Micro-Processor
and
Assembly Language

BCA- 203

Directorate of Distance Education
Maharshi Dayanand University
ROHTAK – 124 001

Maharshi Dayanand University
ROHTAK – 124 001

Developed & Produced by EXCEL BOOKS, A-45 Naraina, Phase-I, New Delhi-110028

Contents

UNIT 1 INTRODUCTION TO MICROPROCESSOR 1

Introduction to Microprocessor

	Evolution of Microprocessor	
	Overview of Intel Pro-Pentium	
	Motorola 68000 Series	
	Introduction to DEC Alpha, Power PC	
	RISC & CISC Characteristics	
UNIT 2	BASIC MICROPROCESSOR ARCHITECTURE AND INTERFACE	16
	Internal Architecture	
	External System Bus Architecture	
	Memory and Input/Output Interface	
UNIT 3	PROGRAMMING MODE	44
	Register Organization of 8086	
	Memory Addressing and Instruction Formats	
	Memory Interfacing	
	Cache Memory and Cache Controllers	
UNIT 4	BASIC I/O INTERFACE	63
	I/O Interface	
	8255 Programmable Interface	
	8254 Programmable Timer	
	8251 Programmable/Communication Interface	
	Interrupts	
	8259 Programmable Interrupts Controller	
	Real Time Clock	
	DMA	
	8237/8257 DMA Controller	
UNIT 5	8086 ASSEMBLY LANGUAGE PROGRAMMING	101
	Instruction set of 8086	
	Assembler Directives and Operators	
	A Few Machine Level Programs	
	Machine coding and Programs	

Programming with an Assembler

Assembly Language Example Programs

APPENDIX 170

Introduction
Evolution of Microprocessor
Overview of Intel Pro-pentium
Motorola 68000 Series
DEC
PowerPC
RISC / CISC Architecture

Unit 1

Introduction to Microprocessor

Learning Objectives

After reading this unit you should appreciate the following:

- Introduction to Microprocessor
- Evolution of Microprocessor
- Overview of Intel Pro-Pentium
- Motorola 68000 Series
- Introduction to DEC Alpha, Power PC
- RISC & CISC Architecture

Top

Introduction

Microprocessor acts as a CPU in a microcomputer. It is present as a single IC chip in a microcomputer. Microprocessor is the heart of the machine.

A Microprocessor is a device, which is capable of

1. Receiving Input
2 Performing Computations
3. Storing data and instructions
4. Display the results
5. Controlling all the devices that perform the above 4 functions.

The device that performs tasks is called Arithmetic Logic Unit (ALU). A single chip called Microprocessor performs these tasks together with other tasks.

A MICROPROCESSOR is a multipurpose programmable logic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to those instructions and provides results as input

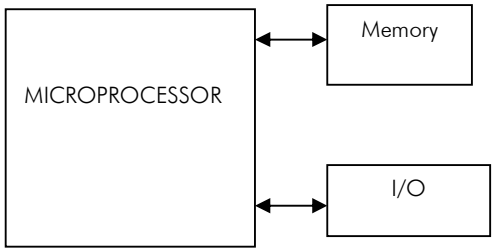


Figure 1.1

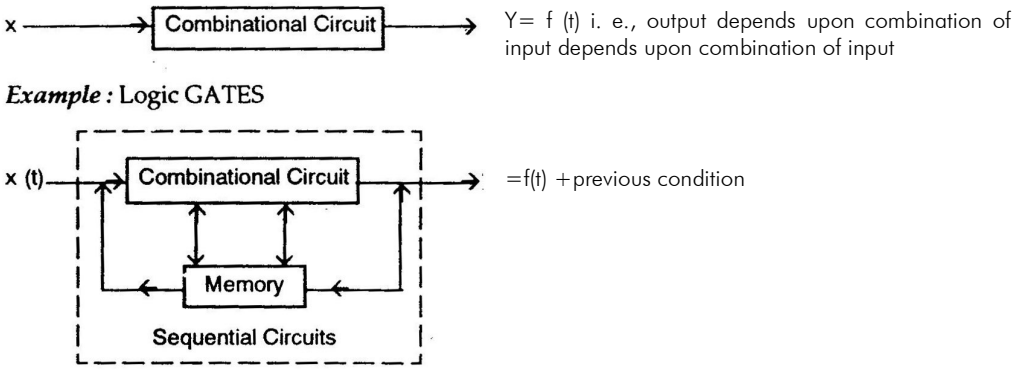
Figure shows a programmable machine, which consists of a microprocessor, memory, I/O. All these three component work together to perform a given task.

Top

Evolution of Microprocessor

The digital circuits and systems can be broken into:

- Combinational Circuits: and
- Sequential Circuits



Example: Flip-Flops Registers, Counters etc

It is the notion that the digital circuits and systems are the by product of the Boolean functions. Let any Boolean function be expressed as:
f_{ki} (A,B,C,D …….).

Where, k → Number of Boolean Variables,

and i → Total Boolean functions generated by these variables.

If, for example, k = 4, then i = 2^4 = 0,1.....15.

Hence, a Boolean function can be expressed as:

f_{ki} (A,B,C,D)

where i = 0,1.....15.

Example: Consider a function as:

F_{41} (A, B, C, D) = AB + $\overline{B}C$ + CD + $\overline{A}B$

Here k = 4, and i = 1

This function can be realized in a number of ways depending upon the types of the technologies used:

(i) Discrete Element: The use of discrete element for realizing any function was the first technology used before 1960 and had to generate each Boolean variable by discrete elements to realize the whole function.

In this method, each variable was realised independently and these are combined to get the functions. The performance of the assembled circuit depended upon the individuals how neatly he could do it apart from the complexity of the circuit. It required often much more time for assembling and every one had to assemble it separately in hi$ own way. This created problems in automated working environment.

(ii) SSI: The innovation in the semiconductor technology forced the scientists & engineers to think many a times to put them as a package either in the form of a single IC or in hybrid form. In the year 1965, the development of Integrated circuit technology came into existence which gave the possibility of packing 100 transistors on the single chip. By this time the two technologies, namely, SSI (< 10) and MSI (< 100) were used to generate and realise functions. In SSI technology, AB, B'C, CD, A'B were generated independently for realising the whole function. Thus, SSI required 5 - chips, one each for AB, B'C, CD, A'B, and AB +B'C+CD+A'B.

(iii) MSI: In the MSI technology, AB+B'C and CD+A'B were generated independently for realising the whole function. Thus in SSI technology more than 4-chips were required whereas in MSI technology (just) more than 2-chips were required for realising the same function. This technology was used between 1965 to 1970.

(vi) LSI: The continued search in semiconductor technology resulted into realisation of more functions due to high packing density. Hence the LSI technology further increased the facility of packing transistors on a single chip upto a few thousands. With this technology, whole functions could be realised with only one chip. For large values of k, the function became more complicated and it may not be possible to realise the whole function by a single chip. The continued search in the semiconductor technology resulted in the development of the VLSI technology wherein many more components could be packed on a single chip.

The continued development of IC technology resulted in realisation of more complicated functions with better reliability, compactness, low cost and low power dissipation. Figure 1.2. shows different stages of integration. Chips such as counters, memory devices, etc. were developed using MSI and LSI technologies. With passage of time, the IC technology developed at an incredible pace and all Boolean functions could be realized on a single chip. The chip capable of processing all Boolean functions was given the name of Processor. On the same line, the chip which processed the data in a controlled manner was called the microprocessor.

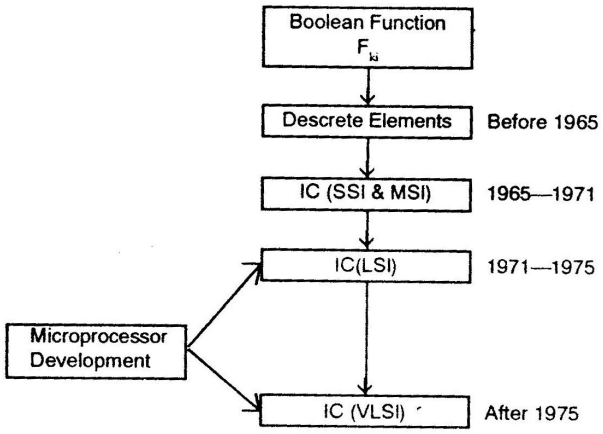


Figure 1.2: Microprocessor family Development

There were people who said that the microprocessor was the accidental by product of the general procedure for 'manufacturing the high density semiconductor memories using MOS-LSI technology. At the end of the 1960-70 decade, the need for the type of technology which placed thousands of transistors on a single chip became obvious. However, it was not yet completely clear as to which function could use so many devices effectively. Memories, of course, were one among such functions. With the design and development of such semiconductor memories, the need of an equally complex and efficient processor was felt which not only were able to use them effectively, but could pave the way for the selling of semiconductor memories easily.

The microprocessor age began with the advancement in the IC technology to put all necessary functions of a CPU into a single chip. Advancement in semiconductor technology increased the capacity to include more and more logic on a single chip. Although the cost of the microprocessor increased with its complexities, yet it was much lower than the cost of the equivalent logic scattered over several less capable chips. In addition to reduced number of ICs needed to perform a given function, the total number of pins are reduced and hence assembly cost was also reduced.

1st Generation Microprocessor

At the end of the 70s a group of engineers developed a chip capable of processing data. This chip was given the name, processor chip. The large processors were developed using VLSI technology. Another successful attempt of engineers, in developing a processor which worked in a con trolled manner, was given the name of microprocessor. Thus a single LSI/ VLSI/VVLSI chip, capable of processing data in a controlled manner was called the microprocessor.

The design team headed by Ted Hoff of Intel Corporation developed the 1st such controlled processor in the year 1969, but Intel started marketing its first microprocessor in the name of Intel 4004 in 1971. This was a 4-bit microprocessor having 16-pins hocused in a single chip of pMOS technology. This was called the first generation microprocessor. The 4-bit microprocessor worked with 4-bit word .The Intel 4004 along with few other devices was used for making calculators. The ability of changing functions of any system by just changing the programming rather than redesigning the hardware was the key behind the evolution of the microprocessors.

The Intel 8008 was developed in the year 1972 that worked with 8-bit word.

It required about 20 or more additional devices(chips) to design a functional CPU.

A few first generation microprocessors are listed in Table 1.1.

Table 1.1

Microprocessor	Word Size	Microprocessor	Word Size
INTEL 4004 &4040	4- bit	INTEL 8008	8-bit
FAIRCHILD PPS-25	″ ″	NATIONAL IMP-8	″ ″
NATIONAL IMP-4	″ ″	ROCKWELL PPS	″ ″
ROCKWELL PPS-4	″ ″	AMI 7200	″ ″
MICROSYSTEM	″ ″	MOSTEK 5065	″ ″

Types of microprocessors

Microprocessors fall into three categories:

- Single Chip Microcomputers → Contains microprocessor, ROM, RWM, I/O port, clock and timer.
- General purpose microprocessor.
- Bit slice microprocessor.

The general purpose microprocessor contain ALU with one or more registers which functioned as accumulator, a control unit, an instruction decoder which handled a fixed instruction set and general and special purpose registers which varied significantly from microprocessor to microprocessor. A microprocessor may have an internal stack of fixed length or use external memory for stack. The general purpose microprocessor are available of word lengths of 1, 4, 8, 16, 32, and 64 bits.

The Bit slice microprocessor divide the functions of ALL, general purpose and special purpose registers and control unit into several ICs. For this general purpose registers and ALU were packed in separately from controls. Each register of ALU (RALU) package was essentially equivalent to 2 or 4-bit wide slice of registers and the ALU of the microprocessor. Bit slice processor could be cascaded to produce any unconventional or conventional word length of the microprocessor such as 4, 8, 10, 12, 16, 32 or higher bits. The control portion ofbit slice processor was constructed from microprocessor sequencer IC and other logics.

IInd Generation Microprocessor

The second generation microprocessor using nMOS technology appeared in the market in the year 1973.

Table 1.2: List of 2nd generation microprocessors.

Microprocessors	Word Size
INTEL 8080 / 8085	8 - bit
FAIRCHILD F8	″ ″
MOTOROLA M 6800	″ ″
NATIONAL CMP-8	″ ″
RCA COSMAC	″ ″
MOS Tech. 6500	″ ″
SIGNETICS 2650	″ ″
ZILOG Z-80	″ ″
INTERSIL 6100	12-BIT
TOSHIBA TLCS-12	″ ″

The Intel 8080, an 8-bit microprocessor, of nMOS technology was developed in the year 1974 which required only two additional devices to design a functional CPU. Since 8080 was nMOS device, it was

much faster and had many more instructions than 8008 that facilitated the programming. The advantages of IInd generation microprocessors were

- Large chip size (170x200 mil) with 40-pins.
- More chips on decoding circuits.
- Ability to address large memory space (64-K Byte) and I/O ports(256).
- More powerful instruction sets.
- Dissipate less power.
- Better interrupt handling facilities.
- Cycle time reduced to half (1.3 to 9 μ sec.)
- Sized 70x200 mil) with 40-pins.
- Less Support Chips Required
- Used Single Power Supply
- Faster Operation

IIIrd Generation Microprocessor

The single chip 3rd generation microprocessor having 64-pins started with the introduction of 16-bit Intel 8086 in the year 1978. The other important IIIrd generation microprocessors were Zilog Z-8000, Motorola M68000, National NS16016, and Texas Instruments TMS 99000 series, etc. The 16-bit microprocessor using HMOS technology achieved enhanced performance parameters w.r.t. the 8-bit microprocessors. In addition to enhanced performance, it contained multiply/divide arithmetic hardware. The memory addressing capabilities were increased i.e. IM Byte to 16 Mbyte through a variety of flexible and powerful addressing modes.

Intel 8088 was identical to 8086 but for the 8-bit data bus. Hence 8088 could read orwrite8-bits data at a time to or from the memory. The Intel 80186 and 80188 were the improved versions of Intel 8086 and 8088, respectively. In addition to 16-bit CPU, the 80186 and 80188 had programmable peripheral devices integrated on the same package. The program written for 80186 and 80188 may not work well on 8086 and 8088, but those written for 8086 and 8088 worked without much difficulties on 80186 and 80188. This means they were upward compatible with 8086 and 8088. The Intel 80286 was the advancedversionof80186. It is designed for use in multi-user/ multitasking environment.

IVth Generation Microprocessor

The single chip 32-bit microprocessor was introduced in the year 1981 by Intel as iAPX 432. The other 4th generation microprocessors were; Bell Single Chip Bellmac-32, Hewlett-Packard, National NSl 6032, Texas Instrument99000. Motorola 68020 and 68030.

The power of the microprocessor went on increasing with the advancement in the integrated circuit technology. The VLSI technology culminated in the extremely complex microprocessor with as many as one billion transistors on a single chip. The Intel in the year 1985 announced the 32-bit microprocessor (80386). The 80486 has already been announced and is also a 32-bit microprocessor.

Most of the microprocessors were manufactured with HMOS(high density short channel MOS) technology because of the following advantages:

(i) Speed-power product was 4-times greater than NMOS. Its typical value was I-pico joule whereas it was 4-pico joules in the case of NMOS technology.

(ii) Circuit density was approximately 2-times greater than NMOS. The typical NMOS density was 4128 um^2 gate whereas it was 1852.5 um^2 for HMOS.

Tabular comparison for μ PS' Parameters

Tables 1.3 (a) and (b) list the characteristic of some Intel microprocessor.

Table 1.3(a)

	8008	8080	8085	8086	80286	80386
Supply	+5V,-9V	+5V&12V	+5V	+5V	+5V	+5V
Year	1972	1974	1976	1978	1982	1985
No.of transistors	2000	4500	6500	20000	—	2,75,000
Pinouts	18	40	40	40	68	132
Technology	PMOS	NMOS	NMOS	HMOS	HMOS	CHMOSIII
Word Size	8	8	8	16	16	32
Direct addressing	16KB	64KB	64KB	1MB	16MB	4GB
Basic 1instructions	48	78	80	97	97	151
GPRs	6	8	8	16	16	8
Max CLK (MHz)	0.8	2.6	5.5	5*	10	24
Gate Delay	30ns	15ns	3ns	3ns	—	—

** 8086-1=10MHz & 8086 6-2=8MHz

Table 1.3(b): Microprocessors Characteristics

Features	38080 3MCS80	8085 MCS8085	8086 iAPX86	8088 iAPX88	80186 iAPX186	80188 iAPX188	80286 iAPX286	80386 iAPX386
Bus inter face (bits)	8	8	16	8	16	8	16	32 & 16
Internal data path	8 Bits	8 Bits	16 Bits	16 Bits	16 Bits	16 Bits	16 Bits	32 Bits
CLK(speed selection	2,2.6 & 3 MHz	3,5 & 6 MHz	5,8 & 10 MHz	5 & 8 MHz	6 & 8 MHz	6 & 8 MHz	6,8 & 10 MHz	12 & 16 MHz
Bus BW (max.)	0.75 Byte/s	1.5M Byte/s	5M Byte/s	2M Byte/s	4M Byte/s	2M Byte/s	10M Byte/s	32M Byte/s
R<->R Add time/word	1.3 μs	0.67 μs	0.3 μs	0.38 μs	0.3 μs	0.3 μs	0.2 μs	125 ns
INTR response time	7.3 μs	2 μs	6.1 μs	8.6 μs	5.25 μs	8.3 μs	3.5 μs	
Memory Add ressability	64K	64K	1MB	1MB	1MB	1MB	16M	4GB
Virtual Memory	NO	NO	NO	NO	NO	NO	1GB/task	64TB
On Chip (Management)	NO	NO	NO	NO	NO	NO	YES	YES
I/OAddressability	256B	256B	64KB	64KB	64KB	64KB	64KB	4GB
Add. Modes	5	5	24	24	24	24	24	7
Co-proce.Interface	NO	NO	YES	YES	YES	YES	YES	YES
REGISTER Arithmet	1	1	8	8	8	8	8	8
REGISTER Index	0	0	4	4	4	4	4	-
REGISTER Segment	0	0	4	4	4	4	4	6
REGISTER GRPs	6	6	8	8	8	8	8	8
Code compatability	8080code				8086 code			

It has a 32 bit address bus and a 64 bit data bus. Some of the features are Superscalar architecture (more than one execution unit),on-chip cache memory for data and code, Branch prediction, high performance floating point unit, Performance monitoring.

Top

Overview of Intel Pro-pentium

The two biggest players in the PC CPU market are Intel and Motorola. Intel has enjoyed tremendous success with its processors since the early 1980s. Most PCs are controlled by Intel processors. The primary exception to this rule is the Macintosh. All Macs use chips made by Motorola. In addition, there are several firms, such as AMD and Cyrix, that make processors which mimic the functionality of Intel's chips. There are also several other chip manufacturers for workstation PC's.

The Intel Processors

The Intel Corporation is the largest manufacturer of microchips in the world, in addition to being the leading provider of chips for PCs. In fact, Intel invented the microprocessor, the so-called "computer on a chip," in 1971 with the 4004 model It was this invention that led to the first microcomputers that began appearing in 1975. However, Intel's success in this market was not guaranteed until 198l, when IBM released the first IBM PC, which was based on the Intel 8088. Since then all IBM machines and the compatibles based on IBM's design have been created around Intel's chips. A list of those chips, along with their basic specifications, is shown in Table 1.4. Although the 80>SS was the first chip to be used in an IBM PC IBM actually used an earlier chip, the 808(1, in a subsequent model, called the IBM PC XT. The chips that came later—the 286, 386, 486, and even the Pentium-I—correspond

To certain design standards that were established by the 8086. this line of chips often referred to as the 80×86 line.

The steady rise in bus size, register size, and addressable memory illustrated in Table 1.4 has also been accompanied by increases in clock speed. For example, the clock attached to the first PCs ran at 4.77 MHz. Whereas clock speeds for Pentium chips started at 60 MHz. In 1993 and quickly rise to 100, 120, and 133, 150, and 166 MHz.

Table 1.4: Intel Chips and their Specifications

MODEL	YEAR INTRODUCED	DATA BUS CAPACITY	REGISTER SIZE	ADDRESSABLE MEMORY
8086	1978	16 bit	16 bit	1 MB
8088	1979	8 bit	16 bit	1 MB
80286	1982	16 bit	16 bit	16 MB
80386	1985	32 bit	32 bit	4 GB
80486	1989	32 bit	32 bit	4 GB
Pentium	1993	64 bit	32 bit	4 GB
Pentium Pro	1995	64 bit	32 bit	64 GB

It is important to realize that these statistics do not convey all the improvements that have been made. The basic design of each chip, known as the architecture, has grown steadily in sophistication and complexity. For example, the architecture of the 386 contained 320,000 transistors, ANCI the 486

contained 1.2 million. With the Pentium, that number grew to more than 3.1 million, and the Pentium Pro's architecture brought the total number of transistors on tile chip to 5.5 million. The growing complexity of the architecture allowed Intel to incorporate some sophisticated techniques for processing. One major improvement that came with the 386 is called virtual 8086 mode. In this mode, a single 386 chip could achieve the processing power of 16 separate 8086 chips each running a separate copy of the operating system. It's like capability for virtual 8086 mode enabled a single 386 chip to run different programs at the same time, a technique known as multitasking. All the chips that succeeded the 386 have had the capacity for multitasking.

The 486

Introduced in 1989 the 80486 did not feature any radically new processor technology. Instead, it combined a 386 processor, a math coprocessor, and a cache memory controller on a single chip. Because these chips were no longer separate, they no longer had to communicate through the bus—which increased the speed of the system dramatically.

The Pentium

I he next member of the Intel family of microprocessors was the Pentium, introduced In 1993. With the Pentium, Intel broke its tradition of numeric model names—partly to prevent other chip manufactures from using similar numeric names, which implied that their products were functionally identical to Intel's chips. The Pentium, however, is still considered part of the 80x86 series.

The Pentium chip itself represented another leap forward for microprocessors. The speed and power of the Pentium dwarfed all of its predecessors in the Intel line. What this means in practical terms is that the Pentium runs application programs approximately five times faster than a 486 at the same clock speed. Part of the Pentium's speed comes from a super-scalar architecture, which allows the chip to process more than one instruction in a single clock cycle.

The Pentium Pro

Introduced in 1995, the Pentium Pro reflected still more design breakthroughs. The Pentium Pro can process three instructions in a single clock cycle—one more than the Pentium. In addition, the Pentium Pro can achieve faster clock speeds—the earliest model available was packaged with a 133 MHz clock. Intel coined the phrase "dynamic execution" to describe the specific innovation that distinguishes the Pentium Pro. Dynamic execution refers to the chip's ability to execute program instructions in the most efficient manner, not necessarily in the order in which they were written. This out-of-order execution means that instructions that cannot be executed immediately are put aside, while the Pentium Pro begins processing other instructions. This is in contrast to the original Pentium chip that can stall because it executes instructions in strict sequence.

Top

Motorola 68000 Series

68000 microprocessor is a 16 bit processor with an addressing space of 65536 locations, each of which holds a 64-bits word; In order to address those locations, 16-bits operands are needed, two of which leave 32 bits free for other purposes; of these 32 bits, 16 are used to hold the opcode.

68000 Architecture has Instructions

- To move or manipulate a block of many locations at one time
- Instructions that perform tasks that are typical of an operating system.

68000 has a total of 47 mnemonic codes, 25 of which denote “ordinary” instructions, 14 denote instructions that perform tasks typical of an operating system, while the last 8 are instructions acting on blocks of words.

There are 13 essentially different addressing modes, further subdivided into 18 varieties; in particular, there are modes to address a special area of the memory array

Subroutines were added in view of the needs of complex programs, and a set of powerful instructions for consistent management of relative addressing was developed.

Top

DEC

A powerful new Alpha 64 bit RISC computer chip was introduced in 1977, as new VAX (Virtual Address Extension) Computer. The VAX was a 32 bit based computer line based on operating system, called VMS. This new computer system was more powerful and had better time-sharing capabilities than past DEC systems DECnet networking technology-enabled customers to connect different types of DEC computers together from small workstations to large corporate servers.

As the 80s drew to a close DECs were not IBM compatible and were designed as a way for users to connect to DEC server systems rather than stand-alone products. In addition, DEC manufactured all the components for its PCs, resulting in systems that were much more expensive than other PC vendors.

From 1984 to 1988 DEC developing its own RISC chip, the PRISM. Using these chips, DEC introduced a UNIX workstation running on its own UNIX, Ultrix

The RISC chips to be used in these new VAXes was a new 64 Bit chip, designed by DEC, code-named the Alpha.

The Alpha

The development of the Alpha chip began in 1988 The new chip used 64 bit technology, allowed users to pack more complexity into their programs than existing 32 Bit technology chips. In addition, though designed to replace the VAX chip, the Alpha chip would have the capacity to support a variety of different operating systems, such as UNIX and Microsoft.

But the design forced all current VAX software to be rewritten to run on the new chip. If DEC wanted other operating systems, such a UNIX or Windows NT, to run on the chip, new versions of the operating systems would have to be developed specifically for the Alpha chip. The software would have to be specially tuned for the chips 64 Bit speed, which was a drawback.

Student Activity 1.1

Before reading the next section, answer the following question.

1. Find the difference between Intel & Motorola Microprocessor.

If your answer is correct, then proceed to the next section.

Top

PowerPC

A PowerPC is a microprocessor designed to meet a standard, which was jointly designed by Motorola, IBM, Apple. The PowerPC standard specifies a common instruction set architecture (ISA), allowing anyone

to design and fabricate PowerPC processors, which will run the same code. The PowerPC architecture is based on the IBM POWER architecture used in IBM's RS/6000 workstations. Currently IBM and Motorola are working on PowerPC chips.

The PowerPC architecture specifies both 32 bit and 64 bit data paths. Early implementations will be 32 bit, future higher performance implementation will be 64 bit. A PowerPC has 32 general purpose (integer) registers (32- or 64 bit) and 32 floating point (IEEE standard 64 bit) registers.

Application

For military and aerospace: weapons and communications systems, more than 150 programs have chosen PowerPC.

For industrial applications: industrial and processing control, transportation, and telecommunications.

Top

RISC / CISC Architecture

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determine the way that machine language programs are constructed. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than hundred and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes. The trend for computer hardware complexity was influenced by various factors, such as upgrading existing models to provide more customer applications, adding instructions that facilitate the translation from high-level language into machine language programs and striving to develop machines that move functions from software implementation into hardware implementation. A computer with a large number of instructions is classified as a Complex Instruction Set Computer, abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a Reduced Instruction Set Computer or RISC.

CISC Characteristics

The design of an instruction set for a computer must take into consideration not only machine language constraints, but also the requirements imposed on the use of high-level programming languages. The translation from high-level to machine language programs is done by means of a compiler program. One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance. The task of a compiler is to generate a sequence of machine instructions for each high-level language statement. The task is simplified if there are machine instructions that implement the statements directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

The major characteristics of CISC architecture are:

1. A large number of instructions-typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes-typically from 5 to 20 different modes

4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

A characteristic of RISC processors is their ability to execute one instruction per clock cycle. This is done by overlapping the fetch, decode and execute phases of two or three instructions by using a procedure referred to as pipelining. A load or store instruction may require two clock cycles because access to memory takes more register operations. Efficient pipelining, as well as a few other characteristics, are sometimes attributed to RISC, although they may exist in non-RISC architectures as well. Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs.

Student Activity 1.2

Answer the following questions.

1. Why microprocessor is called the heart of the computer?
2. What do you mean by 8 bit, 16 bit, 32 bit microprocessor. Why they are named so.
3. Differentiate RISC and CISC.
4. Find out which microprocessor you are using in your computer. Find out its features.
5. What do you mean by pipelining, decoding, instruction format, execution time.

Summary

- Microprocessor acts as a CPU in a microcomputer. It is present as a single IC chip in a microcomputer. Microprocessor is the heart of the machine.

- A MICROPROCESSOR is a multipurpose programmable logic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to those instructions and provides results as input
- The single chip 3rd generation microprocessor having 64-pins started with the introduction of 16-bit Intel 8086 in the year 1978.
- The Intel 8080, an 8-bit microprocessor, of nMOS technology was developed in the year 1974 which required only two additional devices to design a functional CPU.
- The single chip 32-bit microprocessor was introduced in the year 1981 by Intel as iAPX 432
- The Intel Corporation is the largest manufacturer of microchips in the world, in addition to being the leading provider of chips for PCs.
- Introduced in 1989 the 80486 did not feature any radically new processor technology.
- 68000 microprocessor is a 16 bit processor with an addressing space of 65536 locations, each of which holds a 64-bits word;
- An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed.

Self-Assessment Questions

Solved Exercise

I. State True or False

1. Example for 8 bit microprocessor is 8085
2. 8086/8088 is 32 bit microprocessor.
3. 16 bit microprocessor have special features like Memory segmentation and Parallel Processing.
4. 32 bit microprocessor example is Intel 80386
5. RISC processor has complex Instruction set.

II. Fill in the Blanks

1. Microprocessor acts, as _____ is microprocessor.
2. Microprocessor is _____ of computer.
3. ALU is _____
4. Storage device is called _____
5. _____ is a 4 bit microprocessor.

Answers

I. True or False

1. True

2. False
3. True
4. True
5. False

II. Fill in the Blanks

1. CPU
2. Heart
3. Arithmetic Logic Unit
4. Memory
5. 4004

Unsolved Exercise

I. True or False

1. Power PC find its application in military and aerospace.
2. RISC has relatively few instructions.
3. CISC has fixed length instruction formats.
4. RISC processor executes 1 instruction in one clock cycle.
5. Microprocessor is called the heart of the computer.

II. Fill in the Blanks

1. The memory size of 32 bit microprocessor is in_____
2. Features of Pentium processor are _____ and _____
3. Motorola 68000 is _____ microprocessor.
4. Expansion of VAX is _____
5. New 64 bit chip designed by DEC is _____

Detailed Questions

1. Discuss the evolution of microprocessor.
2. Differentiate between RISC and CISC Architecture.
3. Write a short note on 2^{nd} generation microprocessor.

Introduction
Internal Architecture
External System Bus Architecture
Execution Unit (EU) and Bus Interface Unit (BIU)
Memory and Input/Output Interface

Unit 2

Basic Microprocessor Architecture and Interface

Learning Objectives

After reading this unit you should appreciate the following:

- Internal Architecture
- External System Bus Architecture
- Memory and Input/Output Interface

Top

Introduction

Intel introduced its first 4-bit microprocessor 4004 in 1971 and 8-bit microprocessor 8008 in 1972. These microprocessors could not survive as general purpose microprocessors due to their design and performance limitations. Launching of the first general purpose 8-bit microprocessor 8080 in 1974 by Intel is considered to be the first major stepping stone towards the development of advanced microprocessors. The microprocessor 8085 followed 8080, with a few more features added to its architecture, which resulted in a functionally complete microprocessor. The main limitations of the 8-bit microprocessors were their low speed of execution, low memory addressing capability, limited number of general purpose registers and a less powerful instruction set . All these limitations of the 8-bit microprocessors tempted the designers to go for more powerful processors in terms of advanced architecture, more processing capability, larger memory addressing capability and a more powerful instruction set. The 8086 was a result of such developmental design efforts.

In the family of 16-bit microprocessors, Intel's 8086 was the first one launched in 1978. The introduction of the 16-bit processor was a result of the increasing demand for more and more powerful and high speed computational resources. 8086 microprocessor has a much more powerful instruction set along with the architectural developments which imparted substantial programming flexibility and improvement in speed over the 8-bit microprocessors.

The peripheral chips designed earlier for 8085 were compatible with microprocessor 8086 with slight or no modifications. Though there is a considerable difference between the memory addressing techniques of 8085 and 8086, the memory interfacing technique is similar, but includes the use of a few additional signals. The clock requirements are also different as compared to 8085, but the overall minimal system organization of 8086 is similar to that of a general 8-bit microprocessor. In this chapter, the architectures of 8086 and 8088 are discussed in adequate details along with the interfacing of the supporting chips with

them to form a minimum system. The system organization is also discussed in significant details for both the operating modes of 8086 and 8088, along with necessary timing diagrams.

Top

Internal Architecture

The architecture of 8086 provides a number of improvements over 8085 architecture. It supports a 16-bit ALU, a set of 16-bit registers and provides segmented memory addressing capability, a rich instruction set, powerful interrupt structure, fetched instruction queue for overlapped fetching and execution etc. The internal block diagram, shown in Figure 2.1, describes the overall organization of different units inside the chip.

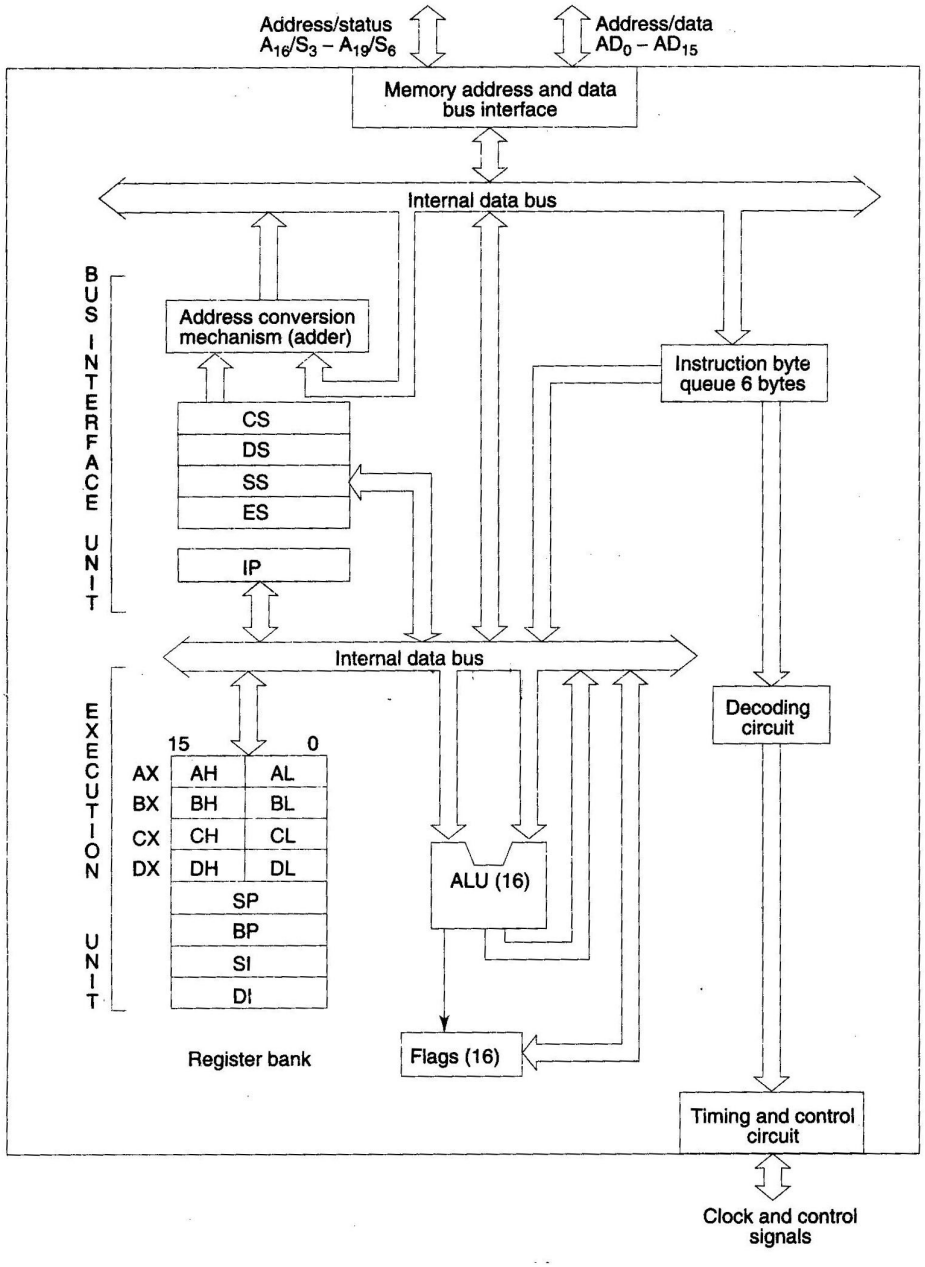


Figure 2.1: 8086 Architecture

The complete architecture of 8086 can be divided into two parts (a) Bus Interface Unit (BIU) and (b) Execution Unit (EU). The bus interface unit contains the circuit for physical address calculations and a predecoding instruction byte queue (6 bytes long). The bus interface unit makes the system bus signals available for external interfacing of the devices. In other words, this unit is responsible for establishing communications with external devices and peripherals including memory via the bus. As already stated, the 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated using segment and offset registers, each 16-bits long.

For generating a physical address from contents of these two registers, the content of a segment register also called as segment address is shifted left bit-wise four times and to this result, content of an offset register also called as offset address is added, to produce a 20-bit physical address. For example, if the segment address is 1005H and the offset is 5555H, then the physical address is calculated as below.

Segment address → 1005H
Offset address → 5555H
Segment address → 1005H → 0001 0000 0000 0101
Shifted by 4 bit positions → 0001 0000 0000 0101 0000
+
Offset address → 0101 0101 0101 0101
Physical address → 0001 0101 0101 1010 0101
1 5 5 A 5

Thus the segment addressed by the segment value 1005H can have offset values from 0000H to FFFFH within it, i.e. maximum 64K locations may be accommodated in the segment. Thus the segment register indicates the base address of a particular segment , while the offset indicates the distance of the required memory location in the segment from the base address. Since the offset is a 16-bit number, each segment can have a maximum of 64K locations. The bus interface unit has a separate adder to perform this procedure for obtaining a physical address while addressing memory. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP or an immediate 16-bit value, depending upon the addressing mode.

In case of 8085, once the opcode is fetched and decoded, the external bus remains free for some time, while the processor internally executes the instruction. This time slot is utilized in 8086 to achieve the overlapped fetch and execution cycles. While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction and arrange it in a queue called as predecoded instruction byte queue. It is a 6 bytes long, first-in first-out structure. The instructions from the queue are taken for decoding sequentially. Once a byte is decoded, the queue is rearranged by pushing it out and the queue status is checked for the possibility of the next opcode fetch cycle. While the opcode is fetched by the bus interface unit (BIU), the execution unit (EU) executes the previously decoded instruction concurrently. The BIU along with the execution unit (EU) thus forms a pipeline. The bus interface unit thus manages the complete interface of execution unit with memory and I/O devices, of course, under the control of the timing and control unit.

The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction opcode received from the queue, depending upon the information made available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in memory.

Memory Segmentation

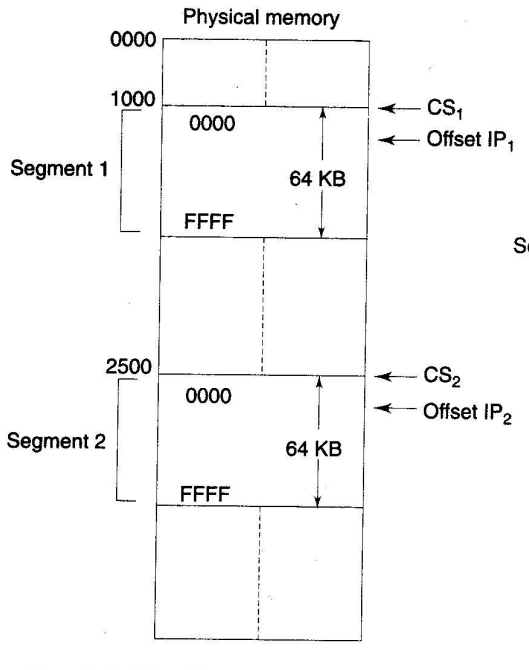
The memory in an 8086/8088 based system is organized as segmented memory. In this scheme, the complete physically available memory may be divided into a number of logical segments. Each segment is 64K bytes in size and is addressed by one of the segment registers. The 16-bit contents of the segment register actually point to the starting location of a particular segment. To address a specific memory location within a segment, we need an offset address. The offset address is also 16-bit long so that the maximum offset value can be FFFFH, and the maximum size of any segment is thus 64K locations.

To emphasize this segmented memory concept, we will consider an example of a housing colony containing say, 100 houses. The simplest method of numbering the houses will be just to assign the numbers from 1 to 100 to each house sequentially. Suppose, now, if one wants to find out house number 67, then he will start from house number 1 and go on till he finds the house, numbered 67. Consider another case where the 100 houses are arranged in the 10 x 10 (rows x columns) pattern. In this case, to find out house number 67, one will directly go to the 6th row and then to the 7th column. In the second scheme, the efforts required for finding the same house will be too less. This second scheme in our example is analogous to the segmented memory scheme, where the addresses are specified in terms of segment addresses analogous to rows and offset addresses analogous to columns.

The CPU 8086 is able to address 1Mbytes of physical memory. The complete 1Mbytes memory can be divided into 16 segments, each of 64Kbytes size. The addresses of the segments may be assigned as 0000H to F000H respectively. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH. In the above said case, the segments are called non-overlapping segments. The non-overlapping segments are shown in Figure 2.2(a). In some cases, however, the segments may be overlapping. Suppose a segment starts at a particular address and its maximum size can be 64Kbytes. But, if another segment starts before this 64Kbytes locations of the first segment, the two segments are said to be overlapping segments. The area of memory from the start of the second segment to the possible end of the first segment is called as overlapped segment area. Figure 2.2(b) explains the phenomenon more clearly. The locations lying in the overlapped area may be addressed by the same physical address generated from two different sets of segment and offset addresses. The main advantages of the segmented memory scheme are as follows:

1. Allows the memory capacity to be 1Mbytes although the actual addresses to be handled are of 16-bit size.
2. Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection.
3. Permits a program and/or its data to be put into different areas of memory each time program is executed, i.e. provision for relocation may be done.

In the Overlapped Area Locations Physical Address = CS + IF = CS + IF + indicates the procedure of physical address formation.



Fieure 2.2(a): Non-overlappoine Segments

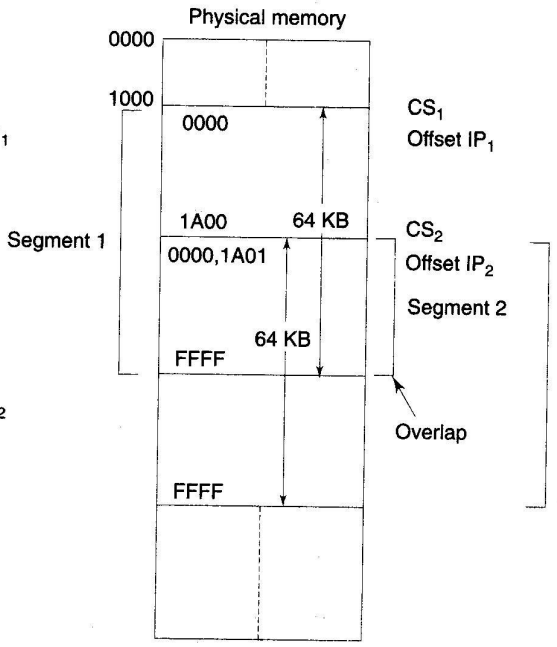
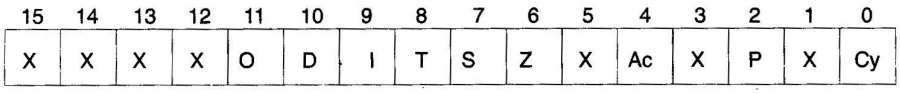


Figure 2.2(b): Overlapping Segments

Flag Register

8086 has a 16-bit flag register which is divided into two parts, viz. (a) condition code or status flags and (b) machine control flags. The condition code flag register is the lower byte of the 16-bit flag register along with the overflow flag. The condition code flag register is identical to 8085 flag register, with an additional overflow flag, which is not present in 8085. This part of the flag register of 8086 reflects the results of the operations performed by ALU. The control flag register is the higher byte of the flag register of 8086. It contains three flags, viz. direction flag (D), interrupt flag (1) and trap flag (T). The complete bit configuration of 8086 flag register is shown in Figure 2.3.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	Cy

O — Overflow flag
D — Direction flag
I — Interrupt flag
T — Trap flag
S — Sign flag
Z — Zero flag
Ac — Auxiliary carry flag
P — Parity flag
Cy — Carry flag
X — Not used

Figure 2.3: Flag Register of 8086

The description of each flag bit is as follows:

S-Sign Flag: This flag is set, when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

Z-Zero Flag: This flag is set, if the result of the computation or comparison performed by the previous instruction/instructions is zero.

P-Parity Flag: This flag is set to 1, if the lower byte of the result contains even number of 1's.

C-Carry Flag: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction. For example, when two numbers are added, a carry may be generated out of the most significant bit position. The carry flag, in this case, will be set to '1'. In case, no carry is generated, it will be '0'. Some other instructions also affect or use this flag and will be discussed later in this text.

T-Trap Flag: If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

1-lnterrupt Flag: If this flag is set, the maskable interrupts are recognised by the CPU, otherwise, they are ignored.

D-Direction Flag: This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e. autoincrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e. autodecrementing mode. We will describe string manipulations later in chapter 2 in more details.

Ac-Auxiliary Carry Flag: This is set, if there is a carry from the lowest nibble, i.e. bit three, during addition or borrow for the lowest nibble, i.e. bit three, during subtraction.

O-Overflow Flag: This flag is set, if an overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register. For example, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e. the result is of more than 7-bits in size in case of 8-bit signed operations and more than 15-bits in size in case of 16-bit signed operations, then the overflow flag will be set.

Student Activity 2.1

Before reading the next section, answer the following questions.

1. Draw and discuss the internal block diagram of 8086.
2. What do you mean by pipelined architecture? How is it implemented in 8086?
3. Explain the concept of segmented memory? What are its advantages?
4. Explain the physical address formation in 8086.
5. Draw the register organization of 8086 and explain typical applications of each register.
6. Draw and discuss flag register of 8086 in brief.

If your answers is correct, then proceed to the next section.

Pin Descriptions of 8086

The microprocessor 8086 is a 16-bit CPU available in three clock rates, i.e. 5, 8 and 10 MHz, packaged in a 40 pin CERDIP or plastic package. The 8086 operates in single processor or multiprocessor configurations

to achieve high performance. The pin configuration is shown in Figure 2.4. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode) configuration.

The 8086 signals can be categorised in three groups. The first are the signals having common functions in minimum as well as maximum mode, the second are the signals which have special functions for minimum mode and the third are the signals having special functions for maximum mode.

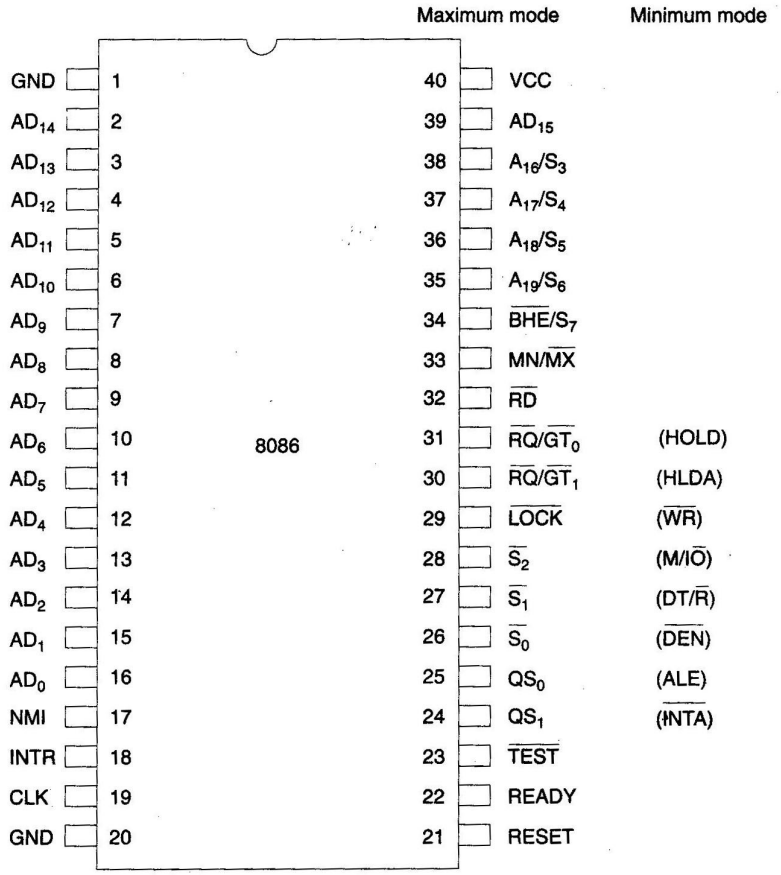


Figure 2.4: Pin Configuration of 8086

The following signal descriptions are common for both the minimum and maximum mode

AD_{15} –AD_0: These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T_1 state, while the data is available on the data bus during T_2,T_3,T_w and T_4. Here T_1,T_2,T_3,T_4 and T_w are the clock states of a machine cycle. T_w is a wait state. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

A_{19}/S_6,A_{18}/S_5,A_{17}/S_4,A_{16}/S_3: These are the time multiplexed address and status lines. During T_1, these are the most significant address lines for memory operations. During l/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T_2,T_3,T_w and T_4. The status of the interrupt enable flag bit(displayed on S_5) is updated at the beginning of each clock cycle. The S_4 and S_3 combinedly indicate which segment register is presently being used for memory accesses as shown in Table 2.1 These lines float to tri-state off (tristated) during the local bus hold acknowledge. The status line S_6 is

always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

Table 2.1: Bus High Enable/status

S_4	S_3	Indications
0	0	Alternate Data
0	1	Stack
1	0	Code or none
1	1	Data

BHE/S_7-Bus High Enable/Status: The bus high enable signal is used to indicate the transfer of data over the higher order (D_{15}-D_8) data bus as shown in Table 2.2. It goes low for the data transfers over D_{15}-D_8, and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T_1 for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on the higher byte of the data bus. The status information is available during T_2, T_3 and T_4. The signal is active low and is tristated during 'hold'. It is low during T_1 for the first pulse of the interrupt acknowledges cycle.

Table 2.2

BHE	A_0	Indication
0	0	Whole word
0	1	Upper byte from or to odd address.
1	0	Lover byte from or to even address.
1	1	None

RD-Read: Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. RD is active low and shows the state for T_2, T_3, T_w of any read cycle. The signal remains tristated during the 'hold acknowledge'.

READY: This is the acknowledgement from the slow devices or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The signal is active high.

INTR-Interrupt Request: This is a level triggered input. This is sampled during the last clock cycle of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resetting the interrupt enable flag. This signal is active high and internally synchronized.

TEST: This input is examined by a 'WAIT' instruction. If the TEST input goes low, execution will continue, else, the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

NMI-Non-maskable Interrupt: This is an edge-triggered input which causes a Type2 interrupt. The NMI is not maskable internally by software. A transition from low to high initiates the interrupt response at the end of the current instruction. This input is internally synchronized.

RESET This input causes the processor to terminate the current activity and start execution from FFFFOH. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronised.

CLK-Clock Input: The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle. The range of frequency for different 8086 versions is from 5MHz to l0MHz.

Vcc +5V: power supply for the operation of the internal circuit.

GND: ground for the internal circuit.

MN/MX: The logic level at this pin decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode.

The following pin functions are for the minimum mode operation of 8086.

M/I/O -Memory/IO: This is a status line logically equivalent to S_2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous T_4 and remains active till final T_4 of the current cycle. It is tristated during local bus "hold acknowledge".

INTA-interrupt Acknowledge: This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during T_2, T_3 and T_w of each interrupt acknowledge cycle.

ALE-Address Latch Enable: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.

DT/R -Data Transmit/Receive: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to S_1 in maximum mode. Its timing is the same as M/I/O. This is tristated during 'hold acknowledge'.

DEN-Data Enable: This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T_2 until the middle of T_4 .

DEN is tristated during 'hold acknowledge' cycle.

HOLD, HLDA-Hold/Hold Acknowledge: When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus (instruction) cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal . HOLD is an asynchronous input, and it should be externally synchronized.

If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T_4 provided:

1. The request occurs on or before T_2 state of the current cycle.
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address).
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A Lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode.

The following pin functions are applicable for maximum mode operation of 8086.

S_2, S_1, S_0-Status Lines: These are the status lines which reflect the type of operation, being carried out by the processor. These become active during T_4 of the previous cycle and remain active during T_1 and T_2 of the current bus cycle. The status lines return to passive state during T_3 of the current bus cycle so that they may again become active for the next bus cycle during T_4. Any change in these lines during T_3 indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in Table 2.3.

Table 2.3

S_2	S_1	S_0	Indication
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Wrtie I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

LOCK This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during "hold acknowledge". When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

QS_1, QS_0-Queue Status: These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table 2.4.

Table 2.4

QS_1	QS_0	Indication
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus even the largest (6-bytes) instruction can be prefetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as instruction pipelining.

At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even. The first byte is a complete opcode in case of some instructions (one byte opcode instruction) and it is a part of opcode, in case of other instructions (two byte long opcode instructions), the remaining part of opcode may lie in the second byte. But invariably the first byte of an instruction is an opcode. These opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied. The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is single opcode byte , the next bytes are treated as data bytes depending upon the decoded instruction length, otherwise, the next byte in the queue is treated as the second byte of the instruction opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.

The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture, there are two separate units, namely, execution unit and bus interface unit, while the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status. Figure 2.5 explains the queue operation.

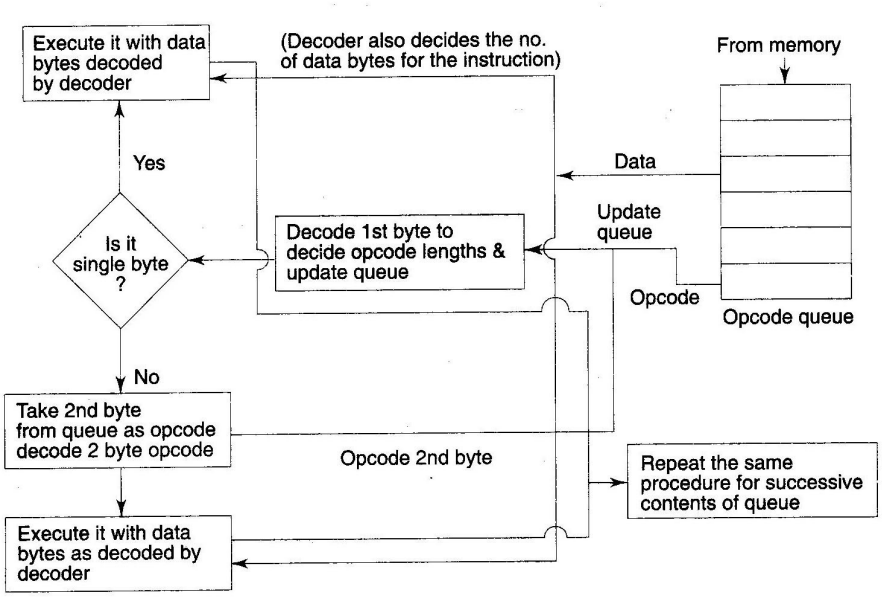


Figure 2.5: The Queue Operation

RQ/CT_0, RQ/G_1-Request/Grant: These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with RQ/GT() having higher priority than RQ/GT_1. RQ/GT pins have internal pull-up resistors and may be left unconnected. The request grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.
2. During 74 (current) or T, (next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at next clock cycle. The CPU's bus interface unit is likely to be disconnected from the local bus of the system.
3. A one clock wide pulse from the another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle.

Thus each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus requests those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as discussed in case of HOLD, and HLDA in minimum mode.

Until now, we have described the architecture and pin configuration of 8086. In the next section, we will study some operational features of 8086 based systems.

Student Activity 2.2

Before reading the next section, answer the following questions.

1. Explain the function of the following signals of 8086.

   (i) ALE (ii) DT/R (iii) DBN (iv) LOCK

(v) TEST (vi) MN/MX (vii) BHE (viii) M/IO

(ix) RQ/GT (x) QS_0 (xi) READY (xii) NMl

(xiii) lNTR (xiv) HOLD (xv) HLDA

2. Explain the function of opcode prefetch queue in 8086,
3. How does 8086 differentiate between an opcode and instruction data?
4. Explain the physical memory organization in an 8086 system.

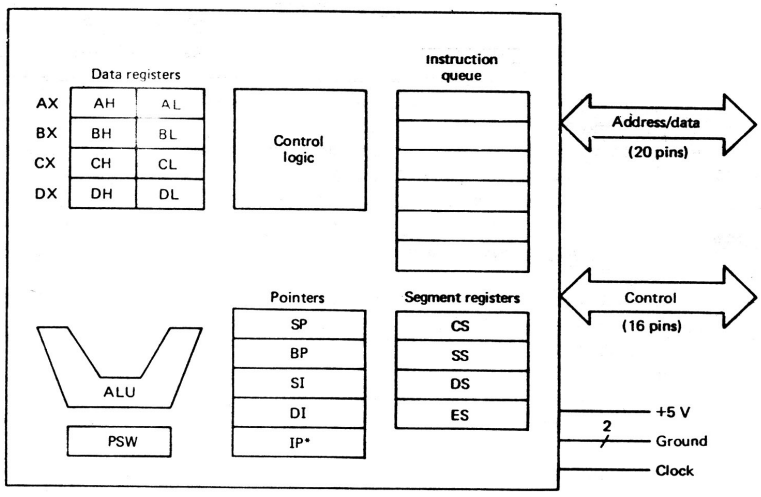
If your answers is correct, then proceed to the next section.

Top

External System Bus Architecture

It is a 16 bit processor with 40 pins. It has 20 address pins and out of which 16are used as data pins. This concept of using same pins for both address and data is called Multiplexing. It has 16 control signals. It can access a memory of 1 MB.(2^{20}).

It has 14 registers which are 16 bits wide. There are a set of arithmetic registers, set of pointers(Base and Index registers),set of segment registers. It has program status word(PSW) or Flag register and a instruction pointer.



Instruction queue: It can queue 6 bytes at a time.

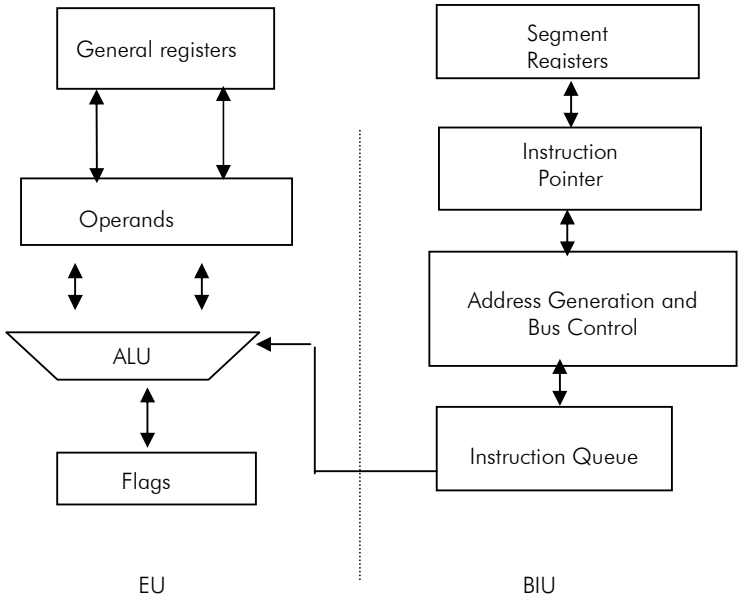
Figure 2.6: 8086's Internal Configuration

Top

Execution Unit (EU) and Bus Interface Unit (BIU)

8086 consist of two processors called EU and BIU. Two Processors can work in parallel. This improves speed of execution. BIU fetches instruction and place them in instruction queue.

Execution unit decodes and execute instruction. When EU is executing an instruction BIU can fetch the next instruction.



Instruction Set

It has a large instruction set which operates on bits, 16 bit or 32 bit word. They have variety of instructions for arithmetic operation , data movement, logical operation ,shift, rotate operation , string manipulation etc.

Top

Memory and Input/Output Interface

General Bus Operation

The 8086 has a combined address and data bus commonly referred to as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed using a few latches and transceivers, whenever required. In the following text, we will discuss a general bus operation cycle.

Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T_1, T_2,T_3 and T_4· The address is transmitted by the processor during T_1. It is present on the bus only for one cycle. During T_2, i.e. the next cycle, the bus is tristated for changing the direction of bus for the following data read cycle. The data transfer takes place during T_3 and T_4. In case, an addressed device is slow and shows 'NOT READY status the wait states T_w are inserted between T_3 and T_4. These clock states during wait period are called idle states (T_i), wait states (T_w) or inactive states. The processor uses these cycles for internal housekeeping. The address latch enable (ALE) signal is emitted during T, by the processor (minimum mode) or the bus controller (maximum mode) depending upon the status of the MN/MX input. The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines S_0, S_1 and S_2 are used to indicate the type of operation as discussed in the pin description of this unit. Status bits BHE/s_7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T_1 while the status bits S_3 to S_7 are valid during T_2 through T_4. The Figure 2.7 shows a general bus operation cycle of 8086.

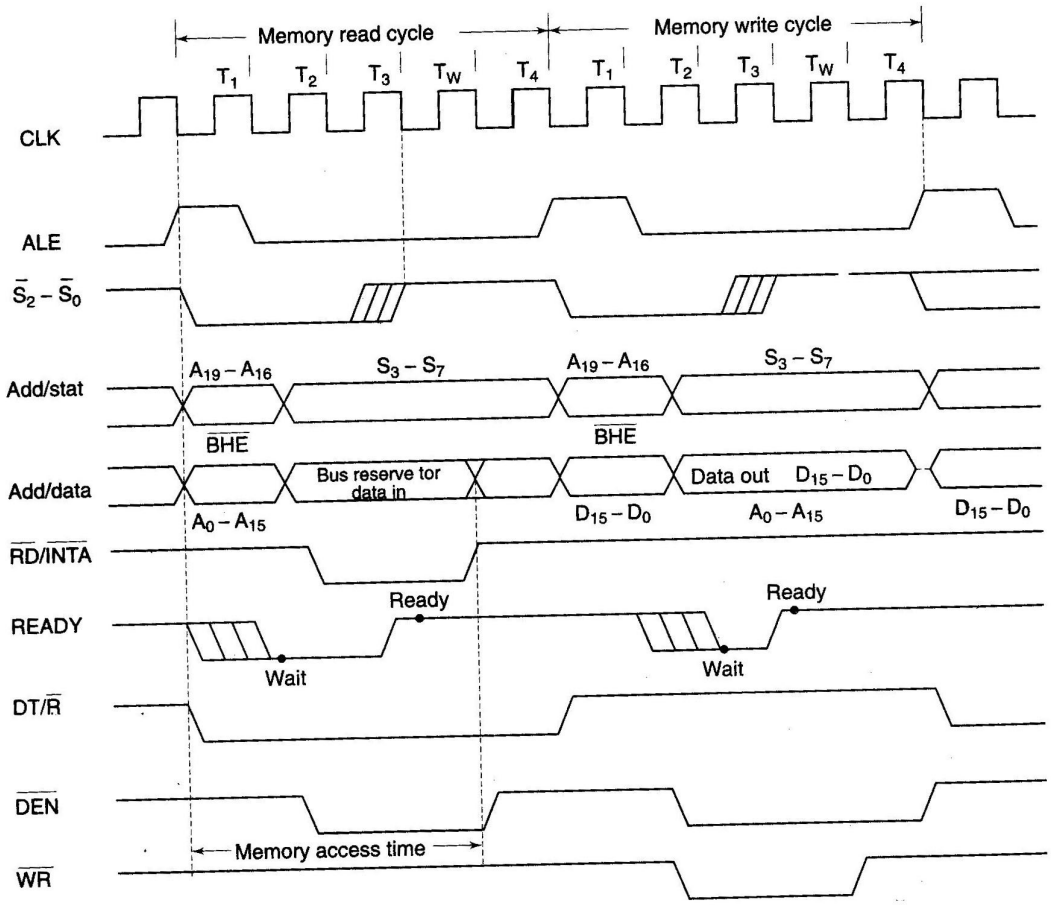


Figure 2.7: General Bus Operation Cycle in Maximum Mode

Minimum Mode 8086 System and Timings

In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1. In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system. The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086. Tran receivers are the bi-directional buffers and some times they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, namely, DEN and DT/R. The DEN signal indicates that the valid data is available on the data bus, while DT/R indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users program storage. A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices. The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system. The clock generator also synchronizes some

external signals with the system clock. The general system organization is shown in Figure 2.8. Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.

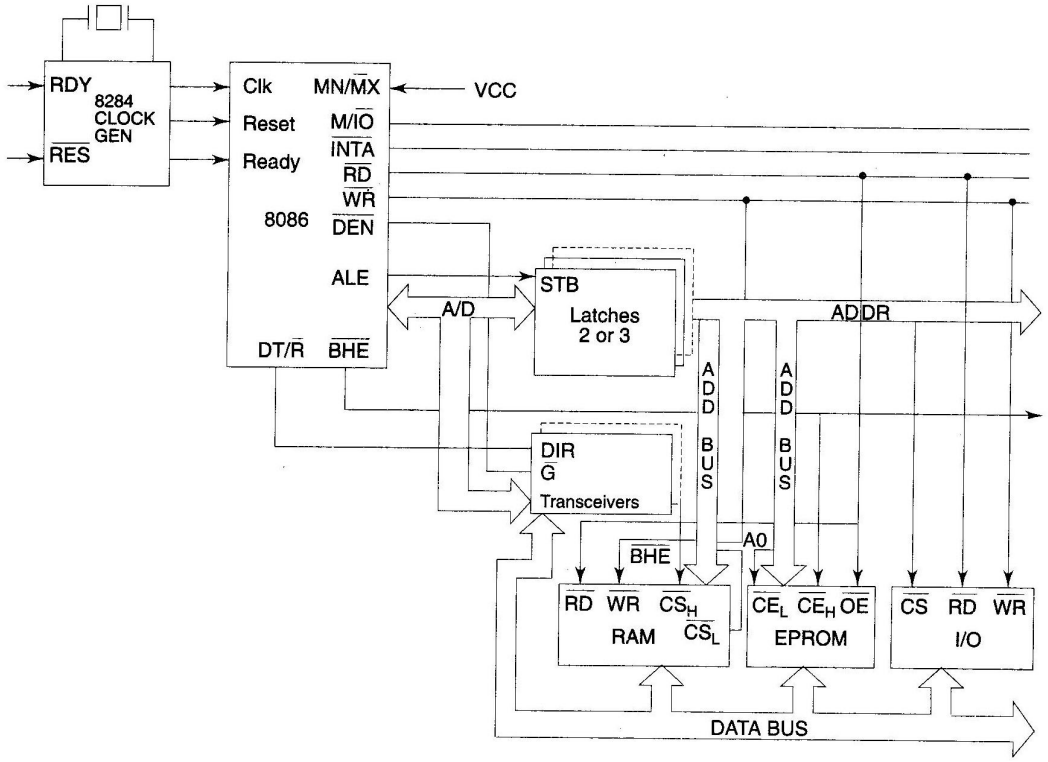


Figure 2.8: Minimum Mode 8086 System

The working of the minimum mode configuration system can be better described in terms of The timing diagrams rather than qualitatively describing the operations. The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts) the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.

The read cycle begins in T_1 with the assertion of the address latch enable (ALE) signal and also M/IO signal. During the negative going edge of this signal, the valid address is latched on the local bus. The BHE and A0 signals address low, high or both bytes. From T_1 to T_4, the M/IO signal indicates a memory or I/O operation. At T_2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T_2. The read (RD) signal causes the addressed device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/O operation. In Tg, after sending the address in T_2, the processor sends the data to be written to the addressed location. The data remains on the bus untill middle of T_4 state. The WR becomes active at the beginning of T_2 (unlike RD is somewhat delayed in T_2 to provide time for floating).

The BHE and AO signals are used to select the proper byte or bytes of memory or I/O word to be read or written as already discussed in the signal description section of this chapter.

The M/IO, RD and WR signals indicate the types of data transfer as specified in Table 2.5.

Table 2.5

M/IO	RD	WR	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory read

Figure 2.9(a): Shows the read cycle while the Figure 2.9(b) shows the write cycle.

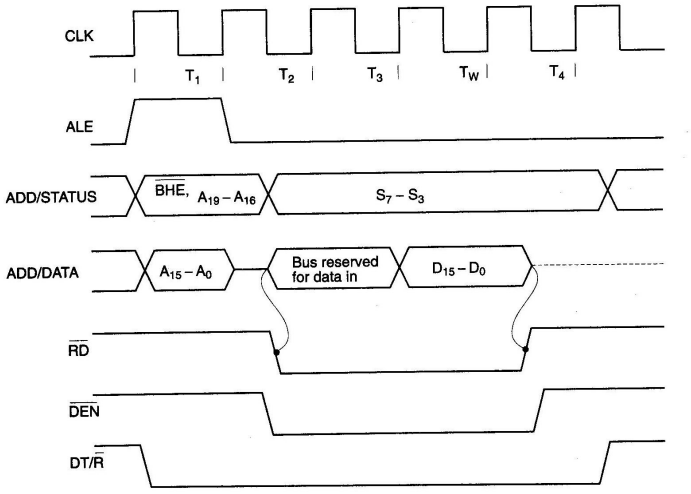


Figure 2.9(a): Read Cyde Timing Diagram for Minimum Mode

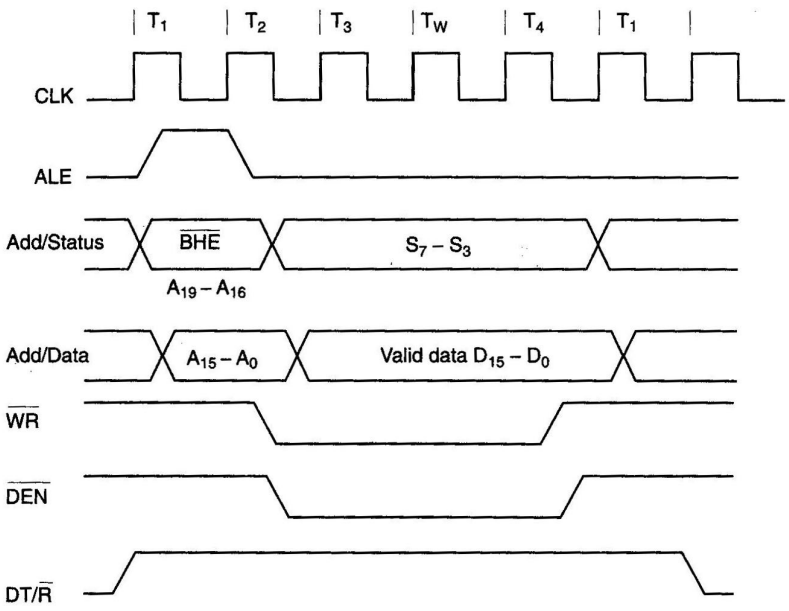


Figure 2.9(b): Write Cycle Timing Diagram for Minimum Operation

Hold Response Sequence

The HOLD pin is checked at leading edge of each clock pulse. If it is received active by the processor before T_4 of the previous cycle or during T_1 state of the current cycle, the CPU activates HLDA in the next clock cycle and for the succeeding bus cycles, the bus will be given to another requesting master. The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock, as shown in Figure 2.9(c). The other conditions have already been discussed in the signal description section for the HOLD and HLDA signals.

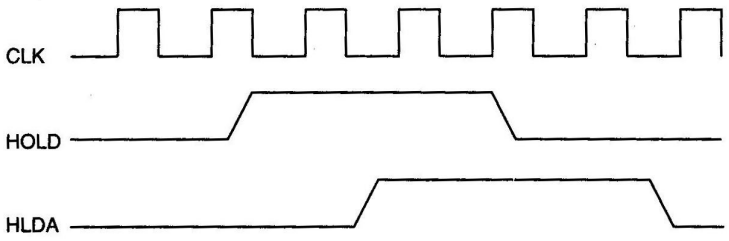


Figure 2.9(c): Bus Request and Bus Grant Timings in Minimum Mode System

Student Activity 2.3

Before reading the next section, answer the following questions.

1. Draw and discuss and write cycle timing diagrams of 8086 minimum mode.
2. Draw and discuss the read and write cycle timing diagram of 8086 in maximum mode.
3. Draw and discuss a typical minimum mode 8086 system.
4. Draw and discuss a typical maximum mode 8086 system. What is the use of a bus controller in maximum mode?

If your answers is correct, then proceed to the next section.

The Processor 8088

The launching of the processor 8086 is seen as a remarkable step in the development of high speed computing machines. Before the introduction of 8086 most of the circuits required for the different applications in computing and industrial control fields were already designed around the 8-bit processor 8085. The 8086 imparted tremendous flexibility in the programming as compared to 8085. So naturally, after the introduction of 8086, there was a search for a microprocessor chip which has the programming flexibility like 8086 and the external interface like 8085, so that all the existing circuits built around 8085 can work as before, with this new chip. The chip 8088 was a result of this demand. The microprocessor 8088 has all the programming facilities that 8086 has, along with some hardware features of 8086, like 1Mbyte memory addressing capability, operating modes (MN/MX), interrupt structure etc. However 8088, unlike 8086, has 8-bit data bus. This feature of 8088 makes the circuits, designed around 8085, compatible with 8088, with little or no modification.

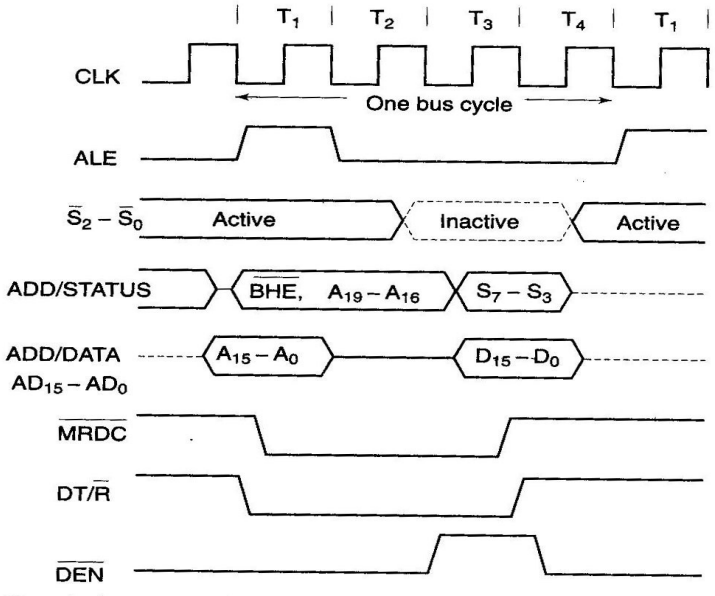


Figure 2.10(a): Memory Read Timing in Maximum Mode

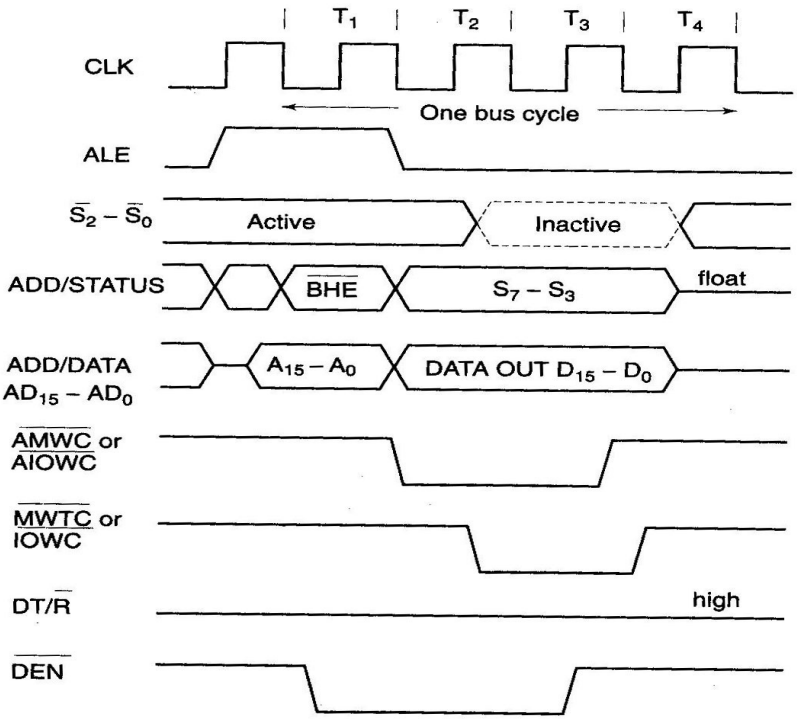


Figure 2.10(b): Memory Write Timing in Maximum Mode

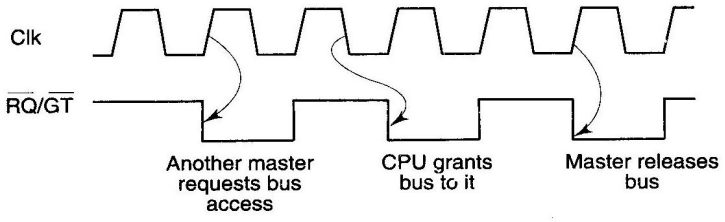


Figure 2.10(c): RQ/GT Timings in Maximum Mode

All the peripheral interfacing schemes with 8088 are the same as those for the 8-bit processors. The memory and I/O addressing schemes are now exactly similar to 8085 schemes except for the increased memory (1Mbyte) and I/O (64Kbyte) capabilities. The architecture shows the developments in 8088 over 8086. Number of abilities and limitations of 8088 are same as 8086. In this section, we will discuss those properties of 8088 which are different from that of 8086 in some respects.

Architecture of 8088

The register set of 8088 is exactly the same as in to 8086. The architecture of 8088 is also similar to 8086 except for two changes; a) 8088 has 4-byte instruction queue and b) 8088 has 8-bit data bus. The function of each block is the same as in 8086. Figure 2.11 shows the 8088 architecture.

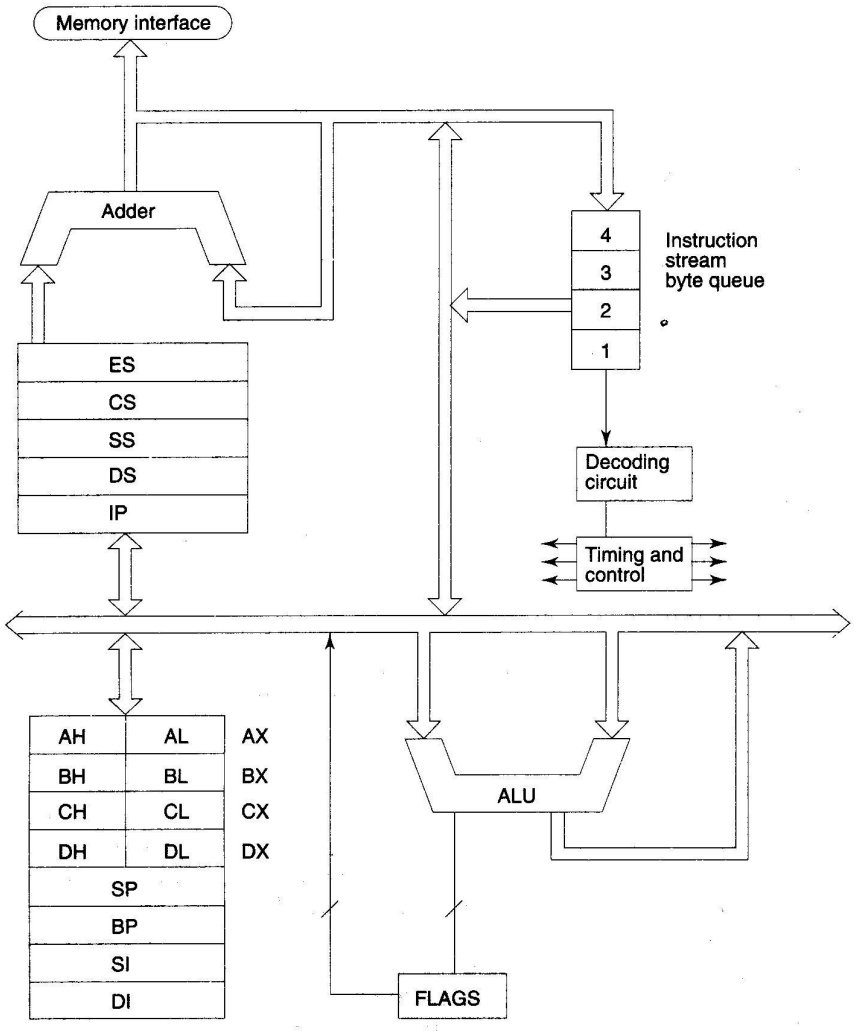


Figure 2.11: Architecture of 8088

The addressing capability of 8088 is 1Mbyte, therefore, it needs 20 address bits, i.e. 20 addressing lines. While handling this 20-bit address, the segmented memory scheme is used and the complete physical address forming procedure is the same as explained in case of 8086. The memory organization and addressing methods of 8088. While physically interfacing memory to 8088, there is nothing like an even address bank or odd address bank. The complete memory is homogeneously addressed as a bank of 1Mbyte memory locations using the segmented memory scheme. This change in hardware is completely transparent to software. As a result of the modified data bus, the 8088 can access only a byte at a time. This fact reduces the speed of operation of 8088 as compared to 8086, but the 8088 can process the 16-bit data internally. On account of this change in bus structure, the 8088 has slightly different timing diagrams than 8086.

The pin diagram of 8088 is shown in Figure 2.12. Most of the 8088 pins and their functions are exactly similar to the corresponding pins of 8086. Hence the pins that have different functions or timings are discussed in this section. Amongst them are the pins that have a common function in minimum and maximum mode.

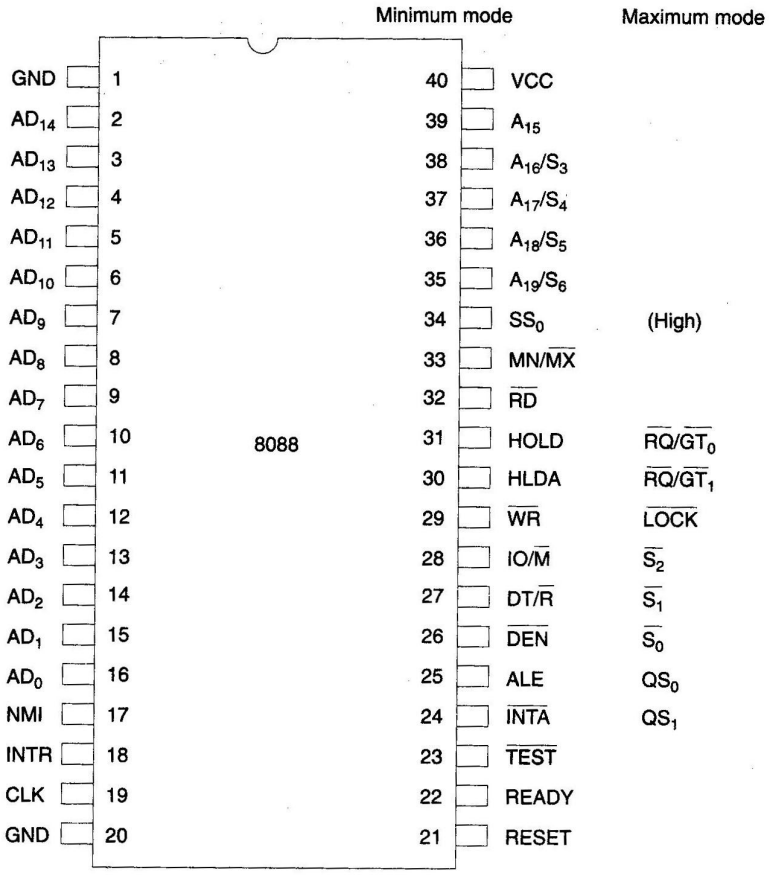


Figure 2.12: Pin Diagram of 8088

AD_7-AD_0 (Address/Data): These lines constitute the address/data time multiplexed bus. During T_1 the bus is used for conducting addresses and during T_2, T_3, T_w and T_4 states these lines are used for conducting data. These are tristate during 'hold acknowledge' and 'interrupt acknowledge' cycles.

A_{15}-A_8 (Address Bus): These lines provide the address bits A_8 to A_{15} in the entire bus cycle. These need not be latched for obtaining a stable valid address. These are active high and are tristated during the 'acknowledge' cycles. Note that, as the 8088 data bus is only of 8 bits, there is no need of the BHE signal.

SS_0: A new pin SS_0 is introduced in 8088 instead of BHE pin in 8086. In minimum mode, the pin 880 is logically equivalent to the S_8 in maximum mode.

IO/M: This pin is similar to M/IO pin of 8086, but it offers an 8085 compatible, memory/IO bus interface.

The signals SS_0, DT/R, IO/M can be decoded to interpret the activities of the microprocessor as given in Table 2.6.

Table 2.6

IO/M	DT/R	SS_0	Operation/Interpretation

1	0	0	Interrupt Acknowledge
1	0	1	Read I/O port
1	1	0	Write I/O port
1	1	1	HALT
0	0	0	Code Access
0	0	1	Read memory
0	1	0	Wrtie memory
0	1	1	Passive

In the maximum mode, the pin 880 is permanently high. The functions and timings of other pins of 8088 are exactly similar to 8086. Due to the difference in the bus structure, the timing diagrams are some what different. The minimum and maximum mode systems are also similar to the respective 8086 systems. The 8088 systems require only one data buffer due to the 8-bit data bus. The minimum and maximum mode systems of 8088 are shown in Figure 2.13(a) and (b) respectively.

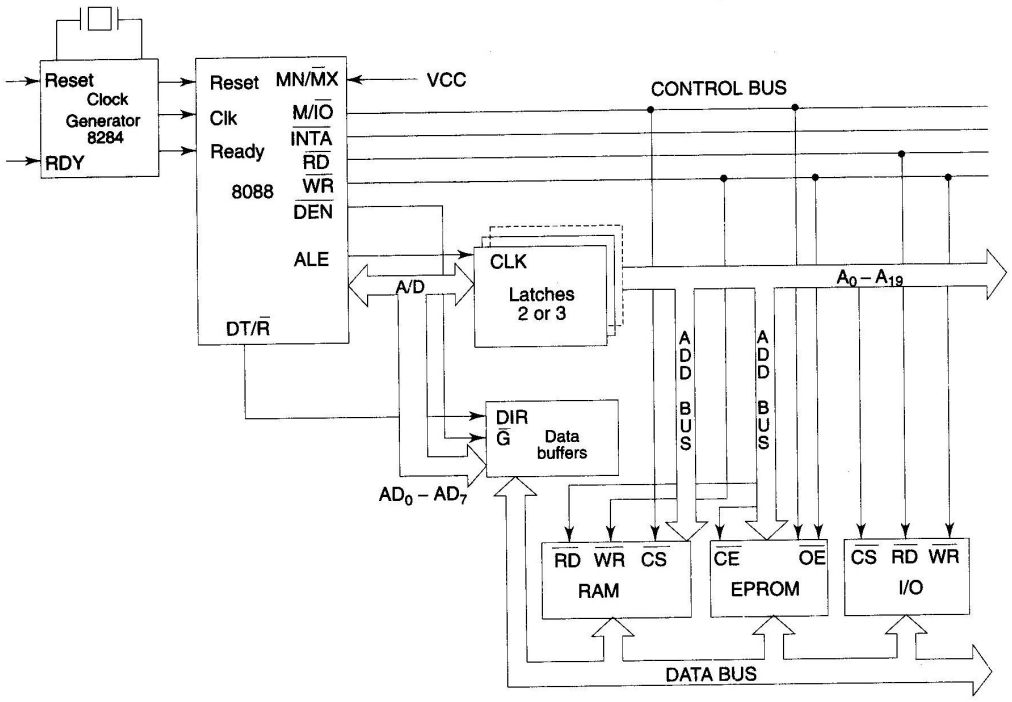


Figure 2.13(a): Minimum Mode 8088 System

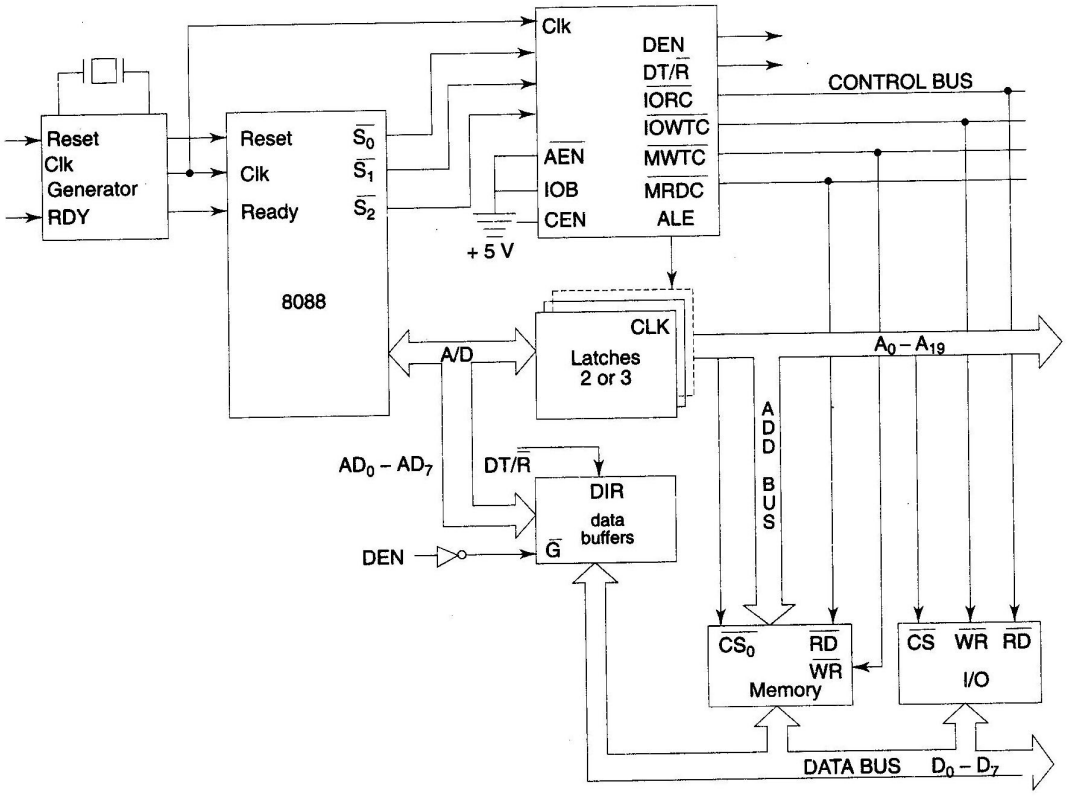


Figure 2.13(b): Maximum Mode 8088 System

General 8088 System Timing Diagram

The 8088 address/data bus is divided in three parts (a) the lower 8 address/data bits, (b) the middle 8 address bits, and (c) the upper 4 address/status bits. The lower 8 lines are time multiplexed for address and data. The upper 4 lines are time multiplexed for address and status. Each of the bus cycles contains T_1, T_2, T_3, T_w and T_4 states. The ALE signal goes high for one clock cycle in T_1. The trailing edge of ALE is used to latch the valid addresses available on the multiplexed lines. They remain valid on the bus for the next cycle (T_2). The middle 8 address bits are always present on the bus throughout the bus cycle. The lower order address bus is tristated after T_2 to change its direction for read data operation. The actual data transfer takes place during T_3 and T_4. Hence the data lines are valid in T_3 or T_4. The multiplexed bus is again tristated to be ready for the next bus cycle. The status lines are valid over the multiplexed address/status bus for T_2, T_3 and T_4 clock cycles.

In case of write cycle, the timing diagram is similar to the read cycle except for the validity of data. In write cycle, the data bits are available on the bus for T_2, T_3, T_w, and T_4. At the end of 14, the bus is tristated. The other signals RD, WR, INTA, DT/R, DEN and READY are similar to the 8086 timing diagram. Figure 2.14 shows the details of read and write bus cycles of 8088.

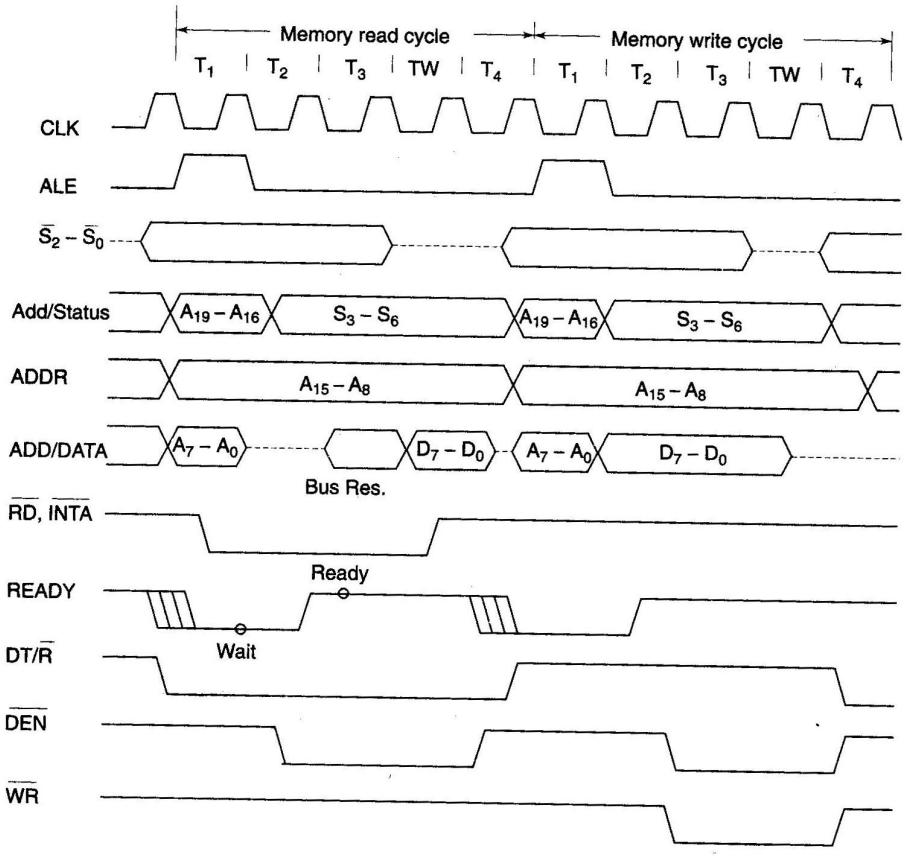


Figure 2.14: Read and Write Cycle Timing Diagram of 8088

Comparison between 8086 and 8088

The 8088, with an 8-bit external data bus, has been designed for internal 16-bit processing capability. Nearly all the internal functions of 8088 are identical to 8086. The 8088 uses the external bus in the same way as 8086, but only 8 bits of external data are accessed at a time. While fetching or writing the 16-bit data, the task is performed in two consecutive bus cycles. As far as the software is concerned, the chips are identical, except in case of timings. The 8088, thus may take more time for execution of a particular task as compared to 8086.

All the changes in 8088 over 8086 are directly or indirectly related to the 8-bit, 8085 compatible data and control bus interface.

1. The predecoded code queue length is reduced to 4 bytes in 8088, whereas the 8086 queue contains 6 bytes. This was done to avoid the unnecessary prefetch operations and optimize the use of the bus by BIU while prefetching the instructions.

2. The 8088 bus interface unit will fetch a byte from memory to load the queue each time, if at least 1 byte is free. In case of 8086, at least 2 bytes should be free for the next fetch operation.

3. The overall execution time of the instructions in 8088 is affected by the 8-bit external data bus. All the 16-bit operations now require additional 4 clock cycles. The CPU speed is also limited by the speed of instruction fetches.

The pin assignments of both the CPUs are nearly identical, however, they have the following functional changes.

1. A_8-A_{15} already latched, all time valid address bus.
2. BHE has no meaning as the data bus is of 8-bits only.
3. SS_0 provides the S_0 status information in minimum mode.
4. IO/M has been inverted to be compatible with 8085 bus structure

Student Activity 2.3

Answer the following questions.

1. Bring out the architectural and signal differences between 8086 and 8088.
2. What may be the reason for developing an externally eight-bit processor like 8088 after the 8086, when a 16-bt processor had already been introduced?
3. Explain the signal SS_0 of 8088.
4. Compare the bus interface of 8085 with 8088.
5. Draw and discuss a typical minimum mode 8088 system.
6. Draw discuss a general 8088 system timing diagram.

Summary

In this chapter, we have presented the internal architecture and signal descriptions of 8086. The functional details of the architecture, like, register set, flags and segmented memory organization are also discussed in significant details. Further, general bus cycle operations have been described with the help of timing diagrams. Then minimal 8086 systems have been presented for the minimum and maximum modes of operation. A software compatible processor-8088 has been discussed in the light of the modifications in it over 8086. To conclude with, the basic bus cycle operations and the timing diagrams of 8088 were discussed along with its comparison with 8086. This chapter is aimed at elaborating the architectural and functional concepts of the processors 8086 and 8088. The instruction set and programming techniques have been discussed in the unit 5.

Self-assessment Questions

Solved Exercise

I. True or False

1. 8086 has four segments.
2. The physical address of 8086 has 16 bits.
3. The size of flag register is 16.

4. 16 bit flag register has 9 flags.
5. 8086 has 40 pins

II. Fill in the Blanks

1. _____ is needed to synchronic the activity within microprocessor and bus control logic
2. Address bus is used for _____
3. Data bus is used for _____
4. _____ is directional.
5. _____ is an interface to CPU.

Answers

I. True of False

1. True
2. False
3. True
4. True
5. True

II. Fill in the Blanks

1. Timing circuitry or clock.
2. Transmitting address of particular device, which CPU wants to access.
3. Sending and receiving data between CPU and other devices.
4. Data bus
5. Bus control logic

Unsolved Exercise

I. True of False

1. 8086 has 14 registers.
2. $A_{16} - A_{20}$ Pins are time multiplexed.
3. 8088 is architecturally similar to 8086.
4. 8086 has 4 data registers.
5. 8086 consists of two processor working in parallel.

II. Fill in the Blanks

1. _____ is used for transmitting & receiving control signals between processor and various devices.
2. _____ performs all arithmetic, logic functions.
3. 8088 is _____bit microprocessor.
4. 8086 can operate in _____ways.
5. The queue size of 8086 is _____Bytes

Detailed Questions

1. What is a Bus?
2. What is memory module?
3. What is I/O subsystem?
4. What is an interface?
5. What is a memory interface?
6. What is I/O interface?

Register Organization of 8086
Memory Addressing and Instruction Formats
Memory Interfacing
Cache Memory
Cache Controller

Unit 3

Programming Mode

Learning Objectives

After reading this unit you should appreciate the following:

- Register Organization of 8086
- Memory Addressing and Instruction Formats
- Memory Interfacing
- Cache Memory and Cache Controllers

Top

Register Organization of 8086

8086 has a powerful set of registers containing general purpose and special purpose registers. All the registers of 8086 are 16-bit registers. The general purpose registers, can be used as either 8-bit registers or 16-bit registers. The general purpose registers are either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. We will categorize the register set into four groups, as follows:

General Data Registers

Figure 3.1 shows the register organization of 8086. The registers AX, BX, CX and DX are the general purpose 16-bit registers. AX is used as 16-bit accumulator, with the lower 8-bits of AX designated as AL and higher 8-bits as AH. AL can be used as an 8-bit accumulator for 8-bit operations. This is the most important general purpose register having multiple functions, which will be discussed later.

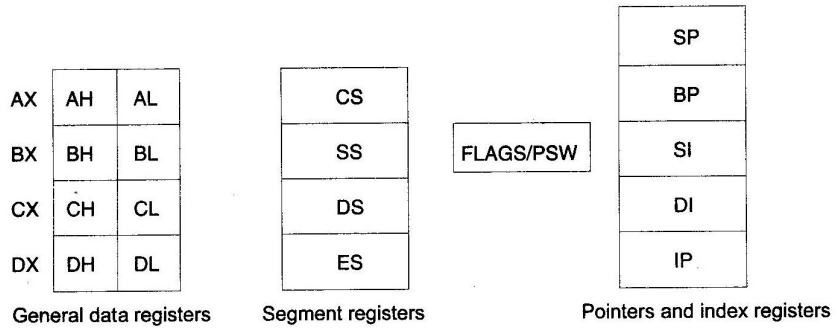


Figure 3.1: Register organisation of 8086

Usually the letters L and H specify the lower and higher bytes of a particular register. For example, CH means the higher 8-bits of the CX register and CL means the lower 8-bits of the CX register. The letter X is used to specify the complete 16-bit register. The register CX is also used as a default counter in case of string and loop instructions. The register BX is used as offset storage for forming physical addresses in case of certain addressing modes. DX register is a general-purpose register which may be used as an implicit operand or destination in case of a few instructions. The detailed uses of these registers will be more clear when we discuss the addressing modes and the instruction set of 8086.

Segment Registers

Unlike 8085, the 8086 addresses a segmented memory. The complete 1 megabyte memory, which the 8086 is able to address, is divided into 16 logical segments. Each segment thus contains 64 Kbytes of memory. There are four segment registers, viz. Code Segment Register (CS), Data Segment Register (DS), Extra Segment Register (ES) and Stack Segment Register (SS). The code segment register is used for addressing a memory location in the code segment of the memory, where the executable program is stored. Similarly, the data segment register points to the data segment of the memory, where the data is resided. The extra segment also refers to a segment which essentially is another data segment of the memory. Thus the extra segment also contains data. The stack segment register is used for addressing stack segment of memory. The stack segment is that segment of memory which is used to store stack data. The CPU uses the stack for temporarily storing important data, e.g. the contents of the CPU registers which will be required at a later stage. The stack grows down, i.e. the data is pushed onto the stack in the memory locations with decreasing addresses. When this information will be required by the CPU, they will be popped off from the stack. While addressing any location in the memory bank, the physical address is calculated from two parts, the first is segment address and the second is offset. The segment registers contain 16-bit segment base addresses, related to different segments. Any of the pointers and index registers or BX may contain the offset of the location to be addressed. The advantage of this scheme is that in place of maintaining a 20-bit register for a physical address, the processor just maintains two 16-bit registers which are within the word length capacity of the machine. Thus the CS, DS, SS and ES segment registers respectively contain the segment addresses for the code, data, stack and extra segments of memory. It may be noted that all these segments are the logical segments. They may or may not be physically separated. In other words, a single segment may require more than one memory chip or more than one segment may be accommodated in a single memory chip.

Pointers and Index Registers

The pointers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets within the code, data and stack segments respectively. The index registers are used as general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed

addressing modes. The register SI is generally used to store the offset of source data in data segment while the register DI is used to store the offset of destination in data or extra segment. The index registers are particularly useful for string manipulations.

Flag Register

The 8086's PSW contains 16 bits, but 7 of them are not used. Each bit in the PSW is called a flag. The 8086 flags are divided into the conditional flags, which reflect the result of the previous operation involving the ALU, and the control flags, which control the execution of special functions. The flags are summarized in Figure 3.2. The lower byte in the PSW corresponds to the 8-bit PSW in the 8080 and contains all of the condition flags except OF.

The condition flags are:

SF (Sign Flag): Is equal to the MSB of the result. Since in 2's complement negative numbers have a 1 in the MSB and for nonnegative numbers this bit is 0, this flag indicates whether the previous result was negative or nonnegative.

ZF (Zero Flag): I's set to 1 if the result is zero and 0 if the result is nonzero.

PF (Parity Flag): 1's set to 1 if the low-order 8 bits of the result contain an even number of 1's; otherwise it is cleared.

CF (Carry Flag): An addition causes this flag to be set if there is a carry out of the MSB, and a subtraction causes it to be set if a borrow is needed. Other instructions also affect this flag and its value will be discussed when these instructions are defined.

AF (Auxiliary Carry Flag): Is set if there is a carry out of bit 3 during an addition or a borrow by bit 3 during a subtraction. This flag is used exclusively for BCD arithmetic.

OF (Overflow Flag): Is set if an overflow occurs, i.e., a result is out of range. More specifically, for addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice versa.

As an example, if the previous instruction performed the addition

$$\begin{array}{r} 0010\ 0011\ 0100\ 0101 \\ +\ 0011\ 0010\ 0001\ 1001 \\ \hline 0101\ 0101\ 01011110 \end{array}$$

then following the instruction:

SF = 0 ZF = 0 PF = 0 CF=0 AF=0 OF=0

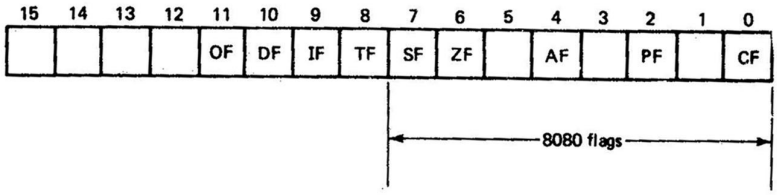


Figure 3.2: 8086's PSW.

If the previous instruction performed the addition

```
  0101 0100 0011 1001
+ 0100 0101 0110 1010
---------------------
  1001 1001 1010 0011
```

then the flags would be:

SF = 1 ZF = 0 PF = 1 CF = 0 AF = 1 OF = 1

The control flags are:

DF (Direction Flag)—Used by string manipulation instructions. If clear, the string is processed from its beginning with the first element having the lowest address. Otherwise, the string is processed from the high address towards the low address.

IF (Interrupt Enable Flag)—If set, a certain type of interrupt (a maskable interrupt) can be recognized by the CPU; otherwise, these interrupts are ignored.

TF (Trap Flag)—If set, a trap is executed after each instruction.

Student Activity 3.1

Before reading the next section, answer the following questions.

1. Explain in detail the registers of 8086.
2. If a physical address is 5A230 when (CS) = 5200, what will it be if the (CS) are changed to 7800?
3. Give the sum and the flag settings for AF, SF, ZF, CF, OF, and PF after hexadecimally adding 62A0 to each of the following:

   (a) 1234 (b) 4321 (c) CFA0 (d) 9D60

4. Give the difference and the flag settings for SF, ZF, CF, OF, and PF after subtracting 4AE0 from each of the following:

   (a) 1234 (b) 5D90 (c) 9090 (d) EA04

If your answers is correct, then proceed to the next section.

Top

Memory Addressing and Instruction Formats

A machine language instruction format has one or more number of fields associated with it. The first field is called as operation code field or opcode field, which indicates the type of the operation to be performed by the CPU. The instruction format also contains other fields known as operand fields. The CPU executes the instruction using the information which reside in these fields.

There are six general formats of instructions in 8086 instruction set. The length of an instruction may vary from one byte to six bytes. The instruction formats are described as follows:

1. One byte Instruction: This format is only one byte long and may have the implied data or register operands. The least significant 3-bits of the opcode are used for specifying the register operand, if any. Otherwise, all the 8-bits form an opcode and the operands are implied.

2. Register to Register: This format is 2 bytes long. The first byte of the code specifies the operation code and width of the operand specified by w bit. The second byte of the code shows the register operands and R/M field, as shown below.

   The register represented by the REG field is one of the operands. The R/M field specifies another

   register or memory location, i.e. the other operand.

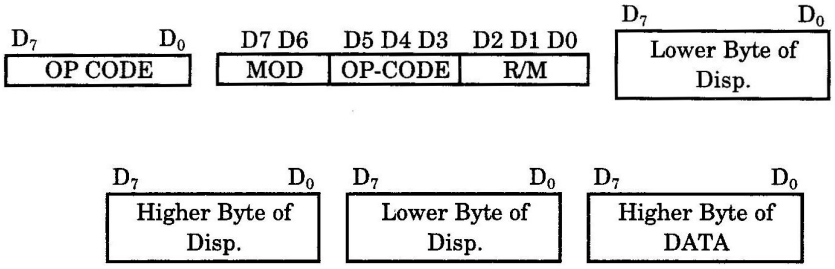
3. Register to/from Memory with no Displacement: This format is also 2 bytes long and similar to the register to register format except for the MOD field as shown.

   The MOD field shows the mode of addressing. The MOD, R/M, REG and the W fields are decided in Table 3.2.

4. Register to/from Memory with Displacement: This type of instruction format contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement. The format is as shown below.

5. Immediate Operand to Register: In this format, the first byte as well as the 3-bits from the second byte which are used for REG field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data. The complete instruction format is as shown below.

6. Immediate Operand to Memory with 16-bit Displacement: This type of instruction format requires 5 or 6 bytes for coding. The first 2 bytes contain the information regarding OPCODE, MOD, and R/M fields. The remaining 4 bytes contain 2 bytes of displacement and 2 bytes of data as shown.



The opcode usually aooears in the first byte. but in a few instructions, a register destination is in the first byte and few other instructions may have their 3-bits of opcode in the second byte. The opcodes have the single bit indicators. Their definitions and significance's are given as follows:

W-bit: This indicates whether the instruction is to operate over an 8-bit or 16-bit data/operands. If bit is 0, the operand is of8-bits and if W is I, the operand is of 16-bits.

D-bit: This is valid in case of double operand instructions. One of the operands must be a register specified by the REG field. The register specified by REG is source operand ifD=0, else, it is a destination operand.

S-bit: This bit is called as sign extension bit. The S bit is used along with W-bit to show the type of the operation. For, example

8-bit operation with 8-bit immediate operand is indicated by S = 0, W = 0;

16-bit operation with 16-bit immediate operand is indicated by S = 0, W = 1 and

16-bit operation with a sign extended immediate data is given by S = 1, W = 1

V-bit: This is used in case of shift and rotate instructions. This bit is set to 0, if shift count is 1 and is set to 1, if CL contains the shift count.

Z-bit: This bit is used by REP instruction to control the loop. If Z bit is equal to I, the instruction with REP prefix is executed until the zero flag matches the Z bit.

The REG code of the different registers (either as source or destination operands) in the opcode byte are assigned with binary codes. The segment registers are only 4 in number hence 2 binary bits will be sufficient to code them. The other registers are 8 in number, so at least 3-bits will be required for coding them. To allow the use of 16-bit registers as two 8-bit registers they are coded with W bit as shown in Table 3.1.

Table 3.1: Assignment of Codes with Different Registers

W bit	Register Address (code)	Registers	Segment 2 bit	Segment Registter
0	000	AL		
0	001	CL	00	ES
0	010	DL	01	CS
0	011	BL	10	SS
0	100	AH	11	DS
0	101	CH		
0	110	DH		
0	111	BH		

1	000	AX		
1	010	CX		
1	100	SP		
1	101	BP		
1	110	SI		
1	111	DI		

Please note that usually all the addressing modes have DS as the default data segment. However, the addressing modes using BP and SP have SS as the default segment register.

To find out the MOD and R/M fields of a particular instruction, one should first decide the addressing mode of the instruction. The addressing mode depends upon the operands and suggests how the effective address may be computed for locating the operand, if it lies in memory. The different addressing modes of the 8086 instructions are listed in Table 3.2. The R/M column and addressing mode row element specifies the R/M field, while the addressing mode column specifies the MOD field.

Table 3.2: Addressing Modes and the Corresponding MOD, REG and R/M Fields

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

Note: 1. D8 & D16 represent 8 and 16 bit displacements respectively.

2. The default segment for the addressing modes using BP and SP is SS, For all other addressing modes the default segments are DS or ES.

When a data is to be referred as an operand, DS is the default data segment register. CS is the default code segment register for storing program codes (executable codes). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available (defined in a particular program) can be read or written as data segments by newly defining the data segment as required. There is no physical difference in the memory structure or no physical separation between the segment areas. They may or may not overlap with each other.

Addressing Modes of 8086

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types. According to the flow of instruction execution, the instructions may be categorised as (i) Sequential control flow instructions and (ii) Control transfer instructions.

Sequential control flow instructions are the instructions which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions. The control transfer instructions, on the other hand, transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example, INT, CALL , RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are explained as follows:

1. Immediate: In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

   Example 1: MOV AX, 0005H

   In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. Direct: In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

   Example 2: MOV AX, [5000H]

   Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H*DS+5000H.

3. Register: In register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

   Example 3: MOV BX, AX.

4. Register Indirect: Sometimes, the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or Dl register. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

   Example 4: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+[BX].

5. Indexed: In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and Dl respectively.

   This mode is a special case of the above discussed register indirect addressing mode.

   Example 5: MOV AX, [SI]

   Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H*DS+[SI].

6. Register Relative: In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and Dl in the default (either DS or ES) segment. The example given below explains this mode.

   Example 6: MOV AX, 50H[BX]

   Here, the effective address is given as 10H*DS+50H+[BX].

7. Based Indexed: The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or Dl). The default segment register may be ES or DS.

   Example 7: MOV AX, [BX] [SI]

   Here, BX is the base register and SI is the index register. The effective address is computed as 10H*DS+[BX]+[SI].

8. Relative Based Indexed: The effective address is formed by adding an 8 or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers, in a default segment.

   Example 8: MOV AX, 5 OH [BX][SI]

   Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H*DS+[BX]+[SI]+50H.

   For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. intersegment and intrasegment addressing modes.

   If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode. If the destination location lies in the same segment, the mode is called intrasegment mode. Figure 3.3 shows the modes for control transfer instructions.

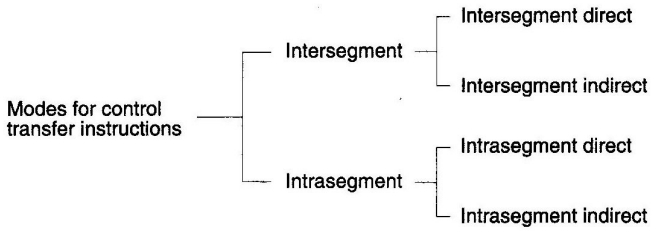


Figure 3.3: Addressing Modes for Control Transfer Instructions

9. Intrasegment Direct Mode: In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

   The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. –128 < d < +128), we term it as short jump and if it is of 16 bits (i.e. –32768 < d < +32768), it is termed as long jump.

10. Intrasegment Indirect Mode: In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

11. Intersegment Direct: In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. Intersegment Indirect: In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

    Forming the Effective Addresses: The following examples explain forming of the effective addresses in the different modes.

    Example 9: The contents of different registers are given below. Form effective addresses for different addressing modes.

    Offset (displacement) = 5000H

    [AXJ-1000H, [BXJ-2000H, [SI]-3000H, [DIJ-4000H, [BP]-5000H,

    [SPJ-6000H, [CSJ-0000H, [DSJ-1000H, [SSJ-2000H, [IPJ-7000H.

    Shifting a number four times is equivalent to multiplying it by 160 or 10,.

(i) Direct addressing mode

```
                                MOV AX, [5000H]
DS:OFFSET ⇔  1000H: 5000H
   10H* DS ⇔   10000
     Offset ⇔ + 5000
              ----------
              15000H - Effective address
```

(ii) Register indirect

```
                                  MOV AX, [BX]
      DS:BX ⇔  1000H:2000H
   10H*DS ⇔    10000
     [BX] ⇔  + 2000
             ----------
             12000H - Effective address
```

(iii) Register relative

```
                                MOV AX, 5000 [BX]
DS: [5000 + BX]
   10H*DS ⇔  10000
     Offset ⇔ + 5000
       [BX] ⇔ + 2000
              ----------
              17000H - Effective address
```

(iv) Based indexed

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


DS:[BX + SI]
  10H*DS ⇔ 10000
     [BX] ⇔+ 2000
     [SI] ⇔+ 3000
              ----------
              15000H - Effective address
```

(v) Relative based indexed

```
                                MOV AX, 5000 [BX] [SI]
DS: [BX + SI + 5000]
        10H*DS ⇔  10000
           [BX] ⇔ + 2000
           [SI] ⇔ + 3000
         Offset ⇔ + 5000
                  ----------
                  1A000 - effective address
```

Below, we present examples of address formation in control transfer instructions.

Example 10: Suppose our main program resides in the code segment where CS=1000H. The main program calls a subroutine which resides in the same code segment. The base register contains offset of the subroutine, i.e. BX= 0050H . Since the offset is specified indirectly, as the content of BX, this is indirect addressing. The instruction CALL [BX] calls the subroutine located at an address 10H*CS+[BX]=10050H, i.e. in the same code segment. Since the control goes to the subroutine which resides in the same segment, this is an example of intrasegment indirect addressing mode.

Example 11: Let us now assume that the subroutine resides in another code segment, where CS=2000H. Now CALL 2000H:0050H is an example of intersegment direct addressing mode, since the control now goes to different segment and the address is directly specified in the instruction. In this case, the address of the subroutine is 20050H.

Student Activity 3.2

Before reading the next section, answer the following questions.

1. How many address and data pious are there is 8086?
2. What is multiplexing?
3. What is PSW?
4. What is an instruction queue?
5. What is a Instruction pointer?
6. What is the use of SI, DI?
7. What is the use of SP, BP?
8. Which acts as count register?
9. Why segment registers are needed?
10. What are the 2 units of 8086?
11. Give some types of 8086 Instruction.
12. How physical address is computed in 8086?

If your answers is correct, then proceed to the next section.

Top

Memory Interfacing

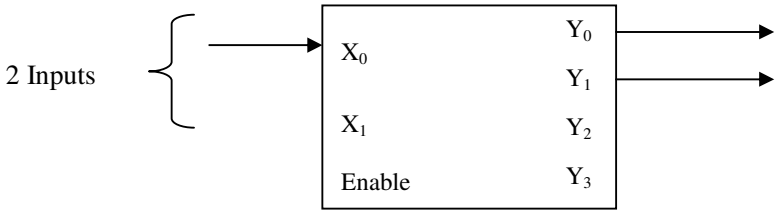
Memory Address Decoding

Binary Decoders **-** Decoders have n-inputs and 2^n outputs, each input combination results in a single output line having a 1, all other lines have a 0 on the output. Examples of use are decoding CPU instructions and memory addresses. Decoders typically have an *enable* when 1 enables decoding the input to 1 on a single output, when not enabled all outputs are 0. The switching function for an *enabled* two-input binary decoder is:

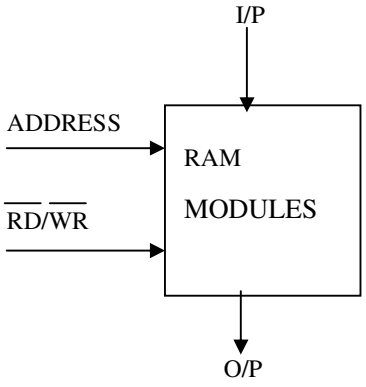
Switching function

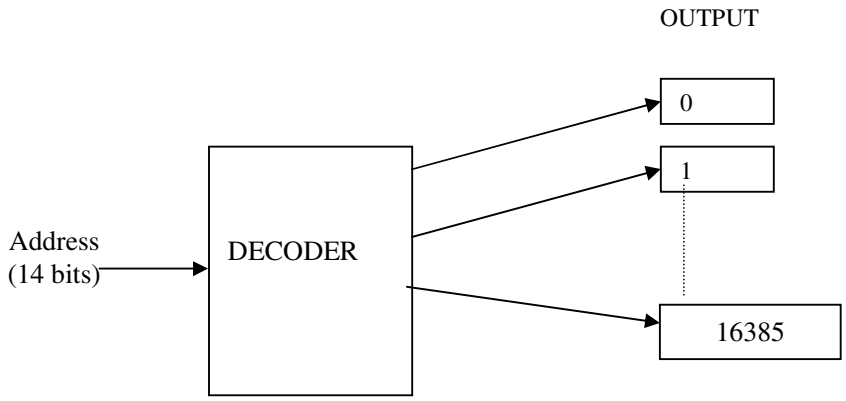
Enabled	x_1	x_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	-	-	0	0	0	0

The *2 to 4* decoder representation is:



Memory Address Decoding – Figure illustrates a 16K by 1 bit word memory (8 bit words are implemented by selecting 8 bits as a group, for example). Since 2^{14} is approximately 16K, a single decoder would require 14 inputs and 2^{14} output





The memory decoder is connected to the CPU by the *address bus.* Each memory cell is connected to an input and output *data bus,* a *read/write control,* and the decoder which enables the memory cell when the appropriate address appears. The decoder ensures that only a single memory cell is activated at a time for either input or output.

Top

Cache Memory

Caching is a technology based on the memory subsystem of your computer. The main purpose of a cache is to accelerate the computer while keeping the price of the computer low. Caching allows to do computer tasks more rapidly.

Cache technology is the use of a faster but smaller memory type to accelerate a slower but larger memory type.

Cache is small high speed memory usually Static RAM that contains the most recently accessed pieces of

the time it takes to an instruction (or piece of data) into the processor is very long when compared to the time to execute the instruction.

When using a cache, we must check the cache to see if the item is in the cache. If it is, that is called a cache hit. If not, it is called a cache miss and the computer must wait for a round trip from the larger, slower memory area.

A cache has some maximum size that is much smaller that the larger storage area.

Cache memory helps by decreasing the time it takes to move information to and from the processor. **cache** memory allows small portions of main memory to be accessed 3 to 4 times faster than DRAM It applies "Locality of Reference." At any time the processor will be accessing memory in a small or localized region of memory. The **cache** loads this region allowing the processor to access the memory region faster.

some of the common terms used when talking about cache.

Cache Hits

When the **cache** contains the information requested, the transaction is said to be a **cache** hit.

Cache Miss

When the **cache** does not contain the information requested, the transaction is said to be a **cache** miss.

Cache Consistency

Since **cache** is a photo or copy of a small piece main memory, it is important that the **cache** always reflects what is in main memory.

Some common terms used to describe the process of maintaining **cache** consistency are:

Snoop

When a **cache** is watching the address lines for transaction, this is called a snoop. This function allows the **cache** to see if any transactions are accessing memory it contains within itself.

Snarf

When a **cache** takes the information from the data lines, the **cache** is said to have snarfed the data. This function allows the **cache** to be updated and maintain consistency.

Snoop and snarf are the mechanisms the **cache** uses to maintain consistency.

Two other terms are commonly used to describe the inconsistencies in the **cache** are:

Dirty Data

When data is modified within **cache** but not modified in main memory, the data in the **cache** is called "dirty data."

Stale Data

When data is modified within main memory but not modified in **cache**, the data in the **cache** is called stale data.

Cache Architecture

Caches have two characteristics , a read architecture and a write policy.

The read architecture may be either "Look Aside" or "Look Through."

The write policy may be either "Write-Back" or "Write-Through."

Both types of read architectures may have either type of write policy, depending on the design.

Read Architecture

Look Aside Cache

In "look aside" **cache** architecture the main memory is located opposite the system interface. Both the main memory and the **cache** see a bus cycle at the same time. Hence the name "look aside."

Look Aside **Cache** Example

When the processor starts a read cycle, the **cache** checks to see if that address is a **cache** hit.

HIT: If the **cache** contains the memory location, then the **cache** will respond to the read cycle and terminate the bus cycle.

MISS: If the **cache** does not contain the memory location, then main memory will respond to the processor and terminate the bus cycle. The **cache** will snarf the data, so next time the processor requests this data it will be a **cache** hit. Look aside caches are less complex, which makes them less expensive. This architecture also provides better response to a **cache** miss since both the DRAM and the **cache** see the bus cycle at the same time. The draw back is the processor cannot access **cache** while another bus master is accessing main memory.

Read Architecture: Look Through

Main memory is located opposite the system interface. The discerning feature of this **cache** unit is that it sits between the processor and main memory. It is important to notice that **cache** sees the processors bus cycle before allowing it to pass on to the system bus. Look Through Read Cycle Example When the processor starts a memory access, the **cache** checks to see if that address is a **cache** hit.

HIT: The **cache** responds to the processor's request without starting an access to main memory.

MISS: The **cache** passes the bus cycle onto the system bus. Main memory then responds to the processors request. **Cache** snarfs the data so that next time the processor requests this data, it will be a **cache** hit. This architecture allows the processor to run out of **cache** while another bus master is accessing main memory, since the processor is isolated from the rest of the system. However, this **cache** architecture is more complex because it must be able to control accesses to the rest of the system. The increase in complexity increases the cost. Another down side is that memory accesses on **cache** misses are slower because main memory is not accessed until after the **cache** is checked. This is not an issue if the **cache** has a high hit rate and their are other bus masters.

Write Policy

A write policy determines how the **cache** deals with a write cycle. The two common write policies are Write-Back and Write-Through. In Write-Back policy, the **cache** acts like a buffer. That is, when the processor starts a write cycle the **cache** receives the data and terminates the cycle. The **cache** then writes the data back to main memory when the system bus is available. This method provides the greatest performance by allowing the processor to continue its tasks while main memory is updated at a later time. However, controlling writes to main memory increase the cache's complexity and cost. The second method is the Write-Through policy. As the name implies, the processor writes through the **cache** to main memory. The **cache** may update its contents, however the write cycle does not end until the data is stored into main memory. This method is less complex and therefore less expensive to implement. The performance with a Write-Through policy is lower since the processor must wait for main memory to accept the data.

Cache Components

The **cache** sub-system can be divided into three functional blocks: SRAM, Tag RAM, and the

Cache Controller. In actual designs, these blocks may be implemented by multiple chips or all may be combined into a single chip.

SRAM

Static Random Access Memory (SRAM) is the memory block which holds the data. The size of the SRAM determines the size of the **cache**.

Tag RAM

Tag RAM (TRAM) is a small piece of SRAM that stores the addresses of the data that is stored in the SRAM.

Top

Cache Controller

The **cache controller** is the brain behind the **cache**. Its responsibilities include: performing the snoops and snarfs, updating the SRAM and TRAM and implementing the write policy. The **cache controller** is also responsible for determining if memory request is cacheable and if a request is a **cache** hit or miss. The cache controller detects cache misses and controls sending and receiving the cells. This device also controls interface. The cache controller contains a status register, which can be read by the processor. The bits of the status register may be used to signal to the processor.

The cache controller accepts commands from the processor. Examples of the commands are

- Reset the Tag Rams
- Set interrupt mask

The cache controller will send a cell to request a cache line when a miss occurs. This cell may also flush an existing dirty line. It expects a cell to be returned containing the data, and the CPU is stalled until such a cell is received.

Student Activity 3.4

Before reading the next section, answer the following questions.

1. What is cache memory?
2. How it increases speed of fetching?
3. What is read/write policy of cache?

Summary

- 8086 has a powerful set of registers containing general purpose and special purpose registers. All the registers of 8086 are 16-bit registers. The general purpose registers, can be used as either 8-bit registers or 16-bit registers.
- The 8086's PSW contains 16 bits, but 7 of them are not used. Each bit in the PSW is called a flag. The 8086 flags are divided into the conditional flags, which reflect the result of the previous operation involving the ALU, and the control flags, which control the execution of special functions.
- Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes
- Caching is a technology based on the memory subsystem of your computer. The main purpose of a cache is to accelerate the computer while keeping the price of the computer low. Caching allows to do computer tasks more rapidly.
- The **cache controller** is the brain behind the **cache**. Its responsibilities include: performing the snoops and snarfs, updating the SRAM and TRAM and implementing the write policy.

Self-Assessment Questions

Solved Exercise

I. True or False

1. PF (Parity Flag)—Is set to 1 if the low-order 8 bits of the result contain an even number of Is; otherwise it is cleared.
2. Carry Flag is to be set if there is a carry out of the MSB, and a subtraction causes it to be set if a borrow is needed.
3. AF (Auxiliary Carry Flag)—Is set if there is a carry out of MSB during an addition
4. OF (Overflow Flag)—Is set if an overflow occurs, i.e., a result is out of range.
5. Addressing modes is used to fetch a location.

II. Fill in the blanks.

1. 8086 is _____ microprocessor.
2. 8086 has _____pins.
3. Memory access capacity of 8086 is _____.
4. Flags are divided into _____and _____.
5. Register can be used as _____bit and _____bit register.

Answer

I. True or False

1. True
2. False
3. True
4. True
5. False

II. Fill in the Blanks

1. 16 bit.
2. 40 pins.
3. 1 MB (2^{20}).
4. Conditional flags and control flags.
5. 8 and 16.

Unsolved Exercise

I. True or False

1. Cache is low speed memory
2. Physical address has same size as Segment register.
3. V-bit is used to cause shift and rotate instructions.
4. Z-bit is used for REP instructions.
5. D-bit is sign extensions bit.

II. Fill in the blanks.

1. BHE stands for _____.
2. AD_0-AD_{15} is used for _____.
3. Pins used in interrupt are _____and _____.
4. _____is Accumulator.
5. Segments registers are _____, _____, _____, and _____.

Detailed Questions

1. What are the 2 modes of 8086?
2. Which pin determines the mode.
3. Which mode is used for multiprocessor configuration.
4. What is bus cycle?
5. Why driver or receiver is needed?
6. What are the pins that specifies various types of transfer in minimum mode.
7. What are the types of transfer of 8086?
8. What is the use of Bus controller?

I/O Interface
Physical Memory Mapped I/O and Port Addressed I/O
8255 Programmable Peripheral Interface
8279 Keyboard/Display Controller
8254 Programmable Timer
8251 Programmable/Communication Interface
Interrupts
Interrupt Priority Management
PC Bus and Interrupt System
The Real Time Clock (RTC)
DMA
DMA Controller

Unit 4

Basic I/O Interface

Learning Objectives

After reading this unit you should appreciate the following:

- I/O Interface
- 8255 Programmable Interface
- 8254 Programmable Timer
- 8251 Programmable/Communication Interface
- Interrupts
- 8259 Programmable Interrupts Controller
- Real Time Clock
- DMA
- 8237/8257 DMA Controller

Top

I/O Interface

I/O devices such as keyboards and displays establish communication of computer with outside world. Devices can be interfaced in two ways I/O MAPPED I/O and Memory mapped I/O. In I/O mapped I/O, device is identified with a unique device number and data are transferred thru IN/OUT instruction. Memory

mapped I/O each device is identified with 16 bit address. I/O devices are considered to be a part of memory and memory related instruction is used for data transfer.

An I/O interface must be able to

- Determine whether or not it is being interfaced
- Determine whether it has to send data to CPU or receive data from CPU
- Send ready signal informing CPU that transfer is over
- Send interrupt Requests to CPU and receive interrupt acknowledgement and send an interrupt type.

An Interface can be divided into two parts. A part that interfaces to the I/O device and a part that interfaces to the system bus. There must be drivers and receivers to maintain signal quality, logic for translating the interface control signals to proper handshaking signals, logic for decoding address that appear on the bus.

Handshaking signals are used to determine in which direction transfer has to take place whether from CPU or to CPU. It should determine whether it is a READ or WRITE operation. Interrupt signals also must be handled here.

Address decoder determine whether it is I/O mapped I/O or Memory mapped I/O from one of the bits. If the decoder finds that an interface is referenced it sends signal to the appropriate device.

Interfaces can be categorized according to the way I/O devices transfer data either in serial or parallel form.

Memory Interface

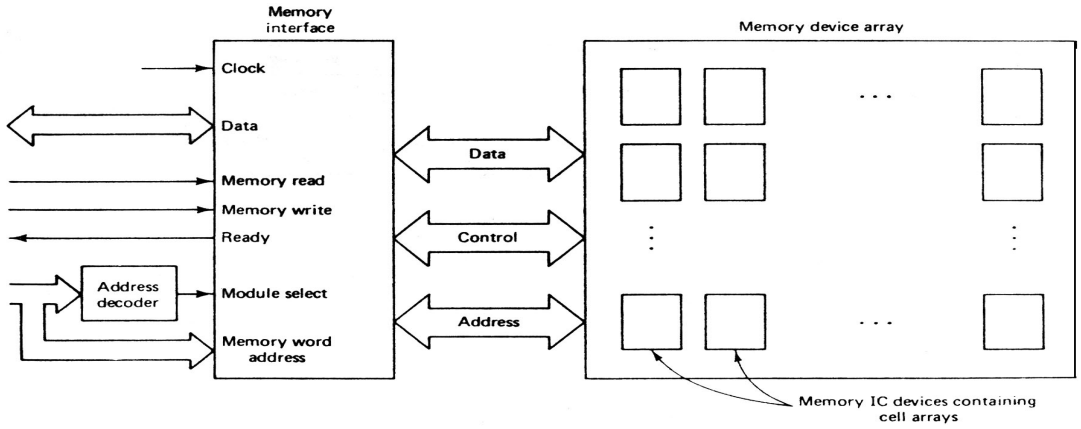


Figure 4.1: Memory Module Design

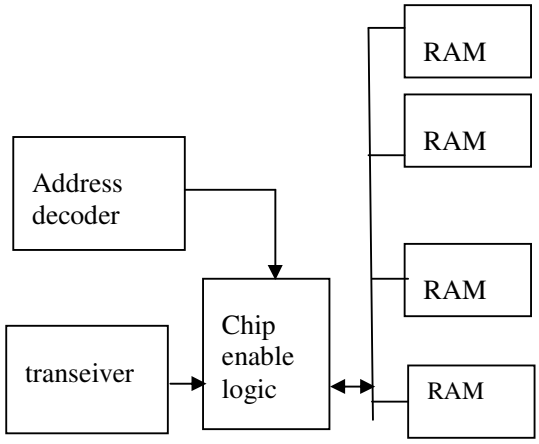
The memory of a computer consists of number of memory modules. Each module consists of an interface and an array of memory IC devices. Each IC device consists of an array of memory cells as shown in Figure 4.1. Each cell can store 1 bit.

Microprocessor communicates with memory through memory interface.

The primary function of memory interface is that the microprocessor must be able to read from or write to a given register of memory chip. The microprocessor must be able to select the memory chip, send control signals for read or write operation.

Certain signals to indicate whether a Memory read or write operation has to be performed. Whenever a communication with memory is required, a set of signals has to be sent by CPU.

Chip select logic which is used to select the particular chip based on the signal it receives from the transceiver



Student Activity 4.1

Before reading the next section, answer the following questions.

1. What is the use of memory interface?
2. Chip select logic is used for_____.

If your answers are correct, then proceed to the next

Top

Physical Memory Mapped I/O and Port Addressed I/O

CPU controlled I/O comes in two ways. The difference is simply whether we use the normal memory addresses for I/O, this is referred to as physical memory mapped I/O, or whether we set up a totally separate section of memory addresses which can ONLY be used for I/O, called port addressed I/O or I/O mapped I/O. Thus the latter has two separate addressing areas for the CPU, one for normal (physical) memory, one for I/O, while the former has only one which serves both purposes.

The other difference between these two techniques is that with memory mapped I/O the same processor instructions that are used for transferring data to and from the processor registers and memory, or for testing the content of a memory register etc, are used for the I/O operation, whereas port-addressed I/O has specific instructions (load and store) which can only be used for I/O operations, and have different mnemonics and codes. In the latter case data must always be transferred into a processor register, and in the latter instructions like ADD, SUB (subtract) and CMP (compare) can refer to the input/output registers.

The peripheral controller serves to present the data at the appropriate memory locations, memory mapped or port addressed as may be. It will also provide control registers to determine whether data can be inputted

or outputed, that is whether the external connections are for input or output of data, since they are often shared. The processor cannot read and write at the same location , it must do one or the other.

There will also be a 'status' register which have various 'flags' which signal to the processor that some data has been placed for reading. Some of these may also be observable as connections to the outside world, to show, for example, that the processor has supplied data for output. The registers are the same size as the computer memory locations, usually eight-bit,-byte sized , so that they usually occupy one memory address each. The actual number of addresses depends on the complexity and functions provided by the peripheral controller

Memory Mapped I/O

Memory I/O devices are mapped into the system memory map along with RAM and ROM. To access a hardware device, simply read or write to those 'special' addresses using the normal memory access instructions. The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. The disadvantage to this method is that the entire address bus must be fully decoded for every device. For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device. This increases the cost of adding hardware to the machine.

Port Mapped I/O or I/O Mapped I/O

I/O devices are mapped into a separate address space. This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access.

The advantage to this system is that less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine.

Handshaking

Handshaking, or two-way handshaking, is one type of strobed operation. It typically involves two handshaking lines: an output line to indicate when the board is ready and an input line that indicates when the peripheral device is ready. Depending on the timing and the property of the handshaking lines, there are different handshaking protocols.

Student Activity 4.2

Before reading the next section, answer the following question.

1. Differentiate I/O mapped I/O, Memory mapped I/O.
2. What are instruction used to I/O mapped I/O?

If your answer is correct, then proceed to the next section.

Top

8255 Programmable Peripheral Interface

Intel’s 8255 A programmable peripheral interface provides a good example of a parallel interface. As shown in Figure 4.2, the interface contain a control register and three separately addressable ports, denoted A, B, and C. Whether or not an 8255A is being accessed is determined by the signal on the CS pin and the direction of the access is according to the RD ad WR signals. Which of the four registers is being addressed is determined by the signals applied to the pins A1 and A0. Therefore, the lowest port address assigned to an 8255 A must be divisible by 4. A summary of the 8255 A’s addressing is:

A1	A0	RD	WR	CS	Transfer Description
0	0	0	1	0	Port A to data bus
0	1	0	1	0	Port B to data bus
1	0	0	1	0	Port C to data bus
0	0	1	0	0	Data bus to port A
0	1	1	0	0	Data bus to port B
1	0	1	0	0	Data bus to port C
1	1	1	0	0	Data bus to control register if D7= 1; if D7=0 input from the data bus is treated as a Set/Reset instruction
x	x	x	x	1	D7-D0 go to high-impedance state
1	1	0	1	0	Illegal combination
x	x	1	1	0	D7-D0 go to high-impedance state

Where 0 is low and 1 is high

Because the bits in port C are sometimes used as control bits, the 8255 A is designed so that they can be output to individually using a Set/Reset instruction. When the 8255 A receives a byte that is directed to its control register, it examines the data bit 7.

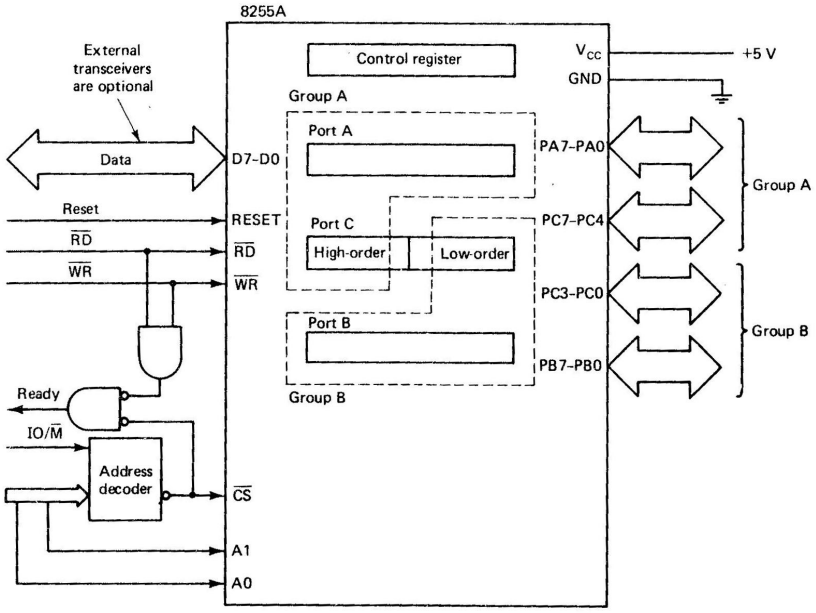


Figure 4.2: Diagram of the 8255 A

If this bit is 1, the data are transferred to the control register, but if it is 0, the data re treated as a Set/Reset instruction and is used to set or clear the port C it specified by the instruction. Bits 3-1 give the bit number of the bit to be changed and bit 0 indicates whether it is to be set or cleared. The remaining bits are unused.

The bits in the three ports are attached to pins that may be connected to the I/O device. These bits are divided into groups A and B, with group A consisting of the bits in port A and the four MSBs of port C and group b consisting of port B and the four LSBs of port C. the use of each of the two groups is controlled by the mode associated with it. Group A is associated with one of three modes, mode 0, mode 1, and mode 2, and group B with one of two modes, mode 0 and mode 1. The modes are determined by the contents of the control register, whose format is given in Fig. 4.3. These modes are:

Mode 0: If a group is in mode 0, it is divided into two sets. For group A these sets are port A ad the upper 4 bits of port C, and for group B they are port B ad the lower 4 bits of port C. each set may be used for inputting or outputting, but not both. Bits D4, D3, D1, and D0 in the control register specify which sets are for input and which are for output.

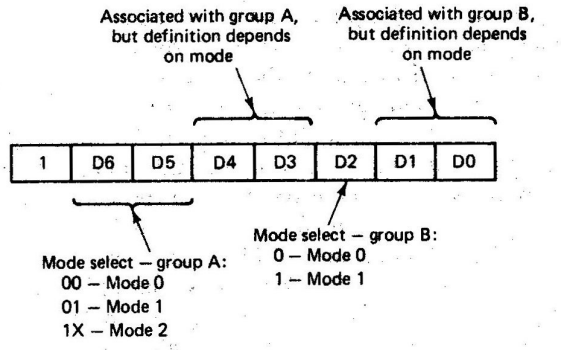


Figure 4.3: Format of the 8255 A's control register.

These bits are associated with the sets as follows:

D4 - Port A.

D3 - Upper half of port C.

D1 - Port B.

D0 – Lower half of port C.

If a bit is 0, then the corresponding set is used for output; if it is 1, the set is for input.

Mode 1: When group A is in this mode port A is used for input or output according to bit D4 (d4=1 indicates input), and the upper half of ports C is used for handshaking and control signals. For inputting, the four MSBs of port C are assigned the following symbols and definition:

PC4	STB_A	A 0 applied to this pin causes PA7-PA0 to be latched, or "strobed," into port A.
PC5	IBF_A	Indicates that the input buffer is full. It is 1 when port A contains data that have not yet been input to the CPU. When a 0 is on this pin the device can input a new byte to the interface.
PC6, PC7		May be used to output control signals to the device or input status from the device. If D3 of the control register is 0, they are for outputting control signals; otherwise, they are for inputting status.

For outputing

PC4,	PC5	Serve the same purpose as described above for PC6 and PC7.

PC7	OBF_A	indicates that the output buffer is full. It outputs a 0 to the device when port A is outputting new data to be taken by the device.
PC6	ACK_A	Device puts a 0 on this pin when it accepts data from port A.

In mode 1, PC3 is denoted $INTER_A$ and is associated with group A. It is used as an interrupt request line and is tied to one of the IR line in the system bus. When inputting to port A, this pin becomes 1 when new data are put in port A (i. e., it is controlled by PC4) and is cleared when the CPU takes the data. For output, this pin is set to 1 when the contents of port A are taken by the device as cleared when new data are sent from the CPU. If group B is in mode 1, port B is input to or output from according to bit D1 of the control register (D1=1 indicates input). For input, PC2 and PC1 are denoted STB_B and IBF_B, respectively, and serve the same purposes for group B as STB_AAND IFB_A do for group A. Similarly, for output PC1 and PC2 are denoted OBF_B and ACK_B. PC0 becomes $INTER_B$ and its use is analogous to that of $INTR_A$. The interrupt enable for group A is controlled by setting or clearing integral flags. Setting or clearing these flags is simulated by setting or clearing PC4, for input, or PC6, for output, using a Set/Reset instruction. Similarly, the interrupt enable for group B is controlled by set/clear of PC2 for both input and output.

Mode 2: This mode applies only to group A, although it also uses PC3 for making interrupt requests. In mode 2, port A is a bidirectional port and the four MSBs of port C are defined as follows:

PC4	STB	A 0 on this line causes the data on PA7 – PA0 to be strobed into port A.
PC5	IBF_A	Becomes 1 when port A is filled with new data from lines PA7 – PA0 and is cleared when these data are taken by the CPU.
PC6	ACK_A	Indicates that the device is ready to accept data from PA7-PA0.
PC7	OBF_A	Becomes 0 when port A is filled with new data from the CPU and is set to 1 when data are taken by the device.

While group A is in mode 2, group B may be in either mode 0 or mode 1. However, if group B is mode 0, only PC2 – PC0 can be used for input or out put because group A has borrowed PC3 to use as an interrupt request line. Normally, if group A is in mode, PC2 – PC0 would be connected to control and status pons on the device attached to the port A lines. Port B may used also be used for this purpose.

In all three modes port C reflects the signals on PC7 – PC0 and can be read with an IN instruction.

Top

8279 Keyboard/Display Controller

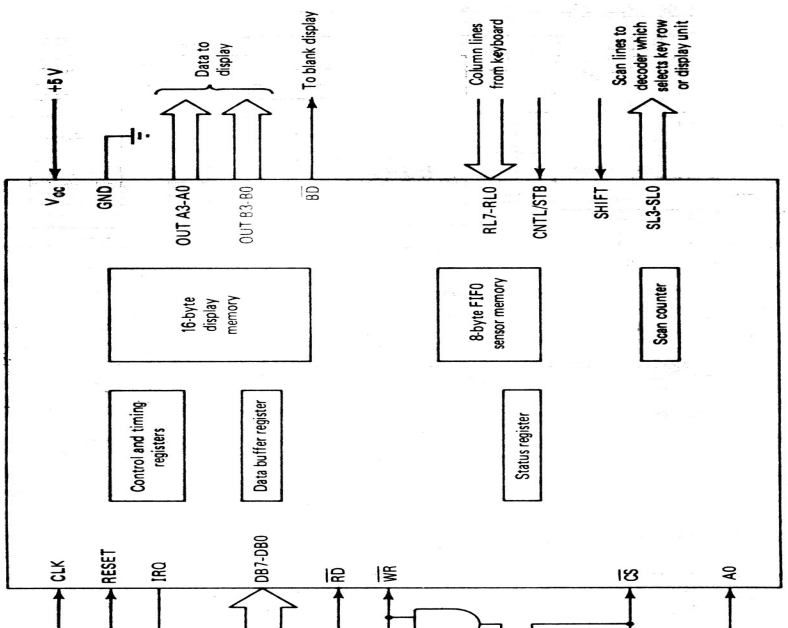


Figure 4.4: Structure of the 8279

Figure shows the structure of 8279 and its interface to the bus. Addressing is according to the following table

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	$\overline{AO}$	Transfer Description
0	1	0	0	Data bus to data bus buffer
0	1	0	1	Data bus to control register
0	0	1	0	Data buffer register to data bus
0	0	1	1	Status register to data bus

8279 scans each row of the keyboard by sending out row addresses on SL2-SL0 and inputting signals on the return lines RL7-RL0, which are column addresses. When a depressed key is detected the key is automatically debounced by weighting 10 ms to check if the same key remains depressed. If a depressed key is detected and 8 bit code word corresponding to key position is assembled by combining column, row position and shift and control status as follows

CNTL	SHFT	R	R	R	C	C	C

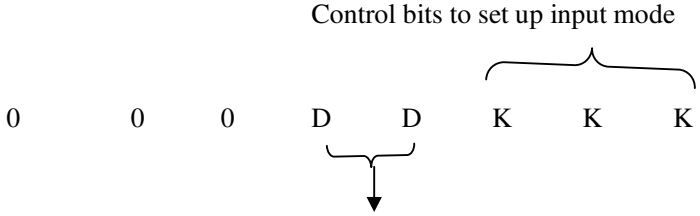
R-row (scan line address)

C-column(return line address)

The SHFT and CNTL pins are used to support shift and control keys. The key position is then entered into 8x8 FIFO sensor memory and IRQ(Interrupt Request) line is activated if sensor memory was previously empty.

The control and timing registers are collection flags and registers that are accessed by commands. The 3 MSBs of command determine its type and meaning of the remaining 5 bits. Out of 8 types of commands 3 commands are important the formats of the 3 commands are as follows

Key board display mode set- it specifies the input and display methods and is used initialize the 8279. Its format is:



DD Control bits to set display modes

00-left entry,8 8 bit display

01- left entry,16 8 bit display

10-right entry ,8 8 bit display

11-right entry ,16 8 bit display

KKK

000 encoded keyboard scan mode with 2-key lockout

001 Decoded keyboard scan mode with 2 key lockout.

010 encoded keyboard scan mode with N-key rollover

011 Decoded keyboard scan mode with N-key rollover

100 encoded sensor matrix scan mode

101 Decoder sensor matrix scan mode

110 strobed input with encoded display scan

111 strobed input with decoded display scan

Read FIFO Sensor Memory:

Indicates that a read operation from the data buffer register inputs a byte from the FIFO memory. In the sensor mode it specifies which row has to be read. This command is needed before inputting data from the FIFO memory. Its format is

0 1 0 1 X A A A

Row address to be read in a sensor mode

X-don't

Bit 4 is autoincrement bit. If 1 next input is from the next byte in the FIFO

Write to display memory:

Indicates that write to data buffer register will put data in display memory

Its format is

1 0 0 1 A A A A

Address of the location in the display memory where the data for the next write will be stored 8279 provides two options for handling the situation in which more than 1 key is depressed at about the same time. With the two-key lockout option ,if another key is depressed while the first key is being debounced, the key which is released last will be entered into the FIFO. If the second key is depressed within two scan cycles after the first is debounced and the first key remains depressed after the second one is released, then the first key is recognized. If more than one is depressed, after they are depressed they are all entered in the order they were sensed.

8279 has a sensor matrix mode in which signals in the return lines are stored into the FIFO at the row corresponding to the scan address.

Display Control

8279 provides a 16 byte display memory and refresh logic. Each address in the display memory corresponds to a display unit with address 0 representing the leftmost display unit. Output is accomplished by 8279 repeatedly sending out characters over the lines OUT A3-A0 and OUT B3-B0 and unit select address is over SL3-SL0.

For the auto increment left entry, after each write to the display the addresses incremented by one, so that the next character appears in the display unit to the right. Auto increment right entry allows character to be displayed in electronic calculators form. It causes the display to be shifted left to one character and stores the next character from the right.

Top

8254 Programmable Timer

A diagram of Intel's 8254 interval timer/event counter is given in Figure 4.5. The 8254 consists of three identical counting circuits, each of which has CLK and GATE inputs and an OUT output. Each can be viewed as containing a Control and Status Register pair, a Counter Register (CR) for receiving the initial count, a Counter Element CCE) which performs the counting but is not directly accessible from the processor, and an Output Latch (OL) for latching the contents of the CE so that they can be read. The CR, CE, and OL are treated as pairs of 8-bit registers. (Physically, the registers are not exactly as depicted, but to the programmer the figure is conceptually accurate.)

The registers can be accessed according to the following table:

CS	RD	WR	A1	AO	Transfer
0	1	0	0	0	To counter 0 CR
0	1	0	0	1	To counter 1 CR
0	1	0	1	0	To counter 2 CR
0	1	0	1	1	To a control register or indicates a command
0	0	1	0	0	From counter 0 OL or status register
0	0	1	0	1	From counter 1 OL or status register
0	0	1	1	0	From counter 2 OL or status register

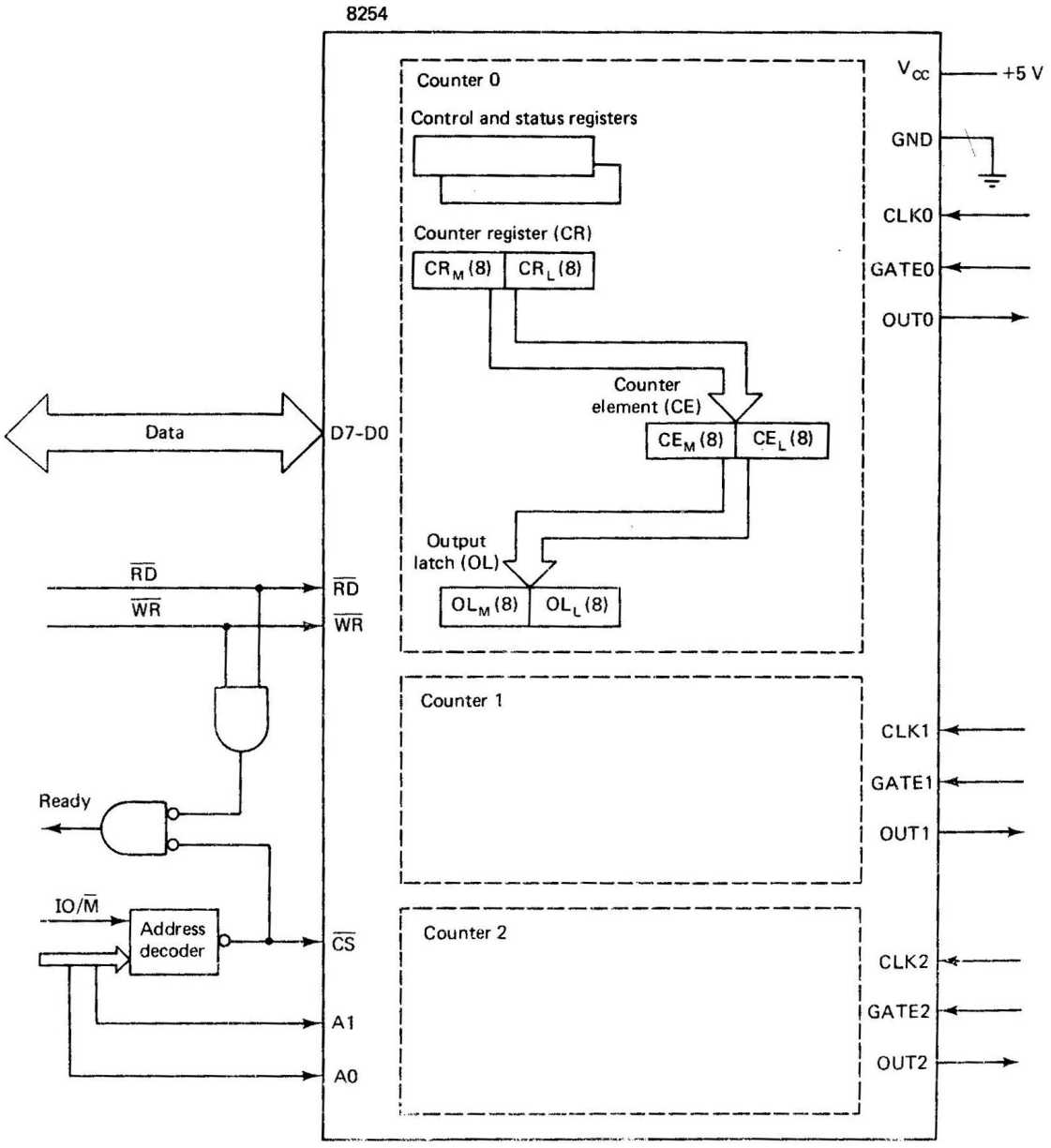


Figure 4.5: Diagram of the 8254

where 0 means low and 1 means high. All other combinations result in the data pins being put into their high-impedance state. When A1 =AO= 1, whether a control register is being written into or a command is being given depends on the MSBs of the byte being output. For the last three combinations, whether an OL or status register is read is determined by a previous command.

There are two types of commands, the counter latch command, which causes the CE in the counter specified by the two MSBs of the command to be latched into the corresponding OL, and the read back

command, which may cause a combination of the CEs to be latched or "prepare" a combination of status registers to be read. To prepare a status register means to cause it to be read the next time a read operation inputs from the counter. When the two MSBs are 00, 01, or 10 a counter latch command is indicated, but if they are 11 a read back is to be performed. In a latch command bits 5 and 4 must be 0 and the remaining bits are unused. The read back command has the format:

1	1	$\overline{\text{COUNT}}$	$\overline{\text{STAT}}$	CNT2	CNT1	CNT0	0

If the COUNT bit is 0, then the CEs for all of the counters whose CNT bits are 1 are latched. If CNTO=CNT2=1 but CNT1=0, then the CEs in counters 0 and 2 are latched but the CE in counter 1 is not latched. Similarly, STAT=0 causes the counters' status registers to be prepared for input. CEs can be latched and status registers can be prepared in the same command.

The formats of the control and status registers are given in Figure 4.6. If the two MSBs of an output are both 1, they indicate that the output is to be a read back command; otherwise, they specify a counter. If they specify a counter and bits 4 and 5 are both 0, then a latch command is indicated and it is directed to the control register of the counter specified by the top 2 bits, but if they are not both 0, then they indicate the type of the input from OL or output to CR. The combination 01 indicates that the Read/Write operations are from/to the OL_L/CR_L, 10 indicates that they are from/to the OL_M/CR_M, and 11 indicates that these operations are to occur in pairs, with the first byte coming from/going to OL_L/CR_L and the second from/to OL_M/CR_M. A 1-byte write to CR will cause the other byte to be zeroed. Bits 1, 2, and 3 determine the mode and bit 0 specifies the format of the count. Given that N is the initial count, the modes are:

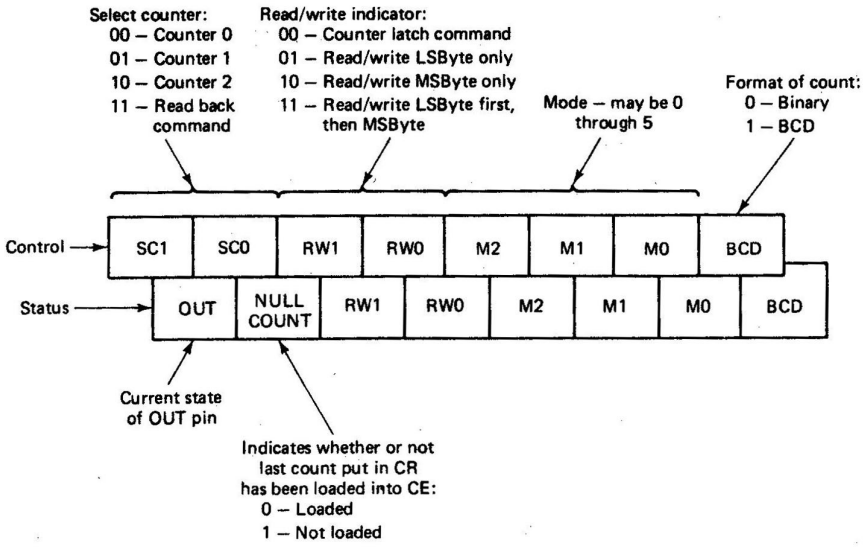


Figure 4.6: Control and status registers for 8254 counters

Mode 0 (Interrupt on Terminal Count)—GATE = 1 enables counting and GATE = 0 disables counting, and GATE has no effect on OUT. The contents of CR are transferred to CE on the first CLK pulse after CR is written into by the processor, regardless of the signal on the GATE pin. The pulse that loads CE is not included in the count. OUT goes low when there is an output to the control register and remains low until the count goes to 0. Mode 0 is primarily for event counting.

Mode 1 (Hardware Retriggerabic One-Shot)—After CR has been loaded with N, a 0-to-1 transition on GATE will cause CE to be loaded, a 1-to-0 transition at OUT, and the count to begin. When the count reaches 0 OUT will go high, thus producing a negative-going OUT pulse N clock periods long.

Mode 2 (Periodic Interval Timer)—After loading CR with N, a transfer is made from CR to CE on the next clock pulse. OUT goes from 1 to 0 when the count becomes 1 and remains low for one CLK pulse; then it returns to 1 and CE is reloaded from CR, thus giving a negative pulse at OUT after every N clock cycles. GATE = 1 enables the count and GATE=0 disables the count. A 0-to-1 transition on GATE also causes the count to be reinitialized on the next clock pulse. This mode is used to provide a programmable periodic interval timer.

Mode 3 (Square-Wave Generator)—It is similar to mode 2 except that OUT goes low when half the initial count is reached and remains low until the count becomes 0. Hence the duty cycle is changed. As before, GATE enables and disables the count and a 0-to-1 transition on GATE reinitializes the count. This mode may be used for baud rate generation.

Mode 4 (Software-Triggered Strobe)—It is similar to mode 0 except that OUT is high while the counting is taking place and produces a one-clock-period negative pulse when the count reaches 0.

Mode 5 (Hardware-Triggered Strobe—Retriggerabic)—After CR is loaded, a O-to-I transition on GATE will cause a transfer from CR to CE during the next CLK pulse. OUT will be high during the counting but will go low for one CLK period when the count becomes 0. GATE can reinitialize counting at any time.

For all modes, if the initial count is 0, it will be interpreted as 2^{16} or 10^4 depending on the format of the count. The above descriptions were only to provide an overall idea of the operation of the 8254 in the various modes.

Top

8251 Programmable/Communication Interface

As an example of a serial interface device let us consider Intel's 8251 A programmable communication interface. The 8251A is diagrammed in Figure 4.7. It is capable of being programmed for asynchronous or synchronous communication. The data in buffer and data-out buffer registers share the same port address. For input, the serial bit stream arriving on the R x D pin is shifted into the receiver shift register and then the data bits are transferred to the data-in buffer register, where they can be input by the CPU. Conversely, on output the data bits put in the data-out buffer register by the CPU are transferred to the transmitter shift register and, along with the necessary synchronization bits, are shifted out through the T x D pin. Among other things the contents of the mode register, which are initialized by the executing program, determine whether the 8251A is in asynchronous mode or synchronous mode and the format of the characters being received and transmitted. The control register, which is also set by the program, controls the operation of the interface, and the status register makes certain information available to the executing program. Clearly, the sync character registers are for storing the sync characters needed for synchronous communication.

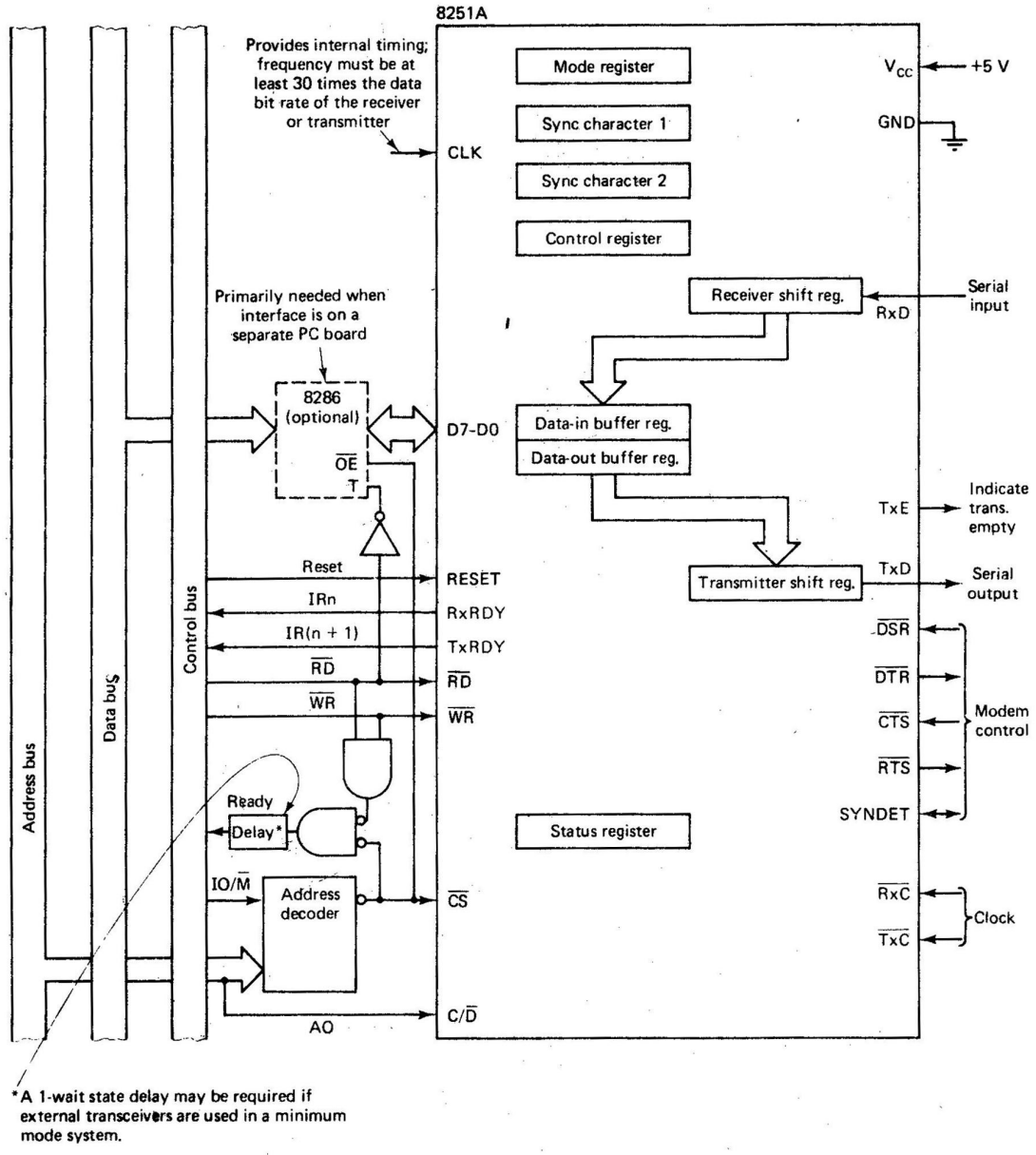


Figure 4.7: 8251 A Serial Communication Interface

Even though all seven of the registers on the left side of Figure 4.7 can be accessed by the processor, the 8251A is associated with only two port addresses. The C/D pin is connected to the address line AO and AO differentiates the two port addresses. The 8251A internally interprets the C/D, RD, and WR signals as follows:

C/D(=AO)	RD	WR	
0	0	1	Data input from the data-in buffer
0	1	0	Data output to the data-out buffer
1	0	1	Status register is put on data bus
1	1	0	Data bus is put in mode, control, or sync character register

where 1 means that the pin is high and 0 means that it is low. All other combinations cause the three-state D7-DO pins to go into their high-impedance state.

Whether the mode, control, or sync character register is selected depends on the accessing sequence. A flowchart of the sequencing is given in Figure 4.8. After a hardware reset or a command is given with its reset bit set to 1, the next output with A0=1 (i.e., with C/D=1, RD=1, and WR=0) is directed to the mode register. The formats of the mode register for both the asynchronous and synchronous cases are defined in Figure 4.9. If the two LSBs of the mode are zero, then the interface is put in its synchronous mode and the MSB determines the number of sync characters. In the synchronous mode, the next 1 or 2 bytes output with A0 = 1 become the sync characters. If the two LSBs of the mode are not both 0, then the 8251A enters its asynchronous mode. In either case, all subsequent bytes prior to another reset go to the control register if A0=1 and the data-out buffer register if A0 = 0.

In the synchronous mode the baud rates of the transmitter and receiver, which are the shift rates of the shift registers, are the same as the frequencies of the signals applied to TxC and RxC, respectively, but in the asynchronous mode the three remaining possible combinations for the two LSBs in the mode register dictate the baud rate factor. The relationship between the frequencies of the TxC and RxC clock inputs and the baud rates of the transmitter and receiver is:

Clock frequency = Baud rate factor x Baud rate

If 10 is in the LSBs of the mode register and the transmitter and receiver baud rates are to be 300 and 1200, respectively, then the frequency applied to TxC should be 4800 Hz and the frequency at RxC should be 19.2 kHz. In both the asynchronous and synchronous modes, bits 2 and 3 indicate the number of data bits in each character, bit 4 indicates whether or not there is to be parity bit, and bit 5 specifies the type of parity (odd or even). For the asynchronous mode the two MSBs indicate the number of stop bits, but for the synchronous mode bit 6 determines whether the SYNDET pin is to be used as an input or as an output and, as mentioned above, bit 7 indicates the number of sync characters.

If the SYNDET pin is used as an output it becomes active when a bit-for-bit match has been found between the incoming bit stream and the sync character(s). If the search for sync characters is conducted by an external device, then SYNDET can be used to input a signal, indicating that a match has been found by the external device. The pin also has a meaning during asynchronous operation, but in this case it can only be an output. This output is called the break detect signal and goes high whenever a character consisting of all 0s is received.

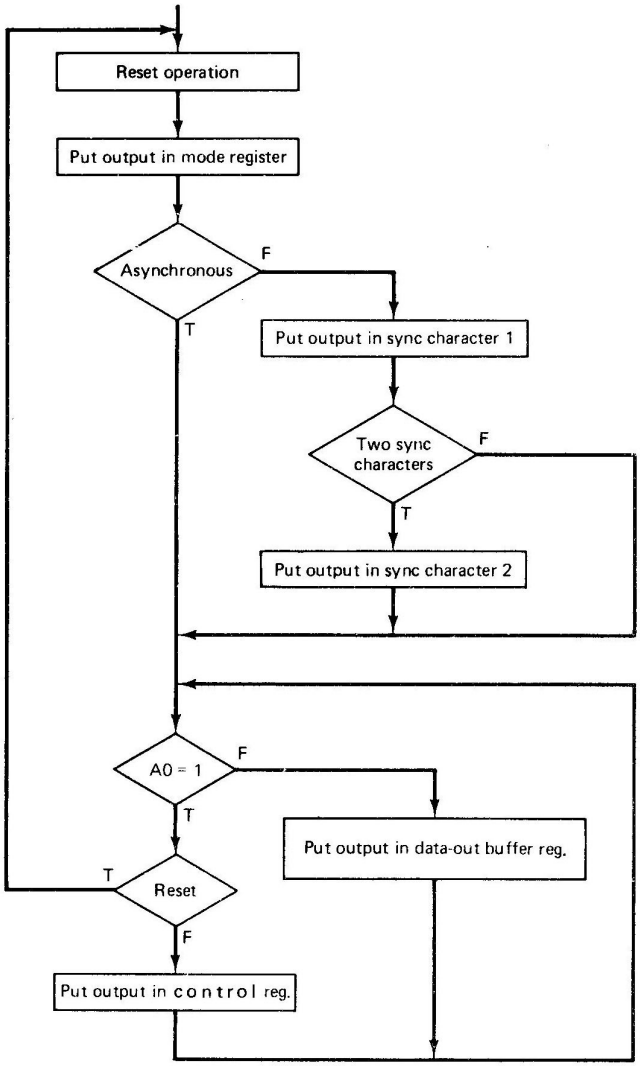


Figure 4.8: Flowchart of the disposition of output

The format for the control register is given in Figure 4.10. Bit 0 of this register must be 1 before data can be output and bit 2 must be 1 before data can be received. Programmed answering of a modem is accomplished by setting bit 1 to 1 since this forces the DTR pin to 0 and the complement of DTR is normally connected to the CD line from the modem. Bit 3 equal to 1 forces TxD to 0, thus causing break characters to be transmitted. Setting bit 4 to 1 causes all the error bits in the status register to be cleared (the bits that are set when framing, overrun, and parity errors occur). Bit 5 is used for sending a Request to Send signal to a modem. If the complement of the RTS pin is connected to a modem's CA line, then a 1 put in bit 5 will cause the CA line to go high. Setting bit 6 causes the 8251 A to be reinitialized and the reset sequence to be reentered (i.e., a return is made to the top of the flowchart shown in Figure 4.8 and the next output will be to the mode register). Bit 7 is used only with the synchronous mode. When set, it causes the 8251A to begin a bit-by-bit search for a sync character or sync characters.

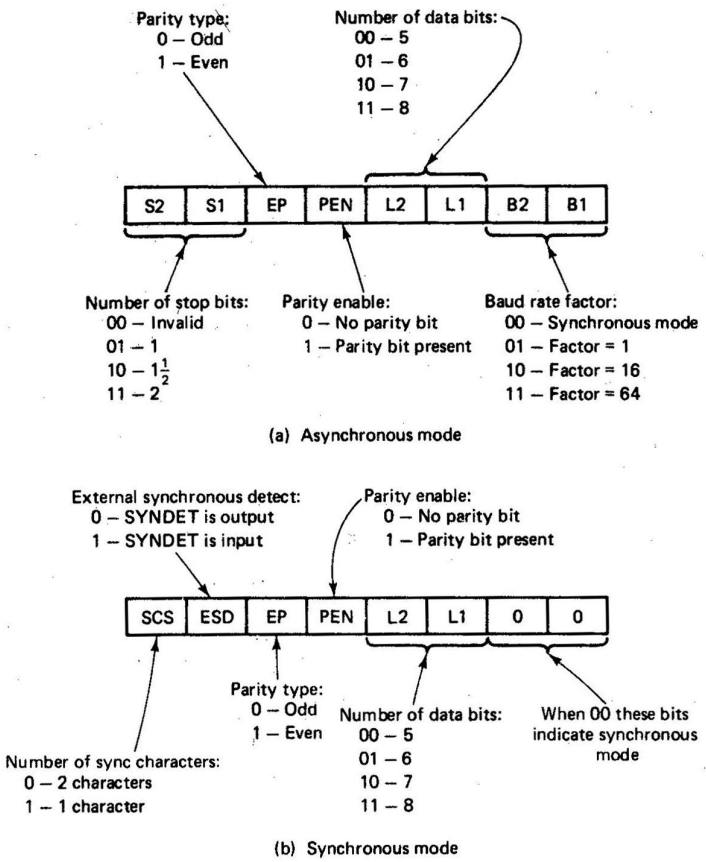


Figure 4.9: Format of the mode register

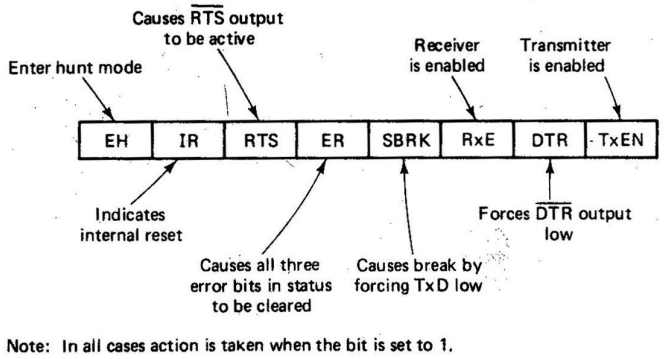


Figure 4.10: Format of the control register

Typical connections to modems for asynchronous and synchronous transmissions are shown in Figure 4.11. With regard to the synchronous connections it is assumed that the timing is controlled by the modem and its related communications equipment. Also, if this equipment is used to detect the sync character(s) at the beginning of a received message, then it can inform the 8251A of its success over the SYNDET line. On

the other hand, if 8251A searches for the sync character(s), then it can use the SYNDET line as an output to tell the modem that the sync character(s) has been found. To satisfy the RS-232-C standard, drivers and receivers are needed to convert the TTL-compatible signals at TxD and RxD to the proper voltage levels.

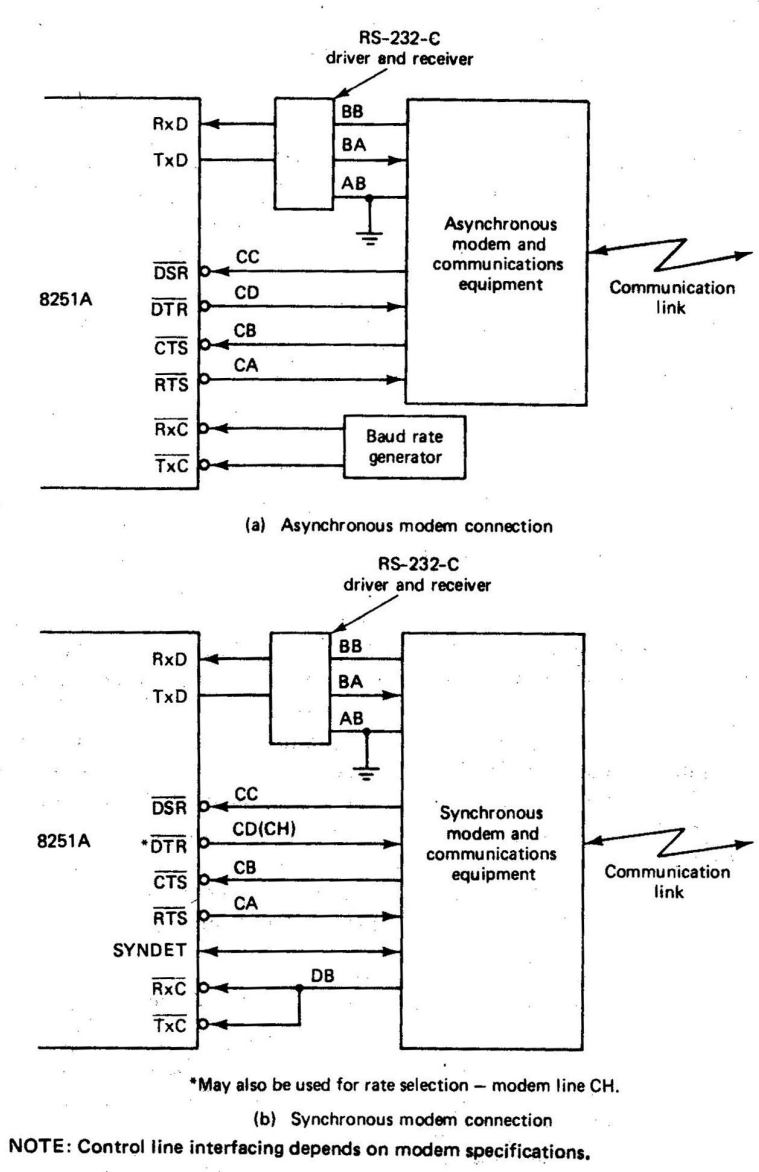


Figure 4.11: 8251 A modem connections

A program sequence which initializes the mode register and gives a command to enable the transmitter and begin an asynchronous transmission of 7-bit characters followed by an even-parity bit and 2 stop bits is:

```
MOV    AL, 11111010B
OUT    51H, AL
```

```
MO      AL, 00110011B
OUT     51H, AL
```

This sequence assumes that the mode and control registers are at address 511-1 and the clock frequencies are to be 16 times the corresponding baud rates. The sequence:

```
MOV     AL, 00111000B
OUT     51H, AL
MOV     AL, 16H
OUT     51H, AL
OUT     51H, AL
MOV     AL, 10010100B
OUT     51H, AL
```

would cause the same 8251A to be put in synchronous mode and to begin searching for two successive ASCII sync characters. As before, the characters are to consist of 7 data bits and an even parity bit, but there will, of course, be no stop bits. The format of the status register is given in Figure 4.12. Bits 1, 2, and 6 reflect the signals on the RxRDY, TxE, and SYNDET pins. TxRDY indicates that the data-out buffer is empty. Unlike the TxRDY pin, this bit is not affected by the CTS input pin or the TxEN control bit. RxRDY indicates that a character has been received and is ready to be input to the processor. Either the TxRDY and RxRDY bits can be used for programmed I/O or the signals on the corresponding pins can be connected to interrupt request lines to provide for interrupt I/O. The TxRDY bit is automatically cleared when a character- is made available for transmitting and the RxRDY bit is automatically cleared when the character that set it is input by the processor. Bit 2 indicates that the transmitter shift register is waiting to be sent a character from the data-out buffer register. During synchronous transmissions, while this bit is set, the transmitter will take its data from the sync character registers until data are put in the data-out buffer register. Bits 3, 4, and 5 indicate parity, overrun, and framing errors, respectively. When an error is detected, the bit having the corresponding error type will be set to 1. If the complement of the DSR pin is connected to the Data Set Ready (CC) line, then bit 7 reflects the state of the modem and is 1 when the modem is turned on and is in its data mode.

Figure 4.13 gives a typical program sequence which uses programmed I/O to input 80 characters from the 8251A, whose data buffer register's address is 0050, and put them in the memory buffer beginning at LINE. The inner loop continually tests the RxRDY bit until it is set by a character being put in the data-in buffer register. Then the newly arrived character is moved to the buffer and the error bits are checked. If the present character arrived before the previous character was input or a parity or framing error occurred during transmission, then the input ceases and a call is made to an error routine that would presumably examine the individual error bits, print an appropriate message, and clear the error bits.

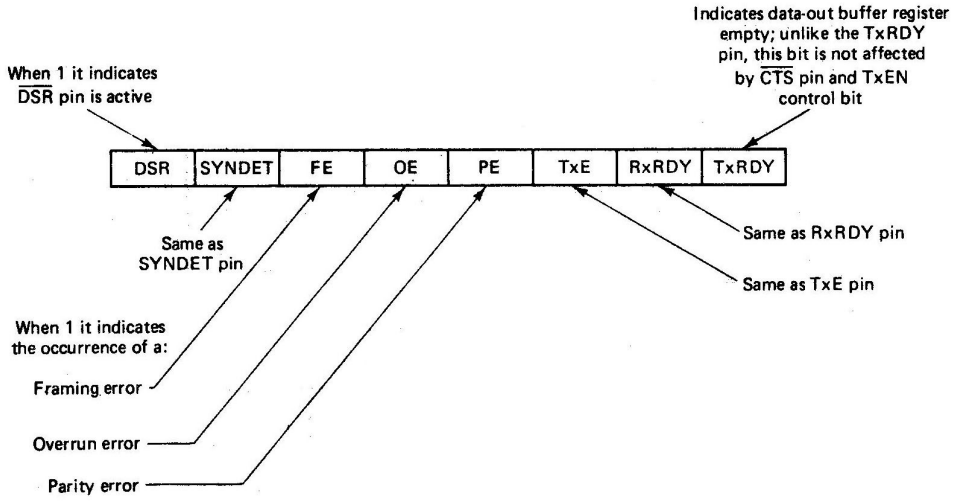


Figure 4.12: Format of the status register

```
        MOV   AL,00110101B        ;ENABLE TRANSMITTER AND RECEIVER
        OUT   51H;AL              ;AND CLEAR ERROR BITS
        MOV   DI,0                ;INITIALIZE INDEX
        MOV   CX,80               ;PUT COUNT IN CX
BEGIN:  IN    AL,51H              ;WAIT FOR INPUT
        TEST  AL,02H
        JZ    BEGIN
        IN    AL,50H              ;INPUT CHARACTER AND
        MOV   LINE[DI],AL         ;PUT IN LINE BUFFER
        INC   DI
        IN    AL,51H              ;CHECK ERROR
        TEST  AL,00111000B        ;BITS AND
        JNZ   ERROR               ;IF NO ERROR IS FOUND
        LOOP  BEGIN               ;CONTINUE INPUTTING
        JMP   SHORT EXIT
ERROR:  CALL  NEAR PTR ERR_ROUT   ;ELSE CALL ERR_ROUT
EXIT:    .
         .
         .
```

Figure 4.13: Inputting a line of characters through an 8251A

Because inputting a character automatically resets the RxRDY bit, unless another character is received before the inner loop is reentered, the inner loop must cycle until the RxRDY bit is reset to 1 by the next incoming character. If the incoming characters have fewer than 8 bits, the unused MSBs in the data buffer register are always zeroed. Also, the parity bit is not passed to the processor and checks for parity errors can only be made by examining the parity error bit in the status register. On output, if a character is less than 8 bits long, the unneeded MSBs in the data-out buffer register are ignored.

Student Activity 4.3

Before reading the next section, answer the following questions.

1. Explain the working of 8251 Programmable/Communication Interface.
2. Discuss the Format of the status register of 8251.
3. Explain the working of 8254 Programmable Timer.

If your answers are correct, then proceed to the next section.

Top

Interrupts

When the CPU detects an interrupt signal, it stops the current activity and jumps to a special routine, called an interrupt handler. This handler then detects why the interrupt occurred and takes the appropriate action. When the handler is finished executing this action, it jumps back to the interrupted process.

Several levels or "types" of interrupts are supported, ranging from 0 to 255. Each type has a reserved memory location, called an interrupt vector. The interrupt vector points to the appropriate interrupt handler. When two or more interrupts occur at the same time, the CPU uses a priority system The 256 priority levels supported by the Intel 8086-processors can be split into three categories:

- Internal Hardware-Interrupts
- External Hardware-Interrupts
- Software Interrupts

Internal Hardware-Interrupts

Internal hardware-interrupts are the result of certain situations that occur during the execution of a program, e.g. divide by zero. The interrupt levels attached to each situation are stored in hardware and cannot be changed.

Divide by zero	00h		
Single Step		04h	
NMI	08h		
Breakpoint		0ch	
Overflow			

External Hardware-Interrupts

External hardware-interrupts are produced by controllers of external devices or coprocessors and are linked to the processor pin for Non Maskable Interrupts (NMI) or to the pin for Maskable Interrupts (INTR). The NMI line is usually reserved for interrupts that occur because of fatal errors like a parity error or a power distortion.

Interrupts from external devices can also be linked to the processor via the Intel 8259A Programmable Interrupt Controller (PIC). The CPU uses a group of I/O ports to control the PIC and the PIC puts its signals on the INTR pin. The PIC makes it possible to enable or disable interrupts and to change the priority levels under supervision of a program.

The instructions STI and CLI can be used to enable/disable interrupts on the INTR pin, this has no effect on NMI interrupts.

Software Interrupts

Software interrupts are the result of an INT instruction in an executed program. This can be seen as a programmer triggered event that immediately stops execution of the program and passes execution over to the INT handler. The INT handler is usually part of the operating system and will determine the action that should be taken (e.g. output to screen, execute file etc.) An example is INT 21h, which is the DOS service interrupt. When the handler is called it will read the value stored in AH (sometimes even AL) and jumps to the right routine.

Interrupt Table

Each interrupt level has a reserved memory location, called an interrupt vector. All these vectors (or pointers) are stored in a table, the interrupt table. This table lies at linear address 0, or with 64KB segments, at 0000:0000. Each vector is 2 words long (4 bytes). The high word contains the offset and the low word the segment of the INT handler.

Since there are 256 levels and each vector is 4 bytes long, the table contains 1024 bytes (256*4=1024). The INT number is multiplied by 4 to fetch the address from the table.

How INT's are Processed

When the CPU registeres an INT it will push the FLAGS register to the stack and it will also push the CS and IP registers. After that the CPU disables the interrupt system. Then it gets the 8-bit value the interrupting device sends and multiplies this by 4 to get the offset in the interrupt table. From this offset it gets the address of the INT handler and carries over execution to this handler.

The handler usually enables the interrupt system immediately, to allow interrupts with higher priority. Some devices also need a signal that the interrupt has been acknowledged. When the handler is finished it must signal the 8259A PIC with an EOI (End Of Interrupt). Then the handler executes an IRET instruction

Student Activity 4.4

Before reading the next section, answer the following questions.

1. What are the various modes of 8254?
2. Brief 8279 features.
3. Write notes on Interrupts.

If your answers are correct, then proceed to the next section.

Top

Interrupt Priority Management

The interrupt priority management logic indicated in Fig. 4.14 can be implemented in several ways. It does not need to be present in systems which use software priority management or simple daisy chaining, but more complex systems may require the efficiency gained by including hardware for managing the I/O interrupts. Many manufacturers have made priority management devices available and Intel is no exception. Although such a device made by one manufacturer could be used with processors made by other manufacturers, generally there are fewer compatibility problems if the CPU and interrupt priority device are produced by the same company. Therefore, we will be concerned with the Intel 8259A programmable interrupt controller (PIC), which has been specifically designed to work with the 8086/8088 as well as other members of the Intel microprocessor family.

The 8259A has been designed so that it can operate alone or in concert with other 8259As. In order to limit the initial discussion, an interrupt system involving a single 8259A device is considered first; then the discussion is extended to systems that can include as many as nine 8259As.

Interrupt System Based on a Single 8259A

The 8259A is contained in a 28-pin dual-in-line package that requires only a + 5-V supply voltage. Its organization is shown in Fig. 4.14 along with its connections to a maximum mode system. Its pins (other than the supply voltage and ground pins) are defined as follows:

D7-DO – For communicating with the CPU over the data bus. On a few systems bus drivers may be needed, but on other systems direct connections can be used.

INT – To send interrupt request signals to the CPU.

INTA – To receive interrupt acknowledge signals from the CPU. The 8259A assumes that an acknowledgment consists of two negative pulses, thus making it compatible with 8086/8088 systems.

RD – To signal the 8259A that it is to place the contents of the IMR, ISR, or IRR register or a priority level on the data bus. Which of these possibilities is placed on the bus depends on the state of the 8259A and is discussed below.

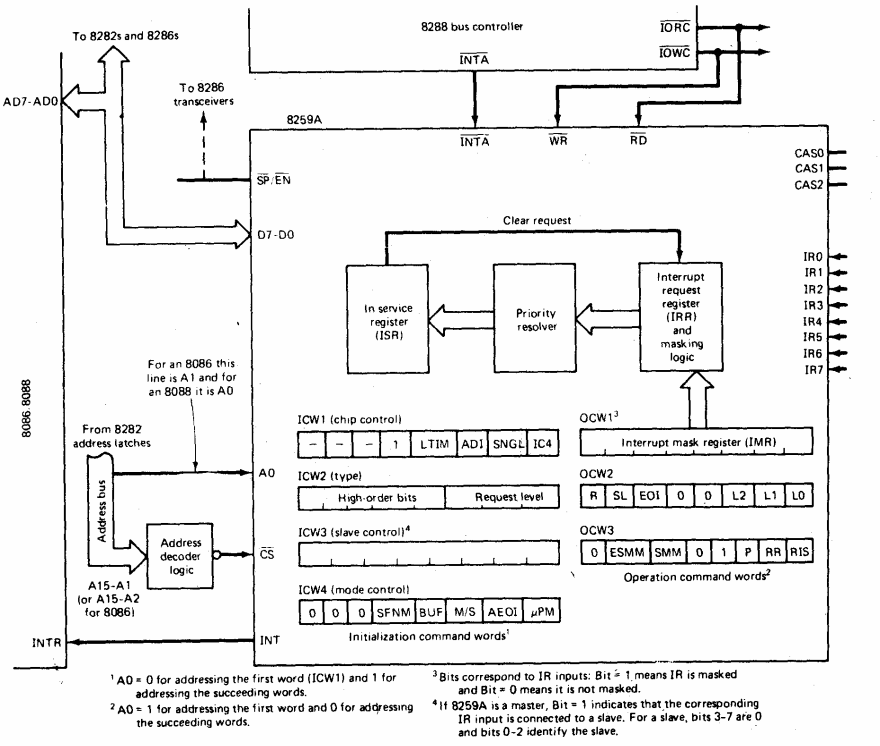


Figure 4.14: Organization of the 8259A programmable interrupt controller

WR – To signal the 8259A that it is to accept data from the data bus and us the data to set the bits in the command words. How the received data are distributed is discussed later.

CS – For indicating that the 8259A is being accessed. This pin is connected to the address bus through the decoder logic that compares the high-order bits of the address of the 8259A with the address currently on the address bus. Input to this pin can be combined with S2 to give the ready signal.

AO – For indicating which port of the 8259A is being accessed. Two addresses must be reserved in the I/O address space for each 8259A in the system.

IR7-IRO – For receiving interrupt requests from I/O interfaces or other 8259As referred to as slaves.

CAS2-CASO – To identify a particular slave device.

SP/EN – For one of two purposes; either as an input to determine whether the 8259A is to be a master (SP/EN = 1) or as a slave (SP/EN = 0), or as an output to disable the data bus transceivers when data are being transferred from the 8259A to the CPU. Whether the SP/EN pin is used as an input or output depends on the buffer mode discussed below.

For an 8088 the two addresses associated with an 8259A are normally consecutive, and the AO line is connected to the AO pin, but because there are only eight data pins on the 8259A and the 8086 always inputs the interrupt pointer from the lower 8 bits of its 16-bit data bus, all data transfers to and from the 8259A must be made over the lower byte of the bus. The easiest way to guarantee that all transfers will use the lower half of the bus is to connect the Al line to AO and use two consecutive even addresses, with the first being divisible by 4. However, to simplify the following discussion, the second address will be referred to as the odd address for both cases.

The control portion of the 8259A contains several programmable bits that can be viewed as being contained in seven 8-bit registers. These registers are divided into two groups, with one group containing the initialization command words (ICWs) and the other group containing the operation command words (OCWs). The initialization command words are normally set by an initialization routine when the computer system is first brought up and remain constant throughout its operation. By contrast, the operation command words are used to dynamically control the processing of interrupts.

The IRR (and its associated masking logic), priority resolver, and ISR are for receiving and controlling the interrupts that arrive at the IR7-IRO pins. The IRR latches the incoming requests and, in conjunction with the priority resolver, allows unmasked requests with sufficient priority to put a 1 on the INT pin. The priority resolver logic determines the priorities of the requests in the IRR and the ISR is for holding the requests currently being processed.

After a bit in the IRR is set to 1 it is compared with the corresponding mask bit in the IMR. If the mask bit is 0, the request is passed on to the priority resolver, but if it is 1, the request is blocked. When an interrupt request is input to the priority resolver its priority is examined and, if according to the current state of the priority resolver the interrupt is to be sent to the CPU, the INT line is activated.

Assuming that the IF flag in the CPU is 1, the CPU will enter its interrupt sequence at the completion of the current instruction and return two negative pulses over the INTA line. Upon the arrival of the first pulse, the IRR latches are disabled so that the IRR will ignore further signals on the IR7-IRO lines. This state is maintained until the end of the second INTA pulse. Also, the first INTA pulse will cause the appropriate ISR bit to be set and the corresponding IRR bit to be cleared. The second INTA pulse causes the current contents of ICW2 to be placed on D7-DO, and the CPU uses this byte as the interrupt type. If the automatic end of interrupt (AEOI) bit in ICW4 is 1, at the end of the second INTA pulse the ISR bit that was set by the first INTA pulse is cleared; otherwise, the ISR bit is not cleared until the proper end of interrupt (EOI) command is sent to OCW2.

As indicated above, the initialization command words are normally filled by an initializing routine when the system is turned on and contain the control bits that are held constant throughout the system's operation. The 8259A has an even address (AO = 0) and an odd address (AO = 1) associated with it and the initialization command words must be filled consecutively by using the even address for ICWI and the odd address for the remaining ICWs.

The definitions of the bits in ICWI are:

Bits 7-5 — Not used in an 8086/8088 system, only in an 8080 or 8085 system.

Bit 4 — Always set to 1. It directs the received byte to ICWI as opposed to OCW2 or OCW3. Which also use the even address (A0 = 0).

Bit 3 (LTIM) — Determines whether the edge-triggered mode (LTIM = 0) or the level-triggered mode (LTIM = 1) is to be used. The edge-triggered mode causes the IRR bit to be cleared when the corresponding ISR bit is set.

Bit 2 (ADD — Not used in an 8086/8088 system, only in an 8080 or 8085 system.

Bit 1 (SNGL) — Indicates whether or not the 8259A is cascaded with other 8259As. SNGL = 1 when only one 8259A is in the interrupt system.

Bit 0 (IC4) — Is set to 1 if an ICW4 is to be output to during the initialization sequence. For an 8086/8088 system this bit must always be set to 1 because bit 0 in JCW4 must be set to 1.

Bits 7-3 of ICW2 are tilled from bits 7-3 of the second byte output by the CPU during the initialization of the 8259A, and bits 2-0 are set according to the level of the interrupt request, e.g., a request on IR6 would cause them to be set to 110. ICW3 is significant only in systems including more than one 8259A and is output to only if SNGL = 0. ICW4 is output to only if IC4 (ICWI) is set to 1; otherwise, the contents of ICW4 is cleared. The bits in ICW4 are defined as follows:

Bits 7-5 — Always set to 0.

Bit 4 (SFNM) — If set to 1, the special fully nested mode is used. This mode is utilized in systems having more than one 8259A and is discussed below.

Bit 3 (BUF) — BUF = 1 indicates that the SP/EN is to be used as an output to disable the system's 8286 transceivers while the CPU inputs data from the 8259A. If no transceivers are present, BUF should be set to 0 and, in systems involving only one 8259A, a 1 should be applied to the SP/EN pin.

Bit 2 (M/S) — This bit is ignored if BUF = 0. For a system having only one 8259A, this bit should be 1; otherwise, it should be 1 for the master and 0 for the slaves.

Bit 1 (AEOI) — If AEOI = 1, then the ISR bit that caused the interrupt is cleared at the end of the second INTA pulse.

Bit 0 (µPM) — µPM = 1 indicates the 8259A is in an 8086/8088 system. This bit being 0 implies an 8080 or 8085 system. A typical program sequence for setting the contents of the ICWs, which assumes that the even address of the 8259A is 0080, is:

```
MOV     AL,13H
OUT     80H,AL
MOV     AL,18H
```

```
OUT     81H,AL
MOV     AL,ODH
OUT     81H,AL
```

The first two instructions cause the requests to be edge triggered, denote that only one 8259A is used, and inform the 8259A that an ICW4 will be output. The next two instructions cause the 5 most significant bits of the interrupt type to be set to 00011. ICWS is not output to because SNGL = 1; therefore, the last two instructions set ICW4 to OD, which informs the 8259A that the special fully nested mode is not to be used, the SP/EN is used to disable transceivers, the 8259A is a master, EOI commands must be used to clear the ISR bit, and the 8259A is part of an 8086/8088 system.

There are three OCWs. The command word OCWI is used for masking interrupt requests; when the mask bit corresponding to an interrupt request is 1, then the request is blocked. OCW2 and OCWS are for controlling the mode of the 8259A and receiving EOI commands. A byte is output to OCWI by using the odd address associated with the 8259A and bytes are output to OCW2 and OCWS by using the even addresses. OCW2 is distinguished from OCWS by the contents of bit S of the data byte. If bit S is 0, the byte is put in OCW2, and if it is 1, it is put in OCWS. Both OCW2 and OCWS are distinguished from ICWI, which also uses the even address, by the contents of bit 4 of the data. If bit 4 is 0, then the byte is put on OCW2 or OCWS according to bit S. There is no ambiguity in ICW2, ICWS, ICW4, and OCWI all using the odd address because the initialization words must always follow ICWI as dictated by the initialization sequence, and an output to OCWI cannot occur in the middle of this sequence.

Referring back to our earlier discussion, bits L2-LO of OCW2 are for designating an IR level, bit 5 is for giving EOI commands, and bits 6 and 7 are for controlling the IR levels. Recall that when the AEOI bit in ICW4 is 1, the ISR bit, which is set by the interrupt request, is reset automatically at the end of the second INTA pulse, but if AEOI = 0, then the ISR bit must be explicitly cleared by an EOI command, which consists of sending an OCW2 with bit 5 equal to 1. When an EOI command is given the meanings of the four possible combinations of bit 7, the R (rotate) bit, and bit 6, the SL (set level) bit, are:

R	SL	
0	0	Nonspecific, normal priority mode
0	1	Specifically clears the ISR bit indicated by L2-LO
1	0	Rotate priority so that a device after being serviced has the lowest priority
1	1	Rotate priority until position specified by L2-LO is lowest

The bits in OCW2 are only temporarily retained by the 8259A until the actions specified by them are carried out. This statement is particularly important with regard to the EOI bit. Let us now examine these possibilities in greater detail beginning with the normal priority mode (00).

Ordinarily, a request on IRO has the highest priority, one on IRI has the next highest priority, and so on. When the first INTA pulse arrives the priority resolver allows only the unmasked request having the highest priority to set its ISR bit. Because the 3 least significant bits of ICW2, where ICW2 indicates the interrupt type and determines the interrupt pointer address, are determined by which ISR bit is set, the address of the interrupt routine depends on which ISR bit is set. Therefore, the interrupt routine associated with the device

connected to the highest-priority IR pin is begun first and the other requests must wait until further interrupts are allowed.

Under the normal priority mode, if ISRn is set, the priority resolver will not recognize any requests on IR7 through IR(n+l), but will recognize unmasked requests on IR(n -1) through IRO, Consequently, if the IF flag in the CPU has been reset to 1, requests having higher priority than the one being processed may cause the current interrupt routine to be interrupted while those having lower priorities are kept waiting. The lower-priority requests are processed according to their priorities as the higher-priority ISR bits are cleared. If AEOI = 1, the ISR bit corresponding to an interrupt is automatically cleared at the end of the second INTA pulse. When AEOI = 0 the ISR must be cleared by the interrupt routine by setting bit 5 of OCW2.

As an example of the normal priority mode, suppose that initially AEOI = 0 and all ISR and IMR bits are clear. Also suppose that, as shown in Fig. 4.15, requests occur simultaneously on IR2 and IR4, then a request arrives at IRI, and last a request arrives at IR3 and that these are the only requests. First, ISR2 will be set and the interrupt routine associated with IR2 will start executing. After this routine resets the IF flag to 1 and when the IRI request is made, ISRI will be set and the IRI routine will be executed in its entirety. While it is executing it should reset the IF bit to 1 and send the necessary command to clear ISRI. Upon the return to the IR2 routine, ISR2 is cleared. Then ISR4 is set and its routine is begun. While this routine is executing IR3is made. It is acknowledged as soon as IF is reset to 1, and ISR3 is set. Then the IR3 routine is initiated. Before the IR3 routine is completed it should clear ISR3 and set IF. The return is made to the IR4 routine, which should clear ISR4 before returning to the IR2 routine. The IR2 routine, having already cleared the ISR2 bit, would simply return to the interrupted program. (Note that if IF is not reset to 1 within the interrupt routine, further interrupts will not be processed until the routine is completed, i.e., the IRET instruction is encountered.)

Although a 1 sent to bit 5 of OCW2 normally causes the highest-priority ISR bit (i.e., the last ISR bit to be set) to be cleared, any ISR bit can be explicitly cleared by sending an OCW2 with the R, SL, and EOI bits set to Oil and putting the number of the bit to be cleared in L2-L0. If

0 1 1 0 0 0 1 1

is sent to OCW2, then ISR3 will be cleared.

In addition to the normal priority mode discussed above, OCW2 can rotate the priority by assigning bottom priority to any one of the IR levels. In this case the other priorities will follow as if the normal ordering had been rotated. For instance, if the lowest priority is given to IR4, then the order of priorities will be:

IR5, IR6, IR7, IR0, IR1, IR2, IR3, IR4

(i.e., IR5 is rotated into the top-priority position). A rotation by one can be obtained by letting the combination for the R and SL bits be 10. If the R and SL bit combination is II, then the IR level with the lowest priority is the one specified by L2-LO. If IR5 currently has top priority and

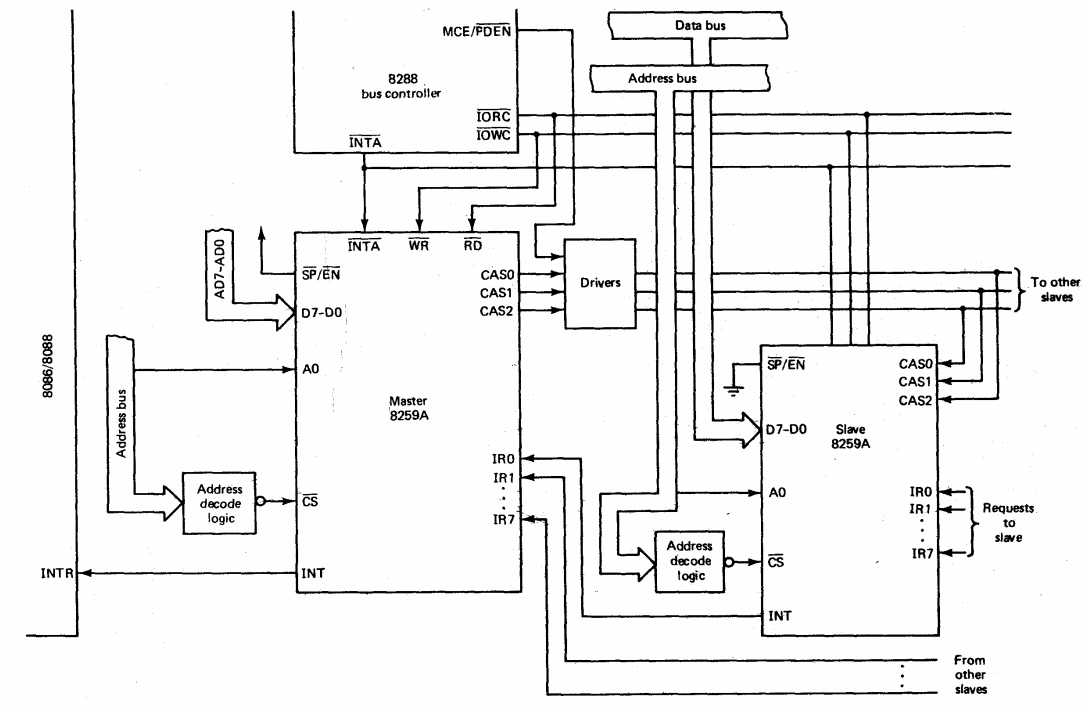


Figure 4.15: Actions taken in the normal operating mode when a typical sequence of interrupts occurs.

1 0 1 0 0 0 0 0

is sent to OCW2, then the new priority ordering would be

IR6, IR7, IR0, IR1, IR2, IR3, IR4, IR5

but if

1 1 1 0 0 0 1 0

is sent, the new ordering would be

IR3, IR4, IR5, IR6, IR7, IR0, IR1, IR2

The R and SL bits may also have significance when EOI = 0. In this case R = 1 and SL = 0 cause automatic rotations when AEOI = 1, and R = SL = 0 turns off this action so that automatic rotations do not take place. R = SL = 1 and EOI = 0 result in the lowest priority being designated by L2-LO without an EOI command being sent. The remaining combination, R = 0 and SL = 1, causes no action.

In OCW3 the ESMM (enable special mask mode) and SMM (special mask mode) bits can be used to negate the priority modes discussed above. If a byte is sent to OCW3 in which both ESMM and SMM are set to 1, then unmasked interrupt requests are processed as they arrive (provided that the processor's IF bit is 1) and the priority order is ignored. By subsequently sending a byte to OCW3 in which ESMM = 1 and SMM = 0, a switch back to the priority ordering of interrupts can be made. If a byte with the ESMM bit equal to 0 is sent to OCW3, then the SMM bit will have no effect and the special mask mode will not change.

The P (polling) bit is used to place the 8259A in polling mode. This mode assumes that the CPU is not accepting interrupts (IF = 0) and it is necessary for the interrupt requests in the IRR to be polled. When the

P bit is 1 the next RD signal would cause" the appropriate bit in the ISR to be set just as if INTA signal had been received, and would return to the AL register in the CPU a byte of the form

I	—	—	—	—	W2	W1	W0

where I = 1 indicates that an interrupt is present and W2, Wl, and W0 give the IR level of the highest-priority interrupt. For example, if P = 1, the priority or daring is

IR3, IR4, IR5, IR6, IR7, IR0, IR1, IR2

there are unmasked interrupts on IR4 and IRI, and the instruction

```
IN    AL, 80H
```

(where 0080 is the even address of the 8259A) is executed, then

1	—	—	—	—	1	0	0

is input to the AL register.

When P = 0, the contents of IRR or ISR can be read into the AL by setting RR = 1 and executing the instruction

```
IN    AL, 80H
```

If at the time the IN instruction is executed RIS = 0, then IRR is input; otherwise, ISR is read. The contents of IMR can always be read by using the of 8259A, e.g., for the address assignment indicated above,

```
IN    AL, 81H
```

would input the contents of IMR to AL.

Because the OCW3 bits (except for ESMM) are used to specify whether or not the 8259A is in special mask mode and which information is to be put on the data bus during a read, these bits are retained until they are reset by the next output to OCW3. For example, if P = 0, RR = 1, and RIS = 0, any read from the even address of the 8259A before a new byte is sent to OCW3 will cause IRR to be read.

As a final note regarding the internal workings of the 8259A, let us examine what happens when noise interferes with a request. An IR input must remain high until after the trailing edge of the first INTA pulse. If it does not, then the 8259A will simulate a 1 on IR7. Therefore, provided that no device is connected to IR7. Requests on this line would indicate improper drops in signals on the other request lines and the interrupt routine associated with IR7 would serve as a noise "cleanup" routine. If a device is connected to IR7, this noise detection scheme could still be used because an IR7 request from a device will set ISR7, while a noise-related request on IR7 will not affect ISR7. Thus, the IR7 routine could distinguish between the two events by reading the ISR and testing bit 7.

Interrupt System Based on Multiple 8259As

A multiple 8259A interrupt system is diagrammed in Fig. 4.16. In this figure data bus drivers are not shown, but they could be inserted. Although the SP/EN pin on the master 8259A is connected to the data bus transceivers, a 0 is applied to the SP/EN pins of the slaves. Only one slave is shown, but up to seven more slaves could be similarly connected into the system, permitting up to 64 distinct interrupt request lines. When designing the address decoder logic, each 8259A must be given its own address pair in the I/O address space. The drivers inserted in the CAS2-CASO lines may or may not be needed, depending on the proximity of the master to the slaves.

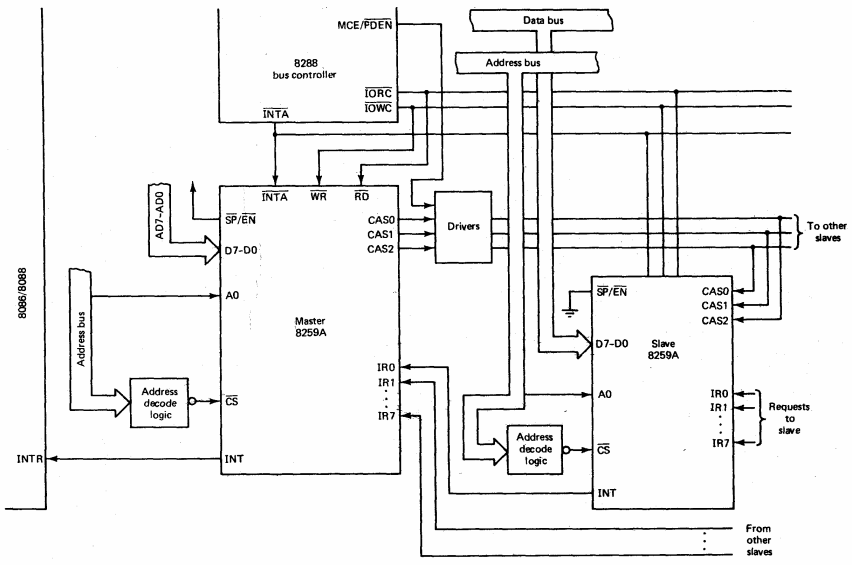


Figure 4.16

In a multiple 8259A system the slaves must be initialized as well as the master. The master would be initialized in the same way as indicated above except that SNGL would be set to 0 and ICW3 would need to be filled. A 1 would be put in each ICW3 bit for which the corresponding IR bit is connected to a slave and 0s would be put in the remaining bits. The SFNM bit may be set to 1 to activate the special fully nested mode. The SNGL bit should also be set to 0 when initializing the slaves. Thus, an ICW3 will be required for each slave, but for a slave ICW3 has a different meaning. For a slave, ICW3 has the form

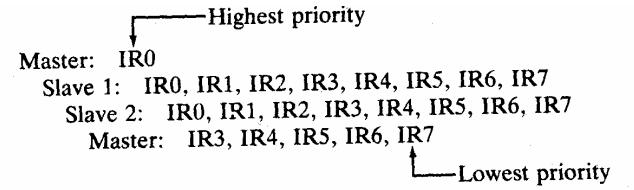
0	0	0	0	0	ID2	ID1	ID0

where the 3 least significant bits provide the slave with an identification number. The identification number given to the slave should be the same as the number of the master request line to which its INT pin is connected.

When a slave puts a 1 on its INT pin this signal is sent to the appropriate IR pin on the master. Assuming that the IMR and priority resolver do not block this signal, it is sent to the CPU through the INT pin on the master. When the CPU returns the INTA signal the master will not only set the appropriate ISR bit and clear the corresponding IRR bit, it will also check the corresponding bit in ICW3 to determine whether or not the interrupt came from a slave. If so, the master will place the number of the IR level on the CAS2-CASO lines; if not, it will put the contents of ICW2on the data bus and no signals will be applied to the CAS2- CASO lines. The INTA signal is also received by all of the slaves, but only that slave whose ID matches the number sent to it by the master over the CAS2-CASO lines will accept the signal. In the selected slave the appropriate ISR bit will be set, the corresponding IRR bit will be cleared, and its ICW2 will be put on the data bus. Because ICW2 contains the interrupt type, it is important that unique combinations be put in the master and slave ICW2s during the initialization process. EOI commands are required for both the master and the slave if their AEOI bits are 0.

Except for the response to the INTA signal discussed above, the actions taken by all 8259A devices in the system are the same. Also, their modes are controlled and their registers are read in the same way. There is one exception, however. If the SFNM bit in the ICW4 of the master is initialized to 1, the master will enter the special fully nested mode. This mode is ordinarily used with the normal priority mode and AEOI = 0. In this case, the master will allow unmasked requests of sufficient priority to be passed on to the INT pin even

if the corresponding ISR bit is already 1. This means that if a higher-priority request arrives at a slave while one or more of the slave's requests is being processed, the new request will be allowed to send its INT signal through the master. When using the special fully nested mode, two EOI commands may need to be sent by the interrupt routine. First, a nonspecific EOI command would be sent to the slave that caused the interrupt and then the slave's ISR would be tested. If and only if the ISR contains all 0s would a nonspecific EOI command be sent to the master. Therefore, assuming that slave 1 is connected to IRI and slave 2 is connected to IR2 of a master, they are the only slaves, and the highest priority in all three 8259As is assigned to IRQ, the fully nested order of priorities would be:



The masks in the master and slaves may, of course, be used to block out some of the requests.

Top

PC Bus and Interrupt System

The PC Bus uses a bus controller, address latches, and data transceivers (bidirectional data buffers).

- Bus controller:(Intel 8288 Bus Controller) coordinates activities on bus. It converts CPU status and clock signal into bus control signals. These control signals direct operations of latches, data transceivers, and the I/O bus
- Address latches: these are buffers for address lines. They serve two purposes, fill the speed gap between the CPU and other devices; and allow the CPU pins to be used for other purposes.
- Data transceivers: bidirectional data buffers

Interrupt processing: It follows the steps:

- When external device requests an interrupt, the CPU initiates a special sequence of bus cycles, called *interrupt-acknowledge sequence*
- The external device recognizes the interrupt-acknowledge sequence by decoding the bus control signals
- Once the external device recognizes the acknowledge, it then places the interrupt vector number on the data bus (through interrupt controller, in the case of IBM PC)
- After the CPU receives the interrupt vector, it begins the standard *interrupt-initiation sequence*: forming the interrupt vector address; then starting execution of the interrupt handler routine.

Intel 8259 interrupt controller: The 8088 processor has only two interrupt control inputs, nonmaskable interrupt (NMI) and interrupt request (INTR). NMI are interrupts that cannot be masked (examples: memory parity error, power loss). The interrupt controller has several functions:

- it receives interrupt requests from up to eight different sources.
- it prioritizes and queues interrupts that come at the same time
- it masks interrupt requests when instructed by the CPU

- it processes the CPU's interrupt-acknowledge signal by sending the interrupt vector number to the CPU.

Top

The Real Time Clock (RTC)

Calling the clock real-time is somewhat of a misnomer because it only reflects the time setting it has been given. The RTC is the other half of the chip that has the CMOS memory and can be thought of as a set of counters.

The first one counts from 0 to 9 and then tells the next counter (the 10's place counter) to count up once. The first counter then starts counting again and counts from 0 to 9 and again telling the next higher counter (the 10's counter) to set its counter up one more (now at 2), and so on.

Now the 10's place counter (which is counting 10's of seconds) only counts from 0 to 5 and then it tells the minute counter to do its increment, which will then start its journey from 0 to 9 and so on.

Of course the next counter after the minute counter is the minute 10's place counter, which also counts from 0 to 5 and then tells the hours counter to count once etc, etc.

The process goes on through the hour 10's place as it counts from 0 to 2, the day counter-which goes from 1 to 30, or 31, or sometimes 28 or maybe 29 depending on what the rules for which month has how many days and which years have 29 day Februarys. The month counter proceeds to go from 1 to 12, and then of course the year counter starts its journey with the good old 0 to 9, and finally we have the year 10's counter again with values going from 0 to 9.

Top

DMA

- **DMA** stands for **D**irect **M**emory **A**ccess
  - Uses same Address/Data lines on ISA bus
  - Controls the ISA bus instead of the processor ("bus master")
  - Floppy Drive (DRQ2)
  - Hard Drive (varies)
  - Sound Card (varies)
- DMA vs. Progammed I/O
  - Requires less processor load
  - Can continually transfer the same block (loop indefinitely)
  - Devices which are demand-based can be serviced more efficiently
- DMA characteristics:
  - 8-bit or 16-bit transfers

HOW DMA WORKS to transfer data from memory to a peripheral:

1. Peripheral Device requests DMA service by pulling DREQ line high
2. DMA controller requests CPU go on hold by pulling CPU's HRQ line high
3. CPU finishes current bus cycle, then acknowledges with HLDA

   The CPU relinquishes ALL bus control now (3-state)
4. DMA controller activates DACK line to Acknowledge DMA to the peripheral
5. DMA begin transferring data to the peripheral

Top

DMA Controller

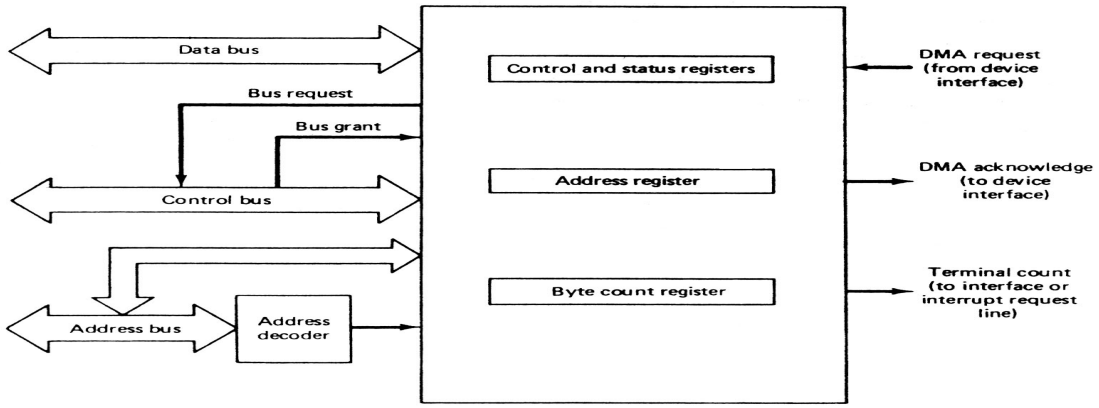


Figure 4.17: General Organization of a DMA Controller

Steps involved in transferring a block of data from I/O devices (e.g. a disk) to memory:

1. the CPU sends a signal to initiate a disk transfer through the I/O interface
2. the CPU sends the starting address of the block
3. the disk driver reads the starting address, and reads a block of data and puts it in its own buffer
4. the disk driver sends a interrupt signal to the CUP
5. the CPU reads the datum into its registers (accumulator)
6. the CPU checks if there is more to transfer, if yes, the CPU signals the disk driver to do so meanwhile
7. the CPU transfer the datum from register to memory and increments its pointer to memory
8. The DMA controller takes care of the last few steps (from signaling disk to transfer) .A

Intel's 8237 DMA controller:

- The 8237 has 4 independent I/O channels
- It has 27 registers, 7 of which are system-wide registers and 5 for each channels.
- out of 5 regs: 4 are 16-bit and 1 is 6-bit. they are

- 6-bit is the mode register
- DMA base address
- DMA current address
- DMA total
- DMA remaining

DMA Hardware (8237 DMAC)

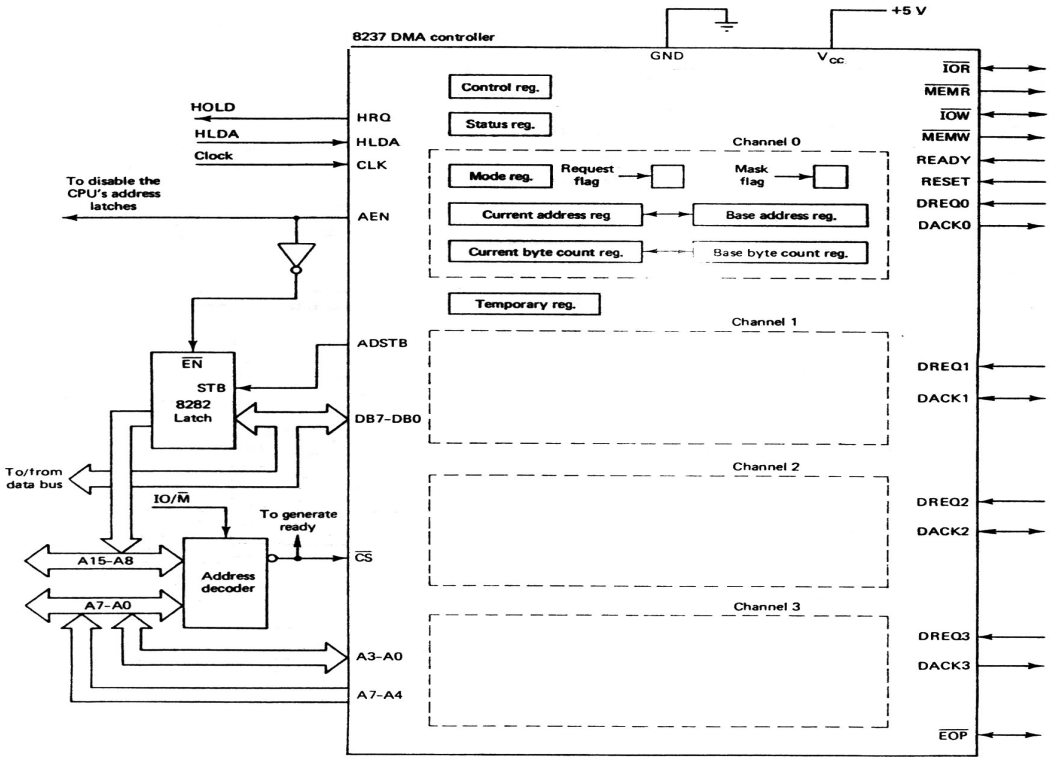


Figure 4.18: Organization of an 8237 and its associated logic

- Processor has HOLD/HOLD Acknowledge lines to interact with 8237
  - DMAC can gain control of ISA bus by asserting HOLD
  - Processor acknowledges with HLDA
- DRQ4 services slave controller
- Priorities are set as fixed
  - DRQ0 highest,DRQ7 lowest
  - Set at POST
  - Can be reprogrammed for rotating priority
- ISA address/data/control lines are also connected (not shown)

- Can access control registers through ports
- Each channel has a page register associated with it

8237 Modes

Intel 8237 can be set to four different style of transfer:

1. Single - One transfer at a time, allow processor access to the bus between transfers
2. Block - Transfer all data, do not allow processor access to the bus (may cause problems with memory refresh)
3. Demand - Keep transferring as long as target keeps DRQ asserted
4. Cascade - allow a slave controller use of the DMAC (used for DRQ4)

- In addition, the DMA controller can be set to make continuous transfers
  - known as auto-initialized DMA
  - normal DMA is known as "single-cycle"
- 8237 is clocked at 1/2 of ISA Bus (0.5 *BLCK)
  - up to 4.166MHz (8.33 Mhz ISA)
  - Maximum transfer rate: 4.166MB/s (16-bit DMA)
  - Maximum Programmed I/O transfer rate: 2.77 MB/s
- Size of transfer
  - Master can only generate word-sized transfers
  - Slave can generate byte-sized transfers
  - Minimum transfer size: 1 byte
  - Maximum transfer size: 64KB (8-bit),128KB (16-bit)

Student Activity 4.5

Answer the following questions.

1. Explain 8237.
2. What is advantage of DMA controller?
3. Write down the steps undivided in transferring a data using DMA?
4. Write down the modes of DMA controller 8236?

Summary

- I/O devices such as keyboards and displays establish communication of computer with outside world. Devices can be interfaced in two ways I/O MAPPED I/O and Memory mapped Memory.

- CPU controlled I/O comes in two ways. The difference is simply whether we use the normal memory addresses for I/O, this is referred to as physical memory mapped I/O.
- When the CPU detects an interrupt signal, it stops the current activity and jumps to a special routine, called an interrupt handler. This handler then detects why the interrupt occurred and takes the appropriate action.
- As an example of a serial interface device let us consider Intel's 8251 A programmable communication interface. It is capable of being programmed for asynchronous or synchronous communication. The data in buffer and data-out buffer registers share the same port address.
- Calling the clock real-time is somewhat of a misnomer because it only reflects the time setting it has been given. The RTC is the other half of the chip that has the CMOS memory and can be thought of as a set of counters.

Self-assessment Questions

Solved Exercises

I. True or False

1. Key board is a output device.
2. Address decoder determine whether I/O is memory mapped or I/O mapped.
3. Interface connecting I/O with memory.
4. Buses are only synchronous.
5. 2^{12} is 2 KB.

II. Fill in the blanks

1. HOLD & HLDA are used for used for _____in minimum mode.
2. H/IO is used for _____.
3. ALE is _____.
4. DEN is _____.
5. BHE is _____.

Answers

I. True or False

1. False
2. True
3. True
4. False
5. False

II. Fill in the blanks

1. DMA transfer
2. Distinguishes memory transfer and I/O transfer
3. Address Latch Enable
4. Data Enable
5. Bus High Enable

Unsolved Exercises

I. True or False

1. Registers in interface are input and control register.
2. Handshaking is used for storage.
3. Memory interface is used for connecting memory with CPU.
4. 8255 is a keyboard controller.
5. 8259 is a PPI.

II. Fill in the blanks

1. NMI is _____.
2. INTR is _____.
3. INTA is _____.
4. HLDA is _____.
5. WR is _____.

Detailed Questions

1. What is handshaking?
2. What is synchronous bus?
3. Which bus is suitable for long distance?
4. Difference between synchronous and asynchronous bus.
5. If an address bus has n lines, how much it can access memory.
6. What is the use of chip select logic?
7. If the data bus size is 16 bit, how many bytes it can fetch in 1 fetch.
8. What is the purpose of DMA controller?
9. What are the pins used for DMA transfer?
10. Write down the steps involved in transferring a data using DMA
11. Write down the modes of DMA controller 8237.

Unit 5

8086 Assembly Language Programming

Learning Objectives

After reading this unit you should appreciate the following:

- Instruction set of 8086
- Assembler Directives and Operators
- A Few Machine Level Programs
- Machine coding and Programs
- Programming with an Assembler
- Assembly Language Example Programs

Instruction set of 8086

The 8086/8088 instructions are categorized into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

(i) Data Copy/Transfer Instructions: This type of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) Arithmetic and Logical Instructions: All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) Branch Instructions: These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

(iv) Loop Instructions: If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) Machine Control Instructions: These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) Flag Manipulation Instructions: All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD,STD,CLI,STI, etc. belong to this category of instructions.

(vii) Shift and Rotate Instructions: These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

(viii) String Instructions: These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

Data Copy/Transfer Instructions

MOV: Move This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general or special purpose registers or a memory location and, another register or memory location may act as destination.

However, in case of immediate addressing mode, a segment register cannot be a destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register. The following example instructions explain the fact.

Example 5.1

Load DS with 5000H.

1. `MOV DS, 5000H;` Not permitted (invalid)

   Thus to transfer an immediate data into the segment register, the correct procedure is given below.

2. `MOV AX, 5000H`

   `MOV DS, AX`

   It may be noted, here, that both the source and destination operands cannot be memory locations (Except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.

3. `MOV AX, 5000H;` Immediate
4. `MOV AX, BX;` Register
5. `MOV AX, [SI];` Indirect
6. `MOV AX, [200 OH];` Direct
7. `MOV AX, 50H [BX];` Based relative, 50H Displacement

PUSH: Push to Stack This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address. The examples of these instructions are as follows:

Example 5.2

1. `PUSH AX`
2. `PUSH DS`
3. `PUSH [500OH]` ; Content of location 5000H and 5001 H in DS are pushed onto the stack.

POP: Pop from Stack This instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

The examples of these instructions are as shown:

Example 5.3

```
1. POP  AX

2. POP  DS

3. POP  [5000H]
```

XCHG: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted. The examples are as shown:

Example 5.4

```
1.  XCHG        [5000H], AX   ; This instruction exchanges data between AX and a
                              ; memory location [5000H] in the data segment.

2.  XCHG        BX            ; This instruction exchanges data between AX and BX.
```

IN: Input the port This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit) which is allowed to carry the port address. The examples are given as shown:

Example 5.5

```
1.  IN AL, 030 OH ; This instruction reads data from an 8-bit port whose address
                  ; is 0300H and stores it in AL.

2.  IN AX         ; This instruction reads data from a 16-bit port whose
                  ; address is in DX (implicit) and stores it in AX.
```

OUT: Output to the Port This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8-D_{15} while that to an even addressed port is transferred on D_0-D_7. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. The examples are given as shown:

Example 5.6

```
1.  OUT         0300H, AL     ; This sends data available in AL to a port whose
                              ; address is 0300H.

2.  OUT         AX            ; This sends data available in AX to a port whose
                              ; address is specified implicitly in DX.
```

XLAT: Translate The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique. We will explain this instruction with the aid of the following example.

Suppose, a hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned in AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 7-segment code corresponding to the key pressed is found out and sent to the display port. This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H, and store the 7-segment codes for 0 to Fat the locations 2000H to 200FH sequentially. For executing the XLAT instruction, the code of the pressed key obtained from the keyboard (i.e. the code to be translated) is moved in AL and the base address of the look up table containing the 7-segment codes is kept in BX. After execution of the XLAT instruction, the 7-segment code corresponding to the pressed key is returned in AL, replacing the key code which was in AL prior to the execution of the XLAT instruction. To find out the exact address of the 7-segment code from the base address of look up table, the content of AL is added to BX internally, and the contents of the address pointed to by this new content of BX in DS are transferred to AL. The following sequence of instructions perform the task.

Example 5.7

```
MOV AX, SEG TABLE          ;Address of the segment containing look-up-table
MOV DS, AX                 ; is transferred in DS.
MOV AL, CODE               ; Code of the pressed key is transferred in AL.
MOV BX, OFFSET TABLE       ; Offset of the code look-up-table in BX.
XLAT                       ; Find the equivalent code and store in AL.
```

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is more useful for assembly language rather than for machine language. Suppose, in an assembly language program, a label ADR is used. The instruction LEA BX, ADR loads the offset of the label ADR in BX.

LDS/LES: Load Pointer to DS/ES The instruction , Load DS/ES with pointer, loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The example in Figure 5.1 explains the operation.

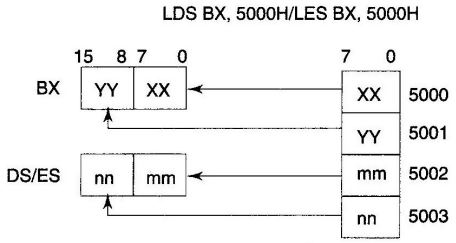


Figure 5.1: LDS/LES Instruction Execution

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except over flow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

Arithmetic Instructions

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The 8086/ 8088 instructions falling under this category are discussed below in significant details. The arithmetic instructions affect all the condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode.

ADD: Add This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected, depending upon the result. The examples of this instruction are given along with the corresponding modes.

Example 5.8

1.	`ADD AX, 0100H`	Immediate
2.	`ADD AX, BX`	Register
3.	`ADD AX, [SI]`	Register indirect
4.	`ADD AX, [5000H]`	Immediate
5.	`ADD [5000H] 0100H`	Immediate
6.	`ADD 0100H`	Destination AX (implicit)

ADC: Add with Carry This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example 5.9

1.	`ADD 0100H`	Immediate (AX implicit)
2.	`ADD AX, BX`	Register
3.	`ADD AX, [SI]`	Register indirect
4.	`ADD AX, [5000H]`	Direct
5.	`ADD [5000H] 0100H`	Immediate

INC: Increment This instruction increments the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction. The examples of this instruction are as follows:

Example 5.10

1.	`INC`	`AX`	Register
2.	`INC`	`[BX]`	Register indirect
3.	`INC`	`[5000H]`	Direct

DEC: Decrement The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction. The examples of this instruction are as follows:

Example 5.11

1.	`DEC`	`AX`	Register
2.	`DEC`	`[500OH]`	Direct

SUB: Subtract The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand can not be an immediate data. All the condition code flags are affected by this instruction. The examples of this instruction along with the addressing modes are as follows:

Example 5.12

1.	`SUB`	`0100H`	Immediate [destination AX]
2.	`SUB`	`AX, BX`	Register
3.	`SUB`	`AX, [5000H]`	Direct
4.	`SUB`	`[5000H], 0100`	Immediate

SBB: Subtract with Borrow The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (Condition code) by this instruction. The examples of this instruction are as follows:

Example 5.13

1.	`SBB`	`O100H`	Immediate [destination AX]
2.	`SBB`	`AX.BX`	Register
3.	`SBB`	`AX, [5000H]`	Direct

4. `SBB [5000H], 0100` Immediate

CMP: Compare This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

Example 5.14

1.	`CMP BX, 0100H`	Immediate
2.	`CMP 0100`	Immediate [AX implicit]
3.	`CMP [5000H],OIOOH`	Direct
4.	`CMP BX, [SI]`	Register indirect
5.	`CMP BX, CX`	Register

AAA: ASCII Adjust After Addition The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by I, the AF and CF flags are set to I, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Figure 5.2. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

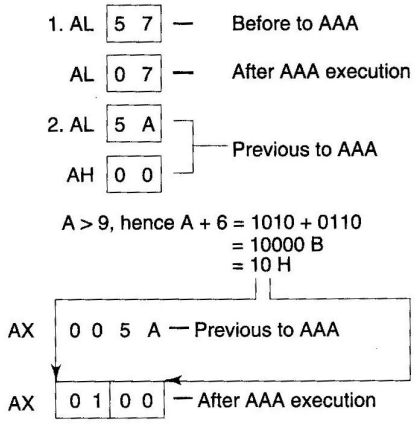


Figure 5.2: ASCII Adjust After Addition Instruction

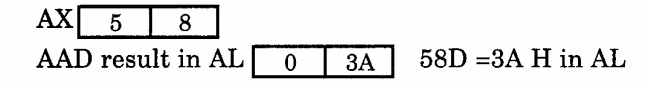
AAS: ASCII Adjust AL After Subtraction AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9.

The procedure is similar to the AAA instruction. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

AAM : ASCII Adjust for Multiplication This instruction, after execution , converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH. The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL = 5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add 6 (0110) to it D + 6 = 13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1 = 6 will be the upper unpacked byte of the result. Thus after the execution, AH = 06 and AL = 03.

AAD: ASCII Adjust for Division Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contain 0508 unpacked BCD for 58 decimal, and DH contain 02H.

Example 5.15



The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Note that 3A is the hexadecimal equivalent of 58 (decimal). Now, instruction DIV DH may be executed. So rather than ASCII adjust for division, it is ASCII adjust before division. All the ASCII adjust instructions are also called as unpacked BCD arithmetic instructions. Now, we will consider the two instructions related to packed BCD arithmetic.

DAA: Decimal Adjust Accumulator This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set , it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL. The examples given below explain the instruction.

Example 5.16

```
(i) AL = 53          CL = 29
    ADD AL, CL       ; AL ← (AL) + (CL)
                     ; AL ← 53 + 29
                     ; AL ← 7C
    DAA              ; AL ← 7C + 06 (as C>9)
                     ; AL ← 82

(ii) AL = 73         CL = 29
    ADD AL,CL        ; AL ← AL + CL
                     ; AL ← 73 + 29
                     ; AL ← 9C
    DAA              ; AL ← 02 and CF = 1
```

```
AL = 7 3
     +
CL = 2 9
     ---
       9 C
     + 6
     ---
       A 2
     + 6 0
     -----
CF = 1   0 2   in AL
```

The instruction DAA affects AF, CF, PF, and ZF flags. The OF is undefined.

DAS: Decimal Adjust After Subtraction This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

Example 5.17

```
(i)  AL = 75          BH = 46
     SUB AL,BH        ; AL ← 2 F = (AL) – (BH)
                      ; AF = 1
     DAS              ; AL ← 2 9 (as F> 9, F – 6 = 9)
(ii) AL = 38          CH = 6 1
     SUB AL,CH        ; AL ← D 7   CF = 1 (borrow)
     DAS              ; AL ← 7 7   (as D> 9, D – 6 = 7)
                      ; CF = 1 (borrow)
```

NEG: Negate: The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand, which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

MLIL: Unsigned Multiplication Byte or Word: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The example instructions are as shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' CF and OF both will be set.

Example 5.18

```
1. MUL BH                ; (AX) ← (AL) x (BH)
2. MUL CX                ; (DX) (AX) ← (AX) × (CX)
3. MUL WORD PTR [SI]     ; (DX) (AX) ← (AX) × ([SI])
```

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result

respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. Sign bit and CF fill the unused higher bits of the result, AF are cleared. The example instructions are given as follows:

Example 5.19

```
1.  IMUL        BH
2.  IMUL        CX
3.  IMUL        [SI]
```

CBW: Convert Signed Byte to Word: This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to Double Word: This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

DIV: Unsigned Division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

IDIV: Signed Division: This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by 0 interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation). All the flags are undefined after IDIV instruction.

Logical Instructions

These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR. The instruction for each of these operations is discussed as follows.

AND: Logical AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand. The examples of this instruction are as follows:

Example 5.20

```
1. AND AX, 0008H
2. AND AX, BX
3. AND AX, [5000H]
4. AND [5000H], DX
```

If the content of AX is 3FOFH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

0011	1111	0000	1111	= 3F0F H [AX]
↓↓↓↓	↓↓↓↓	↓↓↓↓	↓↓↓↓	AND
0000	0000	0000	1000	= 0008 H
0000	0000	0000	1000	= 0008 H [AX]

The result 0008H will be in AX.

OR: Logical OR: The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation. The examples are as follows:

Example 5.21

```
1.  OR  AX,          0098H
2.  OR  R AX,        BX
3.  OR  R AX,        [5000H]
4.  OR [5000H],      0008H
```

The contents of AX are say 3FOFH, then the first example instruction will be carried out as given below.

0011	1111	0000	1111	= 3F0F H [AX]
↓↓↓↓	↓↓↓↓	↓↓↓↓	↓↓↓↓	AND
0000	0000	0000	1000	= 0008 H
0000	0000	0000	1000	= 0008 H [AX]

Thus the result 3F9FH will be stored in the AX register.

NOT: Logical Invert: The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example 5.22

```
NOT   AX
NOT   [5000H]
```

If the content of AX is 200FH, the first example instruction will be executed as shown.

AX	= 0010	0000	0000	1111
invert	↓↓↓↓	↓↓↓↓	↓↓↓↓	↓↓↓↓
	1101	1111	1111	0000
Result in AX =	D	F	F	0

The result DFFOH will be stored in the destination register AX.

XOR: Logical Exclusive OR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

The example instructions are as follows:

Example 5.23

```
1. XOR      AX,0098H
2. XOR      AX, BX
3. XOR      AX, [5000H]
```

If the content of AX is 3FOFH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

AX = 3F0FH =	0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1
XOR	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
0098H =	0 0 0 0	0 0 0 0	1 0 0 1	1 0 0 0
AX = Result =	0 0 1 1	1 1 1 1	1 0 0 1	0 1 1 1
= 3F97H				

TEST: Logical Compare Instruction: The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to I, if the corresponding bits of both operands are I, else the result bit is reset to 0. The result of this anding operation is not available for further use) but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

Example 5.24

```
1.  TEST        AX,  BX
2.  TEST        [0500],  06H
3.  TEST        [BX]   [Dl],  CX
```

SHL/SAL: Shift logical/Arithmetic Left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 5.3 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

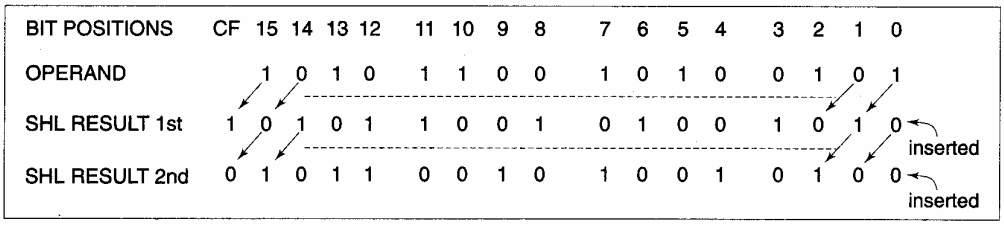


Figure 5.3 Execution of SHL/SAL Instruction

SHR: Shift Logical Right: This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. Figure 5.4 explains execution of this instruction. This instruction shifts the operand through carry flag.

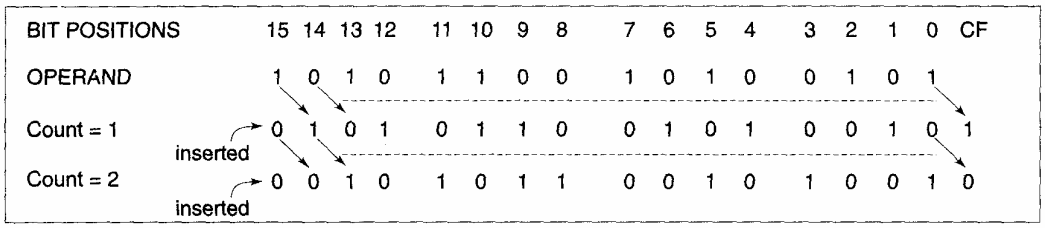


Figure 5.4: Execution SHR Instruction

SAR: Shift Arithmetic Right: This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. Figure 5.5 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through carry flag.

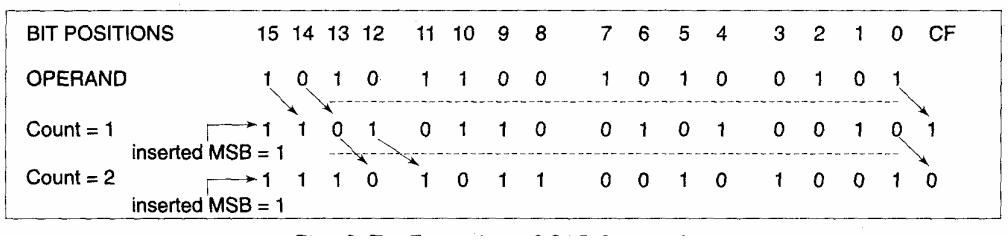


Figure 5.5: Execution of SAR Instruction

Immediate operand is not allowed in any of the shift instructions.

ROR: Rotate Right without Carry: This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. Figure 5.6 explains the operation. The destination operand may be a register (except a segment register) or a memory location.

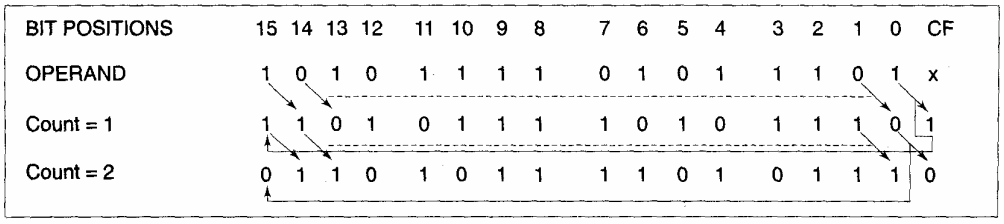


Figure 5.6: Execution of ROR Instruction

ROL: Rotate Left without Carry: This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged by this rotate operation. The

operand may be a register or a memory location. Figure 5.7 explains the operation.

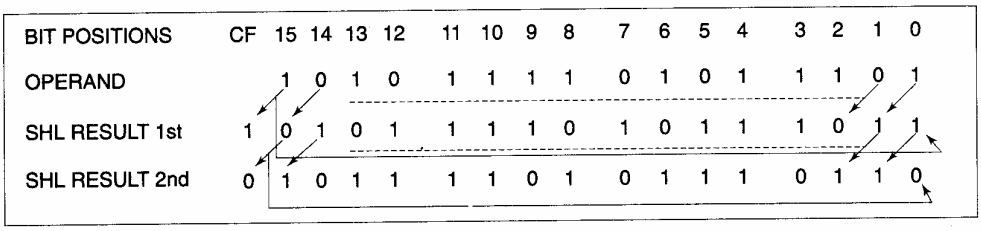


Figure 5.7: Execution of ROL Instruction

RCR: Rotate Right through Carry: This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 5.8 explains this operation.

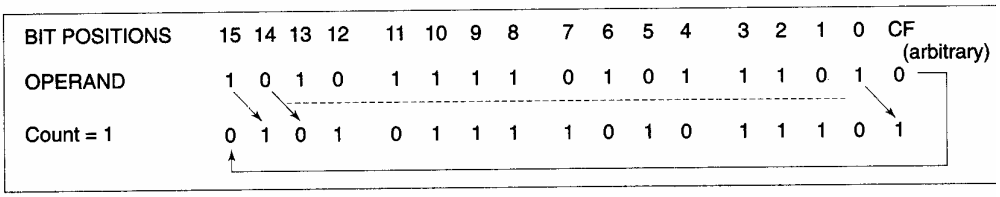


Figure 5.8: Execution of RCR Instruction

RCL: Rotate Left through Carry: This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB , and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left operation. The operand may be a register or a memory location. Figure 5. 9 explains the operation

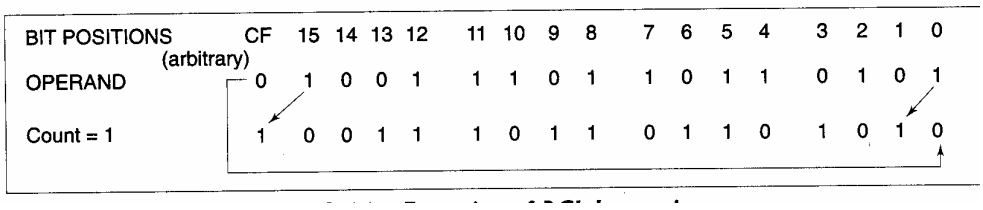


Figure 5.9: Execution of R.CL Instruction

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required,

(a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements. The pointers and counters may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the direction flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation , the index registers are updated by two. The counter in both the cases, is decremented by one.

REP: Repeat Instruction Prefix: This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ, i.e. repeat operation while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word: Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (source index) and DS (data segment) contents. The starting address of the destination locations where this string has to be relocated is given by Dl (destination index) and ES (extra segment) contents. The starting address of the source string is 10H*DS+[SI], while the starting address of the destination string is 10H*ES+[DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/ words pointed to by DS: SI pair (source) to the memory location pointed to by ES: Dl pair (destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and Dl depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all the string manipulation instructions. The following string of instructions explain the execution of the MOVS instruction.

Example 5.25

```
MOV AX,5000H        ; Source segment address is 5000h.
MOV DS,AX           ; Load it to DS.

MOV AX,6000H        ; Destination segment address is 6000h.
MOV ES,AX           ; Load it to ES.
MOV CX,0FFH         ; Move length of the string to counter register CX.
MOV SI,1000H        ; Source index address 1000H is moved to SI.
MOV DI,2000H        ; Destination index address 2000H is moved to DI.
CLD                 ; Clear DF, i.e. set autoincrement mode.
REP MOVSB           ; Move 0FFH string bytes from source address to destination .
```

CMPS: Compare String Byte or String Word The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX(counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explain the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the

direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Example 5.26

```
MOV AX,SEG1              ; Segment address of STRING1, i.e. SEG1 is moved to AX.
MOV DS,AX                ; Load it to DS.
MOV AX,SEG2              ; Segment address of STRING2, i.e. SEG2 is moved to AX.
MOV ES,AX                ; Load it to ES.
MOV SI,OFFSET STRING1    ; Offset of STRING1 is moved to SI.
MOV DI,OFFSET STRING2    ; Offset of STRING2 is moved to DI.
MOV CX,010H              ; Length of the string is moved to CX.
CLD                      ; Clear DF, i.e. set autoincrement mode.
REPE CMPSW               ; Compare 010H words of STRING1 and
                         ; STRING2, while they are equal, If a mismatch is found,
                         ; modify the flags and proceed with further execution .
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

SCAS: Scan String Byte or String Word: This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found. The following string of instructions elaborates the use of SCAS instruction.

Example 5.27

```
        MOV AX,SEG       ; Segment address of the string, i.e. SEG is moved to AX.
        MOV ES,AX        ; Load it to ES.
        MOV DI,OFFSET    ; String offset, i.e. OFFSET is moved to DI.
        MOV CX,010H      ; Length of the string is moved to CX.
        MOV AX,WORD      ; The word to be scanned for, i.e. WORD is in AL.
        CLD              ; Clear DF.
REPNE   SCASW            ; Scan the 010H bytes of the string , till a match to
                         ; WORD is found.
```

This string of instructions finds out, if it contains WORD. If the WORD is found in the word string , before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the programme proceeds further.

LODS: Load String Byte or String Word: The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word: The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES : Dl register pair. The Dl is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index D1 are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and Dl are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending

addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, S1 and D1 are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses.

Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes, the CS may or may not be modified. These type of instructions are classified in two types:

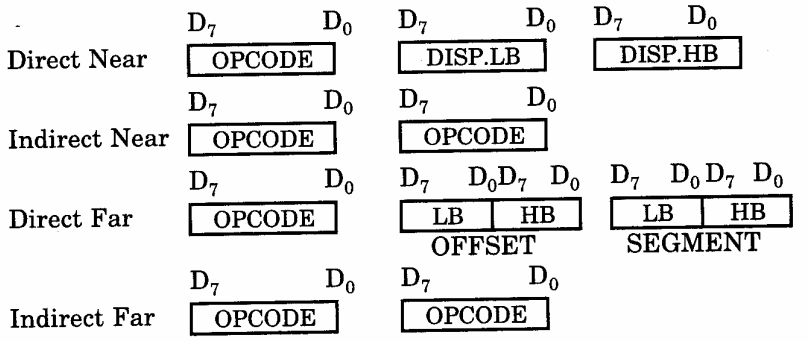
Unconditional Control Transfer (Branch) Instructions: In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

Conditional Control Transfer (Branch) Instructions: In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. Condition code flags replicate the results of the previous operations.

In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Instructions

CALL: Unconditional Call: This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e ± 32K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for them are respectively called as intrasegment and intersegment addressing modes. This instruction comes under unconditional branch instructions and can be described as shown with the coding formats. On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.



RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure . In case of a FAR procedure, the current

contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from OOH to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (Nx4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine.

For the execution of this instruction, the IF must be enabled.

Example 5.28

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT        20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

Figure 5.10 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

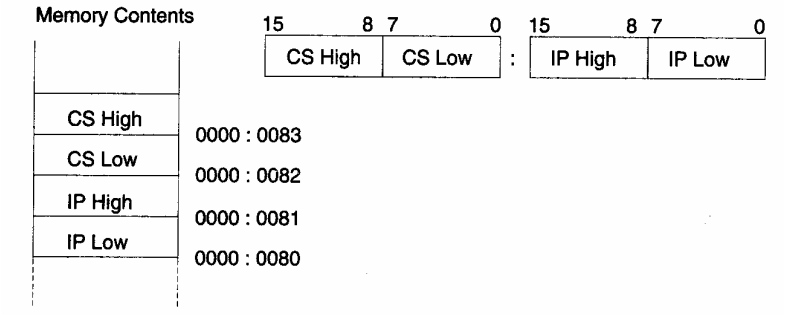
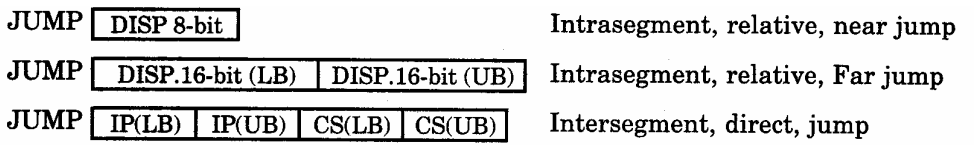


Figure 5.10: Contents of IVT

INTO: Interrupt on Overflow: This is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0000 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

JMP: Unconditional lump: This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS : IP (intersegment direct far). No flags are affected by this instruction. Corresponding to the three methods of specifying jump addresses, the JUMP instruction has the following three formats.



IRET: Return from ISR: When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop Unconditionally: This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMF IF NOT ZERO structure.

Example 5.29

```
        MOV   CX, 0005     ; Number of times in CX
        MOV   BX,0FF7H     ; Data to BX
Label : MOV   AX, CODE1
        OR    BX, AX
        AND   DX, AX
        Loop  Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. Lf CX is already OOH, the execution continues sequentially. No flags are affected by this instruction.

Conditional Branch Instructions

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within -80H to 7FH (or -128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 5.1.

Table 5.1: Conditional Branch Instructions

	Mnemonic	*Displacement*	*Operation*
1.	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1
2.	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0
3.	JS	Label	Transfer execution control to address 'Label' , if SF=1
4.	JNS	Label	Transfer execution control to address 'Label', if SF=0
5.	JO	Label	Transfer execution control to address 'Label', if OF=1
6.	JNO	Label	Transfer execution control to address 'Label', if OF=0
7.	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1
8.	JNP	Label	Transfer execution control to address 'Label', if PF=0
9.	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1
10.	JNB/JAE/JNC	Label	Transfer execution control to address 'Label', if CF=0
11.	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0.
15.	JLE/JNC	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or atleast any one of SF and OF is 1(Botn SF & OF are not 0).

The last four instructions are used in case of decisions based on signed binary number operations, while the remaining instructions can be used for unsigned binary operations. The terms above and below are generally used for unsigned numbers, while the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

```
JCXZ      'Label'     Transfer execution control
                      to address 'Label', if CX=0.
```

The conditional LOOP instructions are given in Table 5.2 with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

Table 5.2: Conditional Loop Instructions

Mnemonic	Displacement	Operation
LOOPZ/LOOPE	Label	Loop through a sequence of instructions from 'Label' while ZF=1 and CX f 0.
LOOPNZ/LOOPENE	Label	Loop through a sequence of instructions from 'Label' while ZF=0 and CX f 0.

The ideas about all these instructions will be more clear with programming practice. This topic is aimed at introducing these instructions to readers. Of course, examples are quoted wherever possible, but the JUMP and the LOOP instructions require a sequence of instructions for explanations and they will be emphasized.

Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

`CLC` - Clear carry flag

`CMC` - Complement carry flag

`STC` - Set carry flag

`CLD` - Clear direction flag

`STD` - Set direction flag

`CLI` - Clear interrupt flag

`STI` - Set interrupt flag

These instructions modify the carry(CF), direction(DF) and interrupt(IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions, are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

`WAIT` - Wait for Test input pin to go low

`HLT` - Halt the processor

`NOP` - No operation

`ESC` - Escape to external device like NDP (numeric co-processor)

`LOCK` - Bus lock instruction prefix.

After executing the HLT instruction, the processor enters the halt state, as explained in Chapter 1. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Student Activity 5.1

Before reading the next section, answer the following question.

1. Bring out the difference between the jump and loop instructions.
2. Which instruction of 8086 can be used for look up table manipulations?

3. What is the difference between the respective shift and rotate instructions?

If your answer is correct, then proceed to the next section.

Assembler Directives and Operators

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he may be able to manage the memory of the system more efficiently. On the other hand, the disadvantages are more prominent. The programming, coding and resource management techniques are tedious. The programmer has to take care of all these functions hence the chances of human errors are more. The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and instruction set.

The assembly language programming is simpler as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement in assembly language over machine language is that the address values and the constants can be identified by labels. If the labels are suggestive, then certainly the program will become more understandable, and each time the programmer will not have to remember the different constants and the addresses at which they are stored, throughout the programs. The labels may help to identify the addresses and constants. Due to this facility, the tedious byte handling and manipulations are got rid of. Similarly, now different logical segments and routines may be assigned with the labels rather than the different addresses. The memory control feature of machine language programming is left unchanged by providing storage define facilities in assembly language programming. The documentation facility which was not possible with machine language programming is now available in assembly language.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks) an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or a variable, logical names of the segments, types of the different routines and modules, end of file, etc. These types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler.

DB: Define Byte The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initialises the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

Example 5.30

```
RANKS  DB OIH, 02H, 03H, 04H
```

This statement directs the assembler to reserve four memory locations for a list named RANKS and initialise them with the above specified four values.

```
MESSAGE DB 'GOOD MORNING
```

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialise those locations by the ASCII equivalent of these characters.

```
VALUE  DB  5 OH
```

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialised for the variable named VALUE.

DW: Define Word: The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Example 5.31

```
WORDS  DW   1234H, 4567H, 78ABH, 045CH,
```

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

```
WDATA  DW 5 DUP (6666H)
```

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initialises all the word locations with 6666H.

DQ: Define Quadword: This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

DT: Define Ten Bytes: The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

ASSUME: Assume Logical Segment Name: The ASSUME directive is used to inform the assembler, the names of the logicals segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS : DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialised by the segment address value decided by the operating system for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program, without which a message 'CODE/DATA EMITTED WITHOUT SEGMENT' may be issued by an assembler.

END: END of Program: The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

ENDP: END of Procedure In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a

name, i.e. label. To mark the end of a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitues the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, and the label can then be used in the program in place of that mnemonic. Suppose, a numerical constant appears in a program ten times. If that constant is to be changed at a later time, one will have to make all these ten corrections. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the EQU statement will give the correct and modified code. The examples given below show the syntax.

Example 5.32

```
LABEL      EQU    050 OH
ADDITION   EQU    ADD
```

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD. EXTRN: External and PUBLIC: Public The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULEI from MODULE 2; in MODULE 1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL . The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MODULE 2. Thus the MODULEI and MODULE 2 must have the following declarations.

```
MODULE1        SEGMENT
PUBLIC         FACTORIAL FAR
MODULE1        ENDS
MODULE2        SEGMENT
EXTRN          FACTORIAL FAR
MODULE2        ENDS
```

GROUP: Group the Related Segments: The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes an information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base.

```
PROGRAM GROUP CODE, DATA, STACK
```

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

```
ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.
```

LABEL: Label: The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initialises a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it

assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

```
CONTINUE    LABEL FAR
```

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA          SEGMENT

DATAS DB 5 OH DUP (?)

DATA-LAST LABEL BYTE FAR

DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATALAST and its type will be byte and far.

LENGTH: Byte Length of a Label: This directive is not available in MASM. This is used to refer to the length of a data array or a string.

```
MOV CX. LENGTH ARRAY
```

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

LOCAL The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that module. At a later time, some other module may declare a particular data type LOCAL, which is previously declared 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 shown.

```
LOCAL a, b, DATA, ARRAY, ROUTINE
```

NAME: Logical Name of a Module: 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 to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

OFFSET: Offset of a Label: When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The segment may also be decided by another operator of similar type, viz, SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

Example 5.33

```
CODE SEGMENT

MOV SI, OFFSET LIST

CODE ENDS

DATA SEGMENT
```

```
LIST DB IOH
DATA ENDS
```

ORG : Origin: The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement. While starting the assembly process for a module, the assembler initialises a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialised to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment. In other words, the location counter will get initialised to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

PROC: Procedure: The PROG directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e. whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

Example 5.34

```
RESULT   PROC     NEAR
ROUTINE PROC     FAR
```

PTR: Pointer: The 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, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16-bit quantity. In other words, the PTR operator is used to specify the data type - byte or word. The examples of the PTR operator are as follows:

Example 5.35

```
MOV AL, BYTE PTR [SI] -        Moves content of memory location
                               addressed by SI (8-bit) to AL
INC BYTE PTR [BX]-             Increments byte contents of memory
                               location addressed by BX
MOV BX, WORD PTR [2000H]-      Moves 16-bit content of memory
                               location 2000H to BX, i.e [2000H] to BL
                               [2001H] to BH

INC WORD PTR [3000H] -         Increments word contents of memory
                               location 3000H considering contents
                               of 3000H (lower byte) and 3001H
                               (higher byte) as a 16-bit number
```

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

```
JMP     WORD PTR [BX]-NEAR Jump
JMP     WORD PTR [BX]-FAR Jump
```

PUBLIC As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be

accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least any one of the other modules of the same program. (Refer to the explanation on EXTRN directive to get the clear idea of PUBLIC.)

SEG: Segment of a Label The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of 'SEG label". The example given below explain the use of SEG operator.

Example 5.36

```
MOV AX, SEG ARRAY     ; This statement moves the segment address of ARRAY in
MOV DS, AX            ; which it is appearing, to register AX and then to DS.
```

SEGMENT: Logical Segment: The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

```
EXE.CODE SEGMENT GLOBAL     ; Start of segment named EXE. CODE,
                            ; that can be accessed by any other module.
EXE.CODE ENDS               ; END of EXE.CODE logical segment.
```

SHORT The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory . Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

```
JMP SHORT LABEL
```

TYPE The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

GLOBAL: The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC    GLOBAL
```

Student Activity 5.2

Before reading the next section, answer the following question.

1. How will you enter the single step mode of 8086?
2. What is LOCK prefix? What is its use?
3. What is REP prefix? What is its use?

If your answer is correct, then proceed to the next section.

A Few Machine Level Programs

In this section, a few machine level programming examples, rather, instruction sequences are presented for comparing the 8086 programming with that of 8085. These programs are ii the form of instruction sequences just like 8085 programs. These may even be hand-coded entered byte by byte and executed on an 8086 based system but due to the complex instruction set of 8086 and its tedious opcode conversion procedure, most of the programmers prefer to use assemblers. However, we will briefly discuss the hand-coding,

Example 5.37

Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

Solution

The flow chart for this problem may be drawn as shown in Figure 5.11

```
MOV  AX, 2000H   ; Initialising DS with value
MOV  DS, AX      ; 2000H.
MOV  AX, [500H]  ; Get first data byte from 0500H offset.
ADD  AX, [600H]  ; Add this to the second byte from 0600H.
MOV  [700H],AX   ; Store AX in 0700H (result).
HLT              ; Stop.
```

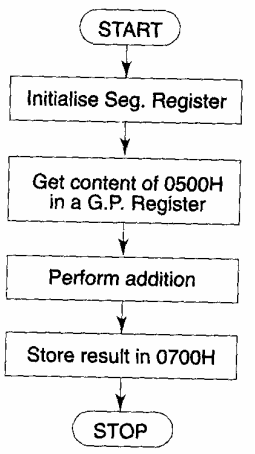


Figure 5.11: Flow Chart

The above instruction sequence is quite straight-for-ward. As the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose resistors, say AX. and then the register general purpose registers, say AX, and then the register content is moved to the segment register DS. Thus the data segment register DS contains 2000H. The instruction MOV AX,[500H] signifies that the contents of the particular location, whose offset is specified in the brackets with the segment pointed to by DS as segment register, is to be moved to AX. The MOV [0700], AX instruction moves the contents of the register AX to an offset 0700H in DS (DS = 2000H). Note that the code segment register CS gets automatically loaded by the code segment address of the program whenever it is executed. Actually it is the monitor program that accepts the CS:IP address of the program and passes it to the corresponding registers at the time of execution. Hence no instructions are required for loading the CS register like DS or SS.

Example 5.38

Write a program to move the contents of the memory location 0500H to register BX and also to CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using DS=2000H and offset=0600H. Store the result of the addition in 0700H. Assume that the data is located in the segment specified by the data segment register which contain 2000H.

Solution

The flow chart for the program is shown in Figure 5.12.

After initializing the data segment register the content of location 0500H are moved to the BX register using MOV instruction. The same data is moved also to the CX register. For this data transfer, there may be two options as shown.

```
MOV AX, 2000H
MOV DS, AX          ; Initialize data segment register.
MOV BX,[0500H]      ; Get contents of 0500H in BX.
MOV CX, BX          ; Copy the same contents in CX.
ADD [0600H],05H     ; Add byte 05H to contents of 0600H.
MOV DX, [0600H]     ; Store the result in DX.
MOV [0700H],DX      ; Store the result in 0700H.
HLT                 ; Stop.
```

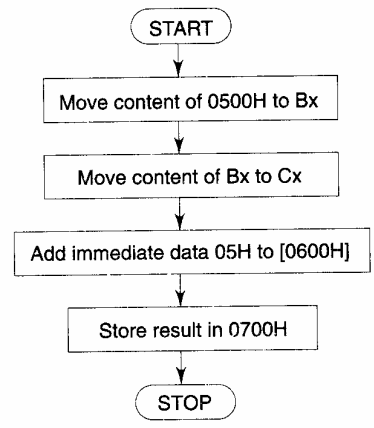


Figure 5.12 Flow Chart

(a) `MOV CX, BX` ; As the contents of BX will be same as 0500H after execution
; of MOV BX,[0500H].

(b) `MOV CX, [050OH]` ; Move directly from 0500H to register CX

The opcode in the first option is only of 2 bytes, while the second option will have 4 bytes of opcode. Thus the second option will require more memory and execution time. Due to these reasons, the first option is preferable.

The immediate data byte 05H is added to content of 0600H using the ADD instruction. The result will be in the destination operand 0600H. This is next stored at the location 0700H. In case of the 8086/8088 instruction set, there is no instruction for the direct transfer of data from the memory source operand to the memory destination operand except, the string instructions. Hence the result of addition which is present at 0600H, should be moved to any one of the general purpose registers, except BX and CX, otherwise the contents of CX and BX will be changed. We have selected DX (we could have selected AX also, because once DS is initialised to 2000H the contents of AX are no longer useful for this purpose. Thus the transfer of result from 0600H to 0700H is accomplished in two stages using successive MOV instructions, i.e., at

first, the content of 0600H is DX and then the content of DX is moved to 0700H. The program ends with the HLT instruction.

Example 5.39

Add the contents of the memory location 2000H:0500H to contents of 3000H:0600H and store the result in 5000H:0700H.

Solution

Unlike the previous example programs, this program refers to the memory locations in different segments, hence, while referring to each location, the data segment will have to be newly initialized with the required value. Figure 5.13 shows the flow chart.

The instruction sequence for the above flow chart is given along with the comments.

```
MOV CX, 2000H      ; Initialize DS at 2000H.
MOV DS, CX
MOV AX, [500H]     ; Get first operand in AX.
MOV CX, 3000H      ; Initialize DS at 3000H.
MOV DS, CX
MOV BX, [0600H]    ; Get second operand in BX.
ADD AX, BX         ; Perform addition.
MOV CX, 5000H      ; Initialize DS at 5000H.
MOV DS, CX
MOV [0700H],AX     ; Store the result of addition in
HLT                ; 0700H and stop.
```

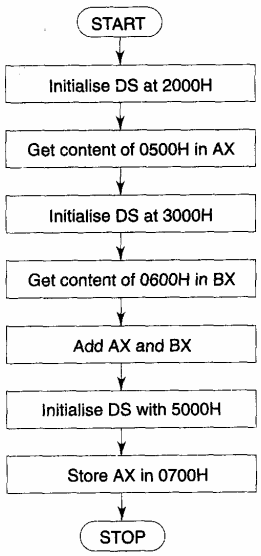


Figure 5.13: Flow Chart

Actually, the program simply performs the addition of two operands which are located in different memory segments. The program has become lengthy only due to data segment register initialization instructions.

Example 5.40

Move a byte string , 16-bytes long, from the offset 0200H to 0300H in the segment 7000H.

Solution

According to the program statement, a string that is 16-bytes long is available at the offset address 0200H in the segment 7000H. The required program should move this complete string at offset 0300H, in the same segment. Let us emphasize this program in the light of comparison between 8085 and 8086 programming techniques.

An 8085 program to perform this task, is given neglecting the segment addresses.

```
        MVI C, 010H    ; Count for the length of string
        LXIH 0200H     ; Initialization of HL pair for source string
        LXID 0300H     ; Initialization of DE pair for destination
BACK :  MOV A, M       ; Take a byte from source in A.
        STAX D         ; Store contents of A to address pointed to by DE pair.
        INX H          ; Increment source pointer.
        INX D          ; Increment destination pointer.
        DCRC           ; Decrement counter.
        JNZ BACK       ; Continue if counter is not zero.
        HLT            ; Stop if counter is zero.
```

The programmers, with fluent hands on 8085 assembly language programming but starting with 8086, may translate the above 8085 assembly language program listings to 8086 assembly language programs using the analogous or comparable instructions. Of course, this method of programming is not efficient, however, it may help those who are familiar to 8085 programming and wish to start writing programs in 8086 assembly language. The reason for the inefficiency of this method is that the special features and capabilities of 8086 have not been taken into account while preparing the 8086 assembly language program. Now, let us think about how the above program may be transferred to 8086 assembly language using analogous instructions. Note that the segment initialization is to be added. Let us consider that the code and data segment address is 7000H. Consider that the code starts at offset 0000H.

```
        MOV AX, 7000H
        MOV DS, AX          ; Data segment initialization
        MOV SI, 0200H       ; Pointer to source string
        MOV DI, 0300H       ; Pointer to destination string
        MOV CX, 0010H       ; Count for length of string
BACK :  MOV AX, [SI]        ; Take a source byte in AX.
        MOV [DI],AX         ; Move it to destination.
        INC SI              ; Increment source pointer.
        INC DI              ; Increment destination pointer.
        DEC CX              ; Decrement count by 1.
        JNZ BACK            ; Continue if count is not 0.
        HLT                 ; Stop if the count is 0.
```

The above listing has been prepared using the program written in 8085 ALP. Indexed addressing mode is used for string byte accesses and transfer in this case. The functions of all the 8086 instructions and the 8086 addressing modes have already been explained in Unit 2. In this program, all the instructions used are more or less analogous to the 8085 program, and the special software capabilities of 8086 like string instructions and loop instructions have not been considered. The 8086 programs based on 8085 codes are inefficient due to the reason that the full capability of the rich 8086 instruction set and the enhanced architecture of 8086 cannot be fully exploited.

The above program uses the decrement and jump-if-not-zero instructions for checking whether the transfer is complete or not. The 8086 instruction set provides LOOP instructions for this purpose. Using these instructions, the program is modified as shown.

```
        MOV AX,7000H
        MOV DS,AX          ; Data segment initialization
        MOV SI,0200H       ; Source pointer initialization
        MOV DI,0300H       ; Destination pointer initialization
        MOV CX,0010H       ; Counter initialization
BACK :  MOV AX,[SI]        ; Take a byte of string from source
        MOV [DI],AX        ; and then move it to destination
        INC SI             ; Update source pointer
        INC DI             ; Update destination pointer continue
        LOOP BACK          ; till CX=0,[DEC CX and JNZ BACK]
        HLT                ; Stop if CX=0
```

Thus the two instructions bracketed in the comment field are replaced by a single loop instruction which results in the saving of memory and execution time. The loop instruction needs the additional instructions for updating the pointers (for example, INC SI, INC Dl). It does not need counter decrement and check-if-zero instruction.

One more feature of the 8086 instruction set is the string instruction, i.e. MOVSB and MOVSW. Using these instructions one can move a string byte/word from source to destination. The length of the string is specified by the CX register. The SI and Dl point to the source and destination locations. The DS and ES registers should be initialised to source and destination segment addresses respectively. Before the use of string instructions, the program should initialise all these registers properly. Using the string byte instruction the same program may be written as shown.

```
        MOV AX, 7000H
        MOV DS, AX         ; Source segment initialisation
        MOV ES,AX          ; Destination segment initialisation
        MOV CX,0010H       ; Counter initialisation
        MOV SI,0200H       ; Source pointer initialisation
        MOV DI,0300H       ; Destination pointer initialisation
        CLD                ; Clear DF
REP     MOVSB              ; Move the complete string
        HLT                ; Stop
```

The MOVSB instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. An experienced programmer will thus directly use the string instructions instead of using other options. The flow chart of the final program is presented in Figure 5.14.

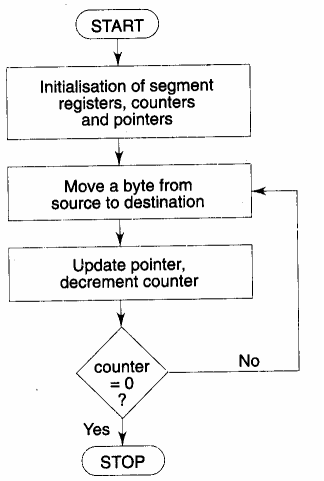


Figure 5.14: Flow Chart

Example 5.41

Find out the largest number from an unordered array of sixteen 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H. Solution The logic for this procedure can be described as follows. The first number of the array is taken in a register, say AX. The second number of the array is then compared with the first one. If the first one is greater than the second one, it is left unchanged. However, if the second one is greater than the first, the second number replaces the first one in the AX register. The procedure is repeated for every number in the array and thus it requires 15 iterations. At the end of 15^{th} iteration the largest number will reside in the register AX. This may be represented in terms of the flow chart as shown in Figure 5.15. The listing is given below.

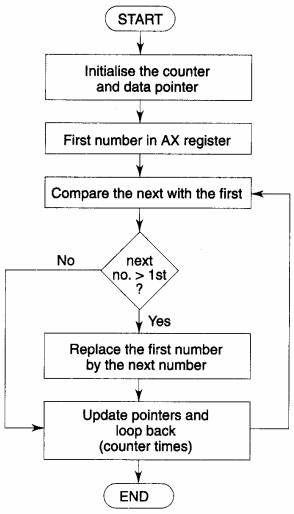


Figure 5.15: Flow Chart

Student Activity 5.3

Before reading the next section, answer the following question.

1. Write a program to move the contents of the memory location 0700H to register BX and also to CX. Add immediate byte 10H to the data residing in memory location, whose address is computed using DS=3000H and offset=0200H. Store the result of the addition in 0500H. Assume that the data is located in the segment specified by the data segment register which contain 3000H.

2. Move a byte string, 16-bytes long, from the offset 0500H to 0010H in the segment 75000H.

If your answer is correct, then proceed to the next section.

Machine Coding the Programs

So far we have discussed five programs which were written for handcoding by a programmer. In this section, we will now have a brief look at how these programs can be translate to machine codes. In Appendix, the instruction set along with the Appendix are presented. This Appendix is self-explanatory to handcode most of the instructions. The S, V, W, D, MOD, REG and R/M fields are suitably decided

depending upon the data types, addressing mode and the registers used. The table 3.2 shows the details about how to select these fields.

Most of the instructions either have specific opcodes or they can be decided only by setting the S, V, W, D, REG, MOD and R/M fields suitably but the critical point is the calculation of jump addresses for intrasegment branch instructions. Before starting the coding of jump or call instructions, we will see some easier coding examples.

Example 5.42

MOV BL, CL

For handcoding this instruction, we will have to first note down the following features.

(i) It fits in the register/memory to/from register format.

(ii) It is an 8-bit operation.

(iii) BL is the destination register and CL is the source register.

Now from the feature (i) using the Appendix, the opcode format is given as shown.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7 D_6 D_5	D_4 D_3	D_2 D_1 D_0
1	0	0	0	1	0	d	w	(MOD)	(REG)	(R/M)

If d =1, then transfer of data is to the register shown by the REG field, i.e. the destination is a register (REG). If d = 0, the source is a register shown by the REG field.

It is an 8-bit operation, hence w bit is 0. If it had been a 16-bit operation, the w bit would have been 1.

Refer to Table 2.2 to search the REG to REG addressing in it, i.e. the last column with MOD 11. According to the Appendix when MOD is 11, the R/M field is treated as a REG field. The REG field is used for source register and the R/M field is used for the destination register, if d is 0. If d =1, the REG field is used for destination and the R/M field is used to indicate source. Now the complete machine code of this instruction comes out to be

	code	dw	MOD	REG	R/M
MOV BL, CL	1 0 0 0 1 0 0 0	1 1	001	0 1 1	= 88 CB

Note that the register codes are to be found out from the Table 3.1.

Examples 5.43

MOVBX.5000H

From Appendix, the coding format is as shown.

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	D_7 D_0	D_7 D_0
1 0 1 1 W REG	DATA LOW BYTE	DATA HIGH BYTE

Following the procedure as in Example 1, the code comes out to be (BB 00 50) as shown.

W (R E G)	Data L B	Data H B
1 0 1 1 1 0 1 1	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0
B B	0 0	5 0

Example 5.44

MOV [SI], DL

This instruction belongs to the register to memory format. Hence from the Appendix, and using the already explained procedure , the machine code can be found as shown.

OPCODE	D	W	MOD	REG	R/M
1000	100	1	00	010	100
8	9		1		4

The machine code is 89 14.

Example 5.45

MOV BP[SI], 0005H

OPCODE	W	MOD	R/M
1100 0111		00	000 010
C	7	0	2
0000	0101	0000	0000
0	5	0	0

The machine code of this instruction is C7 02 05 00.

Example 5.46

MOV BP [SI+ 500H], 7293H

OPCODE	W	MOD	R/M	
1100	0111	10	000010	
C	7	8	2	
0000	0000	0000	0101	
0	0	0	5	Displacement 500H
1001	0011	0111	0010	
9	3	7	2	Data 7293 H

The complete machine code comes out to be C7 82 00 05 93 72.

Example 5.47

ADD AX, BX

The machine code is formed as shown by referring to Appendix and using the Tables 3.1 and 3.2 as has been already described.

OPCODE	D W	MOD	REG	R/M
0000	0011	11	000	011

The machine code is 03 C3.

Example 5.48

ADD AX, 5000H

The code formation is explained as follows:

OPCODE	S W	MOD	R/M
1000	0001	00	000000
8	1	0	0
0000	0000	0101	0000
0	0	5	0

The machine code is 81 00 00 50.

If s bit is 0, the 16-bit immediate data is available in the instruction.

Ifs bit is 1, the 16-bit immediate data is the sign extended form of 8-bit immediate data.

For example, if the eight bit data is 11010001, then its sign extended 16-bit version will be 1111111111010001.

Example 5.49

SHRAX

OPCODE	VW	MOD	REG	R/M
1101	0001	11	101	000
D	1		E	8

The instruction code is D1 E8.

Finding out Machine Code for Conditional JUMP (Intrasegment) Instructions: The Appendix shows that, corresponding to each of the conditional jump instructions, the first byte of the opcode is fixed and the jump displacement must be less than or equal to 127(D) bytes and greater than or equal to -128(D). This type of jump is called as short jump. The following conditional forward jump example explains how to find the displacement. The displacement is an 8-bit signed number. If the displacement is positive, it indicates a forward jump, otherwise it indicates a backward jump. The following example is a sequence of instructions rather than a single instruction to elaborate the procedure of the calculation of positive displacement for a forward jump.

Example 5.50

```
2000 ,01          XOR AX, BX
2002 ,03          JNZ OK
2004              NOP
2005              NOP
2006 ,7,8,9       ADD BX, 05H
200A  OK : HLT
```

The above sequence shows that the programmer wants a conditional jump to label OK, if the zero flag is not set. For finding out the displacement corresponding to the label OK, subtract the address of the jump instruction (2002H), from the address of label (200AH). The required displacement is 200AH - 2002H = 08H. The 08H is the displacement for the forward jump.

Let us find out the displacement for a backward jump. Consider the following sequence of instructions.

Example 5.51

```
2000, 01, 02      MOV    CL, 05H
2003      Repeat : INC    AX
2004              DEC    CL
2005,2006         JNZ    Repeat
```

For finding out the backward displacement, subtract the address of the label (Repeat) from the address of the jump instruction. Complement the subtraction. The lower byte gives the displacement. In this example, the signed displacement for the JNZ instruction comes out to be (2005H-2003H=02, complement-FDH) The magnitude of the displacement must be less than or equal to 127(D). The MSB of the displacement decides whether it is a forward or backward jump. If it is I, it is a backward jump or else it is a forward jump.

A similar procedure is used to find the displacement for intrasegment short calls.

Finding out Machine Code for Unconditional JUMP Intrasegment: For this instruction there are again two types of jump, i.e. short jump and long jump. The displacement calculation procedures are again the same as given in case of the conditional jump. The only new thing here is that, the displacement may be beyond ±127(D). This type of jump is called as long jump. The method of calculation of the displacement is again similar to that for short jump.

Finding out Machine Code for Intersegment Direct Jump: This type of instruction is used to make a jump directly to the address lying in another segment. The opcode itself specifies the new offset and the segment of jump address, directly.

Example 5.52

JUMP 2000:5000

This instruction implies a jump to a memory location in another code segment with CS = 2000H and Offset =5000H. The code formation is as shown.

Code formation	1110 1010	0000 0000	0101 0000	0000 0000	0010 0000
	Opcode	Offset LB	Offset HB	Seg. LB	Seg. HB

The opcode forms the first byte of this instruction and the successive bytes are formed from the segment and the offset of the jump destination. While specifying the segment and offset, the lower byte (LB) is specified first and then the higher byte (HB) is specified. Finally, the opcode comes out to be EA 00 50 00 20. The procedure of coding the CALL instructions is similar.

Hand Coding a Complete Program After studying the hand-coding procedures of different instructions, let us now tries to code a complete program. We will consider Example 5.53 for complete hand-coding. The program and the code corresponding to it is given. These codes, found using hand-coding, may be entered byte-by-byte into an 8086 based system and executed, provided it supports the memory requirements of the program.

Example 5.53

A Hand-coding Program Example

Addresses	Opcodes	Labels	Mnemonics
2000,01,02	B9 0F 00		MOV CX,0F H
2003,04,05	B8 00 02		MOV AX,2000H
2006,07	8E D8		MOV DS,AX
2008,09,0A	BE 00 05		MOV SI,0500H
200B,0C	89 04		MOV AX,[SI]
200D	46	BACK :	INC SI
200E,0F	3B 04		CMP AX,[SI]
2010,11	77 04		JNC NEXT
2012,13	89 04		MOV AX,[SI]
2014,15	E2 F7	NEXT :	LOOP BACK
2016	F4		HLT

Student Activity 5.4

Before reading the next section, answer the following question.

1. Find out the machine code for following instructions.

(i)	ADC AX, BX	(ii)	OR AX, [0500H]	(iii)	AND CX, [SI]
(iv)	TEST AX,5555H	(v)	MUL [SI+5	(vi)	NEG 50[BP]
(vii)	OUT DX, AX	(viii)	LES DI, [0700H]	(ix)	LEA SI, [BX+500H]
(x)	SHL [BX+2000],CL	(xi)	RET 0200H	(xii)	CALL 7000H
(xiii)	JMP 3000h:2000H	(xiv)	CALL [5000H]	(xv)	DIV [5000H]

2. Describe the procedure for coding the intersegment and intrasegment over machine language.
3. Enlist the advantages of assembly language programming over machine language.

If your answer is correct, then proceed to the next section.

Programming with an Assembler

The procedure of hand-coding 8086 programs is somewhat tedious, hence in general a programmer may find it difficult to get a correct listing of the machine codes. Moreover, the procedure of handcoding is time consuming. This programming procedure is called as machine level programming. The obvious disadvantages of machine level programming are as given:

1. The process is complicated and time consuming.
2. The chances of error being committed are more at the machine level in hand-coding and entering the program byte-by-byte into the system.
3. Debugging a program at the machine level is more difficult.
4. The programs are not understood by everyone and the results are not stored in a user-friendly form.

A program called 'Assembler' is used to convert the mnemonics of instructions along with the data into their equivalent object code modules. These object code modules may further be converted in executable code using the linker and loader programs. This type of programming is called assembly level programming. In assembly language programming, the mnemonics are directly used in the user programs. The assembler performs the task of coding.

The advantages of assembly language over machine language are as given:

1. The programming in assembly language is not so complicated as in machine language because the function of coding is performed by an assembler.
2. The chances of error being committed are less because the mnemonics are used instead of numerical opcodes. It is easier to enter an assembly language program.
3. As the mnemonics are purpose-suggestive the debugging is easier.
4. The constants and address locations can be labeled with suggestive labels hence imparting a more friendly interface to user. Advanced assemblers provide facilities like macros, lists, etc. making the task of programming much easier.

5. The memory control is in the hands of users as in machine language.
6. The results may be stored in a more user-friendly form.
7. The flexibility of programming is more in assembly language programming as compared to machine language because of advanced facilities available with the modern assemblers.

Basically, the assembler is a program that converts an assembly input file also called as source file to an object file that can further be converted into machine codes or an executable file using a linker. The recent versions of the assembler are designed with many facilities like macroassemblers, numerical processor assemblers, procedures, functions and so on.

As far as this book is concerned, we will consider the assembly language programming using MASM (Microsoft Macro Assembler). There are a number of assemblers available like MASM, TASM and DOS assembler . MASM is one of the popular assemblers used along with a LINK program to structure the codes generated by MASM in the form of an executable file. MASM reads the source program as its input and provides an object file. The LINK accepts the object file produced by MASM as input and produces an EXE file.

While writing a program, for an assembler, your first step will be to use a text editor and type the program listing prepared by you. Then check the listing typed by you for any typing mistake and syntax error. Before you quit the editor program, do not forget to save it. Once you save the text file with any name (permissible on operating system), you are free to start the assembly process. A number of text editors are available in the market, e.g. Norton's editor [NE], Turbo C [TC], EDLIN, etc. Throughout this book, the NE is used. Any other free form editor may be used for a better user-friendly environment. Thus for writing a program in assembly language, one will need NE editor, MASM assembler, linker and DEBUG utility of DOS. In the following section, the procedures of opening a file for a program, assembling it, executing it and checking its result are described for beginners.

Entering a Program

In this section, we will explain the procedure for entering a small program on IBM PC with DOS operating system. Consider a program of addition of two bytes, as already discussed for handcoding. The same program is written along with some syntax modifications in it for MASM.

Before starting the process, ensure that all the files namely NE.COM (Norton's Editor), MASM.EXE (Assembler), LINK.EXE (linker), DEBUG.EXE (debugger) are available in the same directory in which you are working. Start the procedure with the following command after you boot the terminal and enter the directory containing all the files mentioned.

```
C> NE
```

You will get display on the screen as in Figure 5.16.

Now type the file name. It will be displayed on the screen. Suppose one types KMB.ASM as file name, the screen display will be as shown in Figure 5.17.

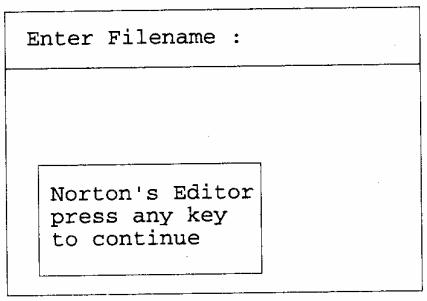


Figure 5.16: Norton's Editor Opening Screen

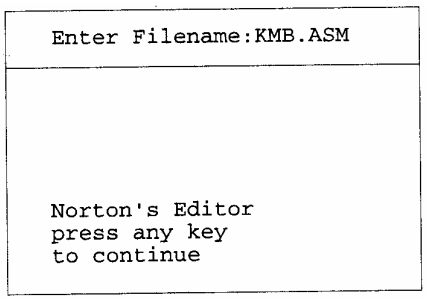


Figure 5.17: Norton's Editor Alternative Opening Screen

Press any of the keys, you will get Figure 5.18 as the screen display.

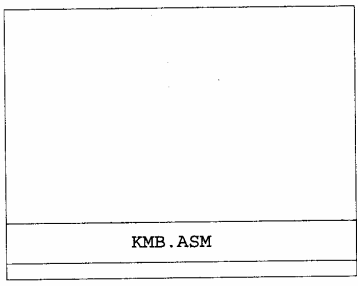


Figure 5.18: Norton's Editor Opens a New KMB.ASM

Note that, for every assembly language program, the extension .ASM must be there Though every time the assembler dose not use the complete name KMB.ASM and uses just KMB to handle the file, the extension shows that it is an assembly language program file. Even if you enter a file name without the .ASM extension, the assembler searches for the file name but with .ASM extension and if it is not available, it issues the message 'File not found'. Once you get the display as in Figure 5.18 you are free to type the program. One may use another type of command line as shown.

```
C> NE KMB.ASM
```

The above command will directly give the display as in Figure 5.18, if KMB.ASM is a newly opened file. Otherwise, if KMB.ASM is already existing in the directory, then it will be opened and the program in it will be displayed. You may modify it and save (command F3-E) it again, if you want the changes to be permanent. Otherwise, simply quit (command F3-Q) it to abandon the changes and exit NE . The entered program in NE looks like in Figure 5.19. We have to just consider that it is an assembly language program to be assembled. Store it with command F3-E. This will generate a new copy of the program in the secondary storage. Then quit the NE with command F3-Q.

Once the above procedure is completed, you may now turn towards the assembling of the above program. Note that all the commands and displays shown in this section are for Norton's Editor. Other editors may require some other commands and their display style may be some what different but the overall procedure is the same.

Note that before quitting the editor program the newly entered or modified file must be saved, otherwise it will be lost and will not be available for further assembling process.

```
ASSUME      CS:CODE, DS:DATA
DATA        SEGMENT
            OPR1 DW 1234 H
            OPR2 DW 0002 H
            RESULT DW 01 H DUP(?)

DATA        ENDS
CODE        SEGMENT
START       MOV AX, DATA
            MOV DS, AX
            MOV AX, OPR 1
            MOV BX, OPR 2
            CLC
            ADD AX, BX
            MOV DI, OFFSET RESULT
            MOV (DI), AX
            MOV AH, 4CH
            INT 21 H
CODE        ENDS
            END START
```

KBM.ASM

Figure 5.19: A Program KMB.ASM in Norton's Editor

Assembling a Program

Microsoft Assembler MASM is one of the easy to use and popular assemblers. All the references and discussions in this section are related to the MASM. As already discussed, the main task of any assembler program is to accept the text assembly language program file as input and prepare an object file. The text assembly language program file is prepared by using any of the editor programs. The MASM accepts the file names only with the extension .ASM. Even if a filename without any extension is given as input, it provides .ASM extension to it. For example, to assemble the program in Figure 5.19, one may enter the following command options-

```
A> MASM KMB
```

or

```
C ——>A> MASM KMB. ASM
```

If any of the command option is entered as above, the programmer gets the display on the screen, as shown in Figure 5.20.

```
C←A>MASM KMB

Microsoft @ Macro Assembler Version 5.10
Copyright (c) Microsoft Corp.1981, 1989.
All Rights Reserved



Object filename[.OBJ]:
List filename[NUL.LST]:
Cross Reference[NUL.CRF]:
```

Figure 5.20: MA.SM Screen Display

Also another command option, available in MASM that does not need any filename in the command line, is given along with the corresponding result display in Figure 5.21.

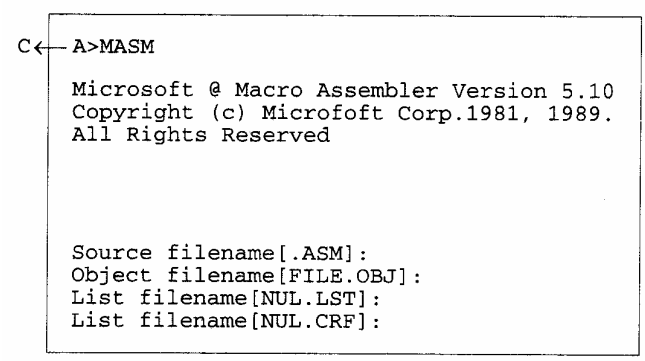


Figure 5.21: MASM Alternative Screen Display

If you do not enter the filename to be assembled at the command line as show in Figure 5.20, then you may enter it as a source filename as shown in Figure 5.21. The source filename is to be typed in the source filename line with or without the extension .ASM. The valid filename entry is accepted with a pressure of enter key. On the next line, the expected .OBJ filename is to be entered. The object file of the assembly language program is created with the .OBJ extension. The .OBJ file is created with the entered name and .OBJ extension the coded filename is entered for the .OBJ file before pressing enter key, the new .OBJ file is created with the same name as source file and extension.OBJ. The .OBJ file contains the coded object modules of the program to be assembled. On the next line, a filename is entered expected listing file of the source file, in the same way as the object filename was entered. The listing file is automatically generated in the assembly process. The listing file is identified by the entered or source filename and an extension .LST. The listing file contains the total offset map of the source file including labels, offset addresses, opcodes, memory allotment for different labels and directives and relocation information. The cross reference filename is also entered in the same way as discussed for listing file. The cross reference file is used for debugging the source program and contains the statistical information size of the file in bytes, number of labels, list of labels, routines to be called, etc. about 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 error code number and the corresponding line number at which they appear. After these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully. The successful assembly process may generate the .OBJ, .LST and .CRF files. Those may further be used by the linker programmer to link the object modules and generate an executable (.EXE) file from a .OBJ file. All the above said files may not be generated during the assembling of every program. The generation of some of them may be suppressed using the specific command line options of MASM. The files generated by the MASM are further used by the program LINK.EXE to generate an executable file of the source program.

Linking a Program

The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. The other supporting information may be obtained from the files generated by MASM. The linker program is invoked using the following options.

```
C ——> A> LINK
```

or

```
C ——> A> LINK KMB.OBJ
```

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first option may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The other option also generates the similar display, but will not ask for the .OBJ filename, as it is already specified at the command line. If no filenames are entered for these files, by default, the source filename is considered with the different extensions. The procedure of entering the filenames in LINK is also similar to that in MASM. The LINK command display is as shown in Figure 5.22.

The option input 'Libraries' in the display of Figure 5.22 expects any special library name of which the functions were used by the source program. The output of the LINK program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced versions of MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options that cannot be detailed here for the obvious reasons. For further details users may refer to "Technical reference and Users' Manual-MASM, Version 5".

```
C←——A>LINK

Microsoft @ Overlay Linker Version. 3.64
Copyright(c)Microsoft Corp. 1983-88. All Rights Reserved.


Object Module[.OBJ]:
Run file[.EXE]:
List file[NUL.MAP]:
Libraries[LIB]:
```

Figure 5.22: Link Command Screen Display

Using DEBUG

DEBUG.COM is a DOS utility that facilitates the debugging and trouble-shooting of assembly language programs. In case of personal computers, all the processor resources and memory resource management functions are carried out by the operating systems. Hence, users have very little control over the computer hardware at lower levels. The DEBUG utility enables you to have the control of these resources up to some extent. In the simplest, rather, crude words, the DEBUG enables you to use the personal computer as a low level microprocessor kit.

The DEBUG command at DOS prompt invokes this facility. A '_' (dash) display signals the successful invoke operation of DEBUG, that is further used as DEBUG prompt for debugging commands. The following command line, DEBUG prompt and the DEBUG command character display explain the DEBUG command entry procedure, as in Figure 5.23.

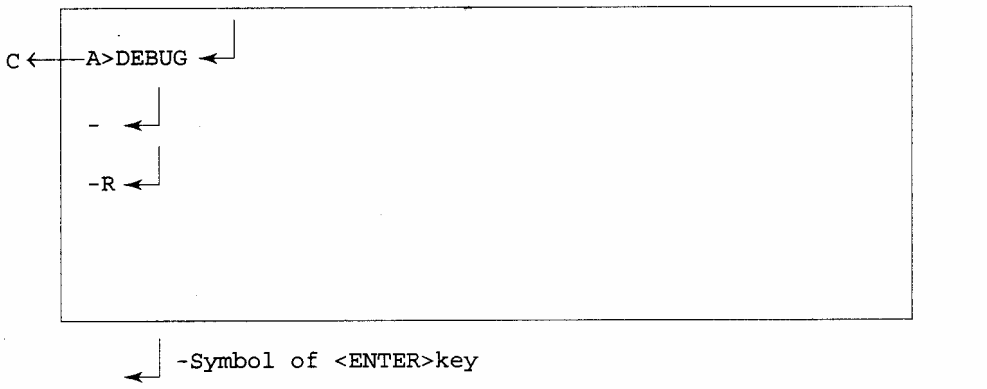


Figure 5.23: DEBUG Command Line and Prompt

A valid command is accepted using the enter key. The list of generally used valid commands of DEBUG is given in Table 5.3 along with their respective syntax.

The program DEBUG may be used either to debug a source program or to observe the results of execution of an .EXE file with the help of the .LST file and the above commands. The .LST file shows the offset address allotments for result variables of a program in the particular segment. After execution of the program, the offset address of the result variables may be observed using the d command. The results available in the registers may be observed using the r command. Thus the DEBUG offers a reasonably good platform for trouble-shooting executing and observing the results of the assembly language programs. Here one should note that the DEBUG is able to trouble-shoot only .EXE files.

Table 5.3: [DEBUG Commands]

COMMAND CHARACTER	*Format / Formats*	*Functions*
-R	<ENTER>	Display all Registers and flags
-R	reg<ENTER> old contents:New contents	Display specified register contents and modify with the entered new contents.
-D	<ENTER>	Display 128 memory locations of RAM starting from the current display pointer.
-D	SEG:OFFSET1 OFFSET2<ENTER>	Display memory contents in SEG from OFFSET1 to OFFSET2.
-E	<ENTER>	Enter Hex data at current display pointer SEG:OFFSET.
-E	SEG:OFFSET1 <ENTER>	Enter data at SEG:OFFSET1 byte by byte. The memory pointer is to be incremented by space key, data entry is to be completed by pressing the <ENTER> key.
-f	SEG:OFFSET1 OFFSET2 BYTE <ENTER>	Fill the memory area starting from SEG:OFFSET1 to OFFSET2 by the byte BYTE.
-f	SEG:OFFSET1 OFFSET2 BYTE1,BYTE2,BYTE3<ENTER>	Fill the memory area as above with the sequence BYTE1, BYTE2, BYTE3, etc.
-a	<ENTER>	Assemble from the current CS:IP.
-a	SEG:OFFSET <ENTER>	Assemble the entered instruction from SEG:OFFSET address.
-u	<ENTER>	Unassemble from the current CS:IP.
-u	SEG:OFFSET <ENTER>	Unassemble from the address SEG:OFFSET.
-g	<ENTER>	Execute from current CS:IP. By modifying CS and IP using R command this can be used for any address.
-g	=OFFSET <ENTER>	Execute from OFFSET in the current CS.
-s	SEG:OFFSET1 to OFFSET2 BYTE/BYTES <ENTER>	Searches a BYTE or string of BYTES separated by ',' in the memory block SEG:OFFSET1 to OFFSET2, and displays all the offsets at which the byte or string of bytes is found.
-q	<ENTER>	Quit the DEBUG and return to DOS
-T	SEG:OFFSET <ENTER>	Trace the program execution by single stepping starting from the address SEG:OFFSET.
-m	SEG:OFFSET1 OFFSET2 NB <ENTER>	Move NB bytes from OFFSET1 to OFFSET2 in segment SEG
-c	SEG:OFFSET OFFSET2 NB <ENTER>	Copy NB bytes from OFFSET1 to OFFSET2 in segment SEG.
-n	FILENAME.EXE <ENTER>	Set filename pointer to FILENAME
-l	<ENTER>	Load the file FILENAME.EXE as set by the -n command in the RAM and set the CS:IP at the address at which the file is loaded.

- Note that, changing the case of the command letters does not change the command option.
- The entered nubers are consiered as hexadecimal

Student Activity 5.5

Before reading the next section, answer the following question.

1. Write an ALP to convert a four digit hexadecimal number to decimal number.

2. Write an ALP to covert a four digit octal number to decimal number.

3. Write an ALP to fund out ASCSII codes of alphanumeric characters from a look up table.

If your answer is correct, then proceed to the next section.

Assembly Language Example Programs

In the previous chapter, we studied the complete instruction set of 8086/88, the assembler directives and pseudo-ops. In the previous sections, we have described the procedure of entering an assembly language program into a computer and coding it, i.e. preparing an EXE file from the source code. In this section, we will study some programs which elucidate the use of instructions, directives and some other facilities that are available in assembly language programming. After studying these programs, it is expected that one should have a clear idea about the use of different directives and pseudo-ops, their syntaxes besides understanding the logic of each program. If one writes an assembly language program and tries to code it, in the first attempt, the chances are rare that there is no error. Rather, errors in programming depend upon the skill of the Programmer. A lot of practice is needed to obtain skills in assembly language programming as compared to high level languages. So, to obtain a command on assembly language one should write and execute a number of assembly programs, besides studying the example programs given in this text.

Before starting the explanation of the written programs, we have to explain one more important point, that is about the DOS function calls available under INT 21H instruction. DOS is an operating system, which is a program that stands between a bare computer system hardware and a user. It acts as a user interface with the available computer hardware resources. It also manages the hardware resources of the computer system. In the Disk Operating System, the hardware resources of the computer system like memory, keyboard, CRT display, hard disk, and floppy disk drives can be handled with the help of the instruction INT 21H. The routines required to refer to these resources are written as interrupt service routines for 21H interrupt. Under this interrupt, the specific resource is selected depending upon the value in AH register. For example, if AH contains 09H, then CRT display is to be used for displaying a message or if, AH contain OAH, then the keyboard is to be accessed. These interrupts are called 'function calls' and the value in AH is called 'function value'. In short, the purpose of 'function calls' under INT 21 His to be decided by the 'function value' that is in AH. Some function values also control the software operations of the machine. The list of the function values available under INT 21 H, their corresponding functions, the required parameters and the returns are given in tabulated form in the Appendix-B. Note that there are a number of interrupt functions in DOS, but INT 21H is used more frequently. The readers may find other interrupts of DOS and BIOS from the respective technical references.

Here, a few example programs are presented. Starting from very simple programs, the chapter concludes with more complex programs.

Program 5.1

Write a program for addition of two numbers.

Solution

The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program.

```
ASSUME CS:CODE,DS:DATA
DATA      SEGMENT
OPR1      DW 1234H                ; 1st operand
OPR2      DW 0002H                ; 2nd operand
RESULT    DW 01 DUP(?)            ; A word of memory reserved for result
DATA      ENDS
CODE      SEGMENT
START:    MOV AX,DATA             ; Initialize data segment
          MOV DS,AX               ;
          MOV AX,OPR1             ; Take 1st operand in AX.
          MOV BX,OPR2             ; Take 2nd operand in BX.
          CLC                     ; Clear previous carry if any.
          ADD AX,BX               ; Add BX to AX.
          MOV DI,OFFSET RESULT    ; Take offset of RESULT in DI.
          MOV [DI],AX             ; Store the result at memory address in DI.
          MOV AH,4CH              ; Return to DOS prompt.
          INT 21H
CODE      ENDS                    ; CODE segment ends.
          END START               ; Program ends
```

Program 5.1: Listings

How to Write an Assembly Language Program

The first step in writing an assembly language program is to define and study the problem. After studying the problem, decide the logical modules required for the program. From the statement of the program one may guess that some data may be required for the program or the result of the program that is to be stored in memory. Hence the program will need a logical space called DATA segment. Invariably CODE segment is a part of a program containing the actual instruction sequence to be executed. If the stack facility is to be used in the program, it will require the STACK segment. The EXTRA segment may be used as an additional destination data segment. Note that the use of all these logical segments is not compulsory except for the CODE segment. Some programs may require DATA and CODE segments, while the others may also contain STACK and EXTRA. For example, Program 5.1 requires only DATA and CODE segment.

The first line of the program containing 'ASSUME' directive declares that the label CODE is to be used as a logical name for CODE segment and the label DATA is to be used for DATA segment. These labels CODE and DATA are reserved by MASM for these purposes only.

They should not be used as general labels. Once this statement is written in the program, CODE refers to the code segment and DATA refers to data segment throughout the program If you want to change it in a program you will have to write another ASSUME statement the program.

The second statement DATA SEGMENT marks the starting of a logical data space DATA Inside DATA segment, OPRI is the first operand. The directive DW defines OPRI as a word operand of value 1234H and OPR2 as a word operand of value 0002H. The third DW directive reserves OIH words of memory for storing the result of the program and leaves it undefined due to the directive DUP(?). The statement DATA ENDS marks the end of the DATA segment. Thus the logical space DATA contains OPRI, OPR2 and RESULT. These labels OPRI, OPR2 and RESULT will be allotted physical memory locations whenever the logical SEGMENT DATA is allocated memory or loaded in the memory of a computer as explained in the previous topic of relocation. The assembler calculates that the above data segment requires 6 bytes, i.e. 2 bytes each for OPRI, OPR2 and RESULT.

The code segment in the above program starts with the statement CODE SEGMENT. The code segment as already explained is a logical segment space containing the instructions. The label STARTS marks the starting point of the execution sequence. The ASSUME statement just informs the assembler that label

CODE is used for the code segment and the label DATA is used for the DATA segment. It does not actually put the address corresponding to CODE in code segment register (CS) and address corresponding to DATA in the data segment register (DS). This procedure of putting the actual segment address values into the corresponding segment registers is called as segment register initialisation. A programmer has to carry out these initializations for DS, SS and ES using instructions, while the CS automatically initialised by the loader at the time of loading the EXE file into the memory for actual execution. The first two instructions in the program are for data segment initialization. Note that, no segment register in 8086 can be loaded with immediate segment address value, instead the address value should be first loaded into any one of the general purpose registers and then the contents of the general purpose register can be transferred to any of the segment registers DS, ES and SS. Also one should note that CS cannot be loaded at all. Its contents can be changed by using a long jump instruction, a call instruction or an interrupt instruction. For each of the segments DS, ES and SS, the programmer will have to carry out initialization if they are used in the program, while CS is automatically initialized by the loader program at the time of loading and execution. Then the two instructions move the two operands OPRI and OPR2 in AX and BX respectively. Carry is cleared before addition operation (optional in this program). The ADD instruction will add BX into AX and store the result in AX. The instruction used to store the result in RESULT uses different addressing mode than that used for taking OPRI into AX. The indexed addressing mode is used to store the result of addition in memory locations labeled RESULT. The instruction MOVDI, OFFSET RESULT stores the offset of the label RESULT into Dl register. Next instruction stores the result available in AX into the address pointed to by Dl, i.e address of the RESULT. A lot has been already discussed about the function calls under INT 21H. The function value 4CH is for returning to the DOS prompt. If instead of these one writes HLT instruction there will not be any difference in program execution except that the computer will hang as the processor goes to HLT state, and the user will not be able to examine the result. In that case, for further operation, one will have to reset the computer and boot it again. To avoid this resetting of the computer every time you run the program and enable one to check the result, it is better to use the function call 4CH at the end of each program so that after executing the program, the computer returns back to DOS prompt. The statement CODE ENDS marks the end of the CODE segment. The statement END START marks the end of the procedure that started with the label START. At the end of each file, the END statement is a must.

Until now we have discussed every line of Program 5.1 in significant details. As we have already said, the program contains two logical segments CODE and DATA, but it is not at all necessary that all the programs must contain the two segments. A programmer may use a single segment to cover up data as well as instructions. The following listings explain the fact.

```
ASSUME CS:CODE
        CODE        SEGMENT
        OPR1        DW   1234H
        OPR2        DW   0002H
        RESULT      DW   01 DUP(?)
START : MOV AX,CODE
        MOV DS,AX
        MOV AX,OPR1
        MOV BX,OPR2
        CLC
        ADD AX,BX
        MOV DI,OFFSET RESULT
        MOV [DI],AX
        MOV AH,4CH
        INT 21H
CODE    ENDS
        END START
```

Program 5.1: Altemative listing for program 1

For this program, we have discussed all clauses and clones in details. For all the following programs, we will not explain the common things like forming segments using directives and operators, etc. Instead, just the logic of the program will be explained. The use of proper syntax of the 8086/8088 assembler MASM is self explanatory. The comments may help the reader in getting the ideas regarding the logic of the program.

Assemble the above written program using MASM after entering it into the computer using the procedure explained earlier. Once you get the EXE file as the output of the LINK program, Your listing is ready for execution. The Program 5.1 is prepared in the form of EXE file with the name KMB.EXE in the directory. Next, it can be executed with the command line as given below.

```
C> KMB
```

This execution method will store the result of the program in memory but will not display it on the screen. To display the result on the screen the programmer will have to use DOS function calls. This will make the programs too lengthy. Hence, another method to check the results is to run the program under the control of DEBUG. To run the program under the control of debug and to observe the results one must prepare the LST file, that gives information about memory allotment to different labels and variables of the program while assembling it. The LST file can be displayed on the screen using NE-Norton's Editor.

Program 5.2

Write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers).

Solution

In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                        ; Data segment starts
NUMLIST DB 52H,23H,—                ; List of byte numbers
COUNT EQU 100D                      ; Number of bytes to be added
RESULT DW 01H DUP(?)                ; One word is reserved for result.
DATA ENDS                           ; Data segment ends
CODE SEGMENT                        ; Code segment starts at relative
ORG 200H                            ; address 0200h in code segment.
START:      MOV AX,DATA             ; Initialize data segment.
            MOV DS,AX
            MOV CX,COUNT            ; Number of bytes to be added in CX.
            XOR AX,AX               ; Clear AX and CF.
            XOR BX,BX               ; Clear BH for converting the byte to word
            MOV SI,OFFSET NUMLIST   ; Point to the first number in the list.
AGAIN:      MOV BL,[SI]             ; Take the first number in BL, BH is zero
            ADD AX,BX               ; Add AX with BX.
            INC SI                  ; Increment pointer to the byte list.
            DEC CX                  ; Decrement counter.
            JNZ AGAIN               ; If all numbers are added,point to result
            MOV DI,OFFSET RESULT    ; destination and store it.
            MOV [DI],AX
            MOV AH,4CH              ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program 5.2: Listings

The use of statement ORG 200H in this program is not compulsory. We have used this statement here just to explain the way to use it. It will not affect the result at all. Whenever the program is loaded into the memory whatever is the address assigned for CODE, the executable code starts at the offset address 0200H due to the above statement. Similar to DW, the directive DB reserves space for the list of 8-bit numbers in the series. The procedure for entering the program, coding and execution has already been explained. The result of addition will be stored in the memory locations allotted to the label RESULT.

Program 5.3

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution

Compare the ith number of the series with the (i+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (i+1)th number is greater. If the ith number is greater than (i+1)th, leave it in AX (any register may be used). Otherwise, load the (i+1)th number in AX, replacing the ith number in AX. The procedure is repeated tilt all the members in the array have been compared.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                          ; Data segment starts
LIST DB 52H,23H,56H,45H,--            ; List of byte numbers
COUNT EQU 0F                          ; Number of bytes in the list
LARGEST DB 01H DUP(?)                 ; One byte is reserved for the largest number.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts.
START:      MOV AX,DATA               ; Initialize data segment.
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT              ; Number of bytes in CL.
            MOV AL,[SI]               ; Take the first number in AL
AGAIN:      CMP AL,[SI+1]             ; and compare it with the next number.
            JNL NEXT
            MOV AL,[SI+1]
NEXT:       INC SI                    ; Increment pointer to the byte list.
            DEC CL                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are compared,point to result
            MOV SI,OFFSET LARGEST     ; destination and store it.
            MOV [SI],AL
            MOV AH,4CH                ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program 5.3 Listings

Program 5.4

Modify the Program 5.3 for a series of words.

Solution

The logic is similar to the previous program written for a series of byte numbers. The program is directly written as follows without any comment leaving it to the reader to find out the use of each instruction and directive used.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 1234H,2354H,0056H,045AH,--
COUNT EQU 0F
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT
            MOV AX,[SI]
AGAIN:      CMP AX,[SI+2]
            JNL NEXT
            MOV AX,[SI+2]
NEXT:       INC SI
            INC SI
            DEC CL
            JNZ AGAIN
            MOV SI,OFFSET LARGEST
            MOV [SI],AX
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END     START
```

Program 5.4 Listings

Program 5.5

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Solution

The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number. If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2357H,0A579H,0C322H,0C91EH,0C000H,0957H
COUNT EQU 006H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI, OFFSET LIST
AGAIN:      MOV AX,[SI]

            ROR AX,01
            JC ODD
            INC BX
            JMP NEXT
ODD:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

Program 5.5: Listings

Program 5.6

Write a program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Solution

Take the ith number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. If CF is 1, the number is negative; otherwise, it is positive.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2579H,0A500H,0C009H,0159H,0B900H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI,OFFSET LIST
AGAIN:      MOV AX,[SI]
            SHL AX,01
            JC NEG
            INC BX
            JMP NEXT
NEG:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

Program 5.6: Listing

The logic of Program 5.6 is similar to that of Program 5.5, hence comments are not given in Program 5.6 except for a few important ones.

Program 5.7

Write a program to move a string of data words from offset 2000H to offset 3000H the length of the string is OFH.

Solution

To write this program, we will use an important facility, available in the 8086 instruction set, i.e. move string byte/word instruction. We will also study the flexibility imparted by this instructions to the 8086 assembly language program. Let us first write the Program 5.7 for 8085, assuming that the string is available at location 2000H and is to be moved at 3000 H.

```
          LXI H , 2000H
          LXI D , 3000H
          MVI C , 0FH
AGAIN :   MOV A , M
          STAX D
          INX  H
          INX  D
          DCR  C
          JNZ  AGAIN
          HLT
```

An 8085 Program for Program 5.7

Now assuming DS is suitably set, let us write the sequence for 8086. At first using the index registers, the program can be written as given.

```
            MOV  SI ,2000H
            MOV  DI ,3000H
            MOV  CX ,0FH
AGAIN :     MOV  AX ,[SI]
            MOV  [DI],AX
            INC  SI
            INC  DI
            DEC  CX
            JNZ  AGAIN
            HLT
```

An 8086 Program for Program 5.7

Comparing the above listings for 8085 and 8086 we may infer that every instruction in 8085 listing is replaced by an equivalent instruction of 8086. The above 8086 listing is absolutely correct but it is not efficient. Let us try to write the listings for the same purpose using the string instruction. Due to the assembler directives and the syntax, one may feel that the program is lengthy, though it eliminates four instructions for a MOVSW instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH

DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV SI,SOURCESTRT
            MOV DI,DESTSTRT
            MOV CX,COUNT
            CLD
REP         MOVSW
            MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.7

An 8086 Program listing for Program 5.7 using String instruction

Compare the above two 8086 listings. Both contain ten instructions. However, in case of the second program, the instruction for the initialisation of segment register and DOS interrupt are additional while the first one neither contains initialisation of any segment registers nor does it contain the DOS interrupt instruction. We can say that the first program uses 9 instructions, while the second one uses only 5 for implementing the same algorithm. This program and the related discussions are aimed at explaining the importance of the string instructions and the method to use them.

Program 5.8

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Solution

There exist a large number of sorting algorithms. The algorithm used here is called bubble sorting. The method of sorting is explained as follows. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series (n-1) times. After (n-1) iterations, you will get the largest number at the end of the series, where n is the length of the series. Again start from the first address of the series. Repeat the same procedure right from the first element to the last element. After (n-2) iterations you will get the second highest number at the last but one place in the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

53 , 25 , 19, 02		n = 4
25 , 53 , 19, 02		1st operation
25 , 19 , 53 , 02		2nd operation
25 , 19 , 02 , 53		3rd operation
largest no.	⇒	4 – 1 = 3 operations
19 , 25 , 02 , 53		1st operation
19 , 02 , 25 , 53		2nd operation

2nd largest number => 4-2=2 operations

02, 19, 25, 53 1st operation

3rd largest number => 4 -3 = 1 operations

Instead of taking a variable count for the external loop in the program like (n - 1), (n – 2), (n-3),...., etc. It is better to take the count (n- 1) all the time for simplicity. The resulting program is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 53H,25H,19H,02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DX,COUNT-1
AGAIN0:     MOV CX,DX
            MOV SI,OFFSET LIST
AGAIN1:     MOV AX,[SI]
            CMP AX,[SI+2]
            JL  PR1
            XCHG [SI+2],AX
            XCHG [SI],AX
PR1:        ADD SI,02
            LOOP AGAIN1
            DEC DX
            JNZ AGAIN0
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.8 Listings

With a similar approach, the reader may write a program to arrange the string in descending order. Just instead of the JL instruction in the above program, one will have to use a JG instruction.

Program 5.9

Write a program to perform a one byte BCD addition.

Solution

It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

```
  92
+ 59
------
  EB      Actual result after addition considering hex. operands
```

```
    1 0 1 1
+   0 1 1 0      As 0BH (LSD of addition) > 09, add 06 to it.
-------------
  1 0 0 0 1      Least significant nibble of result (neglect the auxiliary carry)
                 → AF is set to 1

0110 is added to most significant nibble of the result if it is greater than 9 or AF is set.

                       1      Carry from previous digit (AF)
        E      →   1 1 1 0
               +   0 1 1 0
               -------------
CF is set to 1       0 1 0 1  next significant nibble of result

Result CF      Most significant      Least significant digit
   1                 5                      1
```

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 92H
            OPR2 EQU 52H
RESULT DB 02 DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS,AX
            MOV BL,OPR1
            XOR AL,AL
            MOV AL,OPR2
            ADD AL,BL
            DAA
            MOV RESULT,AL
            JNC MSB0
            INC [RESULT+1]
MSB0:       MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.9 Listings

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after SUB instruction. The reader may try to write a program for BCD subtraction for practice.

Program 5.10

Write a program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

Solution

Here we have directly given the routine for Program 5.10.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 98H
```

```
                OPR2 EQU 49H
                SUM DW 01 DUP(00)
                SUBT DW 01 DUP(00)
                PROD DW 01 DUP(00)
                DIVS DW 01 DUP(00)
DATA            ENDS
CODE            SEGMENT
START:          MOV AX,DATA
                MOV DS,AX
                MOV BL,OPR2
                XOR AL,AL
                MOV AL,OPR1
                ADD AL,BL
                DAA
                MOV BYTE PTR SUM,AL
                JNC MSB0
                INC [SUM+1]
MSB0:           XOR AL,AL
                MOV AL,OPR1
                SUB AL,BL
                DAS
                MOV BYTE PTR SUBT,AL
                JNB MSB1
                INC [SUBT+1]
MSB1:           XOR AL,AL
                MOV AL,OPR1
                MUL BL
                MOV WORD PTR PROD,AX
                XOR AH,AH
                MOV AL,OPR1
                DIV BL
                MOV WORD PTR DIVS,AX
                MOV AH,4CH
                INT 21H
CODE            ENDS
                END START
```

Program 5.10 Listings

Program 5.11

Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Solution

The given string is scanned for the given byte. If it is found in the string, the zero flag is set; else, it is reset. Use of the SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found out. Note that, in this program, the code segment is written before the data segment.

```
ASSUME CS:CODE,DS:DATA
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV CX,COUNT
            MOV DI,OFFSET STRING
            MOV BL,00H
            MOV AL,BYTE1
SCAN1:      NOP
            SCASB [DI]
            JZ XXX
            INC BL
            LOOP SCAN1
XXX:        MOV AH,4CH
            INT 21H
            CODE ENDS
DATA SEGMENT
BYTE1 EQU 25H
COUNT EQU 06H
STRING DB 12H,13H,20H,20H,25H,21H
DATA ENDS
            END START
```

Program 5.11 Listings

Program 5.12

Write a program to convert the BCD numbers 0 to 9 to their equivalent sever segment codes using the look-up table technique. Assume the codes [7-seg] are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

Solution

Refer to the explanation of the XLAT instruction. The statement of the program itself gives the explanation about the logic of the program.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            CODELIST DB 34,45,56,45,23,12,19,24,21,00
            CHAR EQU 05
            CODEC   DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV BX,OFFSET CODELIST
            MOV AL, CHAR
            XLAT
            MOV BYTE PTR CODEC,AL
```

```
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.12 Listings

Program 5.13

Decide whether the parity of a given number is even or odd. If parity is even set DL to 00; else, set DL to 01. The given number may be a multibyte number.

Solution

The simplest algorithm to check the parity of a multibyte number is to go on adding the parity byte by byte with OOH. The result of the addition reflects the parity of that byte of the multibyte number. Adding the parities of all the bytes of the number, one will obtain the over all parity of the number.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            NUM DD 335A379BH
            BYTE_COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DH,BYTE_COUNT
            XOR AL,AL
            MOV CL,00
            MOV SI,OFFSET NUM
NEXT_BYTE:  ADD AL,[SI]
            JP EVENP
            INC CL
EVENP:      INC SI
            MOV AL,00
            DEC DH
            JNZ NEXT_BYTE
            MOV DL,00
            RCR CL,1
            JNC CLEAR
            INC DL
CLEAR:      MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.13 Listings

The contents of CL are incremented depending upon either the parity for that byte is even or odd. If LSB of CL is I, after testing for all bytes, it means the parity of the multibyte number is odd otherwise it is even and DL is modified correspondingly.

Program 5.14

Write a program for the addition of two 3 x 3 matrices . The matrices are stored in the form of lists (row wise). Store the result of addition in the third list.

Solution

In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

$$[A] \quad + \quad [B] \quad = \quad [A+B]$$

The matrix A is stored in the memory at an offset MAT1, as given:

a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{32}, a_{33}, etc.

A total of 3 x3= 9 additions are to be done. The assembly language program is written as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            DIM EQU 09H
            MAT1 DB 01,02,03,04,05,06,07,08,09
            MAT2 DB 01,02,03,04,05,06,07,08,09
            RMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,DIM
            MOV SI,OFFSET MAT1
            MOV DI,OFFSET MAT2
            MOV BX,OFFSET RMAT3
NEXT:       XOR AX,AX
            MOV AL,[SI]
            ADD AL, [DI]
            MOV WORD PTR [BX],AX

            INC SI
            INC DI
            ADD BX,02
            LOOP NEXT
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.14 Listing

Program 5.15

Write a program to find out the product of two matrices. Store the result in the third matrix. The matrices are specified as in the Program 5.14.

Solution

The multiplication of matrices is carried out as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{12}b_{12} + a_{22}b_{12} + a_{13}b_{32} & a_{13}b_{13} + a_{12}b_{23} + a_{13}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

The listings to carry out the above operation is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
ROCOL EQU 03H
MAT1 DB 05H,09H,0AH,03H,02H,07H,03H,00H,09H
MAT2 DB 09H,07H,02H,01H,0H,0DH,7H,06H,02H
PMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CH,ROCOL
            MOV BX,OFFSET PMAT3
            MOV SI,OFFSET MAT1
NEXTROW:    MOV DI,OFFSET MAT2
            MOV CL,ROCOL
NEXTCOL:    MOV DL,ROCOL
            MOV BP,0000H
            MOV AX,0000H
            SAHF
```

```
NEXT_ELE:   MOV AL,[SI]
            MUL BYTE PTR[DI]
            ADD BP,AX
            INC SI
            ADD DI,03
            DEC DL
            JNZ NEXT_ELE
            SUB DI,08
            SUB SI,03
            MOV [BX],BP
            ADD BX,02
            DEC CL
            JNZ NEXTCOL
            ADD SI,03
            DEC CH
            JNZ NEXTROW

            MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.15 Listings

Program 5.16

Write a program to add two multibyte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

Solution

This program is similar to the program written for the addition of two matrices except for the addition instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BYTES EQU 08H
NUM1 DB 05,5AH,6CH,55H,66H,77H,34H,12H
NUM2 DB 04,56H,04H,57H,32H,12H,19H,13H
NUM3 DB 0AH DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,BYTES
            MOV SI,OFFSET NUM1
            MOV DI,OFFSET NUM2
            MOV BX,OFFSET NUM3
            XOR AX,AX
NEXTBYTE:   MOV AL,[SI]
            ADC AL,[DI]
```

```
                MOV BYTE PTR[BX],AL
                INC SI
                INC DI
                INC BX
                DEC CX
                JNZ NEXTBYTE
                JNC NCARRY
                MOV BYTE PTR[BX],01
NCARRY:         MOV AH,4CH
                INT 21H
CODE ENDS
                END START
```

Program 5.16 Listings

Student Activity 5.6

Answer the following question.

1. Write an ALP to perform a sixteen-bit increment operation using 8-bit instructions.
2. Write an ALP to find out average of a given string of data bytes neglecting fractions.
3. Write an ALP to find out decimal addition of sixteen four digit decimal numbers.
4. Write an ALP to convert a four digit decimal number to its binary equivalent.

Summary

- This unit is aimed at introducing the readers with the instruction set of 8086/88 and the most commonly used assembler directives and operators. To start with, the available instruction formats in 8086/88 instruction set are explained in details. Further, the addressing modes available in 8086/88 are discussed in significant details with necessary examples. The 8086/88 instructions can be broadly categorized in six types depending upon their functions, namely data transfer instructions, arithmetic instructions and logical instructions, shift and rotate instructions, string manipulation instructions, control transfer instructions and processor control instructions. All these instruction types have been discussed before proceeding with the assembler directives and operators. The coding information details of all these instructions may be obtained from the Intel Appendix. The necessary assembler directives have been discussed later with their possible syntax, functions and examples. Most of these directives are available in Microsoft MASM. The detailed discussion on every assembler directive and operator is out of scope of this book.
- This unit starts with simple programs written for hand-coding. Further, hand-coding procedures of a few example instructions are studied. Advantages of the assembly language over machine language are then presented in brief. With an overview of the assembler operation, we have initiated the discussion of assembly language programming. The procedures of writing, entering and coding the assembly language programs are then discussed in brief. Further, DOS debugging program - DEBUG, a basic tool for trouble-shooting the assembly language programs, is discussed briefly with an emphasis on the most useful commands. Then we have presented various examples of 8086/8088 programs those highlight the actual use and the syntax of the instructions and directives. Finally, a few programs those enable the programmer to access the computer system resources using DOS function calls are discussed. We do not claim that each program presented here is the most efficient one, rather it just suggests a way to implement the algorithm asked in the problem. There may be a number of

alternate algorithms and program listings to implement a logic but amongst them a programmer should choose one which requires minimum storage, execution time and complexity.

Self-assessment Questions

Solved Exercises

I. State True or False

1. Assembly language depends on the CPU architecture of the machine for which it was designed.
2. An assembly language for solving a problem is very efficient compared to the same program written in a High Level Language.
3. Writing application programs in assembly language is difficult compared to writing in high level language.
4. There are many assemblers available for the IBM PC.
5. In immediate type addressing the time required to fetch the desired data is zero.

II. Fill in the blanks

1. Assembly language use _____ to represent operation codes.
2. Assembly language use _____ to represent operands.
3. One assembly language instruction become _____machine for which it was designed.
4. A translator which translates an assembly language program to machine language is called _____.
5. Early IBM PCs used _____ chip as CUP.

Answers

I. State True or False

1. True
2. True
3. True
4. False
5. False

II. Fill in the blanks

1. mnemonics
2. symbols
3. one

4. compiler
5. Intel 8088/8086

Unsolved Exercises

I. State True or False

1. Direct addressing mode is used when speed is important.
2. Registers are normally directly addressed as the number of registers in CPU is small.
3. A disadvantage of immediate addressing is negative operands cannot be used.
4. Indirect addressing through a register requires 3 memory accesses to fetch data.
5. Subroutines are self-contained programs which can be assembled separately.
6. A subroutine may call itself.
7. A subroutine can be independently debugged.
8. When a program call a subroutine the return address is stored in bottom of stack.

II. Fill in the blanks

1. Assembly language operand can be a _____.

   List all correct answers::

   (a) number (b) character constant
   (c) variable name (d) an absolute address
   (e) symbolic address (f) mnemonic code

2. Directives in an assembly language are required to _____

   (Pick as many as relevant).

   (a) read a program (b) specify values to be stored
   (c) allocate space in memory (d) find execution time
   (e) find error in program

3. Data registers are used to stone _____.

   (a) instructions (b) operands
   (c) operators (d) operators

4. ALU stands for _____.

   (a) All Logic Unit (b) Arithmetic Logic Unit
   (c) Another Legal Unit (d) All-purpose Language Utility

5. The number of bits in an address bus equals the number of bits in _____.

   (a) memory (b) data register

(c) address register (d) flags resister

6. The operation code of an assembly language program is represented by a _____.

(a) code (b) mnemonic

(c) instruction (d) number

7. At the end of the sequence of instructions

MOV AX, 0F289h

MOV BL, AH

BL will store _____.

(a) 89h (b) F2h

(c) 28h (d) F9h

Detailed Questions

1. What is an assembler?
2. What is a linker?
3. Write an ALP to change an already available ascending order byte string to descending order.
4. Write an ALP to convert a given sixteen bit binary number to its GRAY equivalent.
5. Write an ALP to find out transpose of a 3x3 matrix.
6. Write an ALP to find out cube of an 8-bit hexadecimal number.

Instruction set of 8086
Assembler Directives and Operators
A Few Machine Level Programs
Machine Coding the Programs
Programming with an Assembler
Assembly Language Example Programs

Unit 5

8086 Assembly Language Programming

Learning Objectives

After reading this unit you should appreciate the following:

- Instruction set of 8086
- Assembler Directives and Operators
- A Few Machine Level Programs
- Machine coding and Programs
- Programming with an Assembler
- Assembly Language Example Programs

Top

Instruction set of 8086

The 8086/8088 instructions are categorized into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

(i) Data Copy/Transfer Instructions: This type of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) Arithmetic and Logical Instructions: All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) Branch Instructions: These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

(iv) Loop Instructions: If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) Machine Control Instructions: These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) Flag Manipulation Instructions: All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD,STD,CLI,STI, etc. belong to this category of instructions.

(vii) Shift and Rotate Instructions: These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

(viii) String Instructions: These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

Data Copy/Transfer Instructions

MOV: Move This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general or special purpose registers or a memory location and, another register or memory location may act as destination.

However, in case of immediate addressing mode, a segment register cannot be a destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register. The following example instructions explain the fact.

Example 5.1

Load DS with 5000H.

1. `MOV DS, 5000H;` Not permitted (invalid)

   Thus to transfer an immediate data into the segment register, the correct procedure is given below.

2.
```
MOV AX, 5000H
MOV DS, AX
```

   It may be noted, here, that both the source and destination operands cannot be memory locations (Except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.

3. `MOV AX, 5000H;` Immediate

4. `MOV AX, BX;` Register

5. `MOV AX, [SI];` Indirect

6. `MOV AX, [200 OH];` Direct

7. `MOV AX, 50H [BX];` Based relative, 50H Displacement

PUSH: Push to Stack This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address. The examples of these instructions are as follows:

Example 5.2

1. `PUSH AX`

```
2.  PUSH DS

3.  PUSH  [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.
```

POP: Pop from Stack This instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

The examples of these instructions are as shown:

Example 5.3

```
1. POP  AX

2. POP  DS

3. POP  [5000H]
```

XCHG: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted. The examples are as shown:

Example 5.4

```
1.  XCHG        [5000H], AX   ; This instruction exchanges data between AX and a
                              ; memory location [5000H] in the data segment.

2.  XCHG        BX            ; This instruction exchanges data between AX and BX.
```

IN: Input the port This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit) which is allowed to carry the port address. The examples are given as shown:

Example 5.5

```
1.  IN AL, 030 OH ; This instruction reads data from an 8-bit port whose address
                  ; is 0300H and stores it in AL.

2.  IN AX         ; This instruction reads data from a 16-bit port whose
                  ; address is in DX (implicit) and stores it in AX.
```

OUT: Output to the Port This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8-D_{15} while that to an even addressed port is transferred on D_0-D_7. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. The examples are given as shown:

Example 5.6

```
1.  OUT         0300H, AL     ; This sends data available in AL to a port whose
                              ; address is 0300H.
```

```
2.  OUT          AX          ; This sends data available in AX to a port whose
                             ; address is specified implicitly in DX.
```

XLAT: Translate The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique. We will explain this instruction with the aid of the following example.

Suppose, a hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned in AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 7-segment code corresponding to the key pressed is found out and sent to the display port. This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H, and store the 7-segment codes for 0 to Fat the locations 2000H to 200FH sequentially. For executing the XLAT instruction, the code of the pressed key obtained from the keyboard (i.e. the code to be translated) is moved in AL and the base address of the look up table containing the 7-segment codes is kept in BX. After execution of the XLAT instruction, the 7-segment code corresponding to the pressed key is returned in AL, replacing the key code which was in AL prior to the execution of the XLAT instruction. To find out the exact address of the 7-segment code from the base address of look up table, the content of AL is added to BX internally, and the contents of the address pointed to by this new content of BX in DS are transferred to AL. The following sequence of instructions perform the task.

Example 5.7

```
MOV AX, SEG TABLE        ;Address of the segment containing look-up-table
MOV DS, AX               ; is transferred in DS.
MOV AL, CODE             ; Code of the pressed key is transferred in AL.
MOV BX, OFFSET TABLE     ; Offset of the code look-up-table in BX.
XLAT                     ; Find the equivalent code and store in AL.
```

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is more useful for assembly language rather than for machine language. Suppose, in an assembly language program, a label ADR is used. The instruction LEA BX, ADR loads the offset of the label ADR in BX.

LDS/LES: Load Pointer to DS/ES The instruction , Load DS/ES with pointer, loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The example in Figure 5.1 explains the operation.

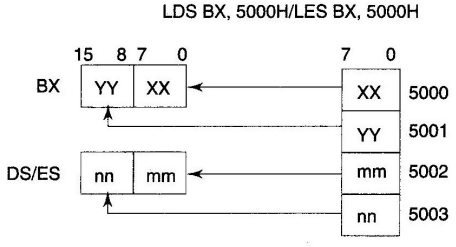


Figure 5.1: LDS/LES Instruction Execution

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except over flow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

Arithmetic Instructions

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The 8086/ 8088 instructions falling under this category are discussed below in significant details. The arithmetic instructions affect all the condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode.

ADD: Add This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected, depending upon the result. The examples of this instruction are given along with the corresponding modes.

Example 5.8

1.	`ADD AX, 0100H`	Immediate
2.	`ADD AX, BX`	Register
3.	`ADD AX, [SI]`	Register indirect
4.	`ADD AX, [5000H]`	Immediate
5.	`ADD [5000H] 0100H`	Immediate
6.	`ADD 0100H`	Destination AX (implicit)

ADC: Add with Carry This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example 5.9

1.	`ADD 0100H`	Immediate (AX implicit)
2.	`ADD AX, BX`	Register
3.	`ADD AX, [SI]`	Register indirect

4. `ADD AX, [5000H]` Direct

5. `ADD [5000H] 0100H` Immediate

INC: Increment This instruction increments the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction. The examples of this instruction are as follows:

Example 5.10

1. `INC AX` Register

2. `INC [BX]` Register indirect

3. `INC [5000H]` Direct

DEC: Decrement The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction. The examples of this instruction are as follows:

Example 5.11

1. `DEC AX` Register

2. `DEC [500OH]` Direct

SUB: Subtract The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand can not be an immediate data. All the condition code flags are affected by this instruction. The examples of this instruction along with the addressing modes are as follows:

Example 5.12

1. `SUB 0100H` Immediate [destination AX]

2. `SUB AX, BX` Register

3. `SUB AX, [5000H]` Direct

4. `SUB [5000H], 0100` Immediate

SBB: Subtract with Borrow The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (Condition code) by this instruction. The examples of this instruction are as follows:

Example 5.13

1. `SBB O100H` Immediate [destination AX]

2. `SBB AX.BX` Register

3. `SBB AX, [5000H]` Direct

4. `SBB [5000H], 0100` Immediate

CMP: Compare This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

Example 5.14

1. `CMP BX, 0100H` Immediate

2. `CMP 0100` Immediate [AX implicit]

3. `CMP [5000H],OIOOH` Direct

4. `CMP BX, [SI]` Register indirect

5. `CMP BX, CX` Register

AAA: ASCII Adjust After Addition The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by I, the AF and CF flags are set to I, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Figure 5.2. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

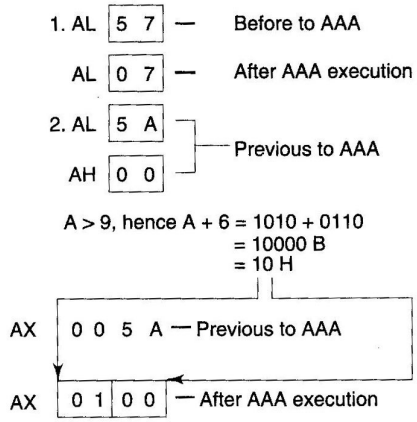


Figure 5.2: ASCII Adjust After Addition Instruction

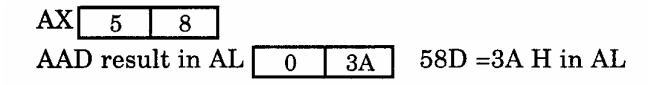
AAS: ASCII Adjust AL After Subtraction AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits of

AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure is similar to the AAA instruction. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

AAM : ASCII Adjust for Multiplication This instruction, after execution , converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH. The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL = 5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add 6 (0110) to it D + 6 = 13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1 = 6 will be the upper unpacked byte of the result. Thus after the execution, AH = 06 and AL = 03.

AAD: ASCII Adjust for Division Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contain 0508 unpacked BCD for 58 decimal, and DH contain 02H.

Example 5.15



The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Note that 3A is the hexadecimal equivalent of 58 (decimal). Now, instruction DIV DH may be executed. So rather than ASCII adjust for division, it is ASCII adjust before division. All the ASCII adjust instructions are also called as unpacked BCD arithmetic instructions. Now, we will consider the two instructions related to packed BCD arithmetic.

DAA: Decimal Adjust Accumulator This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set , it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL. The examples given below explain the instruction.

Example 5.16

```
(i) AL = 53          CL = 29
    ADD AL, CL       ; AL ← (AL) + (CL)
                     ; AL ← 53 + 29
                     ; AL ← 7C
    DAA              ; AL ← 7C + 06 (as C>9)
                     ; AL ← 82

(ii) AL = 73         CL = 29
    ADD AL,CL        ; AL ← AL + CL
                     ; AL ← 73 + 29
                     ; AL ← 9C
    DAA              ; AL ← 02 and CF = 1
```

```
AL = 7 3
     +
CL = 2 9
    9 C
  + 6
    A 2
  + 6 0
CF = 1  0 2  in AL
```

The instruction DAA affects AF, CF, PF, and ZF flags. The OF is undefined.

DAS: Decimal Adjust After Subtraction This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

Example 5.17

```
(i) AL = 75          BH = 46
    SUB AL,BH        ; AL ← 2 F = (AL) – (BH)
                     ; AF = 1
    DAS              ; AL ← 2 9 (as F> 9, F – 6 = 9)
(ii) AL = 38         CH = 6 1
    SUB AL,CH        ; AL ← D 7   CF = 1 (borrow)
    DAS              ; AL ← 7 7   (as D> 9, D – 6 = 7)
                     ; CF = 1 (borrow)
```

NEG: Negate: The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand, which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

MLIL: Unsigned Multiplication Byte or Word: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The example instructions are as

shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' CF and OF both will be set.

Example 5.18

```
1. MUL BH                ; (AX) ← (AL) x (BH)
2. MUL CX                ; (DX) (AX) ← (AX) × (CX)
3. MUL WORD PTR [SI]     ; (DX) (AX) ← (AX) × ([SI])
```

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. Sign bit and CF fill the unused higher bits of the result, AF are cleared. The example instructions are given as follows:

Example 5.19

```
1.  IMUL        BH
2.  IMUL        CX
3.  IMUL        [SI]
```

CBW: Convert Signed Byte to Word: This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to Double Word: This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

DIV: Unsigned Division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

IDIV: Signed Division: This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by 0 interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation). All the flags are undefined after IDIV instruction.

Logical Instructions

These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR. The instruction for each of these operations is discussed as follows.

AND: Logical AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand. The examples of this instruction are as follows:

Example 5.20

```
1. AND AX, 0008H
2. AND AX, BX
3. AND AX, [5000H]
4. AND [5000H], DX
```

If the content of AX is 3FOFH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3F0F H [AX]
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	AND
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H [AX]

The result 0008H will be in AX.

OR: Logical OR: The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation. The examples are as follows:

Example 5.21

```
1.  OR  AX,          0098H
2.  OR  R AX,        BX
3.  OR  R AX,        [5000H]
4.  OR [5000H],      0008H
```

The contents of AX are say 3FOFH, then the first example instruction will be carried out as given below.

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3F0F H [AX]
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	AND
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H [AX]

Thus the result 3F9FH will be stored in the AX register.

NOT: Logical Invert: The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example 5.22

```
NOT   AX
NOT   [5000H]
```

If the content of AX is 200FH, the first example instruction will be executed as shown.

AX	= 0 0 1 0	0 0 0 0	0 0 0 0	1 1 1 1
invert	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
	1 1 0 1	1 1 1 1	1 1 1 1	0 0 0 0
Result in AX =	D	F	F	0

The result DFFOH will be stored in the destination register AX.

XOR: Logical Exclusive OR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

The example instructions are as follows:

Example 5.23

```
1. XOR      AX,0098H
2. XOR      AX, BX
3. XOR      AX, [5000H]
```

If the content of AX is 3FOFH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

AX = 3F0FH =	0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1
XOR	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
0098H =	0 0 0 0	0 0 0 0	1 0 0 1	1 0 0 0
AX = Result =	0 0 1 1	1 1 1 1	1 0 0 1	0 1 1 1

= 3F97H

TEST: Logical Compare Instruction: The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to I, if the corresponding bits of both operands are I, else the result bit is reset to 0. The result of this anding operation is not available for further use) but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

Example 5.24

```
1.  TEST        AX,  BX
2.  TEST        [0500],  06H
3.  TEST        [BX]   [Dl],  CX
```

SHL/SAL: Shift logical/Arithmetic Left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 5.3 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

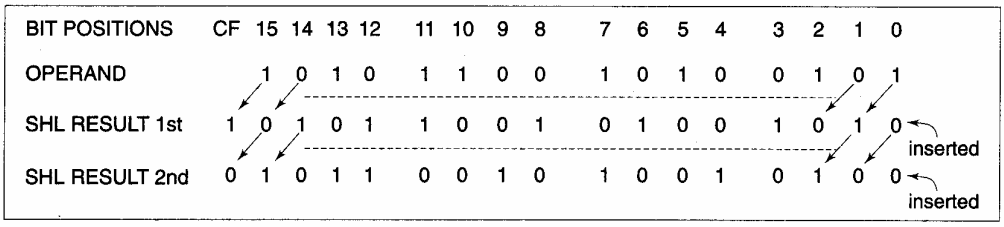


Figure 5.3 Execution of SHL/SAL Instruction

SHR: Shift Logical Right: This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. Figure 5.4 explains execution of this instruction. This instruction shifts the operand through carry flag.

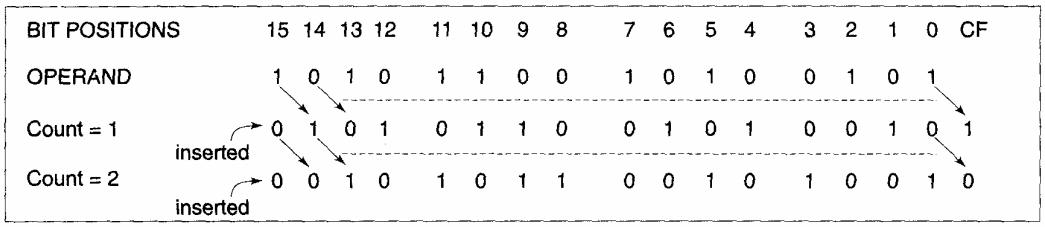


Figure 5.4: Execution SHR Instruction

SAR: Shift Arithmetic Right: This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. Figure 5.5 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through carry flag.

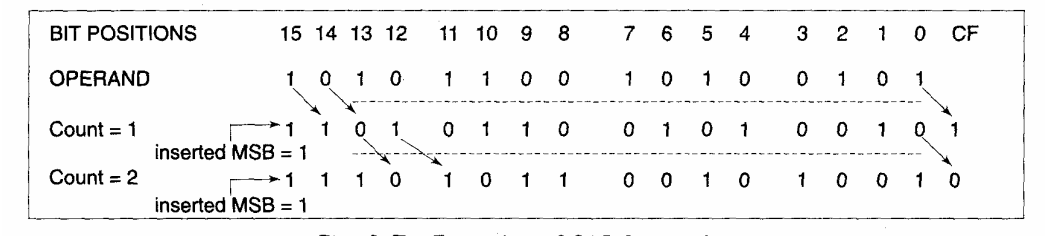


Figure 5.5: Execution of SAR Instruction

Immediate operand is not allowed in any of the shift instructions.

ROR: Rotate Right without Carry: This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. Figure 5.6 explains the operation. The destination operand may be a register (except a segment register) or a memory location.

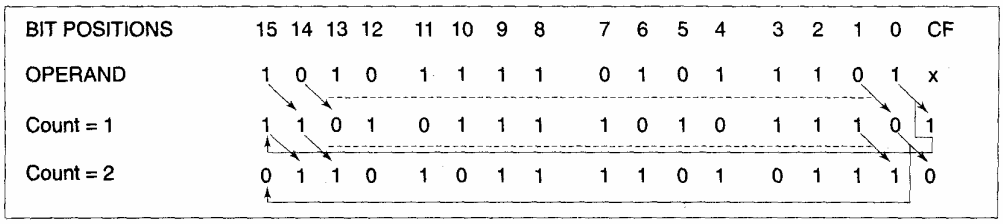


Figure 5.6: Execution of ROR Instruction

ROL: Rotate Left without Carry: This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged by this rotate operation. The operand may be a register or a memory location. Figure 5.7 explains the operation.

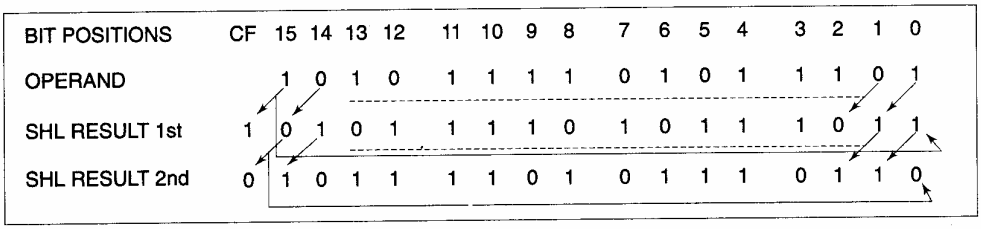


Figure 5.7: Execution of ROL Instruction

RCR: Rotate Right through Carry: This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 5.8 explains this operation.

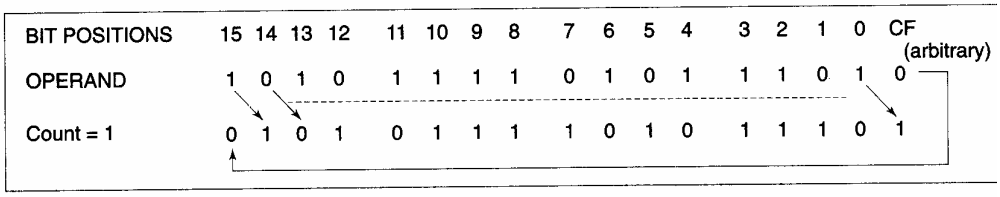


Figure 5.8: Execution of RCR Instruction

RCL: Rotate Left through Carry: This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB , and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left operation. The operand may be a register or a memory location. Figure 5. 9 explains the operation

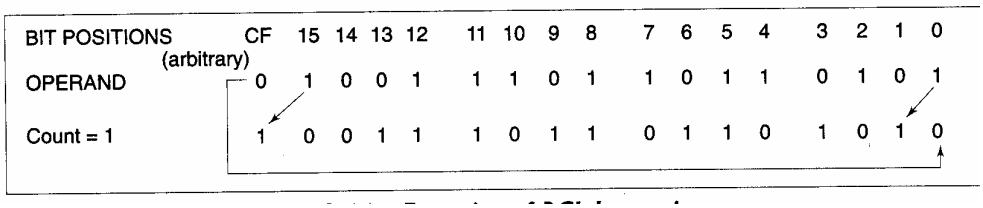


Figure 5.9: Execution of R.CL Instruction

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required, (a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements. The pointers and counters may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the direction flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation , the index registers are updated by two. The counter in both the cases, is decremented by one.

REP: Repeat Instruction Prefix: This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ, i.e. repeat operation while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word: Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (source index) and DS (data segment) contents. The starting address of the destination locations where this string has to be relocated is given by Dl (destination index) and ES (extra segment) contents. The starting address of the source string is 10H*DS+[SI], while the starting address of the destination string is 10H*ES+[DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/ words pointed to by DS: SI pair (source) to the memory location pointed to by ES: Dl pair (destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and Dl depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all the string manipulation instructions. The following string of instructions explain the execution of the MOVS instruction.

Example 5.25

```
MOV AX,5000H        ; Source segment address is 5000h.
MOV DS,AX           ; Load it to DS.

MOV AX,6000H        ; Destination segment address is 6000h.
MOV ES,AX           ; Load it to ES.
MOV CX,0FFH         ; Move length of the string to counter register CX.
MOV SI,1000H        ; Source index address 1000H is moved to SI.
MOV DI,2000H        ; Destination index address 2000H is moved to DI.
CLD                 ; Clear DF, i.e. set autoincrement mode.
REP MOVSB           ; Move 0FFH string bytes from source address to destination .
```

CMPS: Compare String Byte or String Word The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX(counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explain the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Example 5.26

```
MOV AX,SEG1             ; Segment address of STRING1, i.e. SEG1 is moved to AX.
MOV DS,AX               ; Load it to DS.
MOV AX,SEG2             ; Segment address of STRING2, i.e. SEG2 is moved to AX.
MOV ES,AX               ; Load it to ES.
MOV SI,OFFSET STRING1   ; Offset of STRING1 is moved to SI.
MOV DI,OFFSET STRING2   ; Offset of STRING2 is moved to DI.
MOV CX,010H             ; Length of the string is moved to CX.
CLD                     ; Clear DF, i.e. set autoincrement mode.
REPE CMPSW              ; Compare 010H words of STRING1 and
                        ; STRING2, while they are equal, If a mismatch is found,
                        ; modify the flags and proceed with further execution .
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

SCAS: Scan String Byte or String Word: This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found. The following string of instructions elaborates the use of SCAS instruction.

Example 5.27

```
MOV AX,SEG          ; Segment address of the string, i.e. SEG is moved to AX.
MOV ES,AX           ; Load it to ES.
MOV DI,OFFSET       ; String offset, i.e. OFFSET is moved to DI.
```

```
        MOV CX,010H       ; Length of the string is moved to CX.
        MOV AX,WORD       ; The word to be scanned for, i.e. WORD is in AL.
        CLD               ; Clear DF.
REPNE   SCASW             ; Scan the 010H bytes of the string , till a match to
                          ; WORD is found.
```

This string of instructions finds out, if it contains WORD. If the WORD is found in the word string , before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the programme proceeds further.

LODS: Load String Byte or String Word: The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word: The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES : Dl register pair. The Dl is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index D1 are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and Dl are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, S1 and D1 are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses.

Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes, the CS may or may not be modified. These type of instructions are classified in two types:

Unconditional Control Transfer (Branch) Instructions: In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

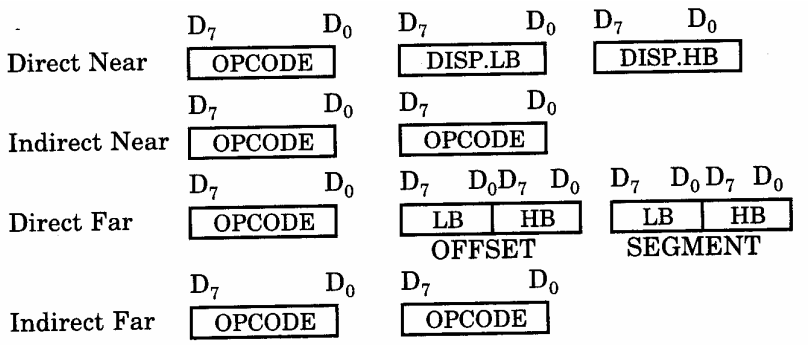
Conditional Control Transfer (Branch) Instructions: In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. Condition code flags replicate the results of the previous operations.

In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Instructions

CALL: Unconditional Call: This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e ± 32K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for

them are respectively called as intrasegment and intersegment addressing modes. This instruction comes under unconditional branch instructions and can be described as shown with the coding formats. On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.



RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure . In case of a FAR procedure, the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from OOH to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (Nx4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine.

For the execution of this instruction, the IF must be enabled.

Example 5.28

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT        20H

Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

Figure 5.10 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

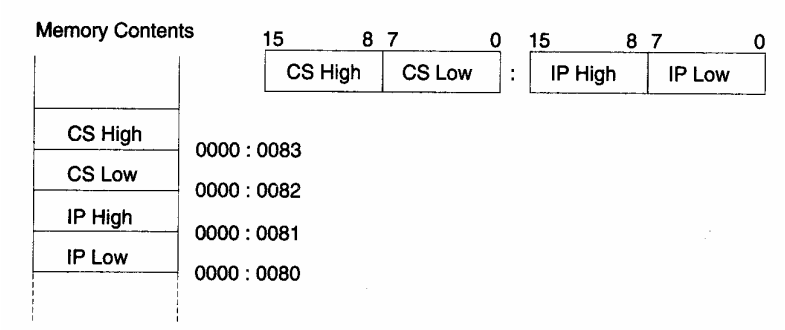


Figure 5.10: Contents of IVT

INTO: Interrupt on Overflow: This is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0000 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

JMP: Unconditional lump: This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS : IP (intersegment direct far). No flags are affected by this instruction. Corresponding to the three methods of specifying jump addresses, the JUMP instruction has the following three formats.

JUMP	DISP 8-bit	Intrasegment, relative, near jump
JUMP	DISP.16-bit (LB) | DISP.16-bit (UB)	Intrasegment, relative, Far jump
JUMP	IP(LB) | IP(UB) | CS(LB) | CS(UB)	Intersegment, direct, jump

IRET: Return from ISR: When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop Unconditionally: This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMF IF NOT ZERO structure.

Example 5.29

```
        MOV   CX, 0005      ; Number of times in CX
        MOV   BX,0FF7H      ; Data to BX
Label : MOV   AX, CODE1
        OR    BX, AX
        AND   DX, AX
        Loop  Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. Lf CX is already OOH, the execution continues sequentially. No flags are affected by this instruction.

Conditional Branch Instructions

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within -80H to 7FH (or -128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 5.1.

Table 5.1: Conditional Branch Instructions

	Mnemonic	*Displacement*	*Operation*
1.	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1
2.	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0
3.	JS	Label	Transfer execution control to address 'Label' , if SF=1
4.	JNS	Label	Transfer execution control to address 'Label', if SF=0
5.	JO	Label	Transfer execution control to address 'Label', if OF=1
6.	JNO	Label	Transfer execution control to address 'Label', if OF=0
7.	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1
8.	JNP	Label	Transfer execution control to address 'Label', if PF=0
9.	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1
10.	JNB/JAE/JNC	Label	Transfer execution control to address 'Label', if CF=0
11.	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0.
15.	JLE/JNG	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or atleast any one of SF and OF is 1(Both SF & OF are not 0).

The last four instructions are used in case of decisions based on signed binary number operations, while the remaining instructions can be used for unsigned binary operations. The terms above and below are generally used for unsigned numbers, while the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

```
JCXZ      'Label'     Transfer execution control
                      to address 'Label', if CX=0.
```

The conditional LOOP instructions are given in Table 5.2 with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

Table 5.2: Conditional Loop Instructions

Mnemonic	Displacement	Operation
LOOPZ/LOOPE	Label	Loop through a sequence of

LOOPNZ/LOOPENE	Label	instructions from 'Label' while ZF=1 and CX f 0. Loop through a sequence of instructions from 'Label' while ZF=0 and CX f 0.

The ideas about all these instructions will be more clear with programming practice. This topic is aimed at introducing these instructions to readers. Of course, examples are quoted wherever possible, but the JUMP and the LOOP instructions require a sequence of instructions for explanations and they will be emphasized.

Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

`CLC` - Clear carry flag

`CMC` - Complement carry flag

`STC` - Set carry flag

`CLD` - Clear direction flag

`STD` - Set direction flag

`CLI` - Clear interrupt flag

`STI` - Set interrupt flag

These instructions modify the carry(CF), direction(DF) and interrupt(IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions, are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

`WAIT` - Wait for Test input pin to go low

`HLT` - Halt the processor

`NOP` - No operation

`ESC` - Escape to external device like NDP (numeric co-processor)

`LOCK` - Bus lock instruction prefix.

After executing the HLT instruction, the processor enters the halt state, as explained in Chapter 1. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the

current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Student Activity 5.1

Before reading the next section, answer the following question.

1. Bring out the difference between the jump and loop instructions.
2. Which instruction of 8086 can be used for look up table manipulations?
3. What is the difference between the respective shift and rotate instructions?

If your answer is correct, then proceed to the next section.

Top

Assembler Directives and Operators

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he may be able to manage the memory of the system more efficiently. On the other hand, the disadvantages are more prominent. The programming, coding and resource management techniques are tedious. The programmer has to take care of all these functions hence the chances of human errors are more. The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and instruction set.

The assembly language programming is simpler as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement in assembly language over machine language is that the address values and the constants can be identified by labels. If the labels are suggestive, then certainly the program will become more understandable, and each time the programmer will not have to remember the different constants and the addresses at which they are stored, throughout the programs. The labels may help to identify the addresses and constants. Due to this facility, the tedious byte handling and manipulations are got rid of. Similarly, now different logical segments and routines may be assigned with the labels rather than the different addresses. The memory control feature of machine language programming is left unchanged by providing storage define facilities in assembly language programming. The documentation facility which was not possible with machine language programming is now available in assembly language.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks) an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or a variable, logical names of the segments, types of the different routines and modules, end of file, etc. These types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program

to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler.

DB: Define Byte The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initialises the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

Example 5.30

```
RANKS  DB OIH, 02H, 03H, 04H
```

This statement directs the assembler to reserve four memory locations for a list named RANKS and initialise them with the above specified four values.

```
MESSAGE DB 'GOOD MORNING
```

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialise those locations by the ASCII equivalent of these characters.

```
VALUE  DB  5 OH
```

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialised for the variable named VALUE.

DW: Define Word: The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Example 5.31

```
WORDS  DW   1234H, 4567H, 78ABH, 045CH,
```

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

```
WDATA  DW 5 DUP (6666H)
```

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initialises all the word locations with 6666H.

DQ: Define Quadword: This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

DT: Define Ten Bytes: The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

ASSUME: Assume Logical Segment Name: The ASSUME directive is used to inform the assembler, the names of the logicals segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS : DATA indicates to the assembler that the data items related to the program, are

available in a logical segment named DATA, and the DS register is to be initialised by the segment address value decided by the operating system for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program, without which a message 'CODE/DATA EMITTED WITHOUT SEGMENT' may be issued by an assembler.

END: END of Program: The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

ENDP: END of Procedure In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitues the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, and the label can then be used in the program in place of that mnemonic. Suppose, a numerical constant appears in a program ten times. If that constant is to be changed at a later time, one will have to make all these ten corrections. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the EQU statement will give the correct and modified code. The examples given below show the syntax.

Example 5.32

```
LABEL      EQU    050 OH
ADDITION   EQU    ADD
```

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD. EXTRN: External and PUBLIC: Public The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULEI from MODULE 2; in MODULE 1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL . The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MODULE 2. Thus the MODULEI and MODULE 2 must have the following declarations.

```
MODULE1        SEGMENT
PUBLIC         FACTORIAL FAR
MODULE1        ENDS
MODULE2        SEGMENT
EXTRN          FACTORIAL FAR
MODULE2        ENDS
```

GROUP: Group the Related Segments: The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the

following segment names. The assembler passes an information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base.

```
PROGRAM GROUP CODE, DATA, STACK
```

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

```
ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.
```

LABEL: Label: The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initialises a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

```
CONTINUE    LABEL FAR
```

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA          SEGMENT

DATAS DB 5 OH DUP (?)

DATA-LAST LABEL BYTE FAR

DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATALAST and its type will be byte and far.

LENGTH: Byte Length of a Label: This directive is not available in MASM. This is used to refer to the length of a data array or a string.

```
MOV CX. LENGTH ARRAY
```

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

LOCAL The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that module. At a later time, some other module may declare a particular data type LOCAL, which is previously declared 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 shown.

```
LOCAL a, b, DATA, ARRAY, ROUTINE
```

NAME: Logical Name of a Module: 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 to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

OFFSET: Offset of a Label: When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The segment may also be decided by another operator of similar type, viz, SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

Example 5.33

```
CODE SEGMENT
MOV SI, OFFSET LIST
CODE ENDS
DATA SEGMENT
LIST DB IOH
DATA ENDS
```

ORG : Origin: The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement. While starting the assembly process for a module, the assembler initialises a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialised to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment. In other words, the location counter will get initialised to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

PROC: Procedure: The PROG directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e. whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

Example 5.34

```
RESULT   PROC     NEAR
ROUTINE PROC     FAR
```

PTR: Pointer: The 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, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16-bit quantity. In other words, the PTR operator is used to specify the data type - byte or word. The examples of the PTR operator are as follows:

Example 5.35

`MOV AL, BYTE PTR [SI] -`	Moves content of memory location addressed by SI (8-bit) to AL
`INC BYTE PTR [BX]-`	Increments byte contents of memory location addressed by BX
`MOV BX, WORD PTR [2000H]-`	Moves 16-bit content of memory location 2000H to BX, i.e [2000H] to BL [2001H] to BH
`INC WORD PTR [3000H] -`	Increments word contents of memory location 3000H considering contents of 3000H (lower byte) and 3001H (higher byte) as a 16-bit number

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

`JMP WORD PTR [BX]`-NEAR Jump

`JMP WORD PTR [BX]`-FAR Jump

PUBLIC As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least any one of the other modules of the same program. (Refer to the explanation on EXTRN directive to get the clear idea of PUBLIC.)

SEG: Segment of a Label The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of 'SEG label". The example given below explain the use of SEG operator.

Example 5.36

```
MOV AX, SEG ARRAY     ; This statement moves the segment address of ARRAY in
MOV DS, AX            ; which it is appearing, to register AX and then to DS.
```

SEGMENT: Logical Segment: The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

```
EXE.CODE SEGMENT GLOBAL     ; Start of segment named EXE. CODE,
                            ; that can be accessed by any other module.
EXE.CODE ENDS               ; END of EXE.CODE logical segment.
```

SHORT The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory . Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

```
JMP SHORT LABEL
```

TYPE The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

GLOBAL: The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC    GLOBAL
```

Student Activity 5.2

Before reading the next section, answer the following question.

1. How will you enter the single step mode of 8086?
2. What is LOCK prefix? What is its use?
3. What is REP prefix? What is its use?

If your answer is correct, then proceed to the next section.

Top

A Few Machine Level Programs

In this section, a few machine level programming examples, rather, instruction sequences are presented for comparing the 8086 programming with that of 8085. These programs are ii the form of instruction sequences just like 8085 programs. These may even be hand-coded entered byte by byte and executed on an 8086 based system but due to the complex instruction set of 8086 and its tedious opcode conversion procedure, most of the programmers prefer to use assemblers. However, we will briefly discuss the hand-coding,

Example 5.37

Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

Solution

The flow chart for this problem may be drawn as shown in Figure 5.11

```
MOV  AX, 2000H   ; Initialising DS with value
MOV  DS, AX      ; 2000H.
MOV  AX, [500H]  ; Get first data byte from 0500H offset.
ADD  AX, [600H]  ; Add this to the second byte from 0600H.
MOV  [700H],AX   ; Store AX in 0700H (result).
HLT              ; Stop.
```

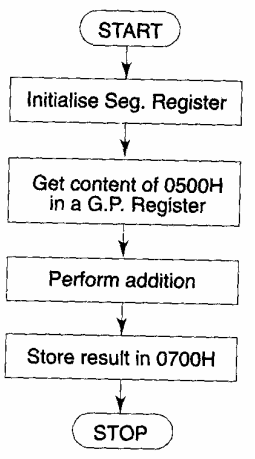


Figure 5.11: Flow Chart

The above instruction sequence is quite straight-for-ward. As the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose resistors, say AX. and then the register general purpose registers, say AX, and then the register content is moved to the segment register DS. Thus the data segment register DS contains 2000H. The instruction MOV AX,[500H] signifies that the contents of the particular location, whose offset is specified in the brackets with the segment pointed to by DS as segment register, is to be moved to AX. The MOV [0700], AX instruction moves the contents of the register AX to an offset 0700H in DS (DS = 2000H). Note that the code segment register CS gets automatically loaded by the code segment address of the program whenever it is executed. Actually it is the monitor program that accepts the CS:IP address of the program and passes it to the corresponding registers at the time of execution. Hence no instructions are required for loading the CS register like DS or SS.

Example 5.38

Write a program to move the contents of the memory location 0500H to register BX and also to CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using DS=2000H and offset=0600H. Store the result of the addition in 0700H. Assume that the data is located in the segment specified by the data segment register which contain 2000H.

Solution

The flow chart for the program is shown in Figure 5.12.

After initializing the data segment register the content of location 0500H are moved to the BX register using MOV instruction. The same data is moved also to the CX register. For this data transfer, there may be two options as shown.

```
MOV AX, 2000H
MOV DS, AX          ; Initialize data segment register.
MOV BX,[0500H]      ; Get contents of 0500H in BX.
MOV CX, BX          ; Copy the same contents in CX.
ADD [0600H],05H     ; Add byte 05H to contents of 0600H.
MOV DX, [0600H]     ; Store the result in DX.
MOV [0700H],DX      ; Store the result in 0700H.
HLT                 ; Stop.
```

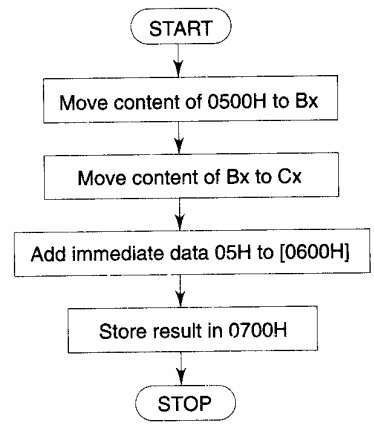


Figure 5.12 Flow Chart

(a) `MOV CX, BX` ; As the contents of BX will be same as 0500H after execution
; of MOV BX,[0500H].

(b) `MOV CX, [050OH]` ; Move directly from 0500H to register CX

The opcode in the first option is only of 2 bytes, while the second option will have 4 bytes of opcode. Thus the second option will require more memory and execution time. Due to these reasons, the first option is preferable.

The immediate data byte 05H is added to content of 0600H using the ADD instruction. The result will be in the destination operand 0600H. This is next stored at the location 0700H. In case of the 8086/8088 instruction set, there is no instruction for the direct transfer of data from the memory source operand to the memory destination operand except, the string instructions. Hence the result of addition which is present at 0600H, should be moved to any one of the general purpose registers, except BX and CX, otherwise the contents of CX and BX will be changed. We have selected DX (we could have selected AX also, because once DS is initialised to 2000H the contents of AX are no longer useful for this purpose. Thus the transfer of result from 0600H to 0700H is accomplished in two stages using successive MOV instructions, i.e., at first, the content of 0600H is DX and then the content of DX is moved to 0700H. The program ends with the HLT instruction.

Example 5.39

Add the contents of the memory location 2000H:0500H to contents of 3000H:0600H and store the result in 5000H:0700H.

Solution

Unlike the previous example programs, this program refers to the memory locations in different segments, hence, while referring to each location, the data segment will have to be newly initialized with the required value. Figure 5.13 shows the flow chart.

The instruction sequence for the above flow chart is given along with the comments.

```
MOV CX, 2000H      ; Initialize DS at 2000H.
MOV DS, CX
MOV AX, [500H]     ; Get first operand in AX.
MOV CX, 3000H      ; Initialize DS at 3000H.
MOV DS, CX
MOV BX, [0600H]    ; Get second operand in BX.
ADD AX, BX         ; Perform addition.
MOV CX, 5000H      ; Initialize DS at 5000H.
MOV DS, CX
MOV [0700H],AX     ; Store the result of addition in
HLT                ; 0700H and stop.
```

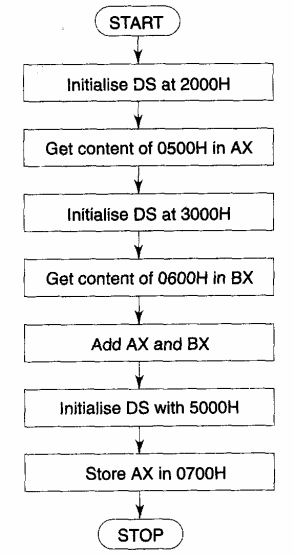


Figure 5.13: Flow Chart

Actually, the program simply performs the addition of two operands which are located in different memory segments. The program has become lengthy only due to data segment register initialization instructions.

Example 5.40

Move a byte string , 16-bytes long, from the offset 0200H to 0300H in the segment 7000H.

Solution

According to the program statement, a string that is 16-bytes long is available at the offset address 0200H in the segment 7000H. The required program should move this complete string at offset 0300H, in the same segment. Let us emphasize this program in the light of comparison between 8085 and 8086 programming techniques.

An 8085 program to perform this task, is given neglecting the segment addresses.

```
        MVI C, 010H    ; Count for the length of string
        LXIH 0200H     ; Initialization of HL pair for source string
        LXID 0300H     ; Initialization of DE pair for destination
BACK :  MOV A, M       ; Take a byte from source in A.
        STAX D         ; Store contents of A to address pointed to by DE pair.
        INX H          ; Increment source pointer.
        INX D          ; Increment destination pointer.
        DCRC           ; Decrement counter.
        JNZ BACK       ; Continue if counter is not zero.
        HLT            ; Stop if counter is zero.
```

The programmers, with fluent hands on 8085 assembly language programming but starting with 8086, may translate the above 8085 assembly language program listings to 8086 assembly language programs using the analogous or comparable instructions. Of course, this method of programming is not efficient, however, it may help those who are familiar to 8085 programming and wish to start writing programs in 8086 assembly language. The reason for the inefficiency of this method is that the special features and capabilities of 8086 have not been taken into account while preparing the 8086 assembly language program. Now, let us think about how the above program may be transferred to 8086 assembly language using analogous instructions. Note that the segment initialization is to be added. Let us consider that the code and data segment address is 7000H. Consider that the code starts at offset 0000H.

```
        MOV AX, 7000H
        MOV DS, AX          ; Data segment initialization
        MOV SI, 0200H       ; Pointer to source string
        MOV DI, 0300H       ; Pointer to destination string
        MOV CX, 0010H       ; Count for length of string
BACK :  MOV AX, [SI]        ; Take a source byte in AX.
        MOV [DI],AX         ; Move it to destination.
        INC SI              ; Increment source pointer.
        INC DI              ; Increment destination pointer.
        DEC CX              ; Decrement count by 1.
        JNZ BACK            ; Continue if count is not 0.
        HLT                 ; Stop if the count is 0.
```

The above listing has been prepared using the program written in 8085 ALP. Indexed addressing mode is used for string byte accesses and transfer in this case. The functions of all the 8086 instructions and the 8086 addressing modes have already been explained in Unit 2. In this program, all the instructions used are more or less analogous to the 8085 program, and the special software capabilities of 8086 like string instructions and loop instructions have not been considered. The 8086 programs based on 8085 codes are inefficient due to the reason that the full capability of the rich 8086 instruction set and the enhanced architecture of 8086 cannot be fully exploited.

The above program uses the decrement and jump-if-not-zero instructions for checking whether the transfer is complete or not. The 8086 instruction set provides LOOP instructions for this purpose. Using these instructions, the program is modified as shown.

```
        MOV AX,7000H
        MOV DS,AX          ; Data segment initialization
        MOV SI,0200H       ; Source pointer initialization
        MOV DI,0300H       ; Destination pointer initialization
        MOV CX,0010H       ; Counter initialization
BACK :  MOV AX,[SI]        ; Take a byte of string from source
        MOV [DI],AX        ; and then move it to destination
        INC SI             ; Update source pointer
        INC DI             ; Update destination pointer continue
        LOOP BACK          ; till CX=0,[DEC CX and JNZ BACK]
        HLT                ; Stop if CX=0
```

Thus the two instructions bracketed in the comment field are replaced by a single loop instruction which results in the saving of memory and execution time. The loop instruction needs the additional instructions for updating the pointers (for example, INC SI, INC Dl). It does not need counter decrement and check-if-zero instruction.

One more feature of the 8086 instruction set is the string instruction, i.e. MOVSB and MOVSW. Using these instructions one can move a string byte/word from source to destination. The length of the string is specified by the CX register. The SI and Dl point to the source and destination locations. The DS and ES registers should be initialised to source and destination segment addresses respectively. Before the use of string instructions, the program should initialise all these registers properly. Using the string byte instruction the same program may be written as shown.

```
        MOV AX, 7000H
        MOV DS, AX         ; Source segment initialisation
        MOV ES,AX          ; Destination segment initialisation
        MOV CX,0010H       ; Counter initialisation
        MOV SI,0200H       ; Source pointer initialisation
        MOV DI,0300H       ; Destination pointer initialisation
        CLD                ; Clear DF
REP     MOVSB              ; Move the complete string
        HLT                ; Stop
```

The MOVSB instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. An experienced programmer will thus directly use the string instructions instead of using other options. The flow chart of the final program is presented in Figure 5.14.

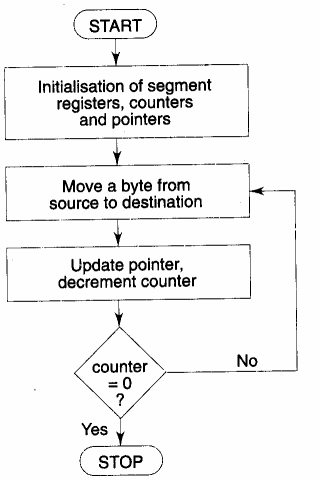


Figure 5.14: Flow Chart

Example 5.41

Find out the largest number from an unordered array of sixteen 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H. Solution The logic for this procedure can be described as follows. The first number of the array is taken in a register, say AX. The second number of the array is then compared with the first one. If the first one is greater than the second one, it is left unchanged. However, if the second one is greater than the first, the second number replaces the first one in the AX register. The procedure is repeated for every number in the array and thus it requires 15 iterations. At the end of 15^{th} iteration the largest number will reside in the register AX. This may be represented in terms of the flow chart as shown in Figure 5.15. The listing is given below.

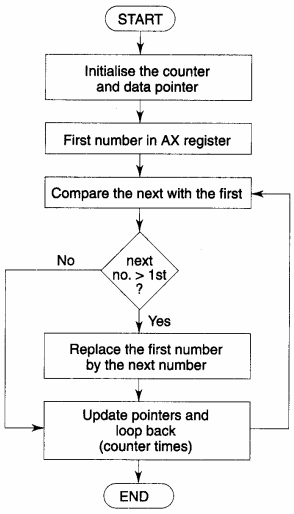


Figure 5.15: Flow Chart

Student Activity 5.3

Before reading the next section, answer the following question.

1. Write a program to move the contents of the memory location 0700H to register BX and also to CX. Add immediate byte 10H to the data residing in memory location, whose address is computed using DS=3000H and offset=0200H. Store the result of the addition in 0500H. Assume that the data is located in the segment specified by the data segment register which contain 3000H.

2. Move a byte string, 16-bytes long, from the offset 0500H to 0010H in the segment 75000H.

If your answer is correct, then proceed to the next section.

Top

Machine Coding the Programs

So far we have discussed five programs which were written for handcoding by a programmer. In this section, we will now have a brief look at how these programs can be translate to machine codes. In Appendix, the instruction set along with the Appendix are presented. This Appendix is self-explanatory to

handcode most of the instructions. The S, V, W, D, MOD, REG and R/M fields are suitably decided depending upon the data types, addressing mode and the registers used. The table 3.2 shows the details about how to select these fields.

Most of the instructions either have specific opcodes or they can be decided only by setting the S, V, W, D, REG, MOD and R/M fields suitably but the critical point is the calculation of jump addresses for intrasegment branch instructions. Before starting the coding of jump or call instructions, we will see some easier coding examples.

Example 5.42

MOV BL, CL

For handcoding this instruction, we will have to first note down the following features.

(i) It fits in the register/memory to/from register format.

(ii) It is an 8-bit operation.

(iii) BL is the destination register and CL is the source register.

Now from the feature (i) using the Appendix, the opcode format is given as shown.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7 D_6 D_5	D_4 D_3	D_2 D_1 D_0
1	0	0	0	1	0	d	w	(MOD)	(REG)	(R/M)

If d =1, then transfer of data is to the register shown by the REG field, i.e. the destination is a register (REG). If d = 0, the source is a register shown by the REG field.

It is an 8-bit operation, hence w bit is 0. If it had been a 16-bit operation, the w bit would have been 1.

Refer to Table 2.2 to search the REG to REG addressing in it, i.e. the last column with MOD 11. According to the Appendix when MOD is 11, the R/M field is treated as a REG field. The REG field is used for source register and the R/M field is used for the destination register, if d is 0. If d =1, the REG field is used for destination and the R/M field is used to indicate source. Now the complete machine code of this instruction comes out to be

	code	dw	MOD	REG	R/M
MOV BL, CL	1 0 0 0 1 0 0 0	1 1	001	0 1 1= 88 CB	

Note that the register codes are to be found out from the Table 3.1.

Examples 5.43

MOVBX.5000H

From Appendix, the coding format is as shown.

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	D_7 D_0	D_7 D_0
1 0 1 1 W REG	DATA LOW BYTE	DATA HIGH BYTE

Following the procedure as in Example 1, the code comes out to be (BB 00 50) as shown.

W (R E G)	Data L B	Data H B
1011 1011	0000 0000	01010000
B B	0 0	5 0

Example 5.44

MOV [SI], DL

This instruction belongs to the register to memory format. Hence from the Appendix, and using the already explained procedure , the machine code can be found as shown.

OPCODE	D	W	MOD	REG	R/M
1000	100	1	00	010	100
8	9		1		4

The machine code is 89 14.

Example 5.45

MOV BP[SI], 0005H

OPCODE	W	MOD	R/M
1100 0111		00	000 010
C	7	0	2
0000 0101		0000	0000
0	5	0	0

The machine code of this instruction is C7 02 05 00.

Example 5.46

MOV BP [SI+ 500H], 7293H

OPCODE	W	MOD	R/M	
1100	0111	10	000010	
C	7	8	2	
0000	0000	0000	0101	
0	0	0	5	Displacement 500H
1001	0011	0111	0010	
9	3	7	2	Data 7293 H

The complete machine code comes out to be C7 82 00 05 93 72.

Example 5.47

ADD AX, BX

The machine code is formed as shown by referring to Appendix and using the Tables 3.1 and 3.2 as has been already described.

OPCODE	D W	MOD	REG	R/M
0000	0011	11	000	011

The machine code is 03 C3.

Example 5.48

ADD AX, 5000H

The code formation is explained as follows:

OPCODE	S W	MOD	R/M
1000	0001	00	000000
8	1	0	0
0000	0000	0101	0000
0	0	5	0

The machine code is 81 00 00 50.

If s bit is 0, the 16-bit immediate data is available in the instruction.

Ifs bit is 1, the 16-bit immediate data is the sign extended form of 8-bit immediate data.

For example, if the eight bit data is 11010001, then its sign extended 16-bit version will be 1111111111010001.

Example 5.49

SHRAX

OPCODE	VW	MOD	REG	R/M
1101	0001	11	101	000
D	1		E	8

The instruction code is D1 E8.

Finding out Machine Code for Conditional JUMP (Intrasegment) Instructions: The Appendix shows that, corresponding to each of the conditional jump instructions, the first byte of the opcode is fixed and the jump displacement must be less than or equal to 127(D) bytes and greater than or equal to -128(D). This type of jump is called as short jump. The following conditional forward jump example explains how to find the displacement. The displacement is an 8-bit signed number. If the displacement is positive, it indicates a forward jump, otherwise it indicates a backward jump. The following example is a sequence of instructions rather than a single instruction to elaborate the procedure of the calculation of positive displacement for a forward jump.

Example 5.50

```
2000 ,01          XOR AX, BX
2002 ,03          JNZ OK
2004              NOP
2005              NOP
2006 ,7,8,9       ADD BX, 05H
200A  OK : HLT
```

The above sequence shows that the programmer wants a conditional jump to label OK, if the zero flag is not set. For finding out the displacement corresponding to the label OK, subtract the address of the jump instruction (2002H), from the address of label (200AH). The required displacement is 200AH - 2002H = 08H. The 08H is the displacement for the forward jump.

Let us find out the displacement for a backward jump. Consider the following sequence of instructions.

Example 5.51

```
2000, 01, 02      MOV    CL, 05H
2003      Repeat : INC    AX
2004              DEC    CL
2005,2006         JNZ    Repeat
```

For finding out the backward displacement, subtract the address of the label (Repeat) from the address of the jump instruction. Complement the subtraction. The lower byte gives the displacement. In this example, the signed displacement for the JNZ instruction comes out to be (2005H-2003H=02, complement-FDH) The magnitude of the displacement must be less than or equal to 127(D). The MSB of the displacement decides whether it is a forward or backward jump. If it is I, it is a backward jump or else it is a forward jump.

A similar procedure is used to find the displacement for intrasegment short calls.

Finding out Machine Code for Unconditional JUMP Intrasegment: For this instruction there are again two types of jump, i.e. short jump and long jump. The displacement calculation procedures are again the same as given in case of the conditional jump. The only new thing here is that, the displacement may be beyond ±127(D). This type of jump is called as long jump. The method of calculation of the displacement is again similar to that for short jump.

Finding out Machine Code for Intersegment Direct Jump: This type of instruction is used to make a jump directly to the address lying in another segment. The opcode itself specifies the new offset and the segment of jump address, directly.

Example 5.52

JUMP 2000:5000

This instruction implies a jump to a memory location in another code segment with CS = 2000H and Offset =5000H. The code formation is as shown.

Code formation	1110 1010	0000 0000	0101 0000	0000 0000	0010 0000
	Opcode	Offset LB	Offset HB	Seg. LB	Seg. HB

The opcode forms the first byte of this instruction and the successive bytes are formed from the segment and the offset of the jump destination. While specifying the segment and offset, the lower byte (LB) is specified first and then the higher byte (HB) is specified. Finally, the opcode comes out to be EA 00 50 00 20. The procedure of coding the CALL instructions is similar.

Hand Coding a Complete Program After studying the hand-coding procedures of different instructions, let us now tries to code a complete program. We will consider Example 5.53 for complete hand-coding. The program and the code corresponding to it is given. These codes, found using hand-coding, may be entered byte-by-byte into an 8086 based system and executed, provided it supports the memory requirements of the program.

Example 5.53

A Hand-coding Program Example

Addresses	Opcodes	Labels	Mnemonics
2000,01,02	B9 0F 00		MOV CX,0F H
2003,04,05	B8 00 02		MOV AX,2000H
2006,07	8E D8		MOV DS,AX
2008,09,0A	BE 00 05		MOV SI,0500H
200B,0C	89 04		MOV AX,[SI]
200D	46	BACK :	INC SI
200E,0F	3B 04		CMP AX,[SI]
2010,11	77 04		JNC NEXT
2012,13	89 04		MOV AX,[SI]
2014,15	E2 F7	NEXT :	LOOP BACK
2016	F4		HLT

Student Activity 5.4

Before reading the next section, answer the following question.

1. Find out the machine code for following instructions.

(i)	ADC AX, BX	(ii)	OR AX, [0500H]	(iii)	AND CX, [SI]
(iv)	TEST AX,5555H	(v)	MUL [SI+5	(vi)	NEG 50[BP]
(vii)	OUT DX, AX	(viii)	LES DI, [0700H]	(ix)	LEA SI, [BX+500H]
(x)	SHL [BX+2000],CL	(xi)	RET 0200H	(xii)	CALL 7000H
(xiii)	JMP 3000h:2000H	(xiv)	CALL [5000H]	(xv)	DIV [5000H]

2. Describe the procedure for coding the intersegment and intrasegment over machine language.
3. Enlist the advantages of assembly language programming over machine language.

If your answer is correct, then proceed to the next section.

Top

Programming with an Assembler

The procedure of hand-coding 8086 programs is somewhat tedious, hence in general a programmer may find it difficult to get a correct listing of the machine codes. Moreover, the procedure of handcoding is time consuming. This programming procedure is called as machine level programming. The obvious disadvantages of machine level programming are as given:

1. The process is complicated and time consuming.
2. The chances of error being committed are more at the machine level in hand-coding and entering the program byte-by-byte into the system.
3. Debugging a program at the machine level is more difficult.
4. The programs are not understood by everyone and the results are not stored in a user-friendly form.

A program called 'Assembler' is used to convert the mnemonics of instructions along with the data into their equivalent object code modules. These object code modules may further be converted in executable code using the linker and loader programs. This type of programming is called assembly level programming. In assembly language programming, the mnemonics are directly used in the user programs. The assembler performs the task of coding.

The advantages of assembly language over machine language are as given:

1. The programming in assembly language is not so complicated as in machine language because the function of coding is performed by an assembler.
2. The chances of error being committed are less because the mnemonics are used instead of numerical opcodes. It is easier to enter an assembly language program.
3. As the mnemonics are purpose-suggestive the debugging is easier.

4. The constants and address locations can be labeled with suggestive labels hence imparting a more friendly interface to user. Advanced assemblers provide facilities like macros, lists, etc. making the task of programming much easier.
5. The memory control is in the hands of users as in machine language.
6. The results may be stored in a more user-friendly form.
7. The flexibility of programming is more in assembly language programming as compared to machine language because of advanced facilities available with the modern assemblers.

Basically, the assembler is a program that converts an assembly input file also called as source file to an object file that can further be converted into machine codes or an executable file using a linker. The recent versions of the assembler are designed with many facilities like macroassemblers, numerical processor assemblers, procedures, functions and so on.

As far as this book is concerned, we will consider the assembly language programming using MASM (Microsoft Macro Assembler). There are a number of assemblers available like MASM, TASM and DOS assembler . MASM is one of the popular assemblers used along with a LINK program to structure the codes generated by MASM in the form of an executable file. MASM reads the source program as its input and provides an object file. The LINK accepts the object file produced by MASM as input and produces an EXE file.

While writing a program, for an assembler, your first step will be to use a text editor and type the program listing prepared by you. Then check the listing typed by you for any typing mistake and syntax error. Before you quit the editor program, do not forget to save it. Once you save the text file with any name (permissible on operating system), you are free to start the assembly process. A number of text editors are available in the market, e.g. Norton's editor [NE], Turbo C [TC], EDLIN, etc. Throughout this book, the NE is used. Any other free form editor may be used for a better user-friendly environment. Thus for writing a program in assembly language, one will need NE editor, MASM assembler, linker and DEBUG utility of DOS. In the following section, the procedures of opening a file for a program, assembling it, executing it and checking its result are described for beginners.

Entering a Program

In this section, we will explain the procedure for entering a small program on IBM PC with DOS operating system. Consider a program of addition of two bytes, as already discussed for handcoding. The same program is written along with some syntax modifications in it for MASM.

Before starting the process, ensure that all the files namely NE.COM (Norton's Editor), MASM.EXE (Assembler), LINK.EXE (linker), DEBUG.EXE (debugger) are available in the same directory in which you are working. Start the procedure with the following command after you boot the terminal and enter the directory containing all the files mentioned.

```
C> NE
```

You will get display on the screen as in Figure 5.16.

Now type the file name. It will be displayed on the screen. Suppose one types KMB.ASM as file name, the screen display will be as shown in Figure 5.17.

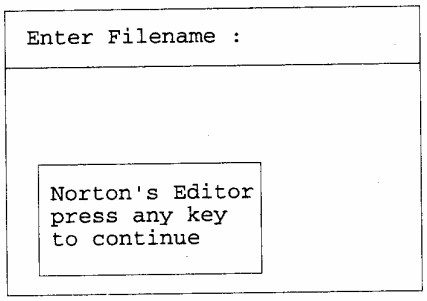


Figure 5.16: Norton's Editor Opening Screen

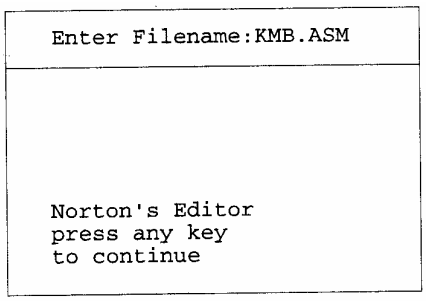


Figure 5.17: Norton's Editor Alternative Opening Screen

Press any of the keys, you will get Figure 5.18 as the screen display.

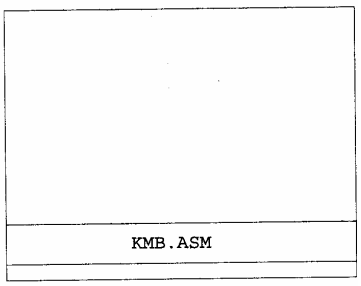


Figure 5.18: Norton's Editor Opens a New KMB.ASM

Note that, for every assembly language program, the extension .ASM must be there Though every time the assembler dose not use the complete name KMB.ASM and uses just KMB to handle the file, the extension shows that it is an assembly language program file. Even if you enter a file name without the .ASM extension, the assembler searches for the file name but with .ASM extension and if it is not available, it issues the message 'File not found'. Once you get the display as in Figure 5.18 you are free to type the program. One may use another type of command line as shown.

```
C> NE KMB.ASM
```

The above command will directly give the display as in Figure 5.18, if KMB.ASM is a newly opened file. Otherwise, if KMB.ASM is already existing in the directory, then it will be opened and the program in it will be displayed. You may modify it and save (command F3-E) it again, if you want the changes to be permanent. Otherwise, simply quit (command F3-Q) it to abandon the changes and exit NE . The entered program in NE looks like in Figure 5.19. We have to just consider that it is an assembly language program to be assembled. Store it with command F3-E. This will generate a new copy of the program in the secondary storage. Then quit the NE with command F3-Q.

Once the above procedure is completed, you may now turn towards the assembling of the above program. Note that all the commands and displays shown in this section are for Norton's Editor. Other editors may require some other commands and their display style may be some what different but the overall procedure is the same.

Note that before quitting the editor program the newly entered or modified file must be saved, otherwise it will be lost and will not be available for further assembling process.

```
ASSUME      CS:CODE, DS:DATA
DATA        SEGMENT
            OPR1 DW 1234 H
            OPR2 DW 0002 H
            RESULT DW 01 H DUP(?)

DATA        ENDS
CODE        SEGMENT
START       MOV AX, DATA
            MOV DS, AX
            MOV AX, OPR 1
            MOV BX, OPR 2
            CLC
            ADD AX, BX
            MOV DI, OFFSET RESULT
            MOV (DI), AX
            MOV AH, 4CH
            INT 21 H
CODE        ENDS
            END START
```

KBM.ASM

Figure 5.19: A Program KMB.ASM in Norton's Editor

Assembling a Program

Microsoft Assembler MASM is one of the easy to use and popular assemblers. All the references and discussions in this section are related to the MASM. As already discussed, the main task of any assembler program is to accept the text assembly language program file as input and prepare an object file. The text assembly language program file is prepared by using any of the editor programs. The MASM accepts the file names only with the extension .ASM. Even if a filename without any extension is given as input, it provides .ASM extension to it. For example, to assemble the program in Figure 5.19, one may enter the following command options-

```
A> MASM KMB
```

or

```
C ——>A> MASM KMB. ASM
```

If any of the command option is entered as above, the programmer gets the display on the screen, as shown in Figure 5.20.

```
C←A>MASM KMB

Microsoft @ Macro Assembler Version 5.10
Copyright (c) Microsoft Corp.1981, 1989.
All Rights Reserved



Object filename[.OBJ]:
List filename[NUL.LST]:
Cross Reference[NUL.CRF]:
```

Figure 5.20: MA.SM Screen Display

Also another command option, available in MASM that does not need any filename in the command line, is given along with the corresponding result display in Figure 5.21.

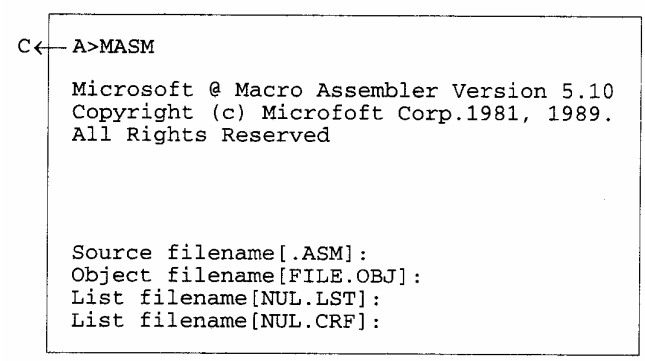


Figure 5.21: MASM Alternative Screen Display

If you do not enter the filename to be assembled at the command line as show in Figure 5.20, then you may enter it as a source filename as shown in Figure 5.21. The source filename is to be typed in the source filename line with or without the extension .ASM. The valid filename entry is accepted with a pressure of enter key. On the next line, the expected .OBJ filename is to be entered. The object file of the assembly language program is created with the .OBJ extension. The .OBJ file is created with the entered name and .OBJ extension the coded filename is entered for the .OBJ file before pressing enter key, the new .OBJ file is created with the same name as source file and extension.OBJ. The .OBJ file contains the coded object modules of the program to be assembled. On the next line, a filename is entered expected listing file of the source file, in the same way as the object filename was entered. The listing file is automatically generated in the assembly process. The listing file is identified by the entered or source filename and an extension .LST. The listing file contains the total offset map of the source file including labels, offset addresses, opcodes, memory allotment for different labels and directives and relocation information. The cross reference filename is also entered in the same way as discussed for listing file. The cross reference file is used for debugging the source program and contains the statistical information size of the file in bytes, number of labels, list of labels, routines to be called, etc. about 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 error code number and the corresponding line number at which they appear. After these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully. The successful assembly process may generate the .OBJ, .LST and .CRF files. Those may further be used by the linker programmer to link the object modules and generate an executable (.EXE) file from a .OBJ file. All the above said files may not be generated during the assembling of every program. The generation of some of them may be suppressed using the specific command line options of MASM. The files generated by the MASM are further used by the program LINK.EXE to generate an executable file of the source program.

Linking a Program

The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. The other supporting information may be obtained from the files generated by MASM. The linker program is invoked using the following options.

```
C ——> A> LINK
```

or

```
C ——> A> LINK KMB.OBJ
```

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first option may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The other option also generates the similar display, but will not ask for the .OBJ filename, as it is already specified at the command line. If no filenames are entered for these files, by default, the source filename is considered with the different extensions. The procedure of entering the filenames in LINK is also similar to that in MASM. The LINK command display is as shown in Figure 5.22.

The option input 'Libraries' in the display of Figure 5.22 expects any special library name of which the functions were used by the source program. The output of the LINK program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced versions of MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options that cannot be detailed here for the obvious reasons. For further details users may refer to "Technical reference and Users' Manual-MASM, Version 5".

```
C←——A>LINK

Microsoft @ Overlay Linker Version. 3.64
Copyright(c)Microsoft Corp. 1983-88. All Rights Reserved.


Object Module[.OBJ]:
Run file[.EXE]:
List file[NUL.MAP]:
Libraries[LIB]:
```

Figure 5.22: Link Command Screen Display

Using DEBUG

DEBUG.COM is a DOS utility that facilitates the debugging and trouble-shooting of assembly language programs. In case of personal computers, all the processor resources and memory resource management functions are carried out by the operating systems. Hence, users have very little control over the computer hardware at lower levels. The DEBUG utility enables you to have the control of these resources up to some extent. In the simplest, rather, crude words, the DEBUG enables you to use the personal computer as a low level microprocessor kit.

The DEBUG command at DOS prompt invokes this facility. A '_' (dash) display signals the successful invoke operation of DEBUG, that is further used as DEBUG prompt for debugging commands. The following command line, DEBUG prompt and the DEBUG command character display explain the DEBUG command entry procedure, as in Figure 5.23.

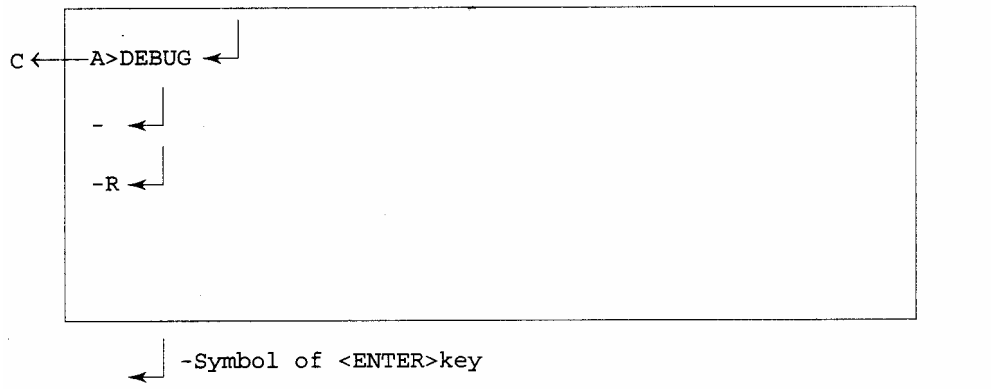


Figure 5.23: DEBUG Command Line and Prompt

A valid command is accepted using the enter key. The list of generally used valid commands of DEBUG is given in Table 5.3 along with their respective syntax.

The program DEBUG may be used either to debug a source program or to observe the results of execution of an .EXE file with the help of the .LST file and the above commands. The .LST file shows the offset address allotments for result variables of a program in the particular segment. After execution of the program, the offset address of the result variables may be observed using the d command. The results available in the registers may be observed using the r command. Thus the DEBUG offers a reasonably good platform for trouble-shooting executing and observing the results of the assembly language programs. Here one should note that the DEBUG is able to trouble-shoot only .EXE files.

Table 5.3: [DEBUG Commands]

COMMAND CHARACTER	*Format / Formats*	*Functions*
-R	<ENTER>	Display all Registers and flags
-R	reg<ENTER> old contents:New contents	Display specified register contents and modify with the entered new contents.
-D	<ENTER>	Display 128 memory locations of RAM starting from the current display pointer.
-D	SEG:OFFSET1 OFFSET2<ENTER>	Display memory contents in SEG from OFFSET1 to OFFSET2.
-E	<ENTER>	Enter Hex data at current display pointer SEG:OFFSET.
-E	SEG:OFFSET1 <ENTER>	Enter data at SEG:OFFSET1 byte by byte. The memory pointer is to be incremented by space key, data entry is to be completed by pressing the <ENTER> key.
-f	SEG:OFFSET1 OFFSET2 BYTE <ENTER>	Fill the memory area starting from SEG:OFFSET1 to OFFSET2 by the byte BYTE.
-f	SEG:OFFSET1 OFFSET2 BYTE1,BYTE2,BYTE3<ENTER>	Fill the memory area as above with the sequence BYTE1, BYTE2, BYTE3, etc.
-a	<ENTER>	Assemble from the current CS:IP.
-a	SEG:OFFSET <ENTER>	Assemble the entered instruction from SEG:OFFSET address.
-u	<ENTER>	Unassemble from the current CS:IP.
-u	SEG:OFFSET <ENTER>	Unassemble from the address SEG:OFFSET.
-g	<ENTER>	Execute from current CS:IP. By modifying CS and IP using R command this can be used for any address.
-g	=OFFSET <ENTER>	Execute from OFFSET in the current CS.
-s	SEG:OFFSET1 to OFFSET2 BYTE/BYTES <ENTER>	Searches a BYTE or string of BYTES separated by ',' in the memory block SEG:OFFSET1 to OFFSET2, and displays all the offsets at which the byte or string of bytes is found.
-q	<ENTER>	Quit the DEBUG and return to DOS
-T	SEG:OFFSET <ENTER>	Trace the program execution by single stepping starting from the address SEG:OFFSET.
-m	SEG:OFFSET1 OFFSET2 NB <ENTER>	Move NB bytes from OFFSET1 to OFFSET2 in segment SEG
-c	SEG:OFFSET OFFSET2 NB <ENTER>	Copy NB bytes from OFFSET1 to OFFSET2 in segment SEG.
-n	FILENAME.EXE <ENTER>	Set filename pointer to FILENAME
-l	<ENTER>	Load the file FILENAME.EXE as set by the -n command in the RAM and set the CS:IP at the address at which the file is loaded.

- Note that, changing the case of the command letters does not change the command option.
- The entered nubers are consiered as hexadecimal

Student Activity 5.5

Before reading the next section, answer the following question.

1. Write an ALP to convert a four digit hexadecimal number to decimal number.

2. Write an ALP to covert a four digit octal number to decimal number.

3. Write an ALP to fund out ASCSII codes of alphanumeric characters from a look up table.

If your answer is correct, then proceed to the next section.

Top

Assembly Language Example Programs

In the previous chapter, we studied the complete instruction set of 8086/88, the assembler directives and pseudo-ops. In the previous sections, we have described the procedure of entering an assembly language program into a computer and coding it, i.e. preparing an EXE file from the source code. In this section, we will study some programs which elucidate the use of instructions, directives and some other facilities that are available in assembly language programming. After studying these programs, it is expected that one should have a clear idea about the use of different directives and pseudo-ops, their syntaxes besides understanding the logic of each program. If one writes an assembly language program and tries to code it, in the first attempt, the chances are rare that there is no error. Rather, errors in programming depend upon the skill of the Programmer. A lot of practice is needed to obtain skills in assembly language programming as compared to high level languages. So, to obtain a command on assembly language one should write and execute a number of assembly programs, besides studying the example programs given in this text.

Before starting the explanation of the written programs, we have to explain one more important point, that is about the DOS function calls available under INT 21H instruction. DOS is an operating system, which is a program that stands between a bare computer system hardware and a user. It acts as a user interface with the available computer hardware resources. It also manages the hardware resources of the computer system. In the Disk Operating System, the hardware resources of the computer system like memory, keyboard, CRT display, hard disk, and floppy disk drives can be handled with the help of the instruction INT 21H. The routines required to refer to these resources are written as interrupt service routines for 21H interrupt. Under this interrupt, the specific resource is selected depending upon the value in AH register. For example, if AH contains 09H, then CRT display is to be used for displaying a message or if, AH contain OAH, then the keyboard is to be accessed. These interrupts are called 'function calls' and the value in AH is called 'function value'. In short, the purpose of 'function calls' under INT 21 His to be decided by the 'function value' that is in AH. Some function values also control the software operations of the machine. The list of the function values available under INT 21 H, their corresponding functions, the required parameters and the returns are given in tabulated form in the Appendix-B. Note that there are a number of interrupt functions in DOS, but INT 21H is used more frequently. The readers may find other interrupts of DOS and BIOS from the respective technical references.

Here, a few example programs are presented. Starting from very simple programs, the chapter concludes with more complex programs.

Program 5.1

Write a program for addition of two numbers.

Solution

The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program.

```
ASSUME CS:CODE,DS:DATA
DATA      SEGMENT
OPR1      DW 1234H                ; 1st operand
OPR2      DW 0002H                ; 2nd operand
RESULT    DW 01 DUP(?)            ; A word of memory reserved for result
DATA      ENDS
CODE      SEGMENT
START:    MOV AX,DATA             ; Initialize data segment
          MOV DS,AX               ;
          MOV AX,OPR1             ; Take 1st operand in AX.
          MOV BX,OPR2             ; Take 2nd operand in BX.
          CLC                     ; Clear previous carry if any.
          ADD AX,BX               ; Add BX to AX.
          MOV DI,OFFSET RESULT    ; Take offset of RESULT in DI.
          MOV [DI],AX             ; Store the result at memory address in DI.
          MOV AH,4CH              ; Return to DOS prompt.
          INT 21H
CODE      ENDS                    ; CODE segment ends.
          END START               ; Program ends
```

Program 5.1: Listings

How to Write an Assembly Language Program

The first step in writing an assembly language program is to define and study the problem. After studying the problem, decide the logical modules required for the program. From the statement of the program one may guess that some data may be required for the program or the result of the program that is to be stored in memory. Hence the program will need a logical space called DATA segment. Invariably CODE segment is a part of a program containing the actual instruction sequence to be executed. If the stack facility is to be used in the program, it will require the STACK segment. The EXTRA segment may be used as an additional destination data segment. Note that the use of all these logical segments is not compulsory except for the CODE segment. Some programs may require DATA and CODE segments, while the others may also contain STACK and EXTRA. For example, Program 5.1 requires only DATA and CODE segment.

The first line of the program containing 'ASSUME' directive declares that the label CODE is to be used as a logical name for CODE segment and the label DATA is to be used for DATA segment. These labels CODE and DATA are reserved by MASM for these purposes only.

They should not be used as general labels. Once this statement is written in the program, CODE refers to the code segment and DATA refers to data segment throughout the program If you want to change it in a program you will have to write another ASSUME statement the program.

The second statement DATA SEGMENT marks the starting of a logical data space DATA Inside DATA segment, OPRI is the first operand. The directive DW defines OPRI as a word operand of value 1234H and OPR2 as a word operand of value 0002H. The third DW directive reserves OIH words of memory for storing the result of the program and leaves it undefined due to the directive DUP(?). The statement DATA ENDS marks the end of the DATA segment. Thus the logical space DATA contains OPRI, OPR2 and RESULT. These labels OPRI, OPR2 and RESULT will be allotted physical memory locations whenever the logical SEGMENT DATA is allocated memory or loaded in the memory of a computer as explained in the previous topic of relocation. The assembler calculates that the above data segment requires 6 bytes, i.e. 2 bytes each for OPRI, OPR2 and RESULT.

The code segment in the above program starts with the statement CODE SEGMENT. The code segment as already explained is a logical segment space containing the instructions. The label STARTS marks the starting point of the execution sequence. The ASSUME statement just informs the assembler that label

CODE is used for the code segment and the label DATA is used for the DATA segment. It does not actually put the address corresponding to CODE in code segment register (CS) and address corresponding to DATA in the data segment register (DS). This procedure of putting the actual segment address values into the corresponding segment registers is called as segment register initialisation. A programmer has to carry out these initializations for DS, SS and ES using instructions, while the CS automatically initialised by the loader at the time of loading the EXE file into the memory for actual execution. The first two instructions in the program are for data segment initialization. Note that, no segment register in 8086 can be loaded with immediate segment address value, instead the address value should be first loaded into any one of the general purpose registers and then the contents of the general purpose register can be transferred to any of the segment registers DS, ES and SS. Also one should note that CS cannot be loaded at all. Its contents can be changed by using a long jump instruction, a call instruction or an interrupt instruction. For each of the segments DS, ES and SS, the programmer will have to carry out initialization if they are used in the program, while CS is automatically initialized by the loader program at the time of loading and execution. Then the two instructions move the two operands OPRI and OPR2 in AX and BX respectively. Carry is cleared before addition operation (optional in this program). The ADD instruction will add BX into AX and store the result in AX. The instruction used to store the result in RESULT uses different addressing mode than that used for taking OPRI into AX. The indexed addressing mode is used to store the result of addition in memory locations labeled RESULT. The instruction MOVDI, OFFSET RESULT stores the offset of the label RESULT into Dl register. Next instruction stores the result available in AX into the address pointed to by Dl, i.e address of the RESULT. A lot has been already discussed about the function calls under INT 21H. The function value 4CH is for returning to the DOS prompt. If instead of these one writes HLT instruction there will not be any difference in program execution except that the computer will hang as the processor goes to HLT state, and the user will not be able to examine the result. In that case, for further operation, one will have to reset the computer and boot it again. To avoid this resetting of the computer every time you run the program and enable one to check the result, it is better to use the function call 4CH at the end of each program so that after executing the program, the computer returns back to DOS prompt. The statement CODE ENDS marks the end of the CODE segment. The statement END START marks the end of the procedure that started with the label START. At the end of each file, the END statement is a must.

Until now we have discussed every line of Program 5.1 in significant details. As we have already said, the program contains two logical segments CODE and DATA, but it is not at all necessary that all the programs must contain the two segments. A programmer may use a single segment to cover up data as well as instructions. The following listings explain the fact.

```
ASSUME CS:CODE
          CODE        SEGMENT
          OPR1        DW   1234H
          OPR2        DW   0002H
          RESULT      DW   01 DUP(?)
START :   MOV AX,CODE
          MOV DS,AX
          MOV AX,OPR1
          MOV BX,OPR2
          CLC
          ADD AX,BX
          MOV DI,OFFSET RESULT
          MOV [DI],AX
          MOV AH,4CH
          INT 21H
CODE      ENDS
          END START
```

Program 5.1: Altemative listing for program 1

For this program, we have discussed all clauses and clones in details. For all the following programs, we will not explain the common things like forming segments using directives and operators, etc. Instead, just the logic of the program will be explained. The use of proper syntax of the 8086/8088 assembler MASM is self explanatory. The comments may help the reader in getting the ideas regarding the logic of the program.

Assemble the above written program using MASM after entering it into the computer using the procedure explained earlier. Once you get the EXE file as the output of the LINK program, Your listing is ready for execution. The Program 5.1 is prepared in the form of EXE file with the name KMB.EXE in the directory. Next, it can be executed with the command line as given below.

```
C> KMB
```

This execution method will store the result of the program in memory but will not display it on the screen. To display the result on the screen the programmer will have to use DOS function calls. This will make the programs too lengthy. Hence, another method to check the results is to run the program under the control of DEBUG. To run the program under the control of debug and to observe the results one must prepare the LST file, that gives information about memory allotment to different labels and variables of the program while assembling it. The LST file can be displayed on the screen using NE-Norton's Editor.

Program 5.2

Write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers).

Solution

In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                        ; Data segment starts
NUMLIST DB 52H,23H,—                ; List of byte numbers
COUNT EQU 100D                      ; Number of bytes to be added
RESULT DW 01H DUP(?)                ; One word is reserved for result.
DATA ENDS                           ; Data segment ends
CODE SEGMENT                        ; Code segment starts at relative
ORG 200H                            ; address 0200h in code segment.
START:      MOV AX,DATA             ; Initialize data segment.
            MOV DS,AX
            MOV CX,COUNT            ; Number of bytes to be added in CX.
            XOR AX,AX               ; Clear AX and CF.
            XOR BX,BX               ; Clear BH for converting the byte to word
            MOV SI,OFFSET NUMLIST   ; Point to the first number in the list.
AGAIN:      MOV BL,[SI]             ; Take the first number in BL, BH is zero
            ADD AX,BX               ; Add AX with BX.
            INC SI                  ; Increment pointer to the byte list.
            DEC CX                  ; Decrement counter.
            JNZ AGAIN               ; If all numbers are added,point to result
            MOV DI,OFFSET RESULT    ; destination and store it.
            MOV [DI],AX
            MOV AH,4CH              ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program 5.2: Listings

The use of statement ORG 200H in this program is not compulsory. We have used this statement here just to explain the way to use it. It will not affect the result at all. Whenever the program is loaded into the memory whatever is the address assigned for CODE, the executable code starts at the offset address 0200H due to the above statement. Similar to DW, the directive DB reserves space for the list of 8-bit numbers in the series. The procedure for entering the program, coding and execution has already been explained. The result of addition will be stored in the memory locations allotted to the label RESULT.

Program 5.3

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution

Compare the ith number of the series with the (i+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (i+1)th number is greater. If the ith number is greater than (i+1)th, leave it in AX (any register may be used). Otherwise, load the (i+1)th number in AX, replacing the ith number in AX. The procedure is repeated tilt all the members in the array have been compared.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT                          ; Data segment starts
LIST DB 52H,23H,56H,45H,—             ; List of byte numbers
COUNT EQU 0F                          ; Number of bytes in the list
LARGEST DB 01H DUP(?)                 ; One byte is reserved for the largest number.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts.
START:      MOV AX,DATA               ; Initialize data segment.
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT              ; Number of bytes in CL.
            MOV AL,[SI]               ; Take the first number in AL
AGAIN:      CMP AL,[SI+1]             ; and compare it with the next number.
            JNL NEXT
            MOV AL,[SI+1]
NEXT:       INC SI                    ; Increment pointer to the byte list.
            DEC CL                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are compared,point to result
            MOV SI,OFFSET LARGEST     ; destination and store it.
            MOV [SI],AL
            MOV AH,4CH                ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program 5.3 Listings

Program 5.4

Modify the Program 5.3 for a series of words.

Solution

The logic is similar to the previous program written for a series of byte numbers. The program is directly written as follows without any comment leaving it to the reader to find out the use of each instruction and directive used.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 1234H,2354H,0056H,045AH,—
COUNT EQU 0F
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV SI,OFFSET LIST
            MOV CL,COUNT
            MOV AX,[SI]
AGAIN:      CMP AX,[SI+2]
            JNL NEXT
            MOV AX,[SI+2]
NEXT:       INC SI
            INC SI
            DEC CL
            JNZ AGAIN
            MOV SI,OFFSET LARGEST
            MOV [SI],AX
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END     START
```

Program 5.4 Listings

Program 5.5

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Solution

The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number. If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2357H,0A579H,0C322H,0C91EH,0C000H,0957H
COUNT EQU 006H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI, OFFSET LIST
AGAIN:      MOV AX,[SI]

            ROR AX,01
            JC ODD
            INC BX
            JMP NEXT
ODD:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

Program 5.5: Listings

Program 5.6

Write a program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Solution

Take the ith number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. If CF is 1, the number is negative; otherwise, it is positive.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2579H,0A500H,0C009H,0159H,0B900H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:      XOR BX,BX
            XOR DX,DX
            MOV AX,DATA
            MOV DS,AX
            MOV CL,COUNT
            MOV SI,OFFSET LIST
AGAIN:      MOV AX,[SI]
            SHL AX,01
            JC NEG
            INC BX
            JMP NEXT
NEG:        INC DX
NEXT:       ADD SI,02
            DEC CL
            JNZ AGAIN
            MOV AH,4CH
            INT 21H
            CODE ENDS
            END START
```

Program 5.6: Listing

The logic of Program 5.6 is similar to that of Program 5.5, hence comments are not given in Program 5.6 except for a few important ones.

Program 5.7

Write a program to move a string of data words from offset 2000H to offset 3000H the length of the string is OFH.

Solution

To write this program, we will use an important facility, available in the 8086 instruction set, i.e. move string byte/word instruction. We will also study the flexibility imparted by this instructions to the 8086 assembly language program. Let us first write the Program 5.7 for 8085, assuming that the string is available at location 2000H and is to be moved at 3000 H.

```
            LXI H , 2000H
            LXI D , 3000H
            MVI C , 0FH
AGAIN :     MOV A , M
            STAX D
            INX  H
            INX  D
            DCR  C
            JNZ  AGAIN
            HLT
```

An 8085 Program for Program 5.7

Now assuming DS is suitably set, let us write the sequence for 8086. At first using the index registers, the program can be written as given.

```
            MOV  SI ,2000H
            MOV  DI ,3000H
            MOV  CX ,0FH
AGAIN :     MOV  AX ,[SI]
            MOV  [DI],AX
            INC  SI
            INC  DI
            DEC  CX
            JNZ  AGAIN
            HLT
```

An 8086 Program for Program 5.7

Comparing the above listings for 8085 and 8086 we may infer that every instruction in 8085 listing is replaced by an equivalent instruction of 8086. The above 8086 listing is absolutely correct but it is not efficient. Let us try to write the listings for the same purpose using the string instruction. Due to the assembler directives and the syntax, one may feel that the program is lengthy, though it eliminates four instructions for a MOVSW instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH

DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV SI,SOURCESTRT
            MOV DI,DESTSTRT
            MOV CX,COUNT
            CLD
REP         MOVSW
            MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.7

An 8086 Program listing for Program 5.7 using String instruction

Compare the above two 8086 listings. Both contain ten instructions. However, in case of the second program, the instruction for the initialisation of segment register and DOS interrupt are additional while the first one neither contains initialisation of any segment registers nor does it contain the DOS interrupt instruction. We can say that the first program uses 9 instructions, while the second one uses only 5 for implementing the same algorithm. This program and the related discussions are aimed at explaining the importance of the string instructions and the method to use them.

Program 5.8

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Solution

There exist a large number of sorting algorithms. The algorithm used here is called bubble sorting. The method of sorting is explained as follows. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series (n-1) times. After (n-1) iterations, you will get the largest number at the end of the series, where n is the length of the series. Again start from the first address of the series. Repeat the same procedure right from the first element to the last element. After (n-2) iterations you will get the second highest number at the last but one place in the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

53 , 25 , 19, 02		n = 4
25 , 53 , 19, 02		1st operation
25 , 19 , 53 , 02		2nd operation
25 , 19 , 02 , 53		3rd operation
largest no.	⇒	4 – 1 = 3 operations
19 , 25 , 02 , 53		1st operation
19 , 02 , 25 , 53		2nd operation

2nd largest number => 4-2=2 operations

02, 19, 25, 53 1st operation

3rd largest number => 4 -3 = 1 operations

Instead of taking a variable count for the external loop in the program like (n - 1), (n – 2), (n-3),...., etc. It is better to take the count (n- 1) all the time for simplicity. The resulting program is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 53H,25H,19H,02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DX,COUNT-1
AGAIN0:     MOV CX,DX
            MOV SI,OFFSET LIST
AGAIN1:     MOV AX,[SI]
            CMP AX,[SI+2]
            JL  PR1
            XCHG [SI+2],AX
            XCHG [SI],AX
PR1:        ADD SI,02
            LOOP AGAIN1
            DEC DX
            JNZ AGAIN0
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.8 Listings

With a similar approach, the reader may write a program to arrange the string in descending order. Just instead of the JL instruction in the above program, one will have to use a JG instruction.

Program 5.9

Write a program to perform a one byte BCD addition.

Solution

It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

```
  92
+ 59
------
  EB      Actual result after addition considering hex. operands
```

```
    1 0 1 1
+   0 1 1 0      As 0BH (LSD of addition) > 09, add 06 to it.
-------------
  1 0 0 0 1      Least significant nibble of result (neglect the auxiliary carry)
                 → AF is set to 1
```

0110 is added to most significant nibble of the result if it is greater than 9 or AF is set.

```
                    1      Carry from previous digit (AF)
      E     →   1 1 1 0
            +   0 1 1 0
          -------------
CF is set to 1    0 1 0 1  next significant nibble of result
```

Result CF	Most significant	Least significant digit
1	5	1

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 92H
            OPR2 EQU 52H
RESULT DB 02 DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS,AX
            MOV BL,OPR1
            XOR AL,AL
            MOV AL,OPR2
            ADD AL,BL
            DAA
            MOV RESULT,AL
            JNC MSB0
            INC [RESULT+1]
MSB0:       MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.9 Listings

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after SUB instruction. The reader may try to write a program for BCD subtraction for practice.

Program 5.10

Write a program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

Solution

Here we have directly given the routine for Program 5.10.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            OPR1 EQU 98H
```

```
                OPR2 EQU 49H
                SUM DW 01 DUP(00)
                SUBT DW 01 DUP(00)
                PROD DW 01 DUP(00)
                DIVS DW 01 DUP(00)
DATA            ENDS
CODE            SEGMENT
START:          MOV AX,DATA
                MOV DS,AX
                MOV BL,OPR2
                XOR AL,AL
                MOV AL,OPR1
                ADD AL,BL
                DAA
                MOV BYTE PTR SUM,AL
                JNC MSB0
                INC [SUM+1]
MSB0:           XOR AL,AL
                MOV AL,OPR1
                SUB AL,BL
                DAS
                MOV BYTE PTR SUBT,AL
                JNB MSB1
                INC [SUBT+1]
MSB1:           XOR AL,AL
                MOV AL,OPR1
                MUL BL
                MOV WORD PTR PROD,AX
                XOR AH,AH
                MOV AL,OPR1
                DIV BL
                MOV WORD PTR DIVS,AX
                MOV AH,4CH
                INT 21H
CODE            ENDS
                END START
```

Program 5.10 Listings

Program 5.11

Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Solution

The given string is scanned for the given byte. If it is found in the string, the zero flag is set; else, it is reset. Use of the SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found out. Note that, in this program, the code segment is written before the data segment.

```
ASSUME CS:CODE,DS:DATA
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV CX,COUNT
            MOV DI,OFFSET STRING
            MOV BL,00H
            MOV AL,BYTE1
SCAN1:      NOP
            SCASB [DI]
            JZ XXX
            INC BL
            LOOP SCAN1
XXX:        MOV AH,4CH
            INT 21H
            CODE ENDS
DATA SEGMENT
BYTE1 EQU 25H
COUNT EQU 06H
STRING DB 12H,13H,20H,20H,25H,21H
DATA ENDS
            END START
```

Program 5.11 Listings

Program 5.12

Write a program to convert the BCD numbers 0 to 9 to their equivalent sever segment codes using the look-up table technique. Assume the codes [7-seg] are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

Solution

Refer to the explanation of the XLAT instruction. The statement of the program itself gives the explanation about the logic of the program.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            CODELIST DB 34,45,56,45,23,12,19,24,21,00
            CHAR EQU 05
            CODEC   DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV BX,OFFSET CODELIST
            MOV AL, CHAR
            XLAT
            MOV BYTE PTR CODEC,AL
```

```
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.12 Listings

Program 5.13

Decide whether the parity of a given number is even or odd. If parity is even set DL to 00; else, set DL to 01. The given number may be a multibyte number.

Solution

The simplest algorithm to check the parity of a multibyte number is to go on adding the parity byte by byte with OOH. The result of the addition reflects the parity of that byte of the multibyte number. Adding the parities of all the bytes of the number, one will obtain the over all parity of the number.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            NUM DD 335A379BH
            BYTE_COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV DH,BYTE_COUNT
            XOR AL,AL
            MOV CL,00
            MOV SI,OFFSET NUM
NEXT_BYTE:  ADD AL,[SI]
            JP EVENP
            INC CL
EVENP:      INC SI
            MOV AL,00
            DEC DH
            JNZ NEXT_BYTE
            MOV DL,00
            RCR CL,1
            JNC CLEAR
            INC DL
CLEAR:      MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.13 Listings

The contents of CL are incremented depending upon either the parity for that byte is even or odd. If LSB of CL is I, after testing for all bytes, it means the parity of the multibyte number is odd otherwise it is even and DL is modified correspondingly.

Program 5.14

Write a program for the addition of two 3 x 3 matrices . The matrices are stored in the form of lists (row wise). Store the result of addition in the third list.

Solution

In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

$$[A] \quad + \quad [B] \quad = \quad [A+B]$$

The matrix A is stored in the memory at an offset MAT1, as given:

a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{32}, a_{33}, etc.

A total of 3 x3= 9 additions are to be done. The assembly language program is written as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            DIM EQU 09H
            MAT1 DB 01,02,03,04,05,06,07,08,09
            MAT2 DB 01,02,03,04,05,06,07,08,09
            RMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,DIM
            MOV SI,OFFSET MAT1
            MOV DI,OFFSET MAT2
            MOV BX,OFFSET RMAT3
NEXT:       XOR AX,AX
            MOV AL,[SI]
            ADD AL, [DI]
            MOV WORD PTR [BX],AX

            INC SI
            INC DI
            ADD BX,02
            LOOP NEXT
            MOV AH,4CH
            INT 21H
CODE        ENDS
            END START
```

Program 5.14 Listing

Program 5.15

Write a program to find out the product of two matrices. Store the result in the third matrix. The matrices are specified as in the Program 5.14.

Solution

The multiplication of matrices is carried out as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{12}b_{12} + a_{22}b_{12} + a_{13}b_{32} & a_{13}b_{13} + a_{12}b_{23} + a_{13}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

The listings to carry out the above operation is given as shown.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
ROCOL EQU 03H
MAT1 DB 05H,09H,0AH,03H,02H,07H,03H,00H,09H
MAT2 DB 09H,07H,02H,01H,0H,0DH,7H,06H,02H
PMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CH,ROCOL
            MOV BX,OFFSET PMAT3
            MOV SI,OFFSET MAT1
NEXTROW:    MOV DI,OFFSET MAT2
            MOV CL,ROCOL
NEXTCOL:    MOV DL,ROCOL
            MOV BP,0000H
            MOV AX,0000H
            SAHF
```

```
NEXT_ELE:   MOV AL,[SI]
            MUL BYTE PTR[DI]
            ADD BP,AX
            INC SI
            ADD DI,03
            DEC DL
            JNZ NEXT_ELE
            SUB DI,08
            SUB SI,03
            MOV [BX],BP
            ADD BX,02
            DEC CL
            JNZ NEXTCOL
            ADD SI,03
            DEC CH
            JNZ NEXTROW

            MOV AH,4CH
            INT 21H
CODE ENDS
END START
```

Program 5.15 Listings

Program 5.16

Write a program to add two multibyte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

Solution

This program is similar to the program written for the addition of two matrices except for the addition instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BYTES EQU 08H
NUM1 DB 05,5AH,6CH,55H,66H,77H,34H,12H
NUM2 DB 04,56H,04H,57H,32H,12H,19H,13H
NUM3 DB 0AH DUP(00)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
            MOV DS,AX
            MOV CX,BYTES
            MOV SI,OFFSET NUM1
            MOV DI,OFFSET NUM2
            MOV BX,OFFSET NUM3
            XOR AX,AX
NEXTBYTE:   MOV AL,[SI]
            ADC AL,[DI]
```

```
                MOV BYTE PTR[BX],AL
                INC SI
                INC DI
                INC BX
                DEC CX
                JNZ NEXTBYTE
                JNC NCARRY
                MOV BYTE PTR[BX],01
NCARRY:         MOV AH,4CH
                INT 21H
CODE ENDS
                END START
```

Program 5.16 Listings

Student Activity 5.6

Answer the following question.

1. Write an ALP to perform a sixteen-bit increment operation using 8-bit instructions.
2. Write an ALP to find out average of a given string of data bytes neglecting fractions.
3. Write an ALP to find out decimal addition of sixteen four digit decimal numbers.
4. Write an ALP to convert a four digit decimal number to its binary equivalent.

Summary

- This unit is aimed at introducing the readers with the instruction set of 8086/88 and the most commonly used assembler directives and operators. To start with, the available instruction formats in 8086/88 instruction set are explained in details. Further, the addressing modes available in 8086/88 are discussed in significant details with necessary examples. The 8086/88 instructions can be broadly categorized in six types depending upon their functions, namely data transfer instructions, arithmetic instructions and logical instructions, shift and rotate instructions, string manipulation instructions, control transfer instructions and processor control instructions. All these instruction types have been discussed before proceeding with the assembler directives and operators. The coding information details of all these instructions may be obtained from the Intel Appendix. The necessary assembler directives have been discussed later with their possible syntax, functions and examples. Most of these directives are available in Microsoft MASM. The detailed discussion on every assembler directive and operator is out of scope of this book.
- This unit starts with simple programs written for hand-coding. Further, hand-coding procedures of a few example instructions are studied. Advantages of the assembly language over machine language are then presented in brief. With an overview of the assembler operation, we have initiated the discussion of assembly language programming. The procedures of writing, entering and coding the assembly language programs are then discussed in brief. Further, DOS debugging program - DEBUG, a basic tool for trouble-shooting the assembly language programs, is discussed briefly with an emphasis on the most useful commands. Then we have presented various examples of 8086/8088 programs those highlight the actual use and the syntax of the instructions and directives. Finally, a few programs those enable the programmer to access the computer system resources using DOS function calls are discussed. We do not claim that each program presented here is the most efficient one, rather it just suggests a way to implement the algorithm asked in the problem. There may be a number of

alternate algorithms and program listings to implement a logic but amongst them a programmer should choose one which requires minimum storage, execution time and complexity.

Self-assessment Questions

Solved Exercises

I. State True or False

1. Assembly language depends on the CPU architecture of the machine for which it was designed.
2. An assembly language for solving a problem is very efficient compared to the same program written in a High Level Language.
3. Writing application programs in assembly language is difficult compared to writing in high level language.
4. There are many assemblers available for the IBM PC.
5. In immediate type addressing the time required to fetch the desired data is zero.

II. Fill in the blanks

1. Assembly language use _____ to represent operation codes.
2. Assembly language use _____ to represent operands.
3. One assembly language instruction become _____machine for which it was designed.
4. A translator which translates an assembly language program to machine language is called _____.
5. Early IBM PCs used _____ chip as CUP.

Answers

I. State True or False

1. True
2. True
3. True
4. False
5. False

II. Fill in the blanks

1. mnemonics
2. symbols
3. one

4. compiler
5. Intel 8088/8086

Unsolved Exercises

I. State True or False

1. Direct addressing mode is used when speed is important.
2. Registers are normally directly addressed as the number of registers in CPU is small.
3. A disadvantage of immediate addressing is negative operands cannot be used.
4. Indirect addressing through a register requires 3 memory accesses to fetch data.
5. Subroutines are self-contained programs which can be assembled separately.
6. A subroutine may call itself.
7. A subroutine can be independently debugged.
8. When a program call a subroutine the return address is stored in bottom of stack.

II. Fill in the blanks

1. Assembly language operand can be a _____.

   List all correct answers::

   (a) number (b) character constant
   (c) variable name (d) an absolute address
   (e) symbolic address (f) mnemonic code

2. Directives in an assembly language are required to _____

   (Pick as many as relevant).

   (a) read a program (b) specify values to be stored
   (c) allocate space in memory (d) find execution time
   (e) find error in program

3. Data registers are used to stone _____.

   (a) instructions (b) operands
   (c) operators (d) operators

4. ALU stands for _____.

   (a) All Logic Unit (b) Arithmetic Logic Unit
   (c) Another Legal Unit (d) All-purpose Language Utility

5. The number of bits in an address bus equals the number of bits in _____.

   (a) memory (b) data register

(c) address register (d) flags resister

6. The operation code of an assembly language program is represented by a _____.

(a) code (b) mnemonic

(c) instruction (d) number

7. At the end of the sequence of instructions

MOV AX, 0F289h

MOV BL, AH

BL will store _____.

(a) 89h (b) F2h

(c) 28h (d) F9h

Detailed Questions

1. What is an assembler?
2. What is a linker?
3. Write an ALP to change an already available ascending order byte string to descending order.
4. Write an ALP to convert a given sixteen bit binary number to its GRAY equivalent.
5. Write an ALP to find out transpose of a 3x3 matrix.
6. Write an ALP to find out cube of an 8-bit hexadecimal number.

Appendix

Mnemonics & Description	*Instruction Code*			
Data Transfer				
MOV = Move	76543210	76543210	76543210	76543210
Register/Memory to/from Register	100010 dw	mod reg r/m		
Immediate to Register/Memory	1100011 w	mod 000 r/m	data	data if w = 1
Immediate to Register	1011 w reg	data	data if w = 1	
Memory to Accumulator	1010000 w	addr-low	addr-high	
Accumulator to Memory	1010001 w	addr-low	addr-high	
Register/Memory to Segment Register	10001110	mod 0 reg r/m		
Segment Register to Register/Memory	10001100	mod 0 reg r/m		
PUSH = Push:				
Register/Memory	11111111	mod 110 r/m		
Register	01010 reg			
Segment Register	000 reg 110			
POP = Pop:				
Register/Memory	10001111	mod 000 r/m		
Register	01011 reg			
Segment Register	000 reg 111			
XCHG = Exchange				
Register/Memory with Register	1000011 w	mod reg r/m		
Register with Accumulator	10010 reg			
IN = Input from:				
Fixed Port	1110010 w	port		
Variable Port	1110110 w			
OUT = Output to				
Fixed Port	1110011 w	port		
Variable Port	1110111 w			
XLAT = Translate Byte to AL	11010111			
LEA = Load EA to Register	10001101	mod reg r/m		
LDS = Load Pointer to DS	11000101	mod reg r/m		
LES = Load Pointer to ES	11000100	mod reg r/m		
LAHF = Load AH with Flags	10011111			
SAHF = Store AH into Flags	10011110			
PUSHF = Push Flags	10011100			
POPF = Pop Flags	10011101			
ARITHMETIC	**76543210**	**76543210**	**76543210**	**76543210**
ADD = Add:				
Reg/Memory with Register to Either	000000 dw	mod reg r/m		
Immediate to Register/Memory	100000 sw	mod 000 r/m	data	data if s w = 01
Immediate to Accumulator	0000010 w	data	data if w = 1	
ADC = Add with Carry:				
Reg/Memory with Register to Either	000100 dw	mod reg r/m		
Immediate to Register/Memory	100000 sw	mod 010 r/m	data	data if s w = 01
Immediate to Accumulator	0001010 w	data	data if w = 1	
INC = Increment:				

(*Contd*)

Mnemonics & Description	*Instruction Code*			
Register/Memory	1111111 w	mod 000 r/m		
Register	01000 reg			
AAA = ASCII Adjust for Addition	00110111			
DAA = Decimal Adjust for Addition	00100111			
SUB = Subtract				
Reg/Memory and Register to Either	001010 dw	mod reg r/m		
Immediate from Register/Memory	100000 sw	mod 101 r/m	data	data if s w = 01
Immediate from Accumulator	0010110 w	data	data if w = 1	
SBB = Subtract with Borrow				
Reg/Memory and Register to Either	000110 dw	mod reg r/m		
Immediate from Register/Memory	100000 sw	mod 011 r/m	data	data if s w = 01
Immediate from accumulator	0001110 w	data	data if w = 1	
DEC = Decrement:				
Register/Memory	1111111 w	mod 001 r/m		
Register	01001 reg			
NEG = Change sign	1111011 w	mod 011 r/m		
CMP = Compare:				
Register/Memory and Register	001110 dw	mod reg r/m		
Immediate with Register/Memory	100000 sw	mod 111 r/m	data	data if s w = 01
Immediate with Accumulator	0011110 w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	00111111			
DAS = Decimal Adjust for Subtract	00101111			
MUL = Multiply (Unsigned)	1111011 w	mod 100 r/m		
IMUL = Integer Multiply (Signed)	1111011 w	mod 101 r/m		
AAM = ASCII Adjust Multiply	11010100	00001010		
DIV = Divide (Unsigned)	1111011 w	mod 110 r/m		
IDIV = Integer Divide (Signed)	1111011 w	mod 111 r/m		
AAD = ASCII Adjust for Divide	11010101	00001010		
CBW = Convert Byte to Word	10011000			
CWD = Convert Word to Double Word	10011001			
LOGICAL	76543210	76543210	76543210	76543210
NOT = Invert	1111011 w	mod 010 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	110100 v w	mod 100 r/m		
SHR = Shift Logical Right	110100 v w	mod 101 r/m		
SAR = Shift Arithemtic Right	110100 v w	mod 111 r/m		
ROL = Rotate Left	110100 v w	mod 000 r/m		
ROR = Rotate Right	110100 v w	mod 001 r/m		
RCL = Rotate Through Carry Flag Left	110100 v w	mod 010 r/m		
RCR = Rotate Through Carry Right	110100 v w	mod 011 r/m		
AND = And:				
Reg/Memory and Register to Either	001000 dw	mod reg r/m		
Immediate to Register/Memory	1000000 w	mod 100 r/m	data	data if w = 1
Immediate to Accumulator	0010010 w	data	data if w = 1	
TEST = And Function to Flags, No Result:				
Register/Memory and Register	1000010 w	mod reg r/m		

(*Contd*)

Mnemonics & Description	*Instruction Code*			
Immediate Data and Register/Memory	1111011 w	mod 000 r/m	data	data if w = 1
Immediate Data and Accumulator	1010100 w	data	data if w = 1	
OR = Or:				
Reg/Memory and Register to Either	000010 dw	mod reg r/m		
Immediate to Register/Memory	1000000 w	mod 001 r/m	data	data if w = 1
Immediate to Accumulator	0000110 w	data	data if w = 1	
XOR = Exclusive or:				
Reg/Memory and Register to Either	001100 dw	mod reg r/m		
Immediate to Register/Memory	1000000 w	mod 110 r/m	data	data if w = 1
Immediate to Accumulator	0011010 w	data	data if w = 1	
STRING MANIPULATIONS				
REP = Repeat	1111001 z			
MOVS = Move Byte/Word	1010010 w			
CMPS = Compare Byte/Word	1010011 w			
SCAS = Scan Byte/Word	1010111 w			
LODS = Load byte/Wd to AL/AX	1010110 w			
STOS = Stor Byte/Wd from AL/A	1010101 w			
CONTROL TRANSFER				
CALL = Call:				
Direct Within Segment	11101000	disp-low	disp-high	
Indirect Within Segment	11111111	mod 010 r/m		
Direct Intersegment	10011010	offset-low	offset-high	
		seg-low	seg-high	
	76543210	76543210	76543210	
Indirect Intersegment	11111111	mod 011 r/m		
JMP = Unconditional Jump:				
Direct Within Segment	11101001	disp-low	disp-high	
Direct Within Segment-short	11101011	disp		
Indirect Within Segment	11111111	mod 100 r/m		
Direct Intersegment	11101010	offset-low	offset-high	
		seg-low	seg-high	
Indirect Intersegment	11111111	mod 101 r/m		
RET = Return from CALL:				
Within Segment	11000011			
Within Seg Adding Immediate to SP	11000010	data-low	data-high	
Intersegment	11001011			
Intersegment Adding Immediate to SP	11001010	data-low	data-high	
JE/JZ = Jump on Equal/Zero	01110100	disp		
JL/JNGE = Jump on Less/Not Greater of Equal	011111100	disp		
JLE/JNG = Jump on Less or Equal/Not Greater	01111110	disp		
JB/JNAE = Jump on Below/Not Above or Equal	01110010	disp		
JBE/JNA = Jump on Below or Equal/Not Above	01110110	disp		

(Contd)

Mnemonics & Description		*Instruction Code*
JP/JPE = Jump on Parity/Parity Even	01111010	disp
JO = Jump on Overflow	01110000	disp
JS = Jump on sign	01111000	disp
JNE/JNZ = Jump on Not Equal/Not Zero	01110101	disp
JNL/JGE = Jump on Not Less/Greater or Equal	01111101	disp
JNLE/JG = Jump on Not Less or Equal/Greater	01111111	disp
JNB/JAE = Jump on Not Below/Above or Equal	01110011	disp
JNBE/JA = Jump on Not Below or Equal/Above	01110111	disp
JNP/JPO = Jump on Not Par/Par Odd	01111011	disp
JNO = Jump on Not Overflow	01110001	disp
JNS = Jump on Not Sign	01111001	disp
LOOP = Loop CX Times	11100010	disp
LOOPZ/LOOPE = Loop While Zero/ Equal	11100001	disp
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	11100000	disp
JCXZ = Jump on CX Zero	11100011	disp
INT = Interrupt		
Type Specified	11001101	type
Type 3	11001100	
INTO = Interrupt on Overflow	11001110	
IRET = Interrupt Return	11001111	
	76543210	76543210
PROCESSOR CONTROL		
CLC = Clear Carry	11111000	
CMC = Complement Carry	11110101	
STC = Set Carry	11111001	
CLD = Clear Direction	11111100	
STD = Set Direction	11111101	
CLI = Clear Interrupt	11111010	
STI = Set Interrupt	11111011	
HLT = Halt	11110100	
WAIT = Wait	10011011	
ESC = Escape (to External Device)	11011xxx	mod xxx r/m
LOCK = Bus Lock Prefix	11110000	

Suggested Readings

1. 16-bit microprocessors: architecture, software, and interface techniques, Walter A Triebel, Pearson Technology Group
2. 8080a 8085 Assembly Language Programming, Lance Leventhal / Osborne & Associates
3. The 8086/8088 family of microprocessors: software, hardware, and system applications, Wunnava Subbarao / Delmar
4. Crash Course in Microcomputers, Louis E. Frenzel, Newnes
5. Embedded Controllers: 80186, 80188, and 80386ex, Barry B Brey, Prentice Hall
6. Introduction to Microprocessors, John Crisp, Butterworth-Heinemann
7. The 8088 and 8086 microprocessors: programming, interfacing, software, hardware, and applications, Walter Triebel, Pearson TechnologyGroup
8. 8088 assembler language programming :the IBM PC, David Willen, Pearson Indiana