions of the 8087 can be

included in the program to be executed by the 8086. The 8086 performs the opcode fetch cycles and identifies the instructions for the 8087. Once the instructions for the 8087 are identified by the 8086, they are assigned to the 8087 for further execution. After the 8087 executes that instruction, the results may be sent to the 8086 or stored in the memory. The 8087 adds 68 new instructions to the instruction set of the 8086.

8.8.1 Architecture of 8087

The simplified block diagram of the 8087 is shown in Fig. 8.14. The 8087 has two internal sections—the control unit (CU) and the numeric extension unit (NEU). The NEU executes all the numeric processor instructions, while the CU receives and decodes the instructions, and reads or writes memory operands. The control unit is also responsible for establishing communication between the CPU (8086) and the memory, and also for coordinating the internal coprocessor execution. The internal data bus in the 8087 is 84 bits wide, including the 68-bit fraction, 15-bit exponent, and sign bit. The microcode control unit in the 8087 generates the control signals required for the execution of the 8087 instructions. The 8087 contains a programmable shifter, which is responsible for shifting the operands during the execution of instructions such as FMUL and FDIV. The data bus interface in the 8087 connects its internal data bus with the system data bus of the 8086.

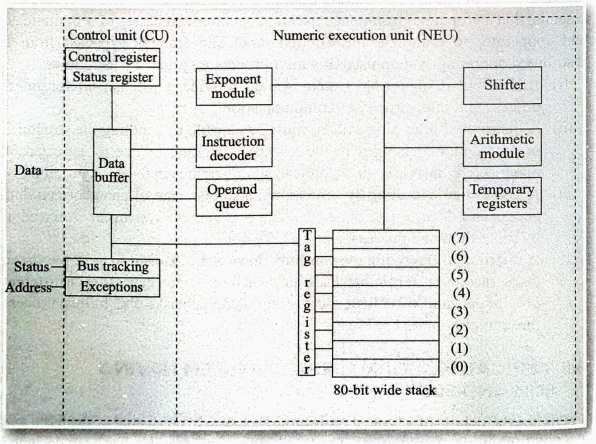


Fig. 8.14 Simplified block diagram of 8087

8.8.2 Pin Details of 8087

The different signals of the 8087 are discussed in detail in this section. Figure 8.15 shows the pin diagram of the 8087.

(i) AD0–AD15: These are the multiplexed address/data lines. These lines carry addresses during the T1 state and data during the T2, T3, TW, and T4 states, as in the 8086. These lines act as the input lines for the 8086-driven bus cycles and become the input/output lines for the NDP-initiated bus cycles.

(ii) A19/S6–A16/S3: These lines are time multiplexed address/status lines and are the same as in the 8086. S3, S4, and S6 are permanently high, while S5 is permanently low.

(iii) $\overline{\text{BHE}}$/S7: During T1, the $\overline{\text{BHE}}$/S7 pin is used to enable the data on the higher-order byte of the 8086 data bus. During the T2–T4 clock cycles, it acts as the status line S7.

(iv) QS1 and QS0: The queue status input signals QS1 and QS0 enable the 8087 to keep track of the instruction queue status of the 8086, to maintain synchronism with it. Their function is same as that of the QS1 and QS0 pins in the 8086. These lines are connected to the corresponding lines of the 8086.

(v) INT: The interrupt output is used by the 8087 to indicate that unmasked exceptions, such as invalid operation, divide-by-0, overflow, etc., have been received during the execution of the instruction.

(vi) BUSY: This output signal indicates to the 8086 that the 8087 is busy with the execution of an allotted instruction. This is usually connected to the $\overline{\text{TEST}}$ input of the 8086.

(vii) READY: This input signal may be used to inform the 8087 that the addressed device such as the memory or the I/O device will complete the data transfer from its side. Usually this signal is synchronized by the clock generator (8284).

(viii) RESET: This input signal is used to discard the internal activities of the coprocessor and prepare it for further execution, whenever needed by the 8086.

(ix) CLK: The CLK input provides the basic timings for the 8087. Its frequency is the same as that of the 8086.

(x) V_{CC}: A +5 V supply is connected to this pin.

(xi) GND: This is used as the return line for the V_{CC} supply.

(xii) $\overline{\text{S2}}$, $\overline{\text{S1}}$, and $\overline{\text{S0}}$: These pins can act either as output pins driven by the 8087 or as input pins driven by the 8086. If these are driven by

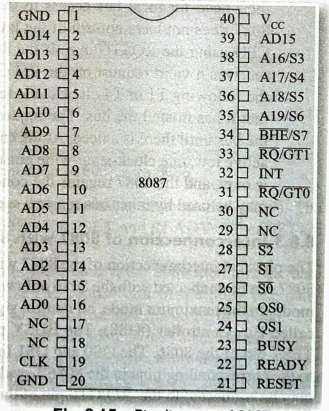


Fig. 8.15 Pin diagram of 8087

the 8087, they can be decoded as shown in Table 8.1. These are used by the bus controllers to derive the Read and Write signals. These signals act as input signals if the CPU is executing a task.

(xiii) $\overline{RQ}/\overline{GT0}$: The request/grant pin is a bidirectional pin used by the 8087 to gain control of the bus from the host 8086 for operand or data transfers. It must be connected to one of the request/grant pins of the 8086. The request/grant sequence is as follows:

Table 8.1 Functions of $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$ in 8087

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Queue status
0	X	X	Unused
1	0	0	Unused
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

An activate low pulse of one clock duration is generated by the 8087 for the host 8086 to inform it that it wants to gain control of the local bus, either for itself or for a coprocessor such as the 8089 (I/O processor) connected to the $\overline{RQ}/\overline{GT1}$ pin of the 8087. The 8087 waits for the grant pulse from the 8086, and when it is received, the 8087 either initiates a bus cycle if the request is for itself or passes the grant pulse to $\overline{RQ}/\overline{GT1}$ if the request is for the other coprocessor. The 8087 releases the bus by sending one more pulse on the $\overline{RQ}/\overline{GT0}$ line to the host 8086, either after completion of the last bus cycle initiated by it, or as a response to a release pulse on the $\overline{RQ}/\overline{GT1}$ line issued by a coprocessor.

(xiv) $\overline{RQ}/\overline{GT1}$: This bidirectional pin is used by the other bus masters to convey their need for local bus access to the 8087. At the time of request, if the 8087 does not have control of the bus, the request is passed on to the host CPU using the $\overline{RQ}/\overline{GT0}$ pin. If the 8087 has control over the bus, when it receives a valid request on the $\overline{RQ}/\overline{GT1}$ pin, it sends a grant pulse during the following T1 or T4 clock cycle to the requesting bus master, indicating that it has floated the bus. The requesting bus master then gains control of the bus until there is a need. At the end, the requesting bus master issues an active low, one clock-state-wide pulse for the 8087, to indicate that the task is over, and the 8087 regains the control of the bus. The request/grant pins may be used by other bus masters such as DMA controllers.

8.8.3 Interconnection of 8087 with 8086

The physical interconnection of the 8087 with the 8086 is shown in Fig. 8.16. The 8087 can be connected with the 8086 only when the 8086 is operating in maximum mode. In the maximum mode, all control signals are derived using a separate chip called Bus Controller (8288). The BUSY pin of the 8087 is connected with the $\overline{TEST}$ pin of the 8086. The QS0 and QS1 lines in the 8087 are directly connected to the corresponding pins in the 8086-based system. The $\overline{RQ}/\overline{GT0}$ pin of the 8087 may be connected to the $\overline{RQ}/\overline{GT0}$ pin of the 8086. The clock pin of the 8087 may be connected to the 8086 clock input. The interrupt output of the 8087 is sent

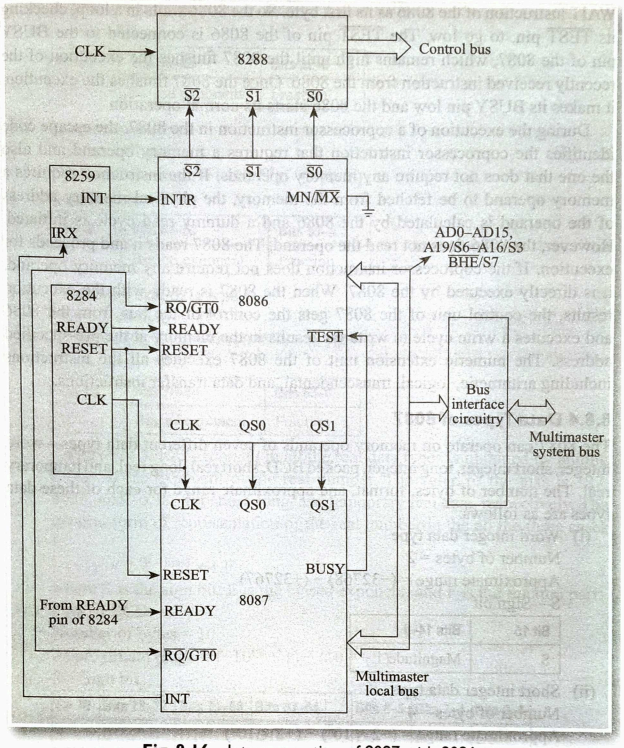


Fig. 8.16 Interconnection of 8087 with 8086

to the 8086 through a chip called programmable interrupt controller (8259). The pins AD15–AD0, A19/S6–A16/S3, $\overline{\text{BHE}}$/S7, RESET, and READY of the 8087 are connected to the corresponding pins of the 8086.

While fetching the instructions from the memory, the 8086 monitors the data bus to check for the 8087 instructions. The control unit of the 8087 internally maintains a parallel queue, identical to the instruction queue of the 8086. The 8087 uses the QS0 and QS1 pins to obtain and identify the instructions fetched by the 8086. The 8086 identifies the coprocessor instructions using the escape code bits embedded in them. The first five bits of the escape code are 11011. Once the 8086 recognizes the escape code, it initiates the execution of the coprocessor instructions in the 8087. Each coprocessor instruction also has the opcode of the

WAIT instruction of the 8086 as its first byte. So the 8086 waits in a loop, checking its $\overline{\text{TEST}}$ pin, to go low. The $\overline{\text{TEST}}$ pin of the 8086 is connected to the BUSY pin of the 8087, which remains high until the 8087 finishes the execution of the recently received instruction from the 8086. Once the 8087 finishes the execution, it makes its BUSY pin low and the 8086 starts its normal operation.

During the execution of a coprocessor instruction in the 8087, the escape code identifies the coprocessor instruction that requires a memory operand and also the one that does not require any memory operands. If the instruction requires a memory operand to be fetched from the memory, the physical memory address of the operand is calculated by the 8086 and a dummy read cycle is initiated. However, the 8086 does not read the operand. The 8087 reads it and proceeds for execution. If the coprocessor instruction does not require any memory operand, it is directly executed by the 8087. When the 8087 is ready with the execution results, the control unit of the 8087 gets the control of the bus from the 8086 and executes a write cycle to write the results in the memory at the pre-specified address. The numeric extension unit of the 8087 executes all the instructions including arithmetic, logical, transcendental, and data transfer instructions.

8.8.4 Data Types of 8087

The 8087 can operate on memory operands of seven different data types—word integer, short integer, long integer, packed BCD, short real, long real, and temporary real. The number of bytes, format, and approximate range for each of these data types are as follows:

(i) Word integer data type
Number of bytes = 2
Approximate range = (−32768) − (+32767)
S—Sign bit

Bit 15	Bits 14–0
S	Magnitude

(ii) Short integer data type
Number of bytes = 4
Approximate range = $(-2 \times 10^9) - (+2 \times 10^9)$
S—Sign bit

Bit 31	Bits 30–0
S	Magnitude

(iii) Long integer data type
Number of bytes = 8
Approximate range = $(-9 \times 10^{18}) - (+9 \times 10^{18})$
S—Sign bit

Bit 63	Bits 62–0
S	Magnitude

(iv) Short real data type
Number of bytes = 4

Approximate range = $(\pm 1 \times 10^{-38}) - (\pm 3 \times 10^{38})$

S—Sign bit

Bit 31	Bits 30–23	Bits 22–0
S	Biased exponent	Fraction

(v) Long real data type

Number of bytes = 8

Approximate range = $(\pm 10^{-308}) - (\pm 10^{308})$

S—Sign bit

Bit 63	Bits 62–52	Bits 51–0
S	Biased exponent	Fraction

(vi) Temporary real data type

Number of bytes = 10

Approximate range = $(\pm 10^{-4932}) - (\pm 10^{4932})$

S—Sign bit

Bit 79	Bits 78–64	Bits 63–0
S	Biased exponent	Fraction

The term *biased exponent* in all the real data types is obtained by adding a bias to the exponent of the real number. The value of the bias is 127, 1023, and 16,383 for short, long, and temporary real data, respectively. The general form of representation of the real number in the all the three cases is

$$(-1)^S \times 2^{(E - \text{bias})} \times 1.F$$

where S is the sign bit, E is the biased exponent, and F is the fraction part.

(vii) Packed BCD

Number of bytes = 10

Approximate range = $(-10^{18} + 1) - (10^{18} - 1)$

S—Sign bit

Bit 79	Bits 78–72	Bits 71–68	Bits 67–64	...	Bits 8–5	Bits 7–4	Bits 3–0
S	0	D17	D16	...	D2	D1	D0

D0, D1, D2... D16, and D17 represent the BCD code of each digit in the packed BCD number. Further details of the 8087 can be obtained by referring to the data sheet of the 8087.

8.9 TYPICAL MULTIPROCESSOR SYSTEM HAVING 8086 AND 8089

While accessing I/O devices by non-DMA data transfer using the serial and parallel ports in the personal computer, the CPU (such as the 8086) is required to set up the interfacing chips used to access the I/O devices and perform the actual data transfer. For high speed devices, data are transferred using DMA, but the CPU has to set up the device controller, initiate the DMA operation, and check

the post-transfer status after the completion of each DMA operation. The 8089 I/O processor is designed to handle the tasks involved in I/O processing. An IOP can fetch and execute its own instructions, unlike a DMA controller.

The instruction set of the 8089 is specifically designed for I/O operations, but in addition to data transfer, it can perform arithmetic and logic operations, branching, searching, and translation. The CPU communicates with the 8089 through memory-based control blocks. The CPU prepares control blocks that describe the task to be performed, and then sends the task to the 8089 through an interrupt-like signal. The 8089 reads the control blocks to locate a program called a *channel program*, which is written using the 8089 instruction set. Then the 8089 performs the assigned task by fetching and executing instructions from the channel program. When the 8089 has finished the task, it informs the CPU either through an interrupt or by updating a status location in the memory.

8.9.1 Pin Details of 8089

The pin diagram of the 8089 is shown in Fig. 8.17.

The details of various pins are as follows:

(i) A0–A15/D0–D15 (multiplexed address/data bus): The function of these lines is defined by the state of the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ lines. The pins are floated after reset and when the bus is not acquired. The signals in A8–A15 remain the same on transfer to a physical 8-bit data bus (used with the 8088 processor as it has an 8-bit data bus and 20-bit address bus) and are multiplexed with data D8–D15 on transfers to a 16-bit physical bus (used with the 8086 processor).

(ii) A19–A16/S6–S3 (address and status bus): The address lines are active only when addressing memory. Otherwise, the status lines are active and are encoded as shown in Table 8.2. The pins are floated after reset and when the bus is not acquired.

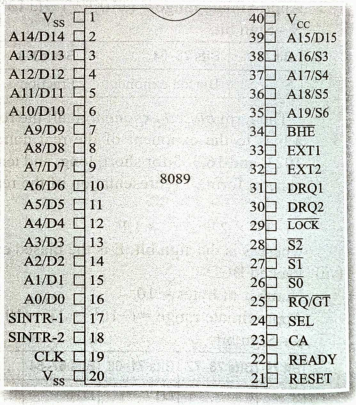


Fig. 8.17 Pin diagram of the 8089

Table 8.2 Status bits and their significance

S6	S5	S4	S3	Significance
1	1	0	0	DMA cycle on CH1
1	1	0	1	DMA cycle on CH2
1	1	1	0	Non-DMA cycle on CH1
1	1	1	1	Non-DMA cycle on CH2

(iii) $\overline{BHE}$ (Bus High Enable): The Bus High Enable signal is used to enable data operations on the most significant half of the data bus (D8–D15). The signal is active low when a byte is to be transferred on the upper half of the data bus. The pin is floated after reset and when the bus is not acquired. $\overline{BHE}$ does not have to be latched.

Table 8.3 Status bits for defining IOP activity

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Significance
0	0	0	Instruction fetch (I/O space)
0	0	1	Data fetch (I/O space)
0	1	0	Data store (I/O space)
0	1	1	Not used
1	0	0	Instruction fetch (System memory)
1	0	1	Data fetch (System memory)
1	1	0	Data store (System memory)
1	1	1	Passive

(iv) $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$ (status pins): These are status pins, which define the IOP activity during any given cycle. They are encoded as shown in Table 8.3.

The status lines are utilized by the bus controller and the bus arbiter to generate all the memory and I/O control signals. The signals change during T4 if a new cycle is to be entered, while the return to passive state in T3 or TW indicates the end of the cycle. The pins are floated after system reset and when the bus is not acquired.

(v) READY: The Ready signal received from the addressed device indicates that the device is ready for data transfer. The signal is active high and is synchronized by the 8284 clock generator.

(vi) $\overline{LOCK}$: The Lock output signal indicates to the bus controller that the bus is needed for more than one contiguous cycle. It is set via the channel control register and during the TSL instruction. The pin floats after reset and when the bus is not acquired. The output is active low.

(vii) RESET: The receipt of a Reset signal causes the IOP to suspend all its activities and enter idle state until a Channel Attention signal is received. The signal must be active for at least four clock cycles.

(viii) CLK (clock): The clock provides all the timing needed for internal IOP operation.

(ix) CA (Channel Attention): This signal gets the attention of the IOP. When the falling edge of this signal is encountered, the SEL input pin is examined to determine master/slave or CH1/CH2 information. This input is active high.

(x) SEL (Select): The first CA received after system reset informs the IOP via the SEL line, whether it is a master or a slave (0 and 1, respectively), and starts the initialization sequence. During any other CA, the SEL line signifies the selection of CH1 and CH2 (0 and 1, respectively).

(xi) DRQ1 and DRQ2 (Data Request): The DMA requests inputs, which signal to the IOP that a peripheral is ready to transfer and receive data using CH1 and CH2, respectively. The signals must be held active high until the appropriate fetch/stroke is initiated.

(xii) $\overline{RQ}/\overline{GT}$ (Request Grant): The Request Grant signal implements the communication dialogue required to arbitrate the use of the system bus (between IOP and CPU in local mode) or I/O bus when two IOPs share the same bus (remote mode). The $\overline{RQ}/\overline{GT}$ signal is active low. An internal pull-up permits $\overline{RQ}/\overline{GT}$ to be left floating, if not used.

(xiii) SINTR-1 and SINTR-2 (Signal Interrupt): The Signal Interrupt signal outputs from CH1 and CH2, respectively. The interrupts may be sent directly to the CPU or through the 8295A interrupt controller. They are used to indicate to the system the occurrence of user-defined events.

(xiv) EXT1 and EXT2 (External Terminate): The External Terminate signal inputs from CH1 and CH2, respectively. The EXT signal causes the termination of the current DMA transfer operation, if the channel is so programmed by the channel control register. The signal must be held active high until the termination is complete.

V_{cc}: Supply voltage (+5 V) V_{ss}: Ground

8.9.2 Local and Remote Operation of 8089

The 8089 assumes all the work involved in an I/O transfer, including device setup, DMA operation, and programmed I/O, thereby relieving the CPU from the burden of I/O processing. This allows the CPU to concentrate on higher-level tasks, while the 8089 takes care of I/O processing. This greatly simplifies system software and hardware efforts, and improves system performance and flexibility, by the distributed processing approach. The 8089 may be operated in a local (closely-coupled)

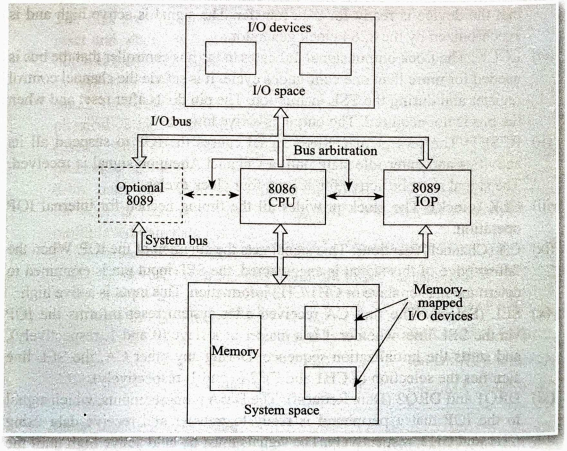


Fig. 8.18 8089 IOP in local configuration

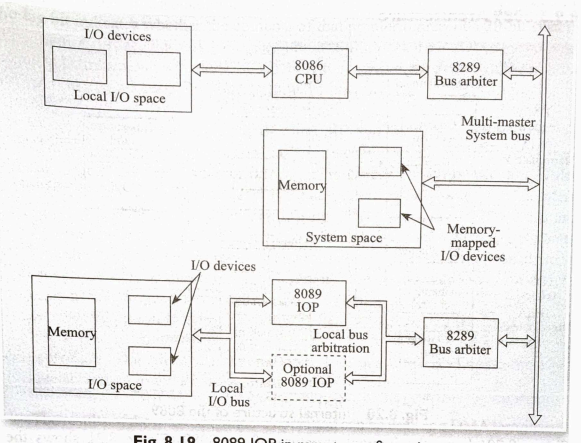


Fig. 8.19 8089 IOP in remote configuration

configuration or a remote (loosely-coupled) configuration. In a local configuration, which is shown in Fig. 8.18, the 8089 shares the bus interface with the host (8086) by using its $\overline{RQ}/\overline{GT}$ pins. All resources are accessed through the system bus.

In a remote configuration, which is shown in Fig. 8.19, the 8089 may have its own local I/O bus and requires a bus arbiter and controller, address latches, and data transceivers for accessing the shared system bus. The $\overline{RQ}/\overline{GT}$ pin on the 8089 can be used to interact with another 8089, which acts as the slave and shares the buses with the host 8089. The 8089 accesses I/O devices dedicated to it through the local bus, while it communicates with the CPU through the system memory. A high speed controller may request a transfer through one of the two DRQ pins (DRQ1 and DRQ2) in the 8089, and terminate a DMA operation through one of two EXT pins (EXT1 and EXT2) in the 8089. To reduce the system bus loading and enhance concurrent processing, local memory can be included to store the channel programs or to provide storage areas. However, the local memory must respond to I/O bus commands instead of memory read and memory write commands (i.e., it must act as the I/O-mapped memory). Unlike the 8086, the 8089 I/O bus need not have the same data width as the memory bus. This allows the 8089 to transfer data from an 8-bit source to a 16-bit destination and vice versa. Since the I/O bus has only 16 address lines, the capacity of the local space (I/O space) is only 64 KB. On the other hand, the system space (i.e., memory space), which is addressed by the system bus, has a capacity of 1 MB. The 8089 instructions access I/O ports using the same addressing modes as are used for the memory operands. Whether an address is in the I/O space or in the system space is determined by the tag bit of the pointer register used.

8.9.3 8089 Architecture

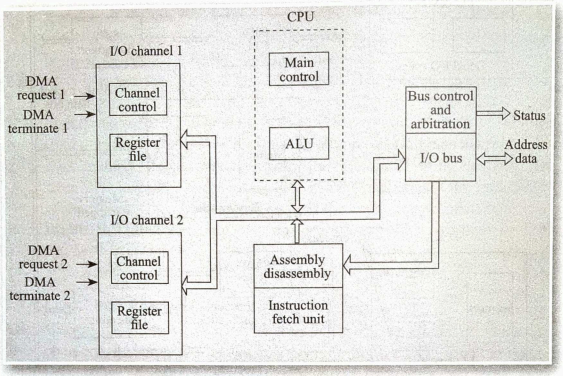


Fig. 8.20 Internal structure of the 8089

Figure 8.20 shows the internal structure of the 8089. Figure 8.21 shows the registers in the 8089 IOP. Each of the two channels can be programmed and operated independently while sharing the common control logic and ALU. The channel control pointer (CCP) cannot be manipulated by the user. It stores the address of the control block (CB) for channel 1 during the initialization sequence. For channel 2, its CB starts at the address that is indicated by adding 8 to the contents of the CCP. To dispatch a task to either channel, the CPU (8086) sends out a Channel Attention (CA) signal along with the Select (SEL) signal, which selects channel 1 (if SEL = 0) or channel 2 (if SEL = 1). Since the channels occupy two consecutive I/O port addresses, the A0 address line of the 8086 is connected to the SEL pin, so that when A0 = 0, one channel is selected and when A0 = 1, another channel is selected.

Each channel has an identical set of registers, each set being divided into two groups according to size. The pointer group consists of those registers having 20 bits and the register group consists of those registers having 16 bits. Each pointer, with the exception of parameter

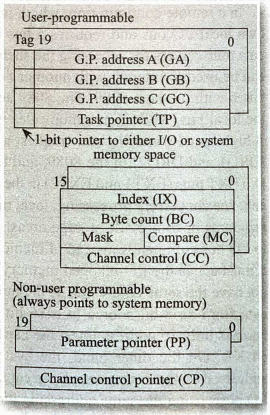


Fig. 8.21 Registers in 8089 IOP

pointer (PP), has an associated tag bit. When used to access a memory operand, the tag bit indicates whether the contents of that pointer represent a 20-bit system (i.e., memory) space address (if tag = 0) or a 16-bit local (i.e., I/O) space address (if tag = 1). In accessing the local space, only the lower-order 16 bits of the pointer are used as the address. Register PP always points to an address in the system space.

The registers GA, GB, GC, IX, BC, and MC can be used as general-purpose registers for arithmetic and logical operations in a channel program. In addition, they perform special functions when addressing memory operands and executing DMA operations. A memory operand can only be addressed by using one of the pointers GA, GB, GC, or PP as a base register. During a DMA operation, GA and GB are used as the source and destination pointers. If GA points to the source, GB points to the destination, and vice versa. When a translation operation is performed along with the DMA transfer, the contents of GC are used as the base address of a 256-byte translation table. Register BC is used as the byte counter during a DMA transfer, and is decremented by 1 after every byte transfer and by 2 after every word transfer.

For a masked compare operation, register MC contains the bit pattern to be compared against in bits 7–0 and a mask in bits 15–8. A masked compare operation is done according to the following expression:

$$((\text{OPERAND BYTE}) \oplus (\text{MC})_{7\text{–}0}) \wedge (\text{MC})_{15\text{–}8}$$

The results of the masked compare operation can be used as a DMA termination condition or to determine whether or not a branch is to be made by a masked compare branch instruction. The register IX is used as an index register. In two of the memory operand addressing modes, its contents are added to those of a base register to form the operand address. The task pointer (TP) stores the address of the next instruction to be executed and is equivalent to the PC in a CPU. It also has a tag bit for indicating whether the next instruction to be executed is stored in the system or I/O space. The parameter pointer (PP) is not programmable by the user, but is automatically filled by the 8089 while initializing a task. PP points to the address of the parameter block.

Each channel also has an 8-bit status register (PSW), which contains the current channel status. This status indicates status descriptors such as the source and destination address widths, channel activity, interrupt control and servicing, bus load limit, and priority. The PSW cannot be manipulated by the user, but can be modified by a channel command. It is saved with TP and the four tag bits in the first two words of the parameter block, when a channel program is suspended. This allows the channel to resume the suspended channel program upon receipt of a resume command.

The 8089 has the capability to perform DMA transfers using different options. The transfer direction can be specified as I/O to I/O, memory to memory, or memory to/from I/O. For each transfer, the 8089 fetches a byte or word, stores the data in the destination and updates GA, GB, and BC accordingly. If data are transferred from an 8-bit source to a 16-bit destination, the 8089 can fetch two bytes and store them as one word. Conversely, if the transfer is from a 16-bit source to an 8-bit destination, a word can be split into two bytes, before the data

are sent to the destination. Between the fetch and the store cycles of the DMA, the data byte can be compared or translated. Further, a DMA operation can be terminated by an external request, a zero byte count, or a match/mismatch detected by a masked compare. These options are specified by the contents of the channel control register (CC) whose format is as follows:

Termination control bits 6–0 These bits specify how the DMA is to be terminated and where to fetch the next instruction from, upon completion of the DMA operation. A DMA transfer can be terminated after the current transfer cycle, based on the comparison (bits 2,1, and 0) shown in Table 8.4 (a), the byte count (bits 4 and 3) shown in Table 8.4 (b), and the external control (bits 5 and 6) shown in Table 8.4 (c), or a combination of the three. If external control is selected, the channel terminates the DMA when the channel's EXT (external termination) input is activated. If the byte count is specified, a 0 in the channel's BC register causes the DMA to terminate. Whether or not a comparison is to result in a termination and whether a match or mismatch is to cause the termination is determined by bits 2 to 0.

Table 8.4 (a) Function of termination control bits 2–0

Bit 2	Bit 1	Bit 0	Termination condition and offset
0	0	0	No termination by masked comparison
0	0	1	Terminates when comparison matches; offset is set to 0
0	1	0	Terminates when comparison matches; offset is set to 4
0	1	1	Terminates when comparison matches; offset is set to 8
1	0	0	No effect
1	0	1	Terminates when there is no match; offset is set to 0
1	1	0	Terminates when there is no match; offset is set to 4
1	1	1	Terminates when there is no match; offset is set to 8

Table 8.4 (b) Function of termination control bits 4 and 3

Bit 4	Bit 3	Termination condition and offset
0	0	No termination by byte counter
0	1	Terminates when BC = 0; offset is set to 0
1	0	Terminates when BC = 0; offset is set to 4
1	1	Terminates when BC = 0; offset is set to 8

Table 8.4 (c) Function of termination control bits 6 and 5

Bit 6	Bit 5	Termination condition and offset
0	0	No external termination
0	1	Terminates when EXT = 1; offset is set to 0
1	0	Terminates when EXT = 1; offset is set to 4
1	1	Terminates when EXT = 1; offset is set to 8

The channel executes the instruction whose address is the contents of TP, plus an offset upon the termination of a DMA operation. Therefore, when more than one termination condition is specified, it is possible to use the offset in conjunction with the branch instruction to enter different DMA completion routines, depending on the actual cause of the DMA termination. If more than one of the selected conditions occurs at the same time, the largest offset that corresponds to a satisfied condition is used. To initiate a DMA transfer, the channel program should contain instructions for setting up the source and destination pointers CC, BC and, if necessary, GC, MC, and the I/O bus width.

Single transfer mode (bit 7) This bit is used to terminate the DMA after a single transfer if it is set to 1 and then execute the next instruction pointed to by TP.

Chaining control (bit 8) This bit gives the other channel (i.e., channel 2, when channel 1 is programmed and vice versa) the highest priority. This bit is not used for DMA operation.

Lock control (bit 9) This bit activates the $\overline{\text{LOCK}}$ output of the 8089 during the DMA transfer cycle, if it is set to 1.

Source/destination indicator (bit 10) This bit specifies whether the register GA is used as the source pointer (i.e., the bit is 0) or the destination pointer (i.e., the bit is 1). In either case, GB is used as the other pointer.

Synchronization control (bits 12 and 11) These bits specify how the data transfer is to be synchronized. An unsynchronized transfer (if the bits are 00) begins the next transfer cycle whenever a bus cycle is available. A source-synchronized transfer (if the bits are 01) starts the read operation of the next transfer cycle upon receiving the DRQ signal. A destination-synchronized transfer (if the bits are 10) starts the write operation of the next transfer cycle when the DRQ is received.

Translation mode (bit 13) This bit indicates that the data bytes are to be translated through a 256-byte look-up table during DMA (if the bit is 1). The base address of the translation table should be stored in register GC.

Function control (bits 15 and 14) These bits specify one of four data transfer modes—memory to memory (if the bits are 11), I/O port to memory (if the bits are 10), memory to I/O port (if the bits are 01), and I/O port to I/O port (if the bits are 00). During a memory to memory data transfer, both the destination and source pointers are auto-incremented, but during and after an I/O to I/O data transfer, both pointers remain unchanged. These two modes are not supported by most conventional DMA controllers. They are useful in moving a block of code or data from one memory area to another and in direct device communications.

8.9.4 Communication between CPU (8086) and IOP (8089)

Inter-processor communication, including the 8089 IOP initialization and task dispatch, is memory-based and is accomplished by means of a linked list of control blocks. The first control block in the linked list is stored beginning at the fixed location FFFF6H in the memory. The others may reside in user-defined areas,

each of which is pointed to by the previous control block. The IOP communication areas are shown in Fig. 8.22. The system configuration pointer block (SCPB) contains three words starting at location FFFF6H in the system memory. The least significant byte (SYSBUS) specifies the width of the system bus, which is eight bits if SYSBUS = 0 and 16 bits if SYSBUS = 1. The succeeding two words store the offset and segment address of the location of the system configuration block (SCB). The SCB does not need to be stored at a fixed location. However, it must reside in the system space. The least significant byte of the SCB is the system operation command (SOC). Bits 0 and 1 of the system operation command define the width of the I/O bus and the $\overline{RQ}/\overline{GT}$ mode as follows:

Bit 1 = 0 indicates the standard $\overline{RQ}/\overline{GT}$ mode.

Bit 1 = 1 indicates the modified $\overline{RQ}/\overline{GT}$ mode for use with multiple 8089s.

Bit 0 = 0 indicates an 8-bit I/O bus.

Bit 0 = 1 indicates a 16-bit I/O bus.

The last two words in the SCB contain the offset and segment address of the beginning of two consecutive channel control blocks (CBs) in the system space. There is one control block for each channel and the first byte of each control block is called the *channel command word* (CCW), which indicates the action to be taken by the channel. The next byte (BUSY) indicates the busy status of the channel (00 for not busy and FF for busy) and the last two words contain the address of a parameter block. A parameter block is used for providing the beginning address of the channel program and passing information to and from this program.

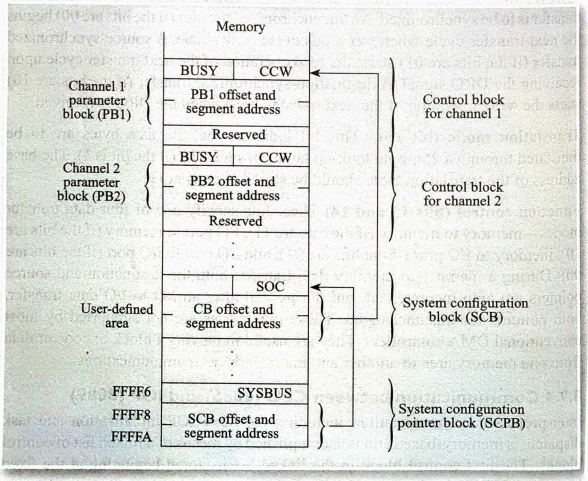


Fig. 8.22 IOP communication structure

The format of a CCW is shown in Fig. 8.23. It includes a 3-bit command field, a 2-bit interrupt control field, a bus load limit bit, and a priority bit.

Bit 7	Bit 6	Bit 5	Bits 5 and 4	Bits 2, 1, and 0
P	0	B	ICF	CF

Fig. 8.23 Channel command word—CCW

The command field specifies one of the six possible commands shown in Table 8.5.

Table 8.5 Function of CF field in channel command word (CCW)

CF	Command field
000	Update PSW—causes PSW to be updated
001	Start channel program (I/O space)—initiates the execution of a channel program that is stored in the I/O space
010	Reserved
011	Start channel program (system space)—initiates the execution of a channel program that is stored in the system space
100	Reserved
101	Resume suspended channel operation—causes a suspended operation to be continued from the point at which it was stopped
110	Suspend channel operation—suspends the operation currently being performed by the channel until a resume command is given
111	Halt channel operation—aborts the current channel operation

The interrupt control field (ICF) is used for enabling (ICF = 10) and disabling (ICF = 11) interrupt requests and for removing previous interrupt requests (ICF = 01). When an IOP sends out an interrupt request, it sets a service bit in its PSW. While the interrupt is being serviced, this bit must be cleared by sending the IOP a command with 01 in the interrupt control field. Otherwise, a request would block other requests. If ICF = 00, it has no effect on interrupts.

When the bus load limit (B) bit is 0, there is no bus load limit and when it is 1, there exists a bus load limit, during which the IOP can execute only one instruction every 128 clock cycles. This prevents the IOP from monopolizing the bus in situations in which there is no need for it to be the dominant processor. The priority bit in the PSW is set or cleared according to the priority bit (P) in the CCW. Further details about the 8089 can be obtained from the data sheet of the 8089.

POINTS TO REMEMBER

- In a multiprocessor system, more than one processor works cooperatively to solve a common task.

- There are different configurations of multiprocessor systems, such as coprocessor, closely-coupled and loosely-coupled multiprocessor system.
- There exist different interconnection topologies between processors and memories in a multiprocessor system.
- The physical interconnections between processors in a multiprocessor system can be different.
- A distributed operating system is commonly used in a multiprocessor system.
- The numeric coprocessor (8087) is used to perform floating-point operations efficiently.
- The I/O processor (8089) is used to perform various I/O operations, relieving the CPU to perform higher-level functions.

KEY TERMS

Bus arbitration This is the method by which allotment of a system bus to a particular requesting master is done amongst the many requesting masters at a time. There are three methods of bus arbitration—daisy chain, polling, and independent requests.

Closely-coupled system A closely-coupled multiprocessor system is one in which the external processor or coprocessor shares not only the entire memory and I/O subsystem in the system, but also the same bus control logic and clock generator of the main processor.

Distributed operating system This is an operating system used in a multiprocessor system to efficiently run a task in the system and to protect the program and data of different modules in the system from unauthorized accesses.

Interconnection topology This is the way in which communication among the microprocessors is performed in a multiprocessor system. There are different methods of interconnection between processors—shared bus architecture, multi-port memory, linked input/output, and crossbar switch.

I/O processor (IOP) An I/O processor is a processor that is mainly used to perform I/O-related operations, to relieve the microprocessor from the relatively slow I/O operations. IOP is also a coprocessor.

Loosely-coupled system Each module in a loosely-coupled system may act as the system bus master and may consist of an 8086 or another processor, capable of being a bus master, a coprocessor, or a closely-coupled configuration. Several modules may share the system resources and the system bus control logic must resolve the bus contention problem.

Multiprocessor system A multiprocessor system is one that contains more than one processor for improving the performance of the system.

Numeric coprocessor or numeric data processor A numeric coprocessor is a processor that works in conjunction with a microprocessor to perform floating-point operations quickly.

REVIEW QUESTIONS

1. What is meant by a multiprocessor system?
2. What are the advantages of a multiprocessor system?
3. What are the different schemes of bus arbitration?

4. What is the advantage and disadvantage of the daisy chain method of establishing priority among modules in a multiprocessor system?
5. What is meant by the polling method of establishing priority among the modules in a multiprocessor system?
6. How is interconnection between processors done in multi-port memory?
7. Draw the diagram showing the linked input/output scheme of interconnection between processors.
8. How is the interconnection between processors done in the crossbar switch scheme?
9. How are the physical interconnections between processors classified?
10. What is the drawback of the completely connected configuration method of physical interconnection among processors?
11. What is meant by the regular topology scheme of physical interconnection among processors?
12. What is an operating system?
13. What are the features of a distributed operating system?
14. What are the two main parts in the 8087 and their function?
15. What is the function of the BUSY and READY pins in the 8087?
16. What is the function of the QS0 and QS1 pins in the 8087?
17. What is the function of the 8089?
18. Explain closely-coupled and loosely-coupled multiprocessor systems, with necessary diagrams.
19. Describe the different bus arbitration schemes used in a loosely-coupled multiprocessor system, with necessary diagrams.
20. With neat diagrams, explain the different interconnection topologies used for communication among the processors in a multiprocessor system.
21. Describe the different physical interconnections between processors in a multiprocessor system, with neat diagrams.
22. Draw and explain the block diagram of the 8087 and the main signals in the 8087.
23. Draw the diagram showing the interconnection of the 8086 with the 8087 and explain the communication between these two processors in detail.
24. Draw and explain the block diagram of the 8089 and the main signals in the 8089.
25. Draw the diagram showing the interconnection of the 8086 with the 8089 and explain the communication between these two processors in detail.

THINK AND ANSWER

1. How is the priority among modules in a multiprocessor system resolved in independent requests scheme?
2. How does the 8086 identify an 8087 instruction when it fetches an instruction from the memory?
3. How is priority among several masters established in daisy chain method?
4. Why is biased exponent used to represent floating point numbers?
5. Why is the capacity of the local I/O space 64 KB in 8089?

8086-based Systems

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Minimum and maximum mode operation of an 8086-based system
- Function of the clock generator (8284A) and the bus controller (8288)
- Bus timings, interrupt acknowledgement, and bus request and grant in the 8086

9.1 INTRODUCTION

To adapt to different situations, the 8086 processors can be operated either in the minimum or the maximum mode. The minimum mode is used for a small system with a single processor (8086) and in any system in which the 8086 generates all the necessary bus control signals directly, thereby minimizing the required bus control logic. The maximum mode is for medium to large size systems, which often include two or more processors. In the maximum mode, the 8086 encodes the basic bus control signals into three status bits ($\overline{S2}$, $\overline{S1}$, and $\overline{S0}$) and uses the remaining control pins to provide the additional information that is needed to support the multiprocessor configuration.

9.2 8086 IN MINIMUM MODE CONFIGURATION

The 8086 is configured in minimum mode when its MN/$\overline{MX}$ pin is connected to +5 V. A typical minimum mode configuration of the 8086 is shown in Fig. 9.1. The figure illustrates the 8284 IC generating the clock, Ready and Reset signals for the 8086. The decoder is used to generate the four control signals $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, and $\overline{IOW}$ using the M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals of the 8086. The chip select (CS) logic is used to generate the Chip Select signals for the odd and even memory banks of the RAM and ROM chips and the I/O devices using $\overline{BHE}$, A0, and a few higher-order address lines of the 8086. The $\overline{CSE}$ and the $\overline{CSO}$ signals represent the Chip Select signal for the even bank and odd bank of the memory, respectively. $\overline{CSIO}$ represents the Chip Select signal for the input/output (I/O) devices.

The address from the 8086 and the $\overline{BHE}$ signals are latched externally using three 74LS373 (octal latch) ICs, since they are available only during the first part of the bus cycle. The ALE signal of the 8086 is used to indicate that the bus contains a valid address and is connected to the clock (G or CLK) input of the 74LS373 as shown in Fig. 9.2.

9.2.1 Formation of Separate Address Bus and Data Bus in 8086

If the 8086-based system includes several interfaces, transceivers (driver and receiver) are required for the data lines. This may not be a requirement for small,

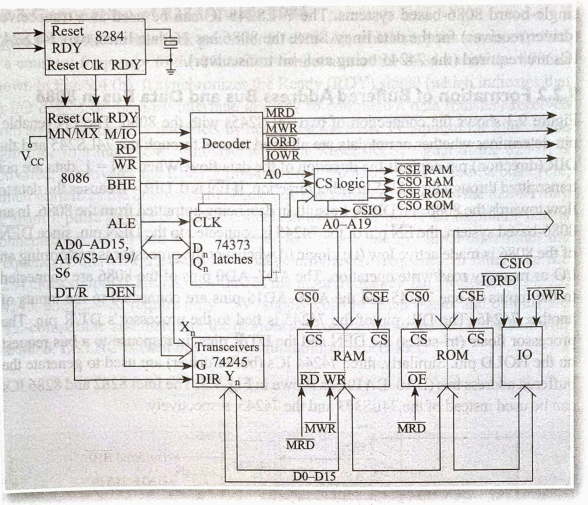


Fig. 9.1 Minimum mode operation of an 8086-based system

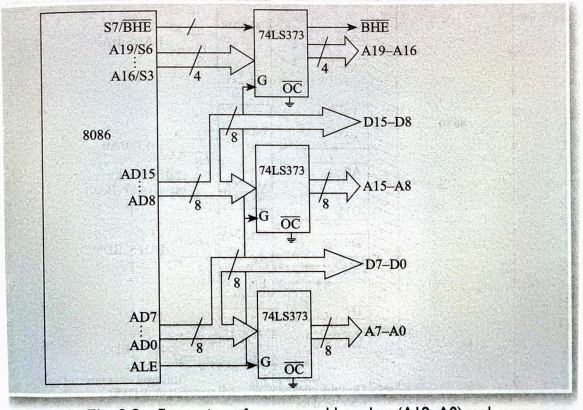


Fig. 9.2 Formation of separate address bus (A19–A0) and data bus (D15–D0) in the 8086

single-board 8086-based systems. The 74LS245 IC can be used as a transceiver (driver/receiver) for the data lines. Since the 8086 has 16 data lines, two 74LS245 ICs are required (the 74245 being an 8-bit transceiver).

9.2.2 Formation of Buffered Address Bus and Data Bus in 8086

Figure 9.3 shows the connection of two 74LS245s with the 8086. The $\overline{EN}$ (enable) pin determines whether or not data are allowed to pass through the 74LS245 and the DIR (direction) pin controls the direction of the data flow. When $\overline{EN} = 1$, data are not transmitted through the 74245 in either direction. If $\overline{EN}$ is 0, DIR = 0 causes the data to flow towards the 8086 and DIR = 1 results in data being outputted from the 8086. In an 8086-based system, the $\overline{EN}$ pin of the 74245 is connected to the $\overline{DEN}$ pin, since $\overline{DEN}$ of the 8086 is made active low (i.e., logic 0) whenever the processor is performing an I/O or memory read/write operation. The AD7–AD0 pins of the 8086 are connected to the inputs of one 74245 and the AD8–AD15 pins are connected to the inputs of another 74245. The DIR pin of the 74245 is tied to the processor's DT/$\overline{R}$ pin. The processor floats (tri-states) the $\overline{DEN}$ and the DT/$\overline{R}$ pins, in response to a bus request on the HOLD pin. Similarly, three 74244 ICs (buffer/driver) are used to generate the buffered address bus (BA0–BA19), as shown in Fig. 9.3. The Intel 8282 and 8286 ICs can be used instead of the 74LS373 and the 74245, respectively.

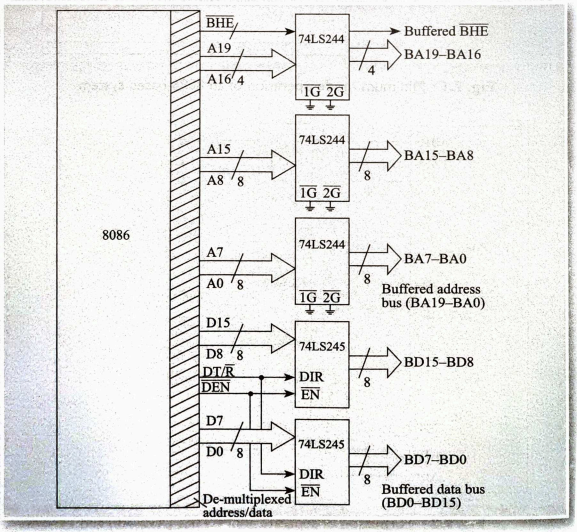


Fig. 9.3 Formation of buffered address bus and data bus in 8086

9.2.3 Connection of 8284A with 8086

Figure 9.4 (a) shows the clock generator IC (8284A), which supplies a train of pulses at a constant frequency to the 8086. The connection of the 8284A with the 8086 is shown in Fig. 9.4 (b). It synchronizes the Ready (RDY) signal (which indicates that an I/O device or memory interface is ready to complete a data transfer) received from an I/O or memory interface, by activating the READY input of the 8086 at the right time in a bus cycle. Similarly, when the Reset ($\overline{RES}$) signal of the 8284A is activated, it activates the RESET input of the 8086 at the right time in a bus cycle, which initializes the 8086 system. The clock pulse source applied to the 8284A may be from a pulse generator that is connected to the EFI pin or an oscillator that is connected across X1 and X2. If the input to F/$\overline{C}$ is 1, the EFI input determines the frequency. Otherwise, the oscillator input determines the frequency. In either case, the 8284 clock output (CLK) is one-third of the input frequency. All the devices, 74373, 74245, and 8284A, require only +5 V supply voltage. Their inputs and outputs are TTL-compatible and therefore the devices are compatible with each other and with the 8086. CSYNC is used in systems with multiple processors.

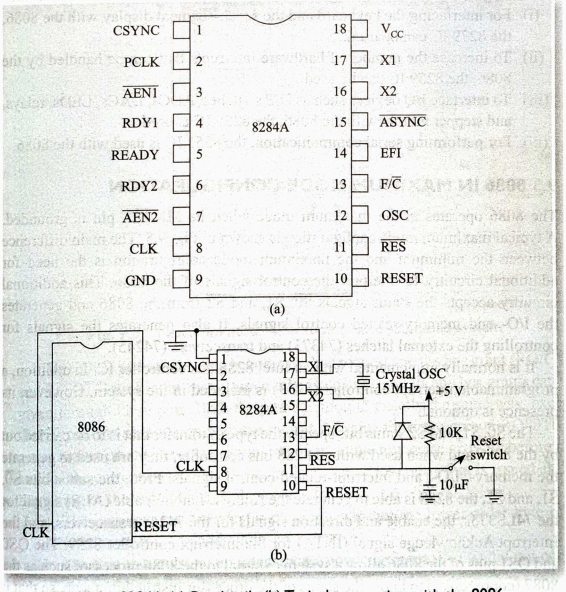


Fig. 9.4 8284A (a) Pin details (b) Typical connection with the 8086

In the minimum mode system, the control lines ($\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$) need not be passed through transceivers, but can be used directly. The $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ lines indicate the type of data transfer, as shown in Table 9.1.

Since the content of CS and IP are FFFFH and 0000H after reset, the first instruction for execution is fetched from the memory address FFFF0H (= CSX10H + IP) by the 8086. Hence, the system start-up program must be stored from the address FFFF0H in the memory. Normally, this address is assigned to a ROM type memory chip, so that the system start-up program is available permanently. The interrupt vector table is stored from the address 00000H in the memory, whenever the interrupt(s) is (are) to be used in the 8086-based system. In addition, depending upon the system requirement, specific interfacing ICs can be used along with the 8086.

Table 9.1 Function of the 8086 control signals in minimum mode operation

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

(i) For interfacing the keyboard and the seven-segment display with the 8086, the 8279 IC can be used.

(ii) To increase the number of hardware interrupts that can be handled by the 8086, the 8259 IC can be used.

(iii) To interface I/O devices such as DIP switches, ADCs, DACs, LEDs, relays, and stepper motors with the 8086, the 8255 IC is used.

(iv) For performing serial communication, the 8251 IC is used with the 8086.

9.3 8086 IN MAXIMUM MODE CONFIGURATION

The 8086 operates in the maximum mode when its MN/$\overline{MX}$ pin is grounded. A typical maximum mode configuration is shown in Fig. 9.5. The main difference between the minimum and the maximum mode configuration is the need for additional circuitry to interpret the control signals of the 8086. This additional circuitry accepts the status signals $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ from the 8086 and generates the I/O- and memory-related control signals. It also generates the signals for controlling the external latches (74373) and transceivers (74245).

It is normally implemented with an Intel 8288 bus controller IC. In addition, a programmable interrupt controller (8259) is included in the system. However, its presence is optional.

The $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status bits specify the type of transfer that is to be carried out by the 8086 and when used with an 8288 bus controller, they are used to generate the memory-, I/O-, and interrupt-related control signals. From the status bits $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$, the 8288 is able to generate the Address Latch Enable (ALE) signal for the 74LS373s, the enable and direction signals for the 74245 transceivers, and the Interrupt Acknowledge signal ($\overline{INTA}$) for the interrupt controller 8259. The QS0 and QS1 pins of the 8086 allow a system external to the 8086 processor such as the 8087 (coprocessor) to know the status of the processor instruction queue, so that it

can determine which instruction is currently executed by the 8086. The $\overline{\text{LOCK}}$ pin indicates that an instruction with a LOCK prefix is being executed and that the bus is not to be used by another master. These pins are needed only in multiprocessor systems.

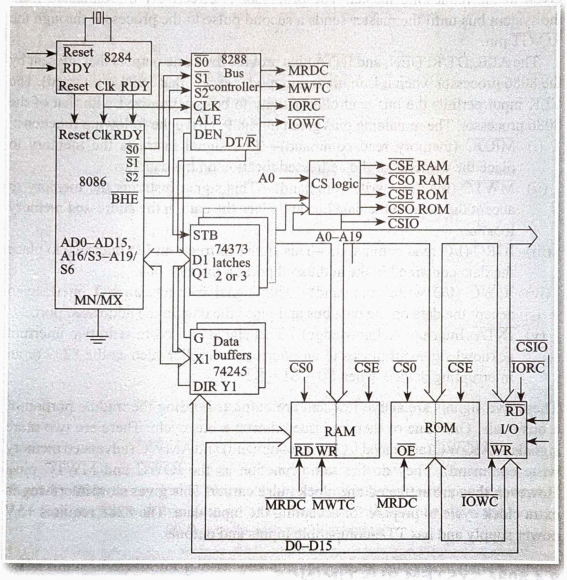


Fig. 9.5 Maximum mode operation of an 8086-based system

The HOLD and HLDA pins become the bus request and the bus grant ($\overline{\text{RQ}}/\overline{\text{GT0}}$ and $\overline{\text{RQ}}/\overline{\text{GT1}}$) pins in the maximum mode. Both bus requests and bus grants can be given through these pins. Both the pins function in exactly the same way, except that if requests are seen on both the pins at the same time, the one on $\overline{\text{RQ}}/\overline{\text{GT0}}$ is given higher priority. A request consists of an active low pulse arriving before the start of the current bus cycle. The grant is an active low pulse that is issued at the beginning of the current bus cycle provided that

(i) The previous bus transfer in the 8086 was not the lower-order byte of a word to or from an odd address.

(ii) The first low pulse of an interrupt acknowledgement ($\overline{\text{INTA}}$) did not occur during the previous bus cycle.

(iii) An instruction with a LOCK prefix is not being executed.

If condition (i) or (ii) is not met, the grant is not given until the next bus cycle; if condition (iii) is not met, the grant waits until the locked instruction is completed. In response to the grant, the tri-state pins of the 8086 (i.e., address, data, and control pins) are placed in their high impedance state and the next bus cycle is given to the requesting master. The processor is effectively disconnected from the system bus until the master sends a second pulse to the processor through the $\overline{RQ}/\overline{GT}$ pin.

The ALE, DT/$\overline{R}$, DEN, and $\overline{INTA}$ pins provide the same outputs that are sent by the 8086 processor when it is in minimum mode (except that DEN is inverted). The CLK input permits the bus controller activity to be synchronized with that of the 8086 processor. The remaining pins given in Fig. 9.5 have the following functions:

(i) $\overline{MRDC}$ (memory read command)—This signal instructs the memory to place the contents of the addressed location on the data bus.

(ii) $\overline{MWTC}$ (memory write command)—This signal instructs the memory to accept the data on the data bus and place the data in the addressed memory location.

(iii) $\overline{IORC}$ (I/O read command)—This signal instructs an I/O interface to place the data contained in the addressed port on the data bus.

(iv) $\overline{IOWC}$ (I/O write command)—This signal instructs an I/O interface to accept the data on the data bus and place the data in the addressed port.

(v) $\overline{INTA}$ (Interrupt Acknowledge)—This signal is used to send two interrupt acknowledgement pulses to an interrupt controller such as the 8259 or an interrupting device, when $\overline{S0} = \overline{S1} = \overline{S2} = 0$.

These five signals are active low and are outputted during the middle portion of a bus cycle. Only one of them is issued during a bus cycle. There are two more signals—$\overline{AIOWC}$ (advanced I/O write command) and $\overline{AMWC}$ (advanced memory write command). They do the same function as the $\overline{IOWC}$ and $\overline{MWTC}$ pins. However, they are activated one clock pulse earlier. This gives slow interfaces an extra clock cycle to prepare for accepting the input data. The 8288 requires +5 V power supply and has TTL-compatible inputs and outputs.

9.4 8086 SYSTEM BUS TIMINGS

This section discusses the timing diagram of the 8086 bus cycles—general bus operation, memory and I/O read cycle, and memory and I/O write cycle in minimum mode operation, and memory and I/O read cycle and memory and I/O write cycle in maximum mode operation. It also discusses the timing diagram for interrupt acknowledgement ($\overline{INTA}$) and the bus request and bus grant timing in minimum and maximum mode operation.

9.4.1 Timing Diagrams for General Bus Operation in Minimum Mode

The 8086 bus cycles are depicted with their T-states in Fig. 9.6. The length of a bus cycle in an 8086 system is four clock cycles, denoted by T1–T4, plus any number of wait state clock cycles, denoted by TW. If the bus is to be inactive or idle after the completion of a bus cycle, the gap between successive bus cycles

is filled with idle state clock cycles denoted by TI. During data transfer, the wait states are inserted between T3 and T4, when a memory or I/O interface is not able to respond quickly.

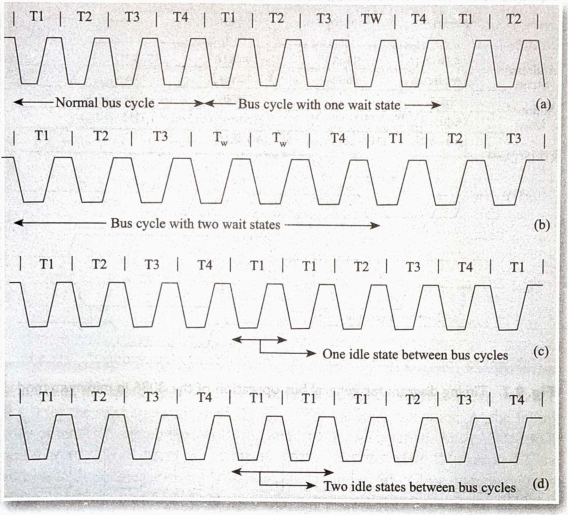


Fig. 9.6 8086 bus cycles (a) Normal bus cycle and bus cycle with one wait state (b) Bus cycle with two wait states (c) Bus cycle with one idle state, and (d) Bus cycle with two idle states

The timing diagram for general bus operation of the 8086 in minimum mode is shown in Fig. 9.7. If the Ready signal is still in low state at the beginning of T3, one or more wait states (TW) will be inserted between T3 and T4, until a Ready has been received (i.e., Ready is made 1). The bus activity during TW is the same as the activity during T3. A signal applied to an RDY input of the 8284A causes a Ready output to the 8086 at the falling edge of the current clock cycle.

The simplified timing diagram for the memory or I/O read cycle, which requires one wait state in the minimum mode is shown in Fig. 9.8.

The simplified timing diagram for the memory or I/O write cycle, which requires one wait state in the minimum mode is shown in Fig. 9.9.

When the processor is ready to initiate a bus cycle, it places the address in the lines AD15–AD0 and A19/S6–A16/S3, and the status of $\overline{\text{BHE}}$ in the line $\overline{\text{BHE}}$/S7, and applies a pulse to the ALE pin during T1. Before the falling edge of the ALE signal, the signals in the address lines $\overline{\text{DEN}}$, DT/$\overline{\text{R}}$, M/$\overline{\text{IO}}$, and $\overline{\text{BHE}}$ are made

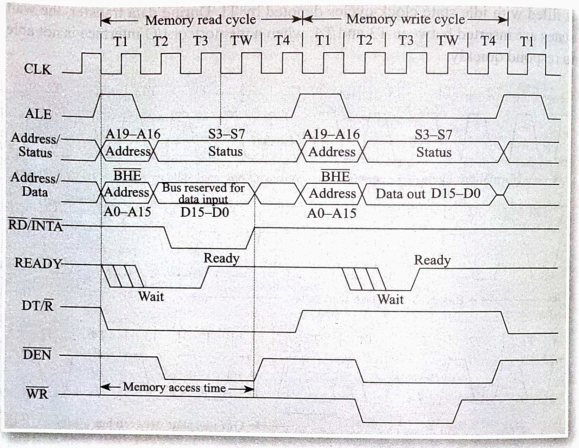


Fig. 9.7 Timing diagram for general bus operation of the 8086 in minimum mode

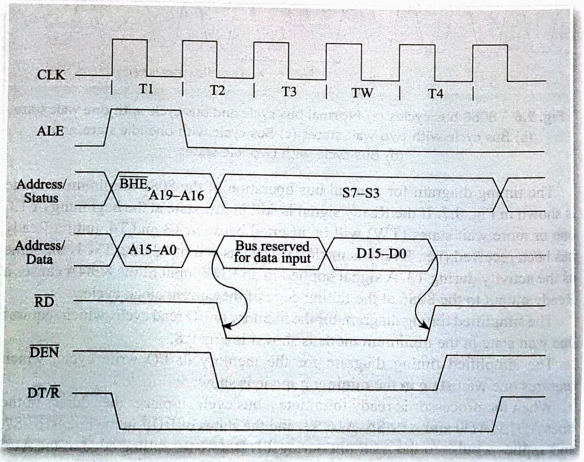


Fig. 9.8 Memory or I/O read cycle in minimum mode operation of the 8086

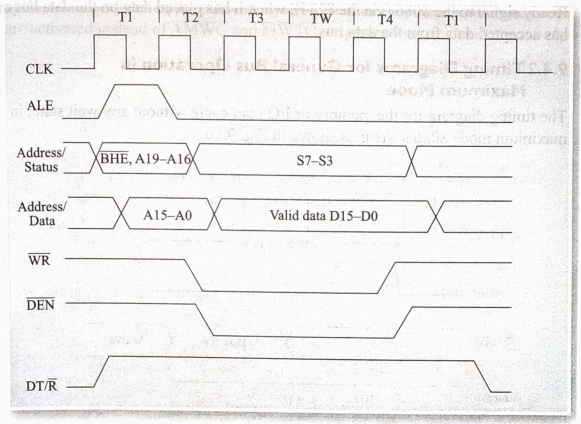


Fig. 9.9 Memory or I/O write cycle in minimum mode operation of the 8086

stable (i.e., the appropriate value, 1 or 0, is placed on the address lines), with DT/$\overline{R}$ = 0 for the read operation, and DT/$\overline{R}$ = 1 for the write operation. At the falling edge of the ALE signal, the 74LS373s latches the address in the lines AD15–AD0 and A19/S6–A16/S3, and the status of $\overline{BHE}$ in the line $\overline{BHE}$/S7. During T2, the address in these lines is removed and the status signals S3–S7 are outputted on the A16/S3–A19/S6 and $\overline{BHE}$/S7 pins. $\overline{DEN}$ is made logic 0 to enable the 74LS245 transceivers. The logic value in the line M/$\overline{IO}$ (which is not shown in Figs 9.7, 9.8, and 9.9) is 1 for memory-related operation and 0 for I/O-related operations.

If an input operation (i.e., read operation) is to be performed, $\overline{RD}$ is active low during T2, and the AD15–AD0 pins should enter a high impedance state in preparation for the receiving of input data. If the memory or I/O interface is ready to transfer data immediately, there are no wait states and the data are put on the bus during T3. After the input data are accepted by the 8086, $\overline{RD}$ is raised to 1 at the beginning of T4, and the memory or I/O interface removes its data upon detecting this transition.

For an output operation (i.e., write operation), the 8086 makes the signal $\overline{WR}$ = 0 and places the output data in the pins AD15–AD0 during T2. During T4, $\overline{WR}$ is made logic 1 and the data are removed.

For both input and output operation, $\overline{DEN}$ is made logic 1 during T4, to disable the transceivers. The M/$\overline{IO}$ signal is set according to the next data transfer at this time.

The bus timing of the 8086 has been designed such that the memory or I/O interface involved in a data transfer can control when data are to be placed on or taken from the bus by the interface. This is done by having the interface send a

Ready signal to the 8086 (via the 8284), when it has placed data on the data bus or has accepted data from the data bus.

9.4.2 Timing Diagrams for General Bus Operation in Maximum Mode

The timing diagram for the memory or I/O read cycle without any wait state, in a maximum mode 8086 system, is shown in Fig. 9.10.

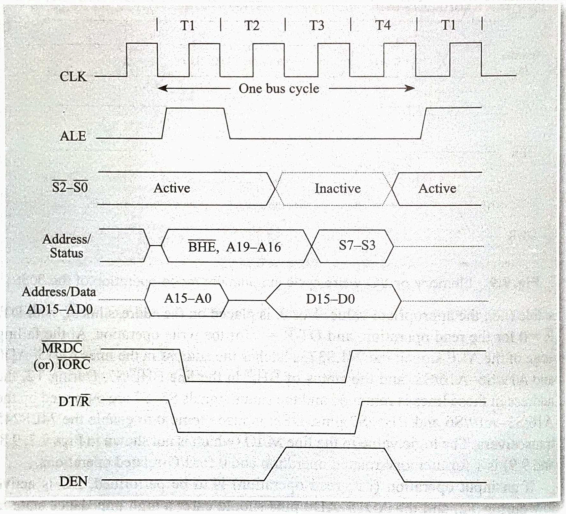


Fig. 9.10 Memory or I/O read cycle in maximum mode operation of the 8086

The timing diagram for the memory write cycle without any wait state, in a maximum mode 8086 system, is shown in Fig. 9.11. The status bits $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ are set just prior to the beginning of the bus cycle. Upon detecting a change from the passive state ($\overline{S0} = \overline{S1} = \overline{S2} = 1$), the 8288 outputs a pulse on its DT/$\overline{R}$ pin during T1. In T2, the 8288 sets DEN = 1, thus enabling the transceivers. For memory read operation, it activates $\overline{MRDC}$, which is maintained until the end of the clock period T4. For a memory write operation, $\overline{AMWC}$ is activated from T2 to T4 and $\overline{MWTC}$ is activated from T3 to T4. The status bits $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ remain active until the end of T3 and become passive (all 1s) during T3 and T4. As with the minimum mode, if the Ready input of the 8086 is not activated before the beginning of T3, the wait states are inserted between T3 and T4.

Similar to the memory read cycle, while performing the I/O read cycle, the control signal $\overline{IORC}$ is activated instead of $\overline{MRDC}$. Similar to the memory write

cycle, while performing the I/O write cycle, the control signals $\overline{\text{AIOWC}}$ and $\overline{\text{IOWC}}$ are activated instead of $\overline{\text{AMWC}}$ and $\overline{\text{MWTC}}$, respectively.

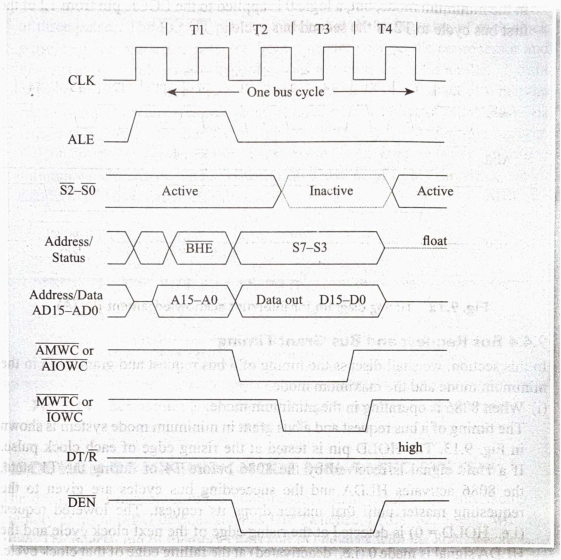


Fig. 9.11 Memory or I/O write cycle in maximum mode operation of the 8086

9.4.3 Interrupt Acknowledgement ($\overline{\text{INTA}}$) Timing

When an interrupt is received through the INTR pin of the 8086 in the minimum and maximum modes, the 8086 generates the interrupt acknowledgement ($\overline{\text{INTA}}$) signal, which we shall now discuss in detail.

(i) When the 8086 is operating in minimum mode:

The timing diagram for the Interrupt Acknowledgement ($\overline{\text{INTA}}$) signals of the INTR interrupt is shown in Fig. 9.12. If an INTR interrupt request has been recognized during the previous bus cycle and an instruction has just been completely executed by the 8086, a negative pulse is applied to the $\overline{\text{INTA}}$ during the current and the next bus cycles. Each of these pulses extends from T2 to T4. Upon receiving the second $\overline{\text{INTA}}$ pulse, the interface receiving the $\overline{\text{INTA}}$ signal puts the interrupt type on the lines AD7–AD0, which are floated for the rest of the time (during the two bus cycles). The interrupt type is available from T2 to T4.

(ii) When 8086 is operating in the maximum mode:

In this mode, the Interrupt Acknowledgement ($\overline{INTA}$) signals are the same as in the minimum mode, but, a logic 0 is applied to the $\overline{LOCK}$ pin from T2 of the first bus cycle to T2 of the second bus cycle.

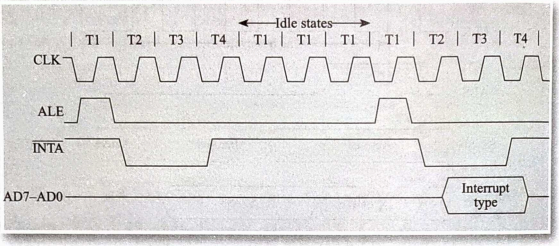


Fig. 9.12 Timing diagram for interrupt acknowledgement ($\overline{INTA}$)

9.4.4 Bus Request and Bus Grant Timing

In this section, we shall discuss the timing of a bus request and grant, both in the minimum mode and the maximum mode.

(i) When 8086 is operating in the minimum mode:

The timing of a bus request and a bus grant in minimum mode system is shown in Fig. 9.13. The HOLD pin is tested at the rising edge of each clock pulse. If a Hold signal is received by the 8086 before T4 or during the T1 state, the 8086 activates HLDA and the succeeding bus cycles are given to the requesting master until that master drops its request. The lowered request (i.e., HOLD = 0) is detected at the rising edge of the next clock cycle and the HLDA signal is made 0 (i.e., deactivated) at the falling edge of that clock cycle. While HLDA = 1, all the three-state outputs of the 8086 are put in their high impedance state. The instructions already in the instruction queue continue to be executed, until one of them requires the use of the bus to access the memory or the I/O device.

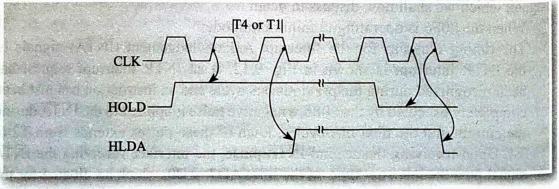


Fig. 9.13 Bus request and bus grant timing in minimum mode operation of the 8086

(ii) When 8086 is operating in the maximum mode:

The timing of a bus request and a bus grant in a maximum mode 8086 system is shown in Fig. 9.14. A request/grant/release is accomplished by a sequence of three pulses. The $\overline{RQ}/\overline{GT}$ pins are checked at the rising edge of each clock pulse, and if a request is detected from a master such as the coprocessor and the necessary conditions discussed earlier are met, the 8086 applies a grant pulse to the $\overline{RQ}/\overline{GT}$ immediately following the next T4 or T1 state. When the requesting master receives this pulse, it takes over the control of the bus. This master may control the bus for one or several bus cycles. When it is ready to relinquish the bus, it sends the release pulse to the 8086 over the same line through which it made the request. $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ are the same, except that $\overline{RQ}/\overline{GT0}$ has higher priority.

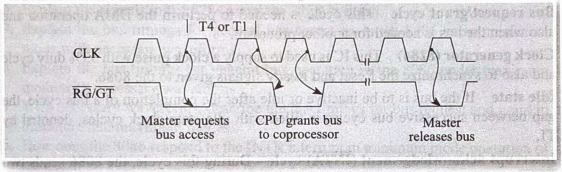


Fig. 9.14 Bus request and bus grant timing in maximum mode operation of the 8086

9.5 DESIGN OF MINIMUM MODE 8086-BASED SYSTEM

The design of a minimum mode 8086-based system requires the interfacing of memory chips and I/O devices such as DIP switches, LEDs, 8255, etc., with the 8086. The interfacing of memory chips and I/O devices with the 8086 has already been explained in Chapter 7. There must be a ROM/EPROM chip at the address FFFF0H, which has the monitor program stored in it, as the 8086 fetches the first instruction from that address for execution after power up and reset. In addition, there must be a ROM/EPROM or RAM chip at the address 00000H, if the system uses interrupts, as the interrupt vector table (IVT) is stored starting at that address. For interfacing the 8255 with the 8086, the concepts used to interface 8-bit I/O devices with the 8086, and the 8255 with the 8085, can be combined.

POINTS TO REMEMBER

- The 8086 can be configured to operate either in minimum or maximum mode.
- The 8284 IC (clock generator) is used to generate the clock and Ready signals for the 8086.
- The 8288 IC (bus controller) is used in the maximum mode of operation of the 8086 to generate the memory and the I/O control signals using the status signals of the 8086.

- The bus cycles of the 8086 may or may not have wait states and idle states.
- There exist different timing diagrams in the 8086, such as the timing diagram for general bus operation (i.e., the memory or I/O read cycle and the memory or I/O write cycle) of the 8086, interrupt acknowledgement, and bus request/grant in minimum and maximum modes.

KEY TERMS

Bus controller (8288) This IC is used to generate the control signals for the memory and I/O device using the status signals $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ in the maximum mode operation of the 8086.

Bus cycle Each bus cycle of the 8086 has four clock periods, T1–T4 plus any number of wait state clock cycles denoted by T_w.

Bus request/grant cycle This cycle is needed to perform the DMA operation and also when the bus is needed for another processor.

Clock generator (8284) This IC is used to supply a clock pulse with 33% duty cycle and also to synchronize the Reset and Ready signals given to the 8086.

Idle state If the bus is to be inactive or idle after the completion of a bus cycle, the gap between successive bus cycles is filled with idle state clock cycles, denoted by TI.

Interrupt acknowledgement ($\overline{INTA}$) cycle During this cycle, the 8086 sends two $\overline{INTA}$ pulses to an external interface after receiving the INTR interrupt, to get the interrupt type number for the INTR interrupt.

I/O read cycle During this cycle, the 8086 reads data from the input device.

I/O write cycle During this cycle, the 8086 writes data into the output device.

Maximum mode In this mode operation, there is more than one processor, and all the control signals for the memory and the I/O device are generated by the bus controller (8288) chip.

Memory read cycle During this cycle, the 8086 reads data or instructions from the memory.

Memory write cycle During this cycle, the 8086 writes data into the memory.

Minimum mode In this mode operation, there is only one 8086 processor, and all the control signals for the memory and the I/O device are generated by the processor itself.

Wait states These states are inserted between T3 and T4, when a memory or I/O interface is not able to respond quickly enough during a data transfer. This is achieved with the help of the Ready input in the 8086.

REVIEW QUESTIONS

1. What is meant by minimum mode operation of the 8086?
2. What is meant by maximum mode operation of the 8086?
3. What is the function of the MN/$\overline{MX}$ pin in the 8086?
4. How are the control signals $\overline{MEMR}$ and $\overline{MEMW}$ generated using the M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals in the minimum mode operation of the 8086?

5. How are the control signals $\overline{IOR}$ and $\overline{IOW}$ generated using the M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals of the 8086?
6. What is the function of the chip select logic and what are the inputs given to it?
7. Write the function of the clock generator IC (8284).
8. What is the role of the bus controller IC (8288)?
9. What is meant by memory read and memory write cycles?
10. What is meant by I/O read and I/O write cycle?
11. Write the function of the signals $\overline{DEN}$ and DT/$\overline{R}$ in the 8086.
12. What is the function of the signals M/$\overline{IO}$ and $\overline{BHE}$ in the 8086?
13. What is the role of the pin F/$\overline{C}$ in the 8284A?
14. Explain the minimum mode configuration of the 8086-based system with the necessary block diagram.
15. Describe the maximum mode configuration of the 8086 with the necessary block diagram.
16. Explain the signals in the 8284A and the 8288 in detail.
17. Explain the bus timings for general bus operation in the 8086 under minimum mode with necessary waveforms.
18. Explain the bus timings for general bus operation in the 8086 under maximum mode with necessary waveforms.
19. With necessary waveforms, describe the bus timings for bus request and grant in minimum and maximum modes.
20. How does the 8086 respond to the INTR interrupt in minimum mode operation of the 8086?

NUMERICAL/DESIGN-BASED EXERCISES

1. Design an 8086-based minimum mode system that contains the following components:
   (i) Two 8K × 8 EPROM chips having the address range FC000H–FFFFFH
   (ii) Two 8K × 8 RAM chips having the address range 80000H–83FFFH
   (iii) Two seven-segment LEDs with common anode connection, having the addresses 80H and 81H
   (iv) An 8-bit DIP switch having the address FF80H
2. Interface an 8255 chip with the 8086 operating in minimum mode so that the addresses 80H, 82H, 84H, and 86H are assigned to port A, port B, port C, and the control register of the 8255, respectively.

THINK AND ANSWER

1. How are wait states introduced in a bus cycle of the 8086?
2. Why and when are the idle states introduced in a bus cycle of the 8086?
3. How many clock periods are present in a bus cycle of the 8086 without wait states?
4. What is the difference between the $\overline{AMWC}$ and $\overline{MWTC}$ signals?
5. How does the 8288 generate the control signals for memory and I/O devices?