

contents

<i>Preface</i>	xxi
<i>Acknowledgements</i>	xxv
<i>About the Author</i>	xxvii

Part I The x86 Microprocessors **0-1**

0 Basics of Computer Systems 0-3
0.1 A Brief History of Microprocessors 0-3
0.2 Basics of Computer Architecture 0-7
0.3 Computer Languages 0-13
0.4 RISC and CISC Architectures 0-14
0.5 Number Systems 0-15
0.6 Number Format Conversions 0-18
0.7 Computer Arithmetic 0-25
0.8 Units of Memory Capacity 0-34
0.9 The 8085 Microprocessor 0-35
<i>Key Points of This Chapter</i> 0-51
<i>Questions</i> 0-52
<i>Exercises</i> 0-53

1 The Architecture of 8086 1-1
1.1 Internal Block Diagram of the 8086 1-1
1.2 Execution Unit 1-2
1.3 Bus Interface Unit 1-8
1.4 Addressing Modes 1-14
<i>Key Points of This Chapter</i> 1-19
<i>Questions</i> 1-20
<i>Exercises</i> 1-21

2	Programming Concepts–I	2-1
2.1	The Assembly Process	2-1
2.2	Assemblers for x86	2-3
2.3	Memory Models	2-5
2.4	Instruction Design	2-17
	<i>Key Points of This Chapter</i>	2-24
	<i>Questions</i>	2-24
	<i>Exercises</i>	2-25
3	Programming Concepts–II	3-1
3.1	Approaches to Programming	3-1
3.2	Data Transfer Instructions	3-4
3.3	Branch Instructions	3-14
3.4	Arithmetic Instructions	3-21
3.5	Logical Instructions	3-38
3.6	Shift and Rotate Instructions	3-39
	<i>Key Points of This Chapter</i>	3-45
	<i>Questions</i>	3-46
	<i>Exercises</i>	3-47
4	Programming Concepts–III	4-1
4.1	String Instructions	4-1
4.2	Procedures	4-7
4.3	Macros	4-18
4.4	Number Format Conversions	4-22
4.5	ASCII Operations	4-25
4.6	Conversions for Computations and Display/Entry	4-28
4.7	Signed Number Arithmetic	4-30
4.8	Programming Using High Level Language Constructs	4-36
	<i>Key Points of This Chapter</i>	4-40
	<i>Questions</i>	4-40
	<i>Exercises</i>	4-41
5	Programming Concepts–IV	5-1
5.1	Input / Output Programming	5-1
5.2	I/O Instructions	5-3

5.3 Modular Programming	5-5
5.4 Programming in C with Assembly Modules	5-12
<i>Key Points of This Chapter</i>	5-15
<i>Questions</i>	5-16
<i>Exercises</i>	5-16

6 The Hardware Structure of 8086 6-1

6.1 Pin Configuration	6-1
6.2 Clock	6-14
6.3 Other Processor Activities	6-18
6.4 Maximum Mode	6-21
6.5 Instruction Cycle	6-25
<i>Key Points of This Chapter</i>	6-30
<i>Questions</i>	6-30
<i>Exercises</i>	6-31

7 Memory and I/O Decoding 7-1

7.1 Memory Device Pins	7-1
7.2 Memory Address Decoding	7-3
7.3 Memory Banks	7-12
7.4 I/O Address Decoding	7-16
<i>Key Points of This Chapter</i>	7-23
<i>Questions</i>	7-23
<i>Exercises</i>	7-23

8 The Interrupt Structure of 8086 8-1

8.1 Interrupts of 8086	8-2
8.2 Dedicated Interrupt Types	8-5
8.3 Software Interrupts	8-8
8.4 Hardware Interrupts	8-8
8.5 Priority of Interrupts	8-11
8.6 Interrupt Type Allocation for Current PCs	8-11
8.7 BIOS 10H Functions	8-17
8.8 Addressing Video Memory Directly	8-22
8.9 Keyboard Interfacing	8-25

8.10 Hooking an Interrupt	8-30
<i>Key Points of This Chapter</i>	8-40
<i>Questions</i>	8-40
<i>Exercises</i>	8-41

9 Peripheral Interfacing—I

9.1 Trainer Kit	9-2
9.2 Programmable Peripheral Interface (PPI)-8255A	9-3
9.3 Modes of Operation	9-8
9.4 Mode 0	9-8
9.5 Mode 1	9-14
9.6 Mode 2 (Strobed Bidirectional Bus I/O)	9-22
9.7 Centronics Printer Interface	9-22
9.8 Interfacing an Analog to Digital Converter to the 8086	9-25
9.9 Interfacing to a Digital to Analog Converter	9-29
9.10 Interfacing Liquid Crystal Displays to the 8086	9-34
9.11 Interfacing a Stepper Motor to the 8086	9-39
9.12 Hex Keyboard Interfacing	9-49
9.13 Interfacing LED Displays	9-53
<i>Key Points of This Chapter</i>	9-59
<i>Questions</i>	9-60
<i>Exercises</i>	9-60

9-1

10 Peripheral Interfacing—II

10.1 The Programmable Interval Timer 8253/8254	10-1
10.2 The Programmable Keyboard Display Interface – 8279	10-15
10.3 The Programmable Interrupt Controller (PIC) 8259	10-30
10.4 Cascade Mode	10-43
<i>Key Points of This Chapter</i>	10-46
<i>Questions</i>	10-46
<i>Exercises</i>	10-47

10-1

11 Peripheral Interfacing—III

11.1 Serial Communication Principles	11-1
11.2 Simplex, Half Duplex and Full Duplex Communication	11-2

11-1

11.3	The Programmable Serial Communication Interface	11-9
11.4	Internal Reset on Power Up	11-22
11.5	Direct Memory Access	11-23
11.6	The DMA Controller – 8237	11-28
11.7	DMA and IBM-PC	11-35
11.8	PCI Based Computers	11-36
	<i>Key Points of This Chapter</i>	11-36
	<i>Questions</i>	11-37
	<i>Exercises</i>	11-38

12 Semiconductor Memory Devices

12-1

12.1	Semiconductor Memory	12-2
12.2	Dynamic RAM	12-5
12.3	Synchronous DRAM (SDRAM)	12-10
12.4	ROM (Read Only Memory)	12-14
12.5	Cache Memory	12-15
12.6	Mapping Techniques	12-17
12.7	Cache and the x86 Family	12-22
	<i>Key Points of This Chapter</i>	12-23
	<i>Questions</i>	12-24
	<i>Exercises</i>	12-24

13 Multiprocessor Configurations

13-1

13.1	Multiprocessor Systems	13-2
13.2	Multiprocessing Using 8086	13-3
13.3	The 8086 and 8089 in a Tightly Coupled Configuration	13-8
13.4	Loosely Coupled Configurations and Bus Arbitration	13-10
13.5	Bus Arbitration Using the 8289 Bus Arbiter IC	13-13
13.6	The Arithmetic Coprocessor 8087	13-18
	<i>Key Points of This Chapter</i>	13-31
	<i>Questions</i>	13-31
	<i>Exercises</i>	13-32

14 Advanced Processors—80286 to Pentium	14-1
14.1 The 80286 Processor	14-1
14.2 The 80386	14-3
14.3 Internal Architecture	14-3
14.4 Programming Enhancements	14-5
14.5 Hardware Features of 80386	14-9
14.6 Virtual Memory	14-13
14.7 Memory Management Unit	14-14
14.8 Converting a Logical Address to a Physical Address	14-21
14.9 Calculating the Size of the Logical Address Space	14-23
14.10 Protection	14-28
14.11 Multi Tasking	14-34
14.12 Interrupts of 80386	14-38
14.13 Privileged Instructions	14-40
14.14 Enhanced Features of 80486	14-42
14.15 Data Alignment	14-45
14.16 The Pentium Processor	14-46
<i>Key Points of This Chapter</i>	14-50
<i>Questions</i>	14-51
<i>Exercises</i>	14-52
15 Microarchitectural Techniques of Advanced Processors	15-1
15.1 Enhancing Performance – Why Pipelining?	15-1
15.2 Replication – The Superscalar Concept	15-19
15.3 Pentium Pro (P6) Architecture	15-37
15.4 VLIW Architecture	15-39
<i>Key Points of This Chapter</i>	15-40
<i>Questions</i>	15-41
<i>Exercises</i>	15-42
16 Multicore Processors	16-1
16.1 Multicore Processing – The Concepts	16-1
16.2 Major Issues in Multicore Processing	16-7

16.3 The x86-based Chip Multiprocessors	16-11
16.4 Conclusion	16-18
<i>Key Points of This Chapter</i>	16-18
<i>Questions</i>	16-18
<i>Exercises</i>	16-19
17 Beyond Pentium—More Advanced Processors	17-1
17.1 Processors Based on the P6 Microarchitecture	17-1
17.2 Features of Pentium Pro	17-2
17.3 Pentium-II and Pentium-III	17-3
17.4 Streaming SIMD Extensions (SSE)	17-5
17.5 Pentium-IV	17-6
17.6 The Continued Dominance of x86	17-9
17.7 ‘Core’ Microarchitecture	17-9
17.8 Multicore Processors	17-9
17.9 Nehalem Microarchitecture	17-10
17.10 Sandy Bridge and IvyBridge	17-11
17.11 Fourth-Generation Core Processor Family	17-11
17.12 Important Technological Features in IA Processors	17-11
17.13 Nanometer Technology	17-12
17.14 Difference Between Core i3, i5 and i7 Processors	17-13
<i>Key Points of This Chapter</i>	17-14
<i>Questions</i>	17-14
<i>Exercises</i>	17-14
18 Atom SoC—Intel’s High-end Embedded Processor	18-1
18.1 History of Intel’s Embedded Processors	18-1
18.2 The Bonnell Microarchitecture	18-3
18.3 The Atom SoC	18-5
18.4 Power Reduction Techniques in Processors	18-14
18.5 Advanced Power and Configuration Interface	18-16
18.6 The Silvermont Micro Architecture	18-19
<i>Key Points of This Chapter</i>	18-20
<i>Questions</i>	18-20
<i>Exercises</i>	18-21

Part II	The 8051 Microcontroller	19-1
19	The 8051 Microcontroller: The Programmer's Perspective	19-3
19.1	History and Family Details of 8051	19-3
19.2	8051: The Programmer's Perspective	19-5
19.3	Assembly Language Programming	19-8
19.4	Internal RAM	19-14
19.5	The 8051 Stack	19-16
19.6	Processor Status Word (PSW)	19-18
19.7	Assembler Directives	19-19
19.8	Storing Data in Code Memory (ROM)	19-20
19.9	The Instruction Set of 8051	19-22
19.10	Port Programming	19-37
19.11	Subroutines (Procedures)	19-43
19.12	Delay Loops	19-45
	<i>Key Points of This Chapter</i>	19-50
	<i>Questions</i>	19-50
	<i>Exercises</i>	19-51
20	Programming the Peripherals of 8051	20-1
20.1	Pin Configuration of 8051	20-1
20.2	Programming the Internal Peripherals	20-7
20.3	Timers of 8051	20-8
20.4	Counter Programming	20-19
20.5	Interrupts of 8051	20-21
20.6	Serial Communication	20-31

0

Basics of Computer Systems

IN THIS CHAPTER, YOU WILL LEARN

- The brief history of Intel's microprocessors and their use in personal computers.
- The general principles of computer architecture.
- The operation of the data, address and control buses of a computer.
- The distinction between RISC and CISC computing.
- The comparison between assembly and high level language programming.
- The binary, hexadecimal and BCD number systems.
- Number format conversions.
- Computer arithmetic using different number systems.
- The processing of negative numbers as done by a computer.
- The programming model and programming of the 8085 microprocessor.
- The pin out and hardware features of 8085.

0.1 | A Brief History of Microprocessors

What is a microprocessor? We usually answer that it is a processor on a single microchip, where the word 'micro' stands for small. A processor is a device which has computing, i.e., data processing capability. Over the years, the computing power of microprocessor chips has increased steeply, along with the speed of processing, and this trend is continuing as newer applications are envisaged.

The idea of a single chip computer was around even as early as the 1950s, but at that time, electronics and integrated circuit technology were still in their infancy. By the end of the 1960s, things had changed and integrated circuits had already made inroads into the technology market. Around that time, there were many companies in the field trying to develop a single chip computer and one of them was Intel. Intel released its 4-bit all-purpose chip, the Intel 4004, in November 1971. It had a clock speed of 108 KHz and 2,300 transistors with ports for ROM, RAM and I/O. This was just the beginning of newer innovations. In April 1972, Intel came up with the 8008, which was just an 8-bit version of the 4004. Since it had nothing very spectacular to offer, this microprocessor did not do much for Intel or the microprocessor field. Meanwhile, other companies started making their presence felt by their own versions of microchips with computing capabilities.

The first major break came with the introduction of the 8080 (again, by Intel) in 1974. Intel put itself back on the map with the 8080, which used the same instruction set as the earlier 8008, but the 8080 is generally considered to be the first truly usable microprocessor. The 8080 had a 16-bit address bus and an 8-bit data bus, a 16-bit stack pointer to memory which replaced the 8-level internal stack of the 8008, and a 16-bit program

counter. It also contained 256 I/O ports, so that I/O devices could be connected without taking away or interfering with the addressing space. It also possessed a signal pin that allowed the stack to occupy a separate bank of memory. These features are what made this a truly modern microprocessor.

Other Players in the Market

All this does not mean that there were no other companies in the microprocessor field – there were, and each of them had contributions to make. We are talking more about Intel's role, because it will take us to the advent and development of Intel's x86 microprocessor family, the class of microprocessors about which we discuss in this book.

Motorola (now Freescale) is another company that came up with its own original design of microprocessors. It started with the 6800 chip, which was followed by more advanced versions of this basic chip. In terms of computational capability, Motorola's processors are as good (or in some ways, better) as any Intel processor. The market segments of both these companies are separate and distinct now.

In contrast, AMD is a company that did not go for original design but worked on improving the 8080 design under license agreements. There was also this very popular microprocessor, Z-80, developed by Zilog Corporation. This 8-bit microprocessor was binary compatible with the 8080 and, surprisingly, is still in widespread use today in many embedded applications. It was supposed to be superior to 8080. Though Zilog made improved versions of this microprocessor, the company did not take off as well as it should have because it could not handle the competition of bigger players in the market.

In 1976, Intel updated the 8080 design with the 8085, and it became very popular as a well-designed and simple microprocessor, though it was never used in a PC. However, it is still very popular and used in various applications just as Zilog's Z-80 processor is.

In 1978, Intel introduced the 16-bit 8086, a 16-bit processor which gave rise to the x86 architecture. In 1979, Intel also released the 8088, a microprocessor similar in every way to 8086, except that its external data bus was 8 bits wide, while being a 16-bit microprocessor internally.

0.1.1 / The x86 Family

The real breakthrough for Intel came when in 1981 when IBM picked Intel's 8088 for its personal computer. With this, the market segment of Intel grew by leaps and bounds. Intel kept on updating and improving the x86 architecture with newer and newer innovations, and since x86 had already established its place in the PC world, PCs also became more and more sophisticated. Table 0.1 gives the timeline of the release of the various x86 processors. The points to remember are that 8088 was the first x86 processor used in a PC and it was a 16-bit architecture with an external data bus of 8 bits. Note that 8086 was manufactured earlier with the same internal architecture as the 8088, but it had an external data bus of 16 bits, which made it difficult to connect it to 8-bit peripherals and was costlier too. That is why it was not chosen for the PC.

Table 0.1 | Year of Release of Various x86 Processors

x86 processor	Year of release
8086	1978
8088	1979
80186	1982
80286	1982
80386	1985
80486	1989
Pentium	1993
Pentium Pro	1995
Pentium-2	1997
Pentium-3	1999
Pentium-M	2002
Pentium-4	2004
Pentium-D	2005
Core-2	2006
Core-2 Quad	2007

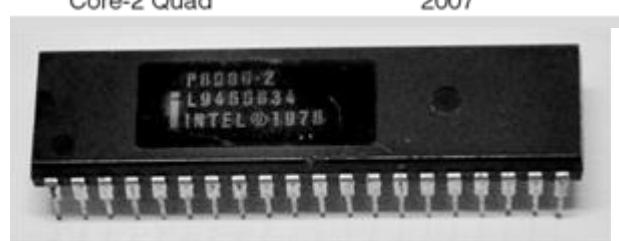


Figure 0.1 | The first x86 processor – the 8086

The 80186 was never used in PCs but found applications in the field of embedded computing. The 80286 was the next x86 processor used in PCs. It was also a 16-bit architecture but also had an external data bus of 16 bits. It marked a step upwards in PC architecture as it brought into PCs the concept of virtual memory and protected mode operations. All later PCs have these features. The next major step came with the 80386, which was a 32-bit processor internally and externally. Current x86 processors have 64-

bit capability and are multi-core processors. The development and changes that occurred till the Pentium processor are covered in Chapter 14, and the innovations that happened after that are covered in Chapters 15 to 17.

0.1.2 / The x86 Philosophy

What is meant by the term x86? It refers to a set of machine language instructions, the philosophy and usage of which was introduced in the 8086 processor. It was designed by Intel, but other companies like AMD also used the same vocabulary, though their internal design could be different. The same was followed by Intel for processors 80186, '286, '386, '486 and the different versions of Pentium. Over the years, the x86 instruction set has grown and expanded, but backward compatibility with earlier members has been maintained. What this means is that a program written for 8086 will run on a Pentium machine (but not the other way round). The current status of the x86 processors is that it is the de facto standard for PCs from desktops to servers and laptops to supercomputers. There are other competitors in the market (some offering superior processors), but their market share is relatively small. However, now the market dynamics for processors has changed drastically with the arrival and popularity of mobile computing devices like smart phones and electronic tablets. x86 processors have made it to this market as well using the Atom processor, though its market share is not very high. The details of this trend are discussed in Chapters 18 and 19.

Now that we have had a peep into the history of the x86 processor family, let us also have a glimpse of the history of PCs, because that is where x86 processors have the maximum number of applications.

0.1.3 / Personal Computers

We have seen that IBM manufactured its first personal computer in August 1981, using the 8088 (see Fig. 0.2). The IBM Personal Computer was introduced as the IBM 5150. During its development, the IBM 5150 had been internally referred to as 'Project Chess' and was created by a team of 12 people headed by Don Estridge and Larry Potter. The first IBM PC had an Intel 8088 processor, 64 KB of RAM (expandable to 256 KB), a floppy disk drive (which could be used to boot the computer with a rebranded version of MS-DOS (PC-DOS) and a CGA or monochrome video card. The machine also had a version of Microsoft BASIC in ROM.



Figure 0.2 | The first IBM PC 5150

In March 1983, the IBM PC-XT was marketed, with the change being that it had a hard disk as well. XT stands for ‘Extended Technology’. The next change came about in August 1984 when the 80286, a 16-bit processor, which also had virtual memory capability, was used by IBM in its PC. They called it the PC-AT with AT standing for ‘Advanced Technology’. Then, one after the other, 80386, 80486 and Pentium came to be used in PCs. The processor developed after the ’486 was to have been named 80586, and Intel wanted to trademark this number, but a court ruling prevented this. So, the word Pentium was coined and Intel did not have to look back after that. Now, in 2014, the Pentium processor is an outdated one. The different generations of the core i7, i5 and i3 processors rule the PC scene. Chapters 15 to 17 discuss these matters and the technologies involved in them.

Incidentally, it was not IBM which first used the term PC. The word ‘Personal Computer’ was in common usage quite some time before IBM developed its first PC, and many other companies also made PCs using various other processors. However, when the IBM PC became a great success, many manufacturers started making similar PCs and software companies also had the inclination to develop software for IBM PC compatible PCs. That is how the PC revolution came about; and we know now that it is not IBM that makes most of the PCs in the world now – various other manufacturers do, but the x86 processor still forms the core of all these PCs. With more and more people using PCs, standardization of software and hardware came about slowly. This has made the world of computing more ordered and comfortable, and compatibility issues for hardware and software are very much less than in the early days.

Note

In this book, whenever we use the word PC, we mean the IBM compatible PC. There is the other great PC, the Apple Mac, which while being low in numbers, is an elite class of computers. ‘Our goal is not to build the most computers. It’s to build the best’ – according to the CEO of Apple Inc. Apple has a worldwide market share of just 10% of the PC market but has more than 90% of the market share of premium computers.

What this Book is About

In this book, we will study the x86 architecture as developed by Intel. We will go through the ideas of programming in assembly language and also study the interfacing of the x86 processors with various devices.

Frequently, we will also see how a particular interface is used in the standard PC. Even though we started by defining a microprocessor as a single chip computer, we know that a PC cannot be made with just a single chip. When we use the phrase ‘single chip computer’, we only mean that all the computing and processing power is held within that single chip which we call the microprocessor – but to get this chip to communicate with the external world, and make it comfortable for us to use, requires a lot more of hardware and software support. We will make a trip to understand the concepts behind how the personal computer has become an integral part of our lives.

0.2 | Basics of Computer Architecture

0.2.1 / *The Block Diagram of a Computer*

A computer, as its name indicates, is a machine used for computing. Computing, which many years ago meant arithmetic calculations, has now given way to large amounts of ‘data processing’. As such, it is more reasonable to designate the computer now as a ‘data processing machine’. For performing its designated tasks, this machine requires many components, which can broadly be divided as hardware and software. Hardware is, obviously, the physical constituents of a computer. Software is the collection of programs which directs the hardware to perform its tasks.

Let us first look at a computer in terms of its hardware. Figure 0.3 shows the architectural description of a computer system. It shows the major parts of the computer and also indicates how these parts are connected together to form the computing machine. The major parts are the CPU, memory and input/output devices.

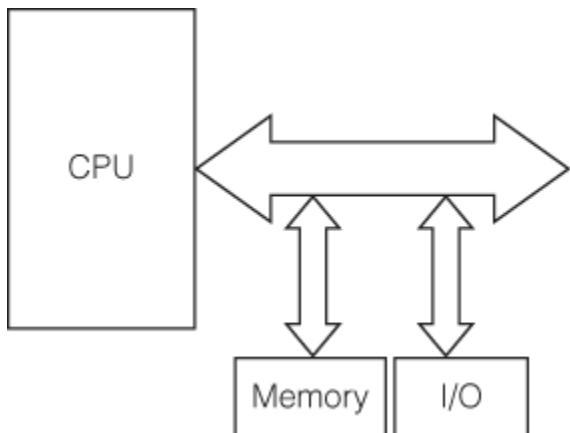


Figure 0.3 | The block diagram of a computer

The heart of a computer is the ‘central processing unit’. It is this unit which gives ‘life’ to a computer. The CPU usually is a ‘microprocessor’, which means that it is usually a separate and self contained chip. The CPU processes the data given to it according to the programs meant to operate on these data. The program consists of ‘instructions’. These instructions are decoded by the CPU, which generates control signals necessary to activate the arithmetic and logic units of the CPU. As such, the CPU contains the arithmetic logic unit and the control unit. All these activities are timed and synchronized by a pulse train of fixed frequency. This is the clock signal, and it also has the job of synchronizing the activity of the CPU with the activity on the bus.

0.2.2 / The System Bus

A bus is collection of signal wires which connect between the components of the computer systems. The figure shows that the CPU is connected to the memory as well as I/O through the system bus, but only one at a time; if the memory and I/O want to use the bus at the same time, there is a conflict, as there is only one system bus. The system bus comprises of the address bus, data bus and the control bus.

The Data Bus

The set of lines used to transfer data is called the data bus. It is a bidirectional bus, as data has to be sent from the CPU to memory and I/O and has to be received as well by the CPU. The width of the data bus determines the data transfer rate, size of the internal registers of the CPU and the processing capability of the CPU. In short, it is a reflection of the complexity of the processor. As we see, the 8086 has a data bus width of 16 bits, while the 80486 has a 32-bit bus width. Thus, the 80486 can process data of 32 bits at a time while the 8086 can only handle 16 bits.

The Address Bus

The address bus width determines the maximum size of the physical memory that the CPU can access. With an address bus width of 20 bits, the 8086 can address 2^{20} different locations. It can use a memory size of 2^{20} bytes or 1 MB. For Pentium with an address bus width of 32 bits, the corresponding numbers are 2^{32} bytes, i.e., 4 GB. When a particular memory location is to be accessed, the corresponding address is placed on the address bus by the CPU. I/O devices also have addresses. In both cases, it is the CPU which supplies the address, and as such, the address bus is unidirectional.

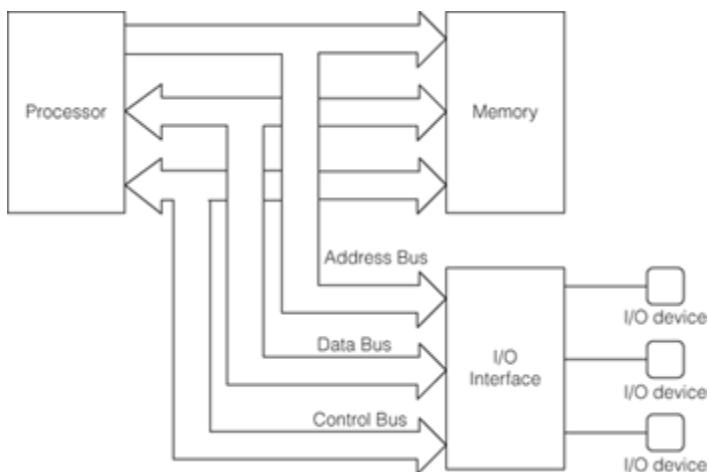


Figure 0.4 | The system bus and its components

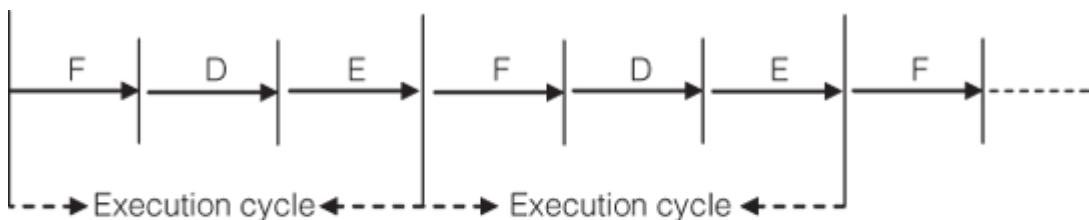
The Control Bus

The control bus is a set of control signals which needs to be activated for activities like writing/reading to/from memory/I/O or special activities of the CPU like interrupts and DMA. Thus, we see signals like Memory Read, I/O Read, Memory Write and Interrupt Acknowledge as part of the control bus. These control signals dictate the actions taking place on the system bus that involve communications with devices like memory or I/O. For example, the Memory Read signal will be asserted for reading from memory. It is sent to memory from the processor. A signal such as 'Interrupt' is received by the processor from an I/O device. Hence in the control bus, we have signals traveling in either direction. Some control lines may be bidirectional too.

Now that we have discussed a computer system in general, let us go a bit deeper into its individual constituents.

0.2.3 / The Processor

The processor, or the microprocessor as we might call it, is the component responsible for controlling all the activity in the system. It performs the following three actions continuously.



F-Fetch
D-Decode
E-Execute

Figure 0.5 | The execution cycle

1. Fetch an instruction from memory.
 2. Decode the instruction.
 3. Execute the instruction.

When we write a program, it is stored in memory. Our code has to be brought to the processor for the required action to be performed. The first step, obviously, is to 'fetch' it from memory. The next step, i.e., decoding, involves the interpretation of the code as to what action is to be performed. After decoding, the action required is performed. This is termed 'instruction execution'. The sequence of these three actions is called the 'execution cycle'. To do all this, the processor has 'control circuitry' to fetch and decode instructions. The ALU part of the processor performs the required arithmetic/logic operations. The sequence of fetch-decode-execute is done continuously and infinitely by the processor. An important implication of this cycle is that instruction execution is 'sequential' in nature – it is only after the first instruction is dealt with will the second one be taken up. However, there will be situations when the sequential nature of program execution is disturbed. This is when a 'branch' instruction appears in the sequence, and a new sequence of instructions will be taken up starting from a new location.

0.2.4 | System Clock

All the activities of the processor and buses are synchronized by a clock, which is, as shown in Fig. 0.6, a square wave with a particular frequency. The reciprocal of the clock frequency is the cycle time T , also called the clock period. $T = 1/f$ where f is the clock frequency. An execution cycle may require many clock periods. This depends on the architectural features of the processor as well as the complexity of the instruction to be executed.

Since an execution cycle also involves fetching instructions and data from memory, it also depends on how many clock cycles are needed to access memory. Obviously, the time for execution depends on the clock speed as well; i.e., a clock speed of 3 GHz implies faster processing than a clock of 1 GHz. However, the technology used for the processor must be able to support the clock frequency used.

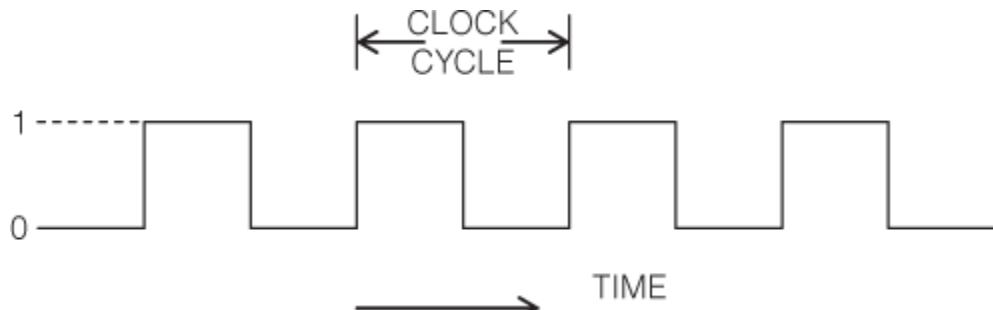


Figure 0.6 | System clock

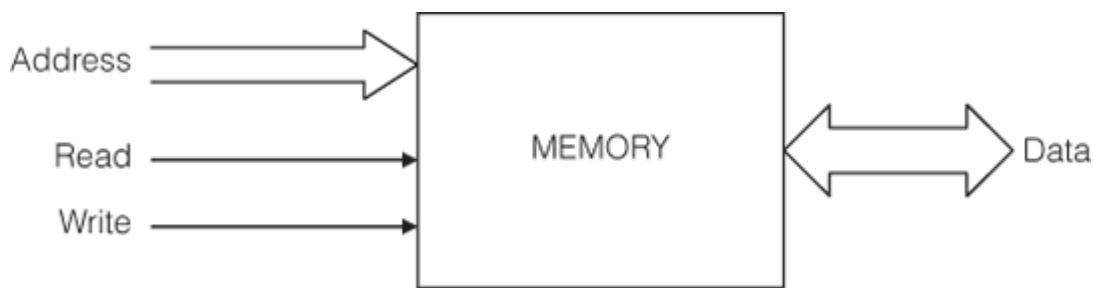


Figure 0.7 | Memory and associated control signals

0.2.5 / Memory

The memory associated with a computer system includes the primary memory as well as secondary memory. However, for the time being, we will think of memory as constituting the primary or main memory only, which is usually RAM (Random Access Memory). Memory is organized as bytes, and the capacity of a memory chip is stated in terms of the number of bytes it can store. Thus, we can have chips of size 256 bytes, 1KB, 1MB and so on. If a computer has a total memory space of 20 MB, it can use RAM chips of the available capacity to get that much of memory.

There are two basic operations associated with memory – read and write. Reading causes a data stored in a memory location to be transferred to the CPU without erasing the content in memory. Writing causes a new data to be placed in a memory location (it overwrites the previous value). There is a certain amount of time required for these operations and this is termed as ‘access time’.

Memory Read Cycle

The steps involved in a typical read cycle are:

1. Place on the address bus the address of the memory location whose content is to be read. This action is performed by the processor.
2. Assert the **memory read** signal which is part of the control bus.
3. Wait until the content of the addressed location appears on the data bus.
4. Transfer the data on the data bus to the processor.
5. De-activate the memory read signal. The memory read operation is over and the address on the address bus is not relevant anymore.

Memory Write Cycle

As a continuation, let us also examine the steps in a typical write cycle.

- Place on the address bus the address of the location to which data is to be written.
- On the data bus, place the data to be written.
- Assert the **memory write** signal which is part of the control bus.
- Wait until the data is stored in the addressed location.
- De-activate the memory write signal. This ends the memory write operation.

At this stage, we should remember that these operations are synchronized with the system clock. An 8086 processor takes at least four clock cycles for reading/writing. These four cycles constitute the ‘memory read’ and ‘memory write’ cycles for the processor. Other processors may require more/less clock cycles for the same operations.

0.2.6 / The I/O System

For a computer to communicate with the outside world, there is the need for what are called peripherals. Some of these peripherals are purely input devices like the keyboard and mouse; some are purely output devices like the printer and video monitor and some like the modem transfer data in both directions. All this just means that such I/O devices are needed for us to use a computer. However, it is difficult for a processor to deal directly with I/O devices, because of their incompatibility with the processor – each peripheral is different and the operating conditions, voltages, speeds and standards are not understandable to the processor. The processor does not have the necessary control signals to deal with different peripherals. Hence, the normal practice is for each peripheral to have a controller which acts as an interface between the peripheral and the processor. This controller, which may be a special purpose chip, understands the characteristics of the particular device and provides the necessary control signals to the processor to communicate with the peripheral. Thus, we have specialized

controllers for most peripherals – like the keyboard display interfacing chip, parallel port interfacing chip and serial communication chip. All these chips are programmable – they have registers for commands, data and status. By suitably programming these chips, we can get the processor to communicate correctly with any peripheral. Figure 0.8 shows the use of an I/O interface between an I/O device and a processor. The processor is not shown in the figure, but the system bus which comes from the processor is shown.

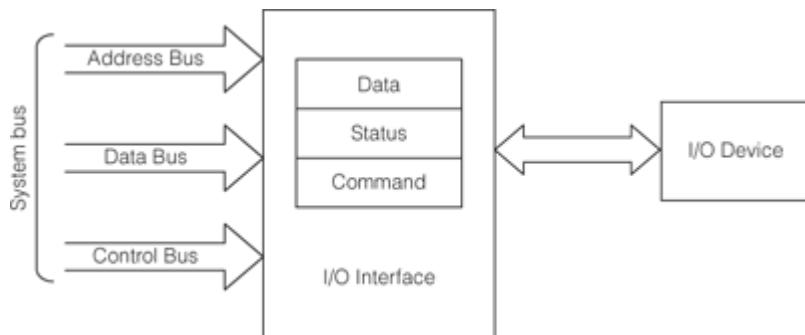


Figure 0.8 | The I/O system

In the final analysis, we can think of a computer that we usually use as a conglomeration of components which include memory and I/O devices of various types, applications and specifications.

0.3 | Computer Languages

0.3.1 | Machine Language, Assembly Language and High Level Language

The computer is just a dumb piece of equipment unless we are able to make it work for us. For that, we must be able to ‘program’ it, so that it will perform the tasks we assign it. Programming a computer entails the use of a language that the computer understands. The language native to computers is ‘machine language’, which consists of binary ones and zeros. The computer knows this language, and the series of ones and zeros fed to it are ‘operation codes’ for it, which tells it what action is to be performed. Thus, there is one binary code for addition and another one for subtraction. These operation codes are called ‘opcodes’ and this language is called ‘machine language’. Programming in machine language means writing the opcodes of the tasks we want to get done by the computer.

However, the problem with machine language, as is obvious, is that it is cumbersome and error-prone. Human beings are not good at remembering or using binary codes. Programming using machine language is not

something that any one of us is likely to enjoy. To make it easier for us to communicate with computers, there is a language at a slightly higher level, and that is called ‘assembly language’. This is more intelligible to users than machine code. This language uses ‘mnemonics’ for specifying the operation the computer is to perform. These mnemonics are a direct translation of the machine code to a symbol. For example, the binary code for addition is replaced by the symbol ‘ADD’ and the binary code for multiplication is given the symbolic name ‘MUL’. The exact mnemonic used depends on the processor type, but it will be related to the operation to be done.

How does this help? A user does not have to remember binary codes or enter binary code for programming. He only needs to remember the symbolic codes and the associated syntax. We say that assembly coding is at a higher level than machine coding. However, does the computer understand the mnemonics? No, which means that there should be an interface between assembly language and machine language. This interface converts the symbolic codes fed in by the user into machine codes. The software which does this is called an ‘**assembler**’.

Since machine language is native to a processor, each processor will have its own machine language and thus it has its own assembly language also. Translating from assembly language to machine language and vice versa is a one to one process – one opcode translates to a unique machine code for a particular processor.

However, we human beings always look for easier ways to get things done. So, there are ‘higher level languages’ which have the vocabulary and grammar similar to the language spoken by us. Such languages are very easy to use because the communication process is similar to English. We have heard of languages like C, FORTRAN, COBOL and, many such ‘high level languages’. The features of such languages are as follows”

1. They are easy to understand and write.
2. They are not processor specific.

Thus, if we write a program in C, we can use it to run on any processor – as long as the ‘compiler’ for the language is available. The compiler is the software which ‘translates the high level language statements’ to statements in a lower level language. The lower level may be assembly or machine language. However, finally the processor needs the machine code.

The program that we write in assembly or high level language is called the source program or source code. A compiler or assembler converts this

into an object code, which is ‘executable’ in the sense that the processor understands the code and performs the tasks indicated.

0.3.2 / Comparison

Programming in machine language is too cumbersome and hence ruled out in the present world. However, assembly language programming is frequently done, so let us now make a comparison between assembly language programming and high level language programming.

Assembly code is specific to a processor – which means that the assembly code of 8086 does not make any sense to 8085 (though both are Intel made). Assembly programs need the programmer to know the architecture of the processor intimately. He should know the registers and flags and the way each instruction handles data. So, doing assembly coding involves the study of the concerned processor. However, once this part is done, coding is very efficient, compact and executes very fast. Speed advantage of a hundred times or more is fairly common. Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

High level language programming, on the other hand, is not processor-specific. It is easy to learn and master. However, high level languages are abstract. Typically, a single high level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. The object code generated by a compiler is usually not compact. However, the advantage of high level languages is that since it is easy to learn, semi-skilled designers can be employed for development activities, and so development and maintenance times are much less.

This book focuses on the x86 architecture and on assembly language programming. The aim is to impart good assembly language skills and a thorough knowledge of the x86 architecture.

0.4 | RISC and CISC Architectures

Two terms that are likely to be encountered frequently while reading about computer architecture are RISC and CISC. RISC stands for Reduced Instruction Set Computer and CISC means Complex Instruction Set Computer. Since a lot of controversy surrounds these two terms, let us try to find out what it is all about.

In the early days of microprocessor development, the trend was to have complex instructions implemented fully using hardware. For example, the multiply instruction is a complex instruction which needs a dedicated hardware multiplier. Because hardware is fast, execution is fast, but with lots of such complex instructions, the hardware budget is naturally high. This is the philosophy and the main feature of CISC.

RISC, on the other hand, views this matter in a different way. On an average, the number of complex instructions a computer uses is relatively less. So, why not realize a complex instruction using a set of simple instructions? This is possible, and the advantage is that the hardware budget is much less. The instruction set is also small. However, software is to be written to realize complex instructions with simple instructions. This amounts to trading software for hardware.

There exists a long history of controversy regarding which is better. The x86 architecture was based on the CISC philosophy, right from the beginning. By the time RISC principles became popular and software development for RISC became established, the x86 CISC processors had already carved a niche for themselves in the processor market. So, even though the supporters of RISC were able to establish their point, most developers did not want to take the risk of switching over to an untested domain. However, most of the newer processors used the RISC philosophy for their architectures – examples are ARM, Power PC, Sun's Sparc processors and the like. Many of them found their applications in the embedded processing field.

The main features of RISC are that they have only simple instructions implemented in a single clock. However, there is an irony in that many RISC processors have as many complex instructions as CISC processors. Probably this can be justified by explaining that such complex instructions have been implemented using microprogramming rather than a direct hardware realization. Microprogramming is a method of implementing the control unit of a computer by breaking down instructions into a sequence of small programming steps.

While the RISC versus CISC controversy is still raging, the distinction between what exactly is RISC and what is CISC is reducing to the extent of being almost indistinguishable except at the basic philosophical level. Intel, which held on to CISC for many years, bowed down to the RISC architecture by designing its Pentium Pro with complex instructions which internally were broken down to simple RISC like instructions. So, the comment on it is that the latest x86 processors are CISC processors that convert their

instructions to micro-operations resembling RISC instruction, whereas ARM is a RISC processor which also has complex instructions that resemble CISC.

0.5 | Number Systems

Motivation

In the study of microprocessors, we will have to use many different number systems, and conversions from one system to the other. Clarity of these ideas is very important for correct computation and the right interpretation of results. This is the motivation for a review on it, though most of you have had an introduction to it already.

We have become quite used to the number system which we call the decimal number system, which is a system with a base (radix) 10. We are so used to this system of numbers that our visualization of quantity is always based on this. Our mental faculties are tuned to perform all calculations in this number system. In contrast, computers are not comfortable with this system – we know that they use the binary system of numbers and all computations are done in the binary format. Thus, we have a problem when we use computers to perform computations for us. So, let us start this discussion by first understanding the intricacies of each of the commonly used number systems. We will discuss the ones that we most often might have to use in the context of computers.

0.5.1 / The Decimal System

The base of this system is 10 (ten) – and it naturally follows that there are ten defined symbols in this system and the combinations of these ten symbols give us various values. The ten symbols here are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, and they are called ‘digits’. The **position** of a digit in a number is what gives its value.

For example, how does the number 346 get its value?

$$\begin{aligned} 346 &= 3 \times 10^2 + 4 \times 10^1 + 6 \times 10^0 \\ &= 300 + 40 + 6 \end{aligned}$$

This means that associated with each position, there is a weight. Here, the weight is a power of 10. Thus

$$\begin{aligned}
 56785 &= 5 \times 10^4 + 6 \times 10^3 + 7 \times 10^2 + 8 \times 10^1 + 5 \times 10^0 \\
 &= 50000 + 6000 + 700 + 80 + 5
 \end{aligned}$$

What about fractional numbers? The positions on the right side of the decimal number also have weights, but the powers (of 10) are negative.

$$\begin{aligned}
 6.785 &= 6 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2} + 5 \times 10^{-3} \\
 &= 6 \times 1 + 7 \times 0.1 + 8 \times 0.01 + 5 \times 0.001 \\
 &= 6 + 0.7 + 0.08 + 0.005
 \end{aligned}$$

These are things that we know very well. They have been reviewed here to show that the same concept applies to other number systems as well.

0.5.2 / The Binary Number System

The base of this system is 2, and so it has two symbols, 0 and 1, each of them being called a bit. So, each position has a weight, which is a power of 2. Take the number 110110. Let us find its value. Since there are 6 bits, there are six positions, with weights as shown for the bits:

Power of 2:	2^5	$+2^4$	$+2^3$	$+2^2$	$+2^1$	$+2^0$
Weight:	32	+16	+8	+4	+2	+1
Number:	1	1	0	1	1	0
Value:	1×32	$+1 \times 16$	$+0 \times 8$	$+1 \times 4$	$+1 \times 2$	$+0 \times 1$

Adding the values in all the bit positions gives $32 + 16 + 0 + 4 + 2 + 0 = 54$. This is the equivalent value in the decimal system. We cannot help putting back everything into the decimal system, because this is the number system with which we are most familiar and comfortable.

Note Sometimes binary numbers are suffixed with B to indicate that they are binary numbers, e.g., 110110B, 1010110B. Sometimes, the notation 110110₂ is also used.

Note Keep the calculator in the PC (Accessories of Windows) open in the scientific mode. This will help to verify all the calculations we are going to do from now on.

Next, let us try to understand the concept of fractional binary numbers.

Example 0.1

Find the decimal values of the binary number 1001.011 B

Solution

Power of 2:	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
Weight	8	4	2	1	0.5	0.25	0.125
Number	1	0	0	1	.	0	1
Value	8	+ 0	+ 0	+ 1	.	0	+ 0.25 + 0.125
	$= 9.375$						

Example 0.1 shows 1001.011 in binary (often written as 1001.011_2).

It also shows the power and weight or value of each digit position. Thus, 1001.001 is equivalent in decimal to 9.375 ($8 + 1 + 0.25 + 0.125$).

Notice that this is the sum of $2^3 + 2^0 + 2^{-2} + 2^{-3}$, but 2^2 and 2^1 are not added, as the bit under these positions is 0. The fractional part is composed of 2^{-2} and 2^{-3} , but there is no digit under 2^{-1} , so 0.5 is not added.

0.5.3 / The Hexadecimal Number System

Next is the hexadecimal system of numbering which has 16 symbols namely 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The base of the system is 16 and each symbol is called a 'hex digit'. Each position in the hexadecimal number system has a weight, which is a power of 16. Let us find the value of 240FCH. The letter 'H' is suffixed to the number if it is needed to make clear that it is a hexadecimal number. A to F have the decimal values of 10 to 15.

Power of 16:	16^4	$+16^3$	$+16^2$	$+16^1$	$+16^0$
Weight:	65536	+4096	+256	+16	+1
Number:	2	4	0	F	C
Value:	2×65536	$+4 \times 4096$	$+0 \times 256$	$+15 \times 16$	$+12 \times 1$
i.e.,	131072	+16384	+0	+240	+12
	$= 147708$				

So, we have calculated the equivalent decimal value of the given hex number by using the concept of positional weights.

Example 0.2

Find the decimal value of the hex number 25.1H

Solution

Power of 16:	16^1	$+16^0$	$+16^{-1}$
Weight:	16	+1	+0.0625
Number:	2	5	.
Value	2×16	$+5 \times 1$	$. +1 \times 0.0625$
i.e.,	32	+5	.
			= 0.0625

There is also an octal system whose base is 8. The equivalent calculations involved in this are left as an exercise for the interested student. In all the above, we have done the conversion to decimal form from other number systems. Now, we will see how we will convert a decimal number to other systems of numbering.

Notes

1. i) In most computers, the default number system for writing numbers is decimal. When we mean decimal numbers, we simply write it as it is – like 35, 687 and 234 and so on. A number in hex form is suffixed with the letter H, for example, 56H, 8FH, 0AH and so on.
2. ii) The numbers from 0 to 9 are the same in the decimal and the hexadecimal system. So, in the forthcoming chapters, you will see that no 'H' is added when writing numbers from 0 to 9, though there is nothing wrong in writing 7H, 8H, 01H and so on.

0.6 | Number Format Conversions

0.6.1 / Conversion from Decimal to Binary

The method is to divide the decimal number by 2 until the quotient is 0. See the technique illustrated below.

Example 0.3

Find the binary value of 13.

Solution

Divide 13 by 2 repeatedly and save the remainders

$$\begin{array}{r}
 2) \underline{13} & \text{remainder} = 1 \\
 2) \underline{6} & \text{remainder} = 0 \\
 2) \underline{3} & \text{remainder} = 1 \\
 2) \underline{1} & \text{remainder} = 1 \\
 0
 \end{array}$$

Now write the remainders from bottom to top, as one line from left to right. We get 1101 as the converted binary number.

Thus, we have been able to convert from decimal to binary by repeated division by 2, the base of the binary number system. To verify, try converting this binary number back to decimal. It should be 13. Or simply use the scientific calculator to verify the conversion. (Make sure it is kept open on your PC's desktop.)

Example 0.4

Convert the number 213 to binary form.

Solution

$$\begin{array}{r} 2) \underline{213} & \text{remainder} = 1 \\ 2) \underline{106} & \text{remainder} = 0 \\ 2) \underline{53} & \text{remainder} = 1 \\ 2) \underline{26} & \text{remainder} = 0 \\ 2) \underline{13} & \text{remainder} = 1 \\ 2) \underline{6} & \text{remainder} = 0 \\ 2) \underline{3} & \text{remainder} = 1 \\ 2) \underline{1} & \text{remainder} = 1 \\ 0 & \end{array}$$

Now, write the remainders from bottom to top in one line, from left to right. The number is 11010101.

0.6.2 / Conversion from Decimal to Hexadecimal

Conversion from decimal to hexadecimal is accomplished by dividing by 16 and finding the remainders. Remainders ranging from 10 to 15 will be written using the hexadecimal symbols A to F. See how 225 is converted to a hexadecimal form.

$$\begin{array}{r} 16) \underline{225} & \text{remainder} = 1 \\ 16) \underline{14} & \text{remainder} = E \\ 0 & \end{array}$$

Result = E1

The method for this is obvious; i.e., divide repeatedly the decimal number by 16 and keep the remainders. Do this until the quotient is 0.

Example 0.5

Convert the decimal number 4152 to hexadecimal.

Solution

16) <u>4152</u>	remainder = 8
16) <u>259</u>	remainder = 3
16) <u>16</u>	remainder = 0
16) <u>1</u>	remainder = 1
0	

Take the remainders from bottom to top and write it in a single line from left to right. The number is 1038H.

0.6.3 / Converting from Binary to Hexadecimal

If we take any hex digit, note that its decimal value ranges from 0 to 15. For example F is 15, A is 10 and so on. If a hex digit has to be converted to a binary number, the maximum number of bits required is 4.

See Table 0.2. Any hex digit can be written as a group of four bits. Taking an example, 4C57FH can be written in binary, by writing the equivalent binary of each of the digits

Hex 4. C 5 7 F

Binary 0100 1100 0101 0111 1111

Table 0.2 | Hex, Binary and Decimal Representations

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

The binary value of 4C57FH is 01001100010101111111. Looking at both the representations tells us the biggest problem with binary numbers – they are long and cumbersome to handle. Putting them into a hex form makes the representation short and concise – we conclude that the binary

representation is an expanded form of the hexadecimal representation where each hex digit is expanded to its 4-bit binary form.

If we have a long binary number, what we can do to convert it into hex form is to divide it into groups of 4 bits (starting from the right, i.e., the LSB). Then, write the hex representation of each 4-bit binary group. Try this technique with the following binary number:

11100101010100011101.

1110	0101	0101	0001	1101	B	binary
E	5	5	1	D	H	hex

Example 0.6

Convert the following binary number to hex form.

10111110000111110001.

Solution

It is the practice to write binary as groups of four with a space between the groups. This increases the readability of the binary number.

1011	1110	0001	1111	0001
A	E	1	F	1

The equivalent hex number is AE1F1H.

Example 0.7

Convert the following hexadecimal number to binary form.

3AF24H

Solution

Take each hexadecimal digit and write its equivalent four-bit binary value.

3	A	F	2	4
0011	1010	1111	0010	0100

From all this, we should realize that the hexadecimal notation is a contracted form of rebinary number representation. Computers do all their processing using binary numbers, but it is easier for us to represent that

binary number in hex form. So, when we ask the computer to add 34H and 5DH, it actually expands these into binary form and does the addition.

0.6.4 / BCD Numbers

BCD stands for ‘Binary Coded Decimal’ but there is more to it than being just a binary representation of a decimal number. Let us look into the details.

Decimal numbers are represented by 10 symbols from 0 to 9, each of them being called a digit. We know the binary code for each of these decimal numbers. Suppose we represent one decimal digit as a byte, it is called ‘unpacked BCD’. Consider the representation of 9 – it is written as 00001001. Now if we want to write 98 in unpacked BCD, it is written as two bytes:

9	8
00001001	00001000

Thus, the binary code of each decimal digit is in one byte.

Packed BCD

What, then, is ‘packed BCD’? When each digit is packed into 4 binary bits, it is packed BCD. Thus 98 is

9	8
1001	1000

Each digit needs a nibble (four bits) to represent it. The packed BCD form of 675 is 0110 0111 0101. The **important point to remember is that since there is no digit greater than 9, no BCD nibble can have a code greater than ‘1001’**. Computers do process BCD numbers, but the user must be aware of the number representation that is being used.

Can we write BCD numbers in hex? Yes, because the hex representation is just a concise representation of binary numbers. The decimal number 675 when written as 675H represents the packed BCD in hex form. There is no need to be confused about this, because the steps involved are as follows:

1. Write the binary equivalent of each decimal number as a nibble.
2. Write the hex equivalent of each nibble.

675 is	0110	0111	0101
	6	7	5
			H

Spend a few moments thinking, to make it clear. **One important point to keep in mind is that when we represent BCD in hex form, no digit will ever take the value of A to F, since decimal digits are limited to 9.** So, there will never be a BCD number such as 8F5H or 56DH or A34H.

Example 0.8

Find the binary, hex and packed BCD representation of the decimal numbers 126 and 245. Also write the packed BCD in the hex format.

Solution

Number	Binary	Hex	BCD	BCD in hex form
126	0111 1110	7EH	0001 0010 0110	126H
245	1111 0101	F5H	0010 0100 0101	245H

Example 0.9

Find the packed BCD value of the decimal number 2347654 and represent the BCD in hex format.

Solution

To find the BCD, each digit is to be coded in 4-bit binary.

Hence, 2347659 is

0010 0011 0100 0111 0110 0101 1001; i.e., 2347659H is the hex representation of the BCD number.

It is very important to keep this in mind when we do programs using BCD arithmetic. Whenever you have doubts, just refer back to this chapter.

0.6.5 / ASCII Code

This word, pronounced as ‘ask-ee’, is the abbreviation of the words ‘American Standard Code for Information Interchange’. This is the code used when entering data through the keyboard and displaying text on the video display. It is very important to know what it is and how this code is used.

ASCII is a seven bit code, which is written as a byte. It has representations for numbers, lower case and upper case English alphabets, special characters (like # , ^ . &) and control characters. For example, there are

ASCII codes for ‘new line’, carriage return and the space bar. A number of characters are related to printing. When we type a character on the keyboard, it is the ASCII value of the key that is read in. The computer must convert it from this form to binary form for processing. The list of ASCII codes is shown in Table 0.3. Note that the ASCII value of numbers from 0 to 9 is 30H to 39H. The ASCII of upper case alphabets starts from 41H and that for lower case starts from 61H. This table will be needed as quick reference for various calculations we will do in the programming chapters.

Table 0.3 | The ASCII Code – Symbols versus Hex Value

Symbol	ASCII (Hex)	Symbol	ASCII (Hex)	Symbol	ASCII (Hex)	Symbol	ASCII (Hex)
NUL	0	DLE	10	(Space)	20	0	30
SOH	1	DC1	11	!	21	1	31
STX	2	DC2	12	*	22	2	32
ETX	3	DC3	13	#	23	3	33
EOT	4	DC4	14	\$	24	4	34
ENQ	5	NAK	15	%	25	5	35
ACK	6	SYN	16	&	26	6	36
BEL	7	ETB	17	.	27	7	37
BS	8	CAN	18	(28	8	38
Tab	9	EM	19)	29	9	39
LF	A	SUB	1A	*	2A	:	3A
VT	B	ESC	1B	+	2B	;	3B
FF	C	FS	1C	,	2C	<	3C
CR	D	GS	1D	-	2D	=	3D
SO	E	RS	1E	.	2E	>	3E
SI	F	US	1F	/	2F	?	3F
@	40	P	50	`	60	P	70
A	41	Q	51	a	61	q	71
B	42	R	52	b	62	r	72
C	43	S	53	c	63	s	73
D	44	T	54	d	64	t	74
E	45	U	55	e	65	u	75
F	46	V	56	f	66	v	76
G	47	W	57	g	67	w	77
H	48	X	58	h	68	x	78
I	49	Y	59	i	69	y	79
J	4A	Z	5A	j	6A	z	7A
K	4B	[5B	k	6B	{	7B
L	4C	/	5C	l	6C	7C	
M	4D]	5D	m	6D	}	7D
N	4E	^	5E	n	6E	~	7E
O	4F	-	5F	o	6F	?	7F

0.6.6 / Representation of Negative Numbers

There are various ways of representing negative numbers – like signed magnitude, one’s complement, two’s complement and so on, but we will

straightway discuss the representation used by computers for this. Computers use the ‘two’s complement’ representation for negative numbers. The method is to complement each bit of the number and add a ‘1’ to this. Let us see how it is done.

We will start with 4-bit numbers. Say we want to represent -6.

1. Write the 4-bit binary value of 6: 0110.
2. Complement each bit: 1001.
3. Add ‘1’ to this: 1010.

So, -6 is ‘1010’, for computers.

Let us try this for all the numbers from 0 to 7. See Table 0.4, which shows the positive and negative number representation of numbers possible to be represented in four bits. A number of observations can be made from Table 0.4.

1. The range of numbers that can be represented by 4 bits is -8 to +7. For an n -bit number, this range works out to be (-2^{n-1}) to $(+2^{n-1}-1)$.
2. In this notation, the most significant bit (MSB) is considered to be the sign bit. The MSB for positive numbers is ‘0’ and for negative numbers is 1.
3. There is a unique representation for 0.

Table 0.4 | Negative and Positive Number Representation in 4-bit Binary

Negative Numbers	Binary	Hex	Positive numbers	Binary	Hex
-8	1000	8			
-7	1001	9	+ 7	0111	7
-6	1010	A	+ 6	0110	6
-5	1011	B	+ 5	0101	5
-4	1100	C	+ 4	0100	4
-3	1101	D	+ 3	0011	3
-2	1110	E	+ 2	0010	2
-1	1111	F	+ 1	0001	1
-0	0000	0	+ 0	0000	0

Since we will deal mostly with bytes and words (16-bit), let’s have a feel of 8-bit negative number representation.

Example 0.10

Find the two's complement number corresponding to -6 when 6 is represented in 8 bits as 0000 0110.

Solution

The steps: 0000 0110

1111 1001	;complement each bit
1111 1010	;add '1' to it
F A	;in hex

Thus, -6 is FAH in 8-bit form, while it is AH in 4-bit form (from Table 0.4)

Note H is the notation for 'hexadecimal'.

One very important point we need to observe and keep in mind is that when a 4-bit number is expanded into an 8-bit form, its sign bit has to be extended into the 8 bits. The sign bit in the 4-bit representation of -6 is '1'. When expanding the number to fill into 8 bits, the 1 is replicated 4 more times to fill the whole byte. Thus, -6, which is AH in 4-bit form, becomes FAH in byte form and will be FFFAH in 16-bit format and FFFF FFFAH in 32-bit format. We need to understand this for negative numbers. For positive numbers, we do it without much thinking. So, +6 is 0110, which expands to be 0000 0110 (byte) or 06H, and 0006H in 16-bit format and 0000 0006 H in the 32-bit format. Note that for positive numbers, the sign bit is 0; effectively we are doing sign extension here too. This concept of 'sign extension' is important and we will deal with it in greater detail later.

Conversion from Two's Complement Form

Given the two's complement representation of a decimal number, how do we find the decimal number which it represents? The answer is – two's complement it again.

Take FA.

1111 1010	...the number
0000 0101	+ ...invert each bit
1	...add 1
0000 0110	...its 2's complement

This is 6. Thus, FA is the two's complement representation of -6.

Example 0.11

Find the decimal number whose two's complement representation is given.

1. FFF2H
2. F9H

Solution

1. FFF2H

Taking two's complement gives 000E

which is 1110, i.e., 14, which means that -14 is the number represented by FFF2H.

2. F9H

Taking two's complement gives

0111, i.e., 7, which means that -7 is the number represented by F9H.

Question Looking at the result of various arithmetic operations on binary numbers, how do we know whether it is a positive or a negative number? What is your observation regarding signed numbers?

Answer We should know how many bits are used for the representation of a signed number in the system. Then, if the MSB is a '1', it is a negative number; if the MSB is a '0', it is a positive number.

0.7 | Computer Arithmetic

0.7.1 / Addition of Unsigned Numbers

When we say that a number is unsigned, it implies that the sign of the number is irrelevant, which actually means that we consider the numbers as having no sign bit – all the bits allotted for the data are used for the magnitude alone; in effect, it turns out that these refer to positive numbers. With 8 bits, numbers from 0 to 255 can be used.

Binary addition is something that you have already learnt. Here, we are reviewing it to bring into focus some important points which we may have to be taken care of in the study of microprocessor programming.

Binary addition is done by adding bits column wise. We will consider byte sized data.

Case 1

Binary	Decimal	Hexadecimal
0101 1001 +	89 +	59H +
0110 1001	105	69H
<u>1100 0010</u>	<u>194</u>	<u>C2H</u>

Addition of the same numbers in the binary, decimal and hexadecimal formats is shown. Since the sum lies within a value of 255, there is no special problem in this case.

Case 2

0111 1000	120 +	78H +
1001 1001	153	99H
<u>10001 0001</u>	<u>273</u>	<u>111H</u>

In this case, the sum is greater than the number of bits allotted for the operand, and the extra bit, beyond the 8 bits of the sum, is called a 'carry'. Whenever a carry appears, it indicates the insufficiency of the space allocated for the result. In microprocessors, there is a flag that indicates this condition.

0.7.2 / Addition of Packed BCD Numbers

Now, let us add packed BCD numbers.

Case 1

Consider the case of two packed BCD bytes that are to be added, say 45 and 22.

Packed BCD	Packed BCD in hex form	Decimal
0100 0101 +	45H +	45 +
0010 0010	22H	22
<u>0110 0111</u>	<u>67H</u>	<u>67</u>

In this case, the upper nibble and lower nibble are within 0 to 9. So, the addition proceeds just like normal decimal addition.

Case 2

Consider the case of two packed BCD bytes that are to be added, say 45 and 27. In BCD form, the correct answer should be 72. However, this is not obtained directly.

Packed BCD	Packed BCD in hex form	Decimal
0100 0101 +	45H +	45 +
0010 0111	27H	27
0110 1100	6CH	72

When adding in binary form, the lower nibble of the sum is greater than 9. Since no BCD digit can have a value greater than 9, a correction needs to be applied here. The correction to get the sum back to BCD form is to add 6 (0110) to the lower nibble alone.

Correction

0110 1100 +
0000 0110
0111 0010

This gives the correct sum of 72.

Case 3

This is when the upper nibble of the sum is greater than 9. The correction is to add 6 to the upper nibble alone.

Add BCD 76 and 62. In binary form, the additions are

0111 0110 +	76H +	
0110 0010	62H	
1101 1000 +	D8H	Now adding 6 to the upper nibble,
0110 0000	60H	
1 0011 1000	138H	

However, note that the data size exceeds 99, which is the maximum number that 8 bits can accommodate for a packed BCD number. Thus, there is a 'carry' generated from the addition operation. However, if the carry is also included in the answer, the sum of 138 is correct. However, more than 8 bits are needed for the sum.

Case 4

When both the upper and lower nibbles of the sum are greater than 9, add 6 to both nibbles. Add BCD 89 and 72.

$$\begin{array}{r}
 \begin{array}{r} 1000 \ 1001 \\ + \\ 0111 \ 0010 \\ \hline 1111 \ 1011 \\ + \\ 0110 \ 0110 \\ \hline 10110 \ 0001 \end{array}
 \quad
 \begin{array}{r} 89H \quad + \\ 72H \\ \hline FBH \quad \text{add 06 to both nibbles} \\ 66H \\ \hline 161H \end{array}
 \end{array}$$

The right answer of 161 is obtained. However, the sum needs more than one byte space.

Example 0.12

Perform the addition of the following numbers after converting to decimal and hexadecimal forms.

1. 39 and 99
2. 117 and 156

Solution

	Decimal	Binary	Hexadecimal
i)	$\begin{array}{r} 39 \\ + \\ 99 \\ \hline 138 \end{array}$	$\begin{array}{r} 0010 \ 0111 \\ + \\ 0110 \ 0011 \\ \hline 1000 \ 1010 \end{array}$	$\begin{array}{r} 27H \\ + \\ 63H \\ \hline 8AH \end{array}$
ii)	$\begin{array}{r} 117 \\ + \\ 156 \\ \hline 273 \end{array}$	$\begin{array}{r} 0111 \ 0101 \\ + \\ 1001 \ 1100 \\ \hline 10001 \ 0001 \end{array}$	$\begin{array}{r} 75H \\ + \\ 9CH \\ \hline 111H \end{array}$

In the second addition, the data has exceeded the size which can be accommodated in 8 bits. Hence, a carry will be generated. In microprocessors, there is a flag which indicates this condition.

0.7.3 / Addition of Negative Numbers

We know now that negative numbers are represented in two's complement notation. Let's consider adding two negative numbers.

Example 0.13

Add -43 and -56.

Solution

Convert the two numbers into their two's complement form, as both are negative numbers.

$$\begin{array}{r}
 -43 + 1101\ 0101 + \\
 -56 \quad 1100\ 1000 \\
 \hline
 -99 \quad 1\ 1001\ 1101
 \end{array}$$

We are adding two 8-bit numbers. If the sum exceeds 8 bits, an extra bit is generated from the addition. Ignore this carry and look at the eight bits of the sum. (This is the rule for two's complement addition.)

It is 1001 1101. The MSB is found to be '1'. So, we know that it is a negative number. To find the decimal number whose two's complement representation this is, take the two's complement of the sum. This comes to be 0110 0011, i.e., 99. Thus, we verify the correctness of our addition procedure.

Example 0.14

Add +90 and -26.

Solution

One number is positive and the other is negative.

$$\begin{array}{r}
 +90 \quad 0101\ 1010 + \\
 -26 \quad 1110\ 0110 \\
 \hline
 64 \quad 1\ 0100\ 0000
 \end{array}$$

Ignore the end around carry. The sum is 0100 0000. Since the MSB of the number is '0', we understand that the sum is positive. So, convert it to decimal. The result is 64.

Example 0.15

Add -120 and +45.

Solution

$$\begin{array}{r}
 -120 \quad 1000\ 1000 + \\
 +45 \quad 0010\ 1101 \\
 \hline
 -75 \quad 1011\ 0101
 \end{array}$$

Look at the sum; the MSB of the sum is '1'. Hence, it is a negative number. The two's complement of this is 0100 1011, i.e., 75. Thus, the result of the calculation is -75.

Note In all the above calculations, we have used data of 8 bits. The result of the calculations was in the range of -128 to +127. Thus, the answers are correct. If the sum goes outside this (for eight-bit data), the answers will be wrong, and havoc will be created if one is not aware of that. Computers have ‘flags’ to let us know of this. This will be discussed in later sections.

0.7.4 | Subtraction

Unsigned Numbers

1. Binary numbers
2. Hexadecimal numbers
3. BCD numbers

The procedure here is similar to addition, i.e., bit by bit, column by column subtraction. Sometimes, borrows from the columns on the left are needed.

Example 0.16

Subtract 56 from 230. Do this subtraction after converting numbers to binary and hex.

Solution

$$\begin{array}{r}
 230 \quad - \quad 1110\ 0110 \quad - \quad E6H \quad - \\
 56 \qquad \qquad 0011\ 1000 \qquad \qquad 38H \\
 \hline
 174 \qquad \qquad \underline{\underline{1010\ 1110}} \qquad \qquad \underline{\underline{AEH}}
 \end{array}$$

In the above subtraction, we are subtracting a smaller number from a bigger number. However, when subtracting column-wise, sometimes there is the issue of having to subtract a bigger number from a smaller number. We know the idea of ‘borrow’ from the left-hand column. However, for the borrowing, with which we append the number depends on the base of the number system. For the decimal system, we borrow 10, for binary 2 and for hex we borrow 16.

Check this hexadecimal subtraction:

$$\begin{array}{r}
 E6H \quad - \\
 38H \\
 \hline
 \underline{\underline{AEH}}
 \end{array}$$

Starting from the rightmost column, we see that we cannot subtract 8 from 6. So, borrowing from E is needed. Borrowing from E leaves E to become D and 6 becomes $6 + 16 = 22$ (in decimal). Subtracting 8 from 22 gives 14,

which is E in hex. That is how we get E in rightmost column of the result. Then, going over to the left, subtract 3 from D (13 in decimal). This is 10 (in decimal) and A in hex. That is how the result of the subtraction is A.

This idea has been explained here in detail, so that we can use a similar idea in BCD subtraction.

0.7.5 / Packed BCD Subtraction

Let us use the same numbers for BCD subtraction as we did in Example 0.16; i.e., subtract 56 from 230. The BCD representation is shown below. Each decimal digit is packed in to 4-bit binary bits.

Decimal	Packed BCD
230	0010 0011 0000
- 56	- 0000 0101 0110
<u>174</u>	<u>0001 0111 0100</u>

The point to remember here is that each group of 4 bits represents a ‘decimal number’, the base of which is ten. Thus, when we try to subtract a bigger number from a smaller number, we have to consider the ‘four bits together’ as a decimal number. Let us review the steps in the above - subtraction.

First step

Thus, when we have to subtract 6 from 0 in the rightmost group of four bits, we need to borrow. Borrow from the group on the left a decimal 10, and add it to the ‘0000’ on the right. That makes it ‘1010’ (because of borrowing, the 0011 on the left is now ‘0010’). Then, subtract 0110 from this. The result is 0100 as seen (within the group, binary subtraction is done).

0010 0010 1010	-
<u>0110</u>	
<u>0100</u>	

Second step

This is the second group. For subtracting 0101 from 0010, borrowing of decimal 10 is taken from the leftmost group. Thus, 0010 is ‘1100’, 12 in decimal. Subtracting ‘0101’ (5) from it, gives ‘0111’ (7) as shown.

$$\begin{array}{r}
 0001 \ 1100 \ 1010 \ - \\
 0101 \ 0110 \\
 \hline
 0111 \ 0100
 \end{array}$$

Third step

The leftmost group is now 0001. Subtract 0000 from it. Thus, the final answer is 174 in packed BCD form.

$$\begin{array}{r}
 0001 \ 1101 \ 1010 \ - \\
 0000 \ 0101 \ 0110 \\
 \hline
 0001 \ 0111 \ 0100
 \end{array}$$

All this shows that BCD subtraction also needs extra care as BCD addition. In computers, special instructions take care of this.

Example 0.17

Express the numbers 53 and 18 in packed BCD and subtract the latter from the former.

Solution

Decimal	Packed BCD
53	0101 0011
18	0001 1000

First step

Borrowing from the left side nibble to the nibble on the right side gives

$$\begin{array}{r}
 0100 \ 1101 \ - \\
 0001 \ 1000 \\
 \hline
 0101
 \end{array}$$

Second step

$$\begin{array}{r}
 0100 \ 1101 \ - \\
 0001 \ 1000 \\
 \hline
 0011 \ 0101
 \end{array}$$

The result is 35, as it should be.

0.7.6 / Subtraction of Signed Numbers

Subtraction is the process of changing the sign of the second number and adding to the first.

65–34 is $65 + (-34)$.

So, when we do subtraction, we actually add the two's complement form (i.e., the negative) of the second number to the first number. This is what computers actually do when they perform subtraction. In the discussion of subtraction in Section 0.7.4, this was not explicitly mentioned, because the idea then was to present certain other intricacies related to subtraction. Now, let us discuss subtraction for 8-bit signed numbers. Keep in mind that the range of signed numbers usable with 8 bits is –128 to + 127.

Example 0.18

Perform subtraction of the following signed numbers:

1. +26 from +68
2. +26 from –68

Solution

1. +26 from +68

This comes to be a computation in the form of $68 + (-26)$. For this, the two's complement form of 26 should be added to 68.

Decimal Binary

$$\begin{array}{r} 68 \text{ is } & 0100 & 0100 & + \\ -26 \text{ is } & 1110 & 0110 \\ \hline 1 & 0010 & 1010 \end{array}$$

Ignore the extra bit generated. Since the MSB is '0', the result is positive. The result is 0010 1010, i.e., 42.

- ii) +26 from –68 i.e., $-68 - (26) = -68 + (-26)$

$$\begin{array}{r} -68 \text{ is (in two's complement form) } 1011 & 1100 & + \\ -26 & 1110 & 0110 \\ \hline -94 & 1 & 1010 & 0010 \end{array}$$

Ignore the extra bit generated. Since the MSB of the 8-bit result 1010 0010 is '1', the difference of the two numbers is negative.

Take the two's complement of this; 0101 1110, i.e., 94. So, the result of the computation is –94.

Example 0.19

Find the result of the following subtraction:

1. -56 from + 23
2. -56 from -23

Solution

1. -56 from + 23

The computation to be done is $+23 - (-56)$ i.e., $23 + 56$. This turns out to be the addition of the two positive numbers 23 and 56.

$$\begin{array}{r} 23 \\ + \quad \quad \quad 0001\ 0111 \\ \hline 56 \\ + \quad \quad \quad 0011\ 1000 \\ \hline 79 \\ + \quad \quad \quad 0100\ 1111 \\ \hline \end{array} \quad \text{i.e., } 79$$

2. -56 from -23

The computation to be done is $-23 - (-56)$ i.e., $-23 + 56$.

$$\begin{array}{r} -23 \\ + \quad \quad \quad 1110\ 1001 \\ \hline 56 \\ + \quad \quad \quad 0011\ 1000 \\ \hline 33 \\ + \quad \quad \quad 1\ 0010\ 0001 \\ \hline \end{array}$$

Ignore the extra bit generated. The MSB of the 8-bit result 0010 0001 is '0'. So, the number is positive and is 33 in decimal, as it should be.

Overflow into the Signed Bit

Whenever we use 8-bit signed numbers in addition or subtraction, the result is found to be correct in sign and magnitude if it is within the range of -128 to +127. However, suppose this is violated? What happens then? A typical case is when two negative numbers are added. Try adding -100 and -55. Both the operands are within the allowed range. See the addition.

$$\begin{array}{r} -100 \\ + \quad \quad \quad 1001\ 1100 \\ -55 \\ + \quad \quad \quad 1100\ 1001 \\ \hline -155 \\ + \quad \quad \quad 1\ 0110\ 0101 \\ \hline \end{array}$$

Ignore the extra carry bit and look at the 8-bit result. The MSB of the result is '0' indicating that it is a positive number. However, we know that the answer is negative. What caused the error? Because the sum was too large (larger than -128) to fit into the 8 bits allotted to it; there was an 'overflow into the sign bit' causing the sign bit to be changed. (A similar issue occurs when we add two positive numbers and the sum is greater than +127). In computers, there is a flag which tells us when there is an overflow into the sign bit causing it to be inverted. These matters will be discussed in detail when we do programming.

0.7.7 / Addition of Numbers of Different Lengths

We have discussed computer arithmetic in detail, because it is very important to be clear about it, so as to be able to understand how the microprocessor responds to different data types and arithmetic operations. Now, let's try to understand how data of different data widths are dealt with.

Data can have different sizes depending on the processor. The 8086 can have data of 8 bits and 16 bits, while Pentium can handle 8, 16 and 32 bits internally. Sometimes, it may be required to add / subtract data of different widths. In these cases, the important thing to do is to equalize the size of the data involved. Processors do not allow addition/subtraction of data of different widths. So, a byte will have to be converted to a 16-bit word if it has to be added to a 16-bit number. The way it is done depends on whether the data is signed or unsigned. For unsigned data, the byte is appended with zeros in the upper byte and converted to a 16-bit word. For signed data, the byte should be 'sign extended' to make it a 16-bit word. Refer Section 0.8.6 once again to convince yourself of the necessity for this.

Example 0.20

Add the unsigned numbers 35H and 7890H.

Solution

In this, 35H is appended with zeros to make it 0035H.

$$\begin{array}{r} 0035H \\ + \\ 7890H \\ \hline \underline{78C5H} \end{array}$$

Example 0.21

Add the following signed numbers:

1. 45H and A87CH
2. A8H and 1045H
3. F5H and B45CH

Solution

1. In this, 45H should be made into a 16-bit number. Check the MSB of this byte. It is '0', meaning that it is a positive number. The sign bit when extended to 16 bits makes the number 0045H. Then, the addition is

$$\begin{array}{r} 0045H \\ + \\ \underline{A87CH} \\ \underline{A8C1H} \end{array}$$

2. In this, the byte is A8H, which has an MSB of '1'. Thus, sign extension makes it FFA8H. Now, the addition is

$$\begin{array}{r} FFA8H \\ + \\ \underline{1045H} \\ \underline{10FEDH} \end{array}$$

The extra bit generated is ignored, like we have done in Section 0.7.3 on signed number computation.

To be sure that this is correct, verification can be done as below.

A8H is -88.

1045H is +4165.

Adding the two, gives us 4077, whose hex representation is 0FEDH.

Add F5H and B45CH.

3. In this, F5H is sign extended to be FFF5H.

$$\begin{array}{r} \text{Adding} \\ FFF5H \\ + \\ \underline{B45CH} \quad \dots \text{note that this is a negative number} \\ \underline{1B451H} \end{array}$$

Ignoring the extra carry bit, the sum is B451H, a negative number. To verify, find the decimal equivalents of the numbers, which are -11 and -19364, which when added give -19375.

Note You may also verify that without extending the negative sign, a wrong result is obtained.

All the calculations we have done can be verified easily using the scientific calculator available on the PC. So, try to be adept in the use of that calculator.

0.8 | Units of Memory Capacity

A memory device is one in which data is stored. How much data a memory device can store depends on its capacity. The capacity of memory is specified as multiples of bytes since memory is byte organized, which means that one byte is stored in one location in memory. So, if there are 100 locations in a memory device, 100 bytes are stored. We all have heard of memory capacity being mentioned in terms such as bytes, kilobytes and megabytes. Now, let us quantify these terms. You will be hearing these terms throughout the use of this book.

A byte is 8 bits. A word is not really defined. It depends on the processor used. For the 8086, a word size is 16 bits. A 32-bit processor may claim to have a word size of 32 bits. Memory capacity is always specified in bytes.

$$2^8 = 256 \text{ bytes}$$

$$2^{10} = 1024 \text{ bytes} = 1 \text{ KiloByte or } 1 \text{ KB}$$

$$2^6 \times 2^{10} = 2^{16} = 64 \text{ KB} = 65,536 \text{ bytes}$$

$$2^{10} \times 2^{10} = 2^{20} = \text{one Mega Byte (1MB)} = 1024 \times 1024 = 1,048,576 \text{ bytes}$$

$$2^{10} \times 2^{20} = 2^{30} = \text{one Giga Byte (1 GB)} = 1024 \times 1024 \times 1024 = 1,073,741,824 \text{ bytes}$$

$$2^{10} \times 2^{30} = 2^{40} = \text{one Tera Byte (TB)} = 1024 \times 1024 \times 1024 \times 1024 = 1,099,511,627,776 \text{ bytes}$$

There are also higher units, which are not so common in usage as yet; but things will change soon, no doubt about it. Some of these units are:

$$\text{Peta Byte (PB)} = 2^{50} \text{ bytes}$$

$$\text{Exa Byte (EB)} = 2^{60} \text{ bytes}$$

$$\text{Zetta Byte (ZB)} = 2^{70} \text{ bytes}$$

0.9 | The 8085 Microprocessor

In this section, we will do a brief review of the 8085 microprocessor. Even though it is not a member of the x86 family, it has an important place in the history of Intel's microprocessors. It was a very popular microprocessor in the days before the advent of the x86 family. It was never used in a PC, but became popular for various uses where computation was involved. It also remained popular in the academic field where it was considered an easy model to teach and understand. Hence, a peep into the attributes of the 8085 would be in order. This section may also serve as a first step in the study of microprocessors, in general.

The 8085 was released by Intel in 1976. It is a 40-pin DIP (dual in line package, in which the pins are mounted as two parallel rows of a rectangular housing). This chip has eight data pins and 16 address pins. Thus, data can be read in and out from the chip as eight bits at a time. The fact that it has 16 address lines imply that the amount of memory it can

access is 2^{16} bytes – i.e., it can have access to memory of 64 KB at the maximum. Besides the data and address pins, it has a number of control pins as well. We will discuss the hardware aspects later and concentrate on the programming model first, because it is easier to understand a processor starting from such a model.

0.9.1 | The Programming Model of 8085

Figure 0.9 shows the programming model of the processor. The functional block diagram shows an arithmetic logic unit (ALU) and its associated blocks. The whole system is controlled by the timing and control unit which synchronizes the activities of the processor with a central clock. All arithmetic and logic computations take place in the ALU. Instructions from memory are brought to an instruction register (not shown) and decoded by the instruction decoder. Accordingly, the specified operation takes place in the ALU. The accumulator is an 8-bit register, called A, and its importance in 8085 is that it serves to hold one of the operands in most two-operand arithmetic and logical operations.

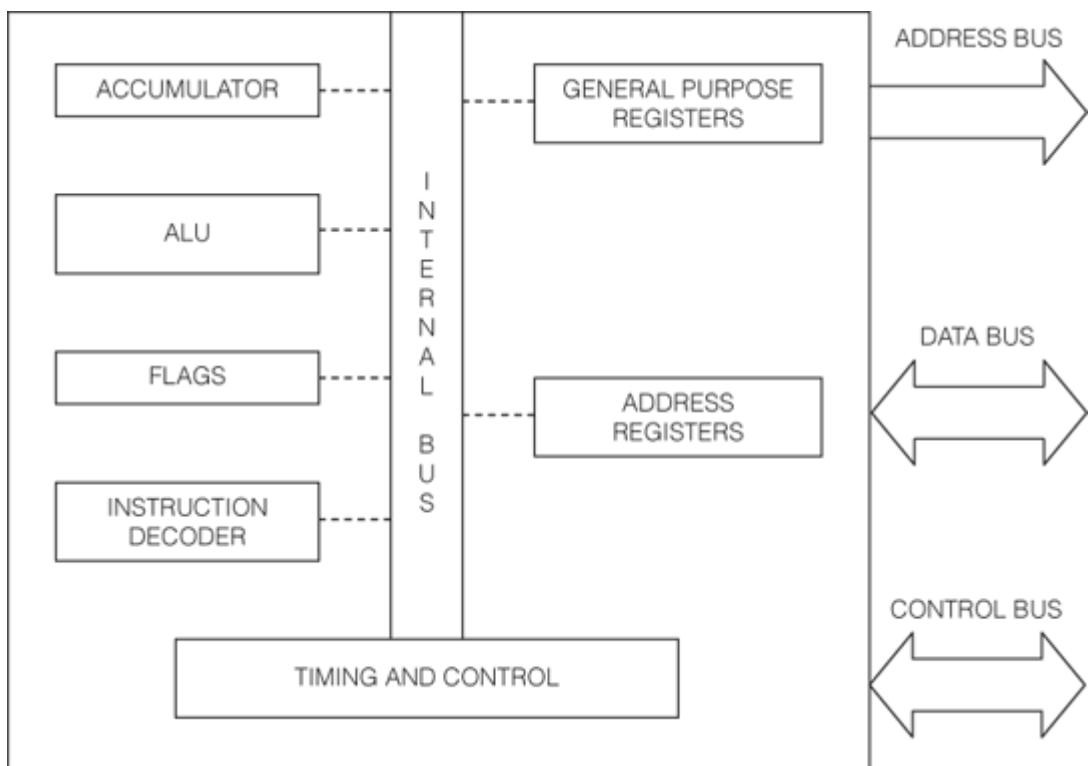


Figure 0.9 | Programming model of the 8085

Also, in many instructions, it is an implied register. There is a flag register which gives information about the result of arithmetic and logical computations. In such cases, 'flag bits' get altered as part of instruction execution.

The model also shows two sets of registers – one is a set of general-purpose registers and the other is a set of address registers (See Fig. 0.10). All processors have registers which are used as temporary storage for operands. ‘General-purpose registers’ are used for data and they are 8-bit registers. The address registers are used for holding memory addresses, and since addresses are 16-bit long, the address registers are 16 bits in size.

PROGRAM COUNTER (16)			
STACK POINTER (16)			
H (8)	L (8)		
D (8)	E (8)		
B (8)	C (8)		
A (8)	FLAGS (8)		

Figure 0.10 | Register structure of the 8085

0.9.1.1 / General-Purpose Registers

They are also designated as scratchpad registers and are 8 bits long. The 8085 is an ‘accumulator’ based machine, which means that for many operations (mostly arithmetic, logical and I/O), it is mandatory to have the operand/one of the operands in the accumulator. In such cases, this register is implied and its name is not written in the instruction. It is denoted as ‘A’ when used in other instructions.

The other 8-bit registers are B, C, D, E, H and L. There is the provision to combine them in pairs to make them 16-bit registers. The allowed combinations are BC, DE and HL where, in the 16-bit form, the first register holds the upper byte of the 16-bit number. Thus, in the BC combination, B holds the upper byte and C the lower byte. Some instructions view the A - register and flag register together as a 16-bit register, called the processor status word (PSW). In that case, the A register is the most significant byte and the flag register is the least significant byte. See Fig. 0.11.

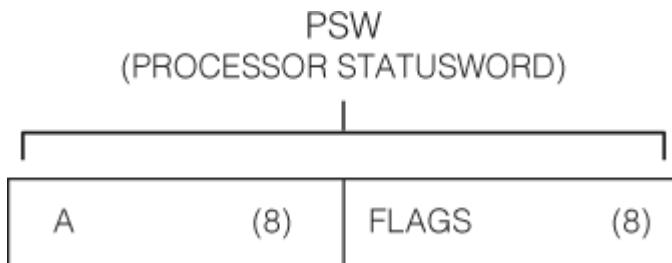


Figure 0.11 | The A and the flag register combined to form the PSW

Flag Register

What is the importance of the flag register? It is an 8-bit register which is a combination of 1-bit flip flops designated as ‘flag bits’. The flags of 8085 are all status or conditional flags, in the sense they are set/reset as a result of the status or condition of the result of certain (not all) instructions. The conditions of the flags are important, because it is based on this that conditional branching is performed. See Fig. 0.12. Now, let us understand each bit of the flag register.

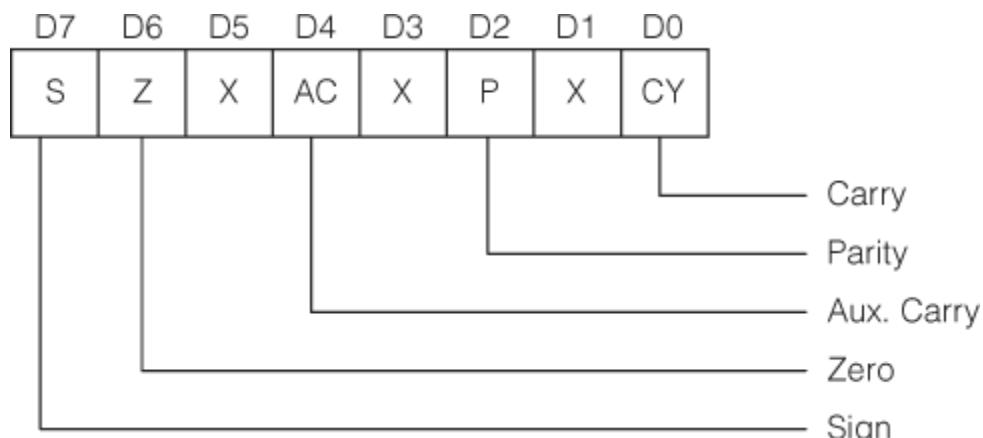


Figure 0.12 | The Flag register bits

Z or Zero flag

When the result of a computation is zero, this flag is set ($Z = 1$); otherwise it is reset.

C or Carry Flag

If an operation causes a bit to overflow from the MSB, this flag is set ($C = 1$); otherwise, it is reset.

S or Sign flag

If the MSB of the result is ‘1’, this flag is set ($S = 1$); otherwise, it is reset.

P or Parity flag

When the result has even parity, this is set ($P = 1$); otherwise it is reset.

AC or Auxiliary Carry flag

This flag is set ($AC = 1$) when there is a carry from D3 to D4 of the result. This flag is used only for BCD calculations, and there is no ‘branching’ operation associated with this flag.

0.9.1.2 / Address Registers

Figure 0.10 shows two 16-bit registers – the program counter and the stack pointer. They are address registers.

Program Counter (PC)

This is a 16-bit register and it is used to ‘sequence’ the instructions being executed. To elaborate this point, think of the process of instruction execution. Instructions are stored in memory, and when execution is to be done, they are fetched one byte at a time from memory to the processor. The PC always points to the ‘next instruction’ to be executed. When a branch instruction appears, it is apparent that the current sequence of instruction fetching is disturbed and a new sequence will be taken up. In that case, PC will be loaded with the ‘branch’ address.

Stack Pointer (SP)

The stack pointer (SP) is a 16-bit register which points to a memory location in RAM which will hold temporary values in an area of RAM called the stack. The stack is explained in detail in Chapter 3.

0.9.2 / Assembly Language Programming

We will now attempt to write a few programs using the assembly language instructions of 8085. Programs that are written are saved in RAM from where instruction bytes are fetched one byte at a time for execution. An instruction may need one or more than one byte space for storage. Executing an instruction may need data from memory addresses as specified in the instruction. Program execution proceeds sequentially one after the other in the order that they are fetched. In this sequence, it may happen that a ‘branch’ instruction is encountered. Branching means ‘jumping’ to a location which is not in the sequence that is followed now. A branch instruction takes control to a different set of instructions written in a different area in memory. Most cases of branching depend on conditions, though unconditional branching is also possible.

The general format of writing assembly code is as shown below:

label: instruction ; comment

Labels are not needed for all lines, and comments are optional – they are used only to enhance the understandability of the instruction line. With these basic ideas, let us go on to the finer aspects of 8085 programming. For most assemblers, a semicolon precedes the comment, and a colon is needed along with a label.

0.9.2.1 / Modes of Addressing

Now that we have the list of registers available in 8085, we can start programming the chip – but we also have to know the modes of addressing before we can actually plunge into programming.

An addressing mode for an instruction implies the way an operand is presented for execution by the instruction. There are single operand as well as double operand instructions for the 8085. In the former case, the operand will be in a register or memory. In the double operand case, the operands will have to be specified to be in some memory address or register and another register. However, there is no mode in which both operands are in memory – this means that if there are two operands, at least one of them has to be in a register.

The general format of a two-operand instruction of any Intel processor can be illustrated by the ‘move’ instruction which is of the form

MOV destination, source

In this, the content of the source is copied to the destination. The source remains unchanged. We will discuss the ideas related to addressing modes using the ‘MOV’ and ADD instructions.

Note: The code lines are not case sensitive.

The addressing modes available are:

1. Register addressing.
2. Direct addressing.
3. Indirect addressing.
4. Immediate addressing.

Register Addressing

In this mode, only registers are involved. Both the source and destination are registers.

MOV A, B	;copy the contents of B to A
MOV C, A	;copy the contents of A to C
ADD A, H	;add the contents of H and A – sum to be in destination i.e., A

Direct Addressing

In this, one of the operands is ‘directly’ addressed. This operand is in memory and its address is ‘directly’ mentioned in the instruction.

LDA 0987H	;load accumulator direct – the address is the 16-bit number 0987H. ;The content of this location is copied to the accumulator (register A)
STA 2345H	;store accumulator direct – the content of the accumulator is stored in the address 2345H
IN 56H	;copy the data in input port with address 56H to register A

Indirect Addressing

Here, one of the operands is ‘indirectly’ specified. It is in memory, but instead of writing its address directly in the instruction, the address is loaded into a 16-bit register and that register is specified in the instruction.

MOV C, M	;copy the content of the ‘memory specified indirectly’ to the C register. For this instruction, the ‘address’ is to be loaded into the HL register pair previous to this instruction
STAX B	;‘store accumulator indirect’ in this, the content of the accumulator is stored in the address specified by the BC register pair.

Immediate Addressing

Here, the source data is written in the instruction itself.

MOV A, 78H	;copy the number 78H to A
ADDA, 67	;add the content of A and the number 67. The sum is to be in A.
LXI H, 3453H	;the immediate number 3453H is loaded in the HL register pair

0.9.3 / Instruction set of 8085

The instruction set can be divided into functional groups as:

1. Data transfer instructions.
2. Arithmetic instructions.
3. Logical instructions.
4. Branch instructions.

We will discuss each group in more detail now, using a sample set of each group.

0.9.3.1 / Data Transfer Instructions

These instructions are responsible for transferring data between registers, register and memory, I/O and accumulator. These also include instructions for loading an 8-bit or 16-bit number into a register of the same size. For certain instructions, the accumulator (A register) is implied. The instructions are listed as:

i) MOV Rd, Rs	;copy data from Rs(source) to destination register(Rd)
ii) MVI Rd,data8	;move 8-bit data to 8-bit register
iii) OUT port address	;send data from accumulator to output port
iv) IN port address	;take in data from input port to accumulator
v) LXI Rp, data16	;load 16-bit data to register pair
vi) MOV R, M	;copy data to register, from indirectly specified memory
vii) MOV M, R	;copy data from register, to indirectly specified memory
viii) LDA address	;load data to accumulator from specified address
ix) STA address	;store data from accumulator to specified address
x) LDAX Rp	;load to accumulator, the data from memory address specified by the register pair (Rp)
xi) STAX Rp	;store into the memory address specified by the register pair Rp, the data in the accumulator

0.9.3.2 / Arithmetic Instructions

For all arithmetic instructions, the A register is implied to contain one operand (if there are two operands for the instruction). The arithmetic group includes instructions to add, subtract, increment, decrement and decimal adjust after addition of BCD. Note that there are no instructions for multiplication and division. **It is important to remember that all conditional flags are affected by arithmetic instructions.**

i) ADD R	;add the content of R to A – sum in A
ii) ADI data8	;add the immediate data to A – sum in A
iii) ADC R	;add the content of R, A and the carry bit – sum in A
iv) ADD M	;add the content of indirectly specified memory to A
v) SUB R	;subtract from A the data in R – result in A
vi) SUI data8	;subtract from A the immediate data – result in A
vii) SBB R	;subtract from A, the content of R and carry – result in A
viii) INR R	;increment the content of the 8-bit register R by 1
ix) DCR R	;decrement the content of the 8-bit register R by 1
x) INR M	;increment by, the content of the indirectly specified memory
xi) DCR M	;increment by 1, the content of the indirectly specified memory
xii) INX Rp	;increment by 1, the content of the register pair Rp
xiii) DCX Rp	;decrement by 1 the content of the register pair Rp
xiv) DAA	;decimal adjust accumulator – to be done after BCD addition

Now let us write a few programs using the instructions we have just learned.

Example 0.22

Bring data from the memory location 4567H, add 56H to it and store the sum in location 0567H.

Solution

```
LDA 4567H      ;copy into the A,data from address 4567H
ADI A,56H       ;add 56H to this - sum is in A
STA 0567H       ;store the content of A in address 0567H.
```

The above is a very simple program in which we have used only the load, store and an immediate add instruction.

Example 0.23

Write a program to add an immediate data to a data in memory address 0987H. Store the sum in the next address in memory.

Solution

In this, an immediate data is loaded into register B. The other operand which is in memory is indirectly addressed. For that, the address of the memory location is loaded into the register pair HL using immediate addressing. Then, the data is moved to A using indirect addressing. This is added to B and the sum is now in A. To store the sum in the next address, increment the address pointer in the HL register pair. See the program below.

```

MVI B,3CH      ;load the immediate data in B
LXI H,0987H    ;load address in the H-L register pair
MOV A,M        ;copy data in the address pointed by HL
ADD A,B        ;add the content of B to A - sum in A
INX H          ;increment the content of HL
MOV M,A        ;store A in the address pointed by HL

```

Example 0.24

There is a byte of data in location 6756H, which is to be overwritten by a number which is less than it by 5. This is done below.

Solution

```

LXI B,6756H    ;load the address of data in the BC pair
LDAX B         ;load in A the content pointed by BC
SUI 05         ;subtract 05 from A-result is in A
STAX 6756H     ;store A in the address pointed by BC

```

Using data transfer and arithmetic instructions, a number of computations can be done. However, a problem arises if we want to do repetitive operations on large chunks of data. The problem is that of having no 'stopping condition'. Say we want to do the same operation on different data ten times – for that, we need a counter which will tell us when the count of 10 is reached. When this happens, the repetitive action can be stopped. The capability of being able to take decisions based on conditions is what gives any computer its power, and the 8085 is no exception. It has instructions for branching which are as listed below. Branching effectively changes the sequential nature of instruction execution by causing control to move to a new set of instructions written at a new address. Branching can be conditional or unconditional – the conditions used are the settings of the flags, as we will see now.

0.9.3.3 / Branch Instructions

JMP address16	;jump unconditionally to the given address
JZ address16	;jump on zero-jump to the given address if Z = 1
JNZ address16	;jump on non-zero-jump to the given address if Z = 0
JC address16	;jump on carry-jump to the given address if C = 1
JNC address16	;jump on no carry-jump to the given address if C = 0
CALL address16	;call a subprogram written in the address given
RET	;return to the original program after executing the subprogram

For 8085, all addresses are 16-bit in length.

Example 0.25

Now, let us write a program for multiplying 24 by 10. Since there is no multiply instruction for 8085, multiplication is achieved by repeated addition. We do it by adding 24 to itself 10 times.

Solution

```
MOV A,0      ;make the content of A = 0
MOV B,10     ;load the multiplier (10) in B
AGAIN:   ADI A,24    ;add the multiplicand(24) to A
         DCR B      ;decrement B
         JNZ AGAIN   ;check if Z (zero flag) is non - zero - if
                     ;so, jump to AGAIN
         LDA 3400H   ;if Z = 0, store the sum in address 3400H
```

In the Example 0.25, B acts as a counter. Initially, A = 0. Then, 24 is added to it, and each time this is done, the counter B is decremented by one. By the time 10 additions are over, the content of B will be equal to 0. When the decrement operation causes the result to be zero, i.e., B = 0, the Zero flag is set (Z = 1). This is the stopping condition for the addition loop. Once the addition is over, the result is saved in memory. Then, the loop is exited and the content of A is copied to a location in memory.

Note Since A has only a data size of 8 bits, the maximum number it can hold is 255. This should be kept in mind.

Example 0.26

Two numbers are stored in memory. Verify if their sum is greater than 255. If yes, send the ASCII value of Y to an output port with address 78H. Otherwise, send N.

Solution

If the sum is greater than 255 (FFH), a carry will be generated which will set the carry flag. So, the verification consists of testing the carry flag. See the program flow.

```

LXI H,0897H ;load HL with the first address
MOV A,M ;load data from that address to A
MOV B,A ;move that data from A to B
INX H ;increment pointer register pair HL
MOV A,M ;load the second data from memory into A
ADD A,B ;add the content of A and B-sum in A
JC YES ;if C = 1, sum >255; jump to YES
MOV A,'N' ;if C = 0, it means sum <255, so A = 'N'
JMP XIT ;jump unconditionally to XIT
YES: MOV A,'Y' ;this is when sum>255.A = 'Y'
XIT: OUT 78H ;send A to output port with address 78H

```

In Example 0.26, a number of points need elaboration. We are already familiar with getting data from memory using indirect addressing. Once the two 8-bit numbers are brought to registers A and B, they are added. If the sum >FFH, the carry flag gets set, which is tested by the JC instruction. If the condition is true, control branches to the label 'YES'. Here, the ASCII value of Y is copied to A. ASCII is represented by single quotes of the character. When we write 'Y', actually the number corresponding to the ASCII of Y, i.e., 59H, is what is being copied. If the YES branch is not taken, the A register is to be loaded with the ASCII of N. Either way, there is an ASCII character in A. The last step is sending this number to an output port. For that, the output port address is mentioned in the instruction, but the register A is implicit. The output port may be a display device.

0.9.3.4 / Logical and Bit Manipulation Instructions

8085 has a set of logical and bit manipulation instructions as well. The list is given below. Flag bits are affected by these instructions:

- i) ANA R ;logically AND the contents of A and R – result in A
- ii) ANI data8 ;logically AND the contents of A and data given – result in A
- iii) ANA M ;logically AND the contents of A and memory – result in A
- iv) ORA R ;logically OR the contents of A and R – result in A
- v) ORI data8 ;logically OR the contents of A and data given – result in A
- vi) ORA M ;logically OR the contents of A and the memory – result in A
- vii) XRA R ;logically XOR the contents of A with that of R – result in A
- viii) XRI data8 ;logically XOR the contents of A and data given – result in A
- ix) XRA M ;logically XOR the contents of A and the memory – result in A
- x) CMP R ;compare the contents of A and R – only flags affected,
- xi) CMP M ;compare the content of A and memory – only flags affected
- xii) CPI data8 ;compare the contents of A and given data – only flags affected
- xiii) RLC ;rotate the bits in A once, to the left
- xiv) RAL ;rotate the bits in A and the carry bit, to the left, once
- xv) RRC ;rotate the bits in A once, to the right
- xvi) RAR ;rotate the bits in A and the carry bit, to the right, once

Most of the above instructions do not need much explanation – but the ‘compare’ instruction does. The format of this instruction is

CMP destination, source

This instruction compares the two operands and causes the conditional flags to be affected, **but neither the destination nor the source changes.** Comparison is done by a subtraction operation, and the flags are set/reset according to the result of this. However, only two flags really matter – they are the Zero flag and the Carry flag. Consider the instruction CMP destination, source. The condition of the flags will be as shown in Table 0.5. Thus, to test for equality of two data, the zero flag can be used, and for comparing the magnitude of the numbers, the carry flag is tested. We have used the latter condition in Example 0.27.

Table 0.5 | Flag Settings After a Compare Instruction

If	C flag	Z flag
destination > source	0	0
destination < source	1	0
destination = source	0	1

Example 0.27

Compare the number in the memory location 4566H with the content of B. Send the bigger number to the output port with address 67H.

Solution

```
LDA 4566H ;load into A, the number in memory
CMP A,B ;compare it to the number in B
JNC YES ;If C = 0, it means that A>B, jump to YES
MOV A,B ;C = 1, means B>A, so copy bigger no to A
YES: OUT 67H ;send the bigger number to the output port
```

With this, we come to an end of our discussion on 8085 programming. There are a few more instructions, but since only a brief introduction to 8085 is possible here, we will straightway go into the hardware aspects of 8085.

0.9.4 / Hardware Aspects of the 8085

Figure 0.13 shows the pin diagram of the processor. Let us discuss the pins in detail. As we do that, the various features of the processor will become clear to us.

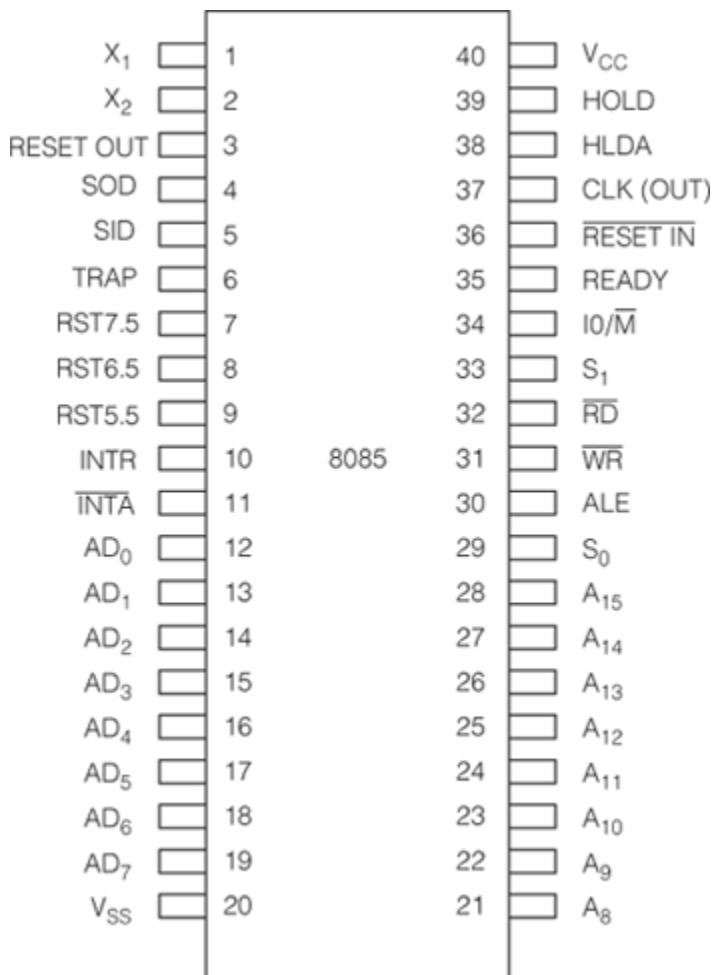


Figure 0.13 | Pin Diagram of the 8085

0.9.4.1 / Multiplexed Address/Data Bus

The 8085 has 16 address lines and 8 data lines, but if we look at the pin diagram, we will not find $16 + 8 = 24$ pins dedicated for address and data. The reason is that the address and data pins are multiplexed. The lower address lines carry data also. See the pins titled AD₀ to AD₇. They are multiplexed pins. They cater to address as well as data but not at the same time. The bus cycle for accessing memory or I/O is arranged in such a way that first the address appears on the address/data pins AD₀ to AD₇ (lower byte of address) as well as on A₈ to A₁₅. Along with this event, a signal named ALE (Address Latch Enable) will go high. An 8-bit latch is to be connected (externally) to the higher order address lines and the ALE is to act as a clock for the latch. So, when ALE goes high, the higher order address information appears at the output of the latch and remains there. Figure

0.14 makes this clear. After this, the address information can be removed from the AD_0 to AD_7 pins, and then they function as data pins named D_0 to D_7 . Thus, using the latch 74LS373, address de-multiplexing has been achieved. However, why was multiplexing done in the first place? In the early days of microprocessor development, packing problems required pin counts to be as small as possible.

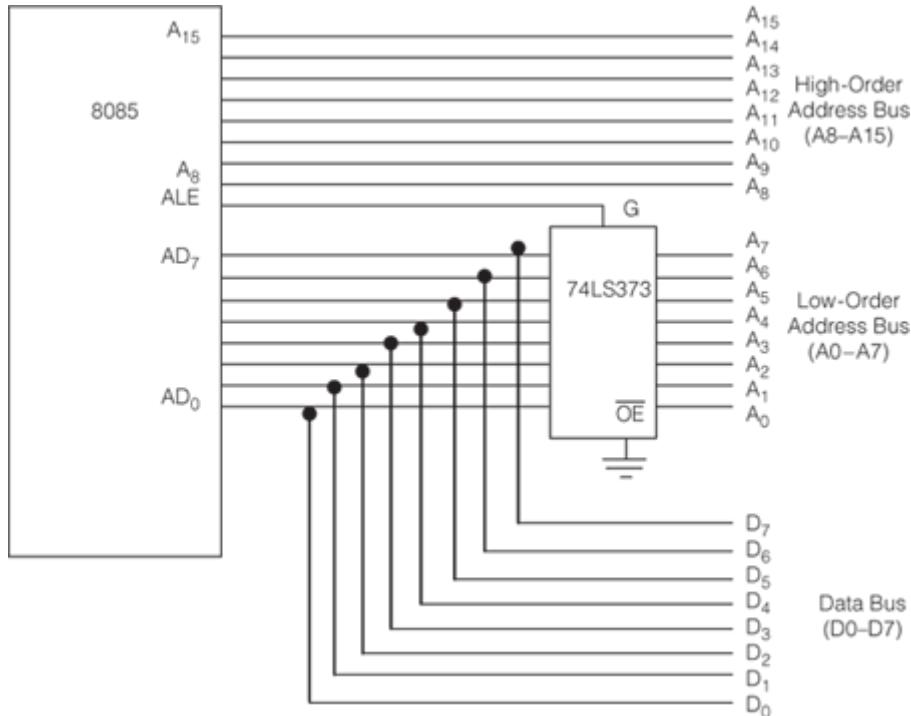


Figure 0.14 | Address and data de-multiplexing

0.9.4.2 / Read and Write

Refer to the pin diagram in Fig. 0.13 again. Pin nos. 31 and 32 are *WR* and *RD*. They are active low signals. They are the control signals for reading from/to memory/I/O. Recollect the way these signals are used - (Section 0.2.5). However, these control signals are used irrespective of whether reading / writing pertains to memory or I/O. The processor has another pin to differentiate between these two cases. See Pin no. 34. It has the designation *IO/M*. This is a control signal sent from the processor and goes high for I/O access and low for memory. External logic is used as shown in Fig. 0.15 to get the four separate control signals – *MEMRD*, *MEMWR*, *IOWR* and *IORD*. Figure 0.16 shows the processor with the separate address bus, data bus and read/write control signals.

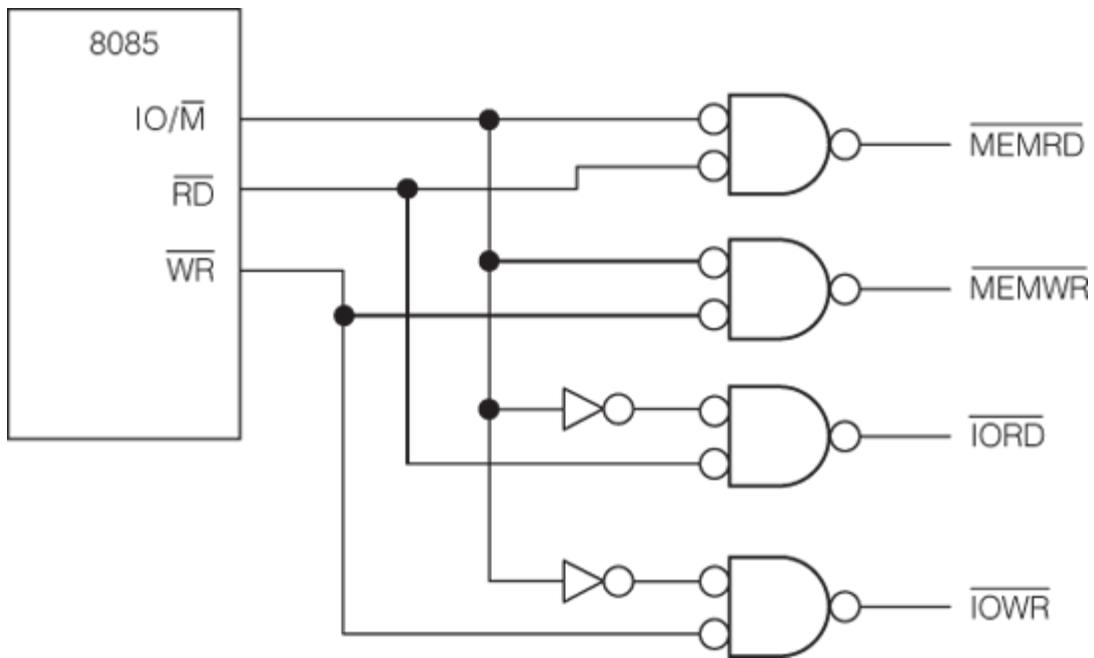


Figure 0.15 | Generating separate control signals for memory and I/O

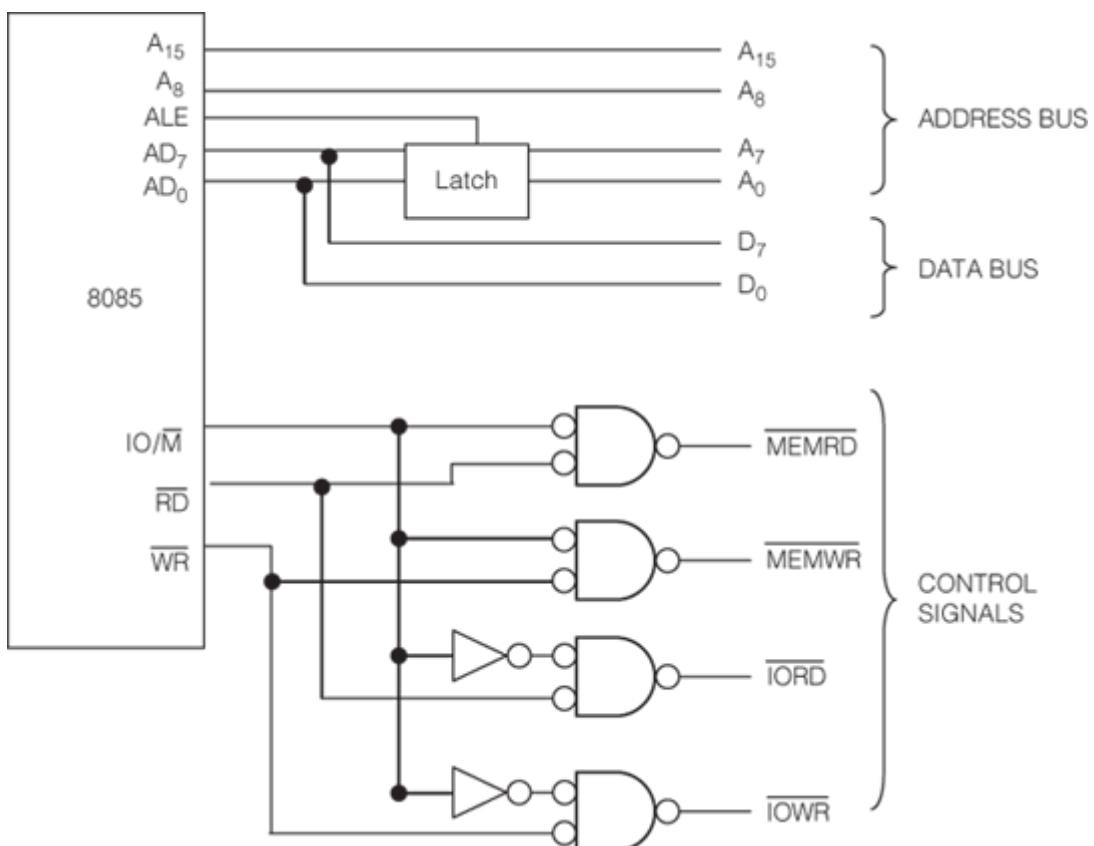


Figure 0.16 | External hardware used for getting separate address and data bus and control signals

0.9.4.3 / Power Supply and Clock

Now, refer Fig. 0.17, which shows the functional block diagram of the processor, power and clock pins. Pins 1 and 2 are the pins between which a

crystal is to be connected. There is an internal crystal oscillator for the chip, and the only thing that we do is to connect a crystal externally. This defines the clock frequency for the processor. Internally, the clock frequency gets divided by 2. So, if we need an internal clock of 4 MHz, an 8 MHz crystal is to be connected. The maximum allowable frequency of operation for the 8085 is 5MHz. For that, a crystal of 10 MHz is to be connected between X₁ and X₂. Pin No. 37 is CLK OUT. From this pin, the clock of the processor can be extracted and used for timing any unit in the system which is required to be synchronized to the processor. The clock can also be divided to get lower frequencies which may be used for devices like an ADC (analog to digital converter), which may be part of the 8085 based system.

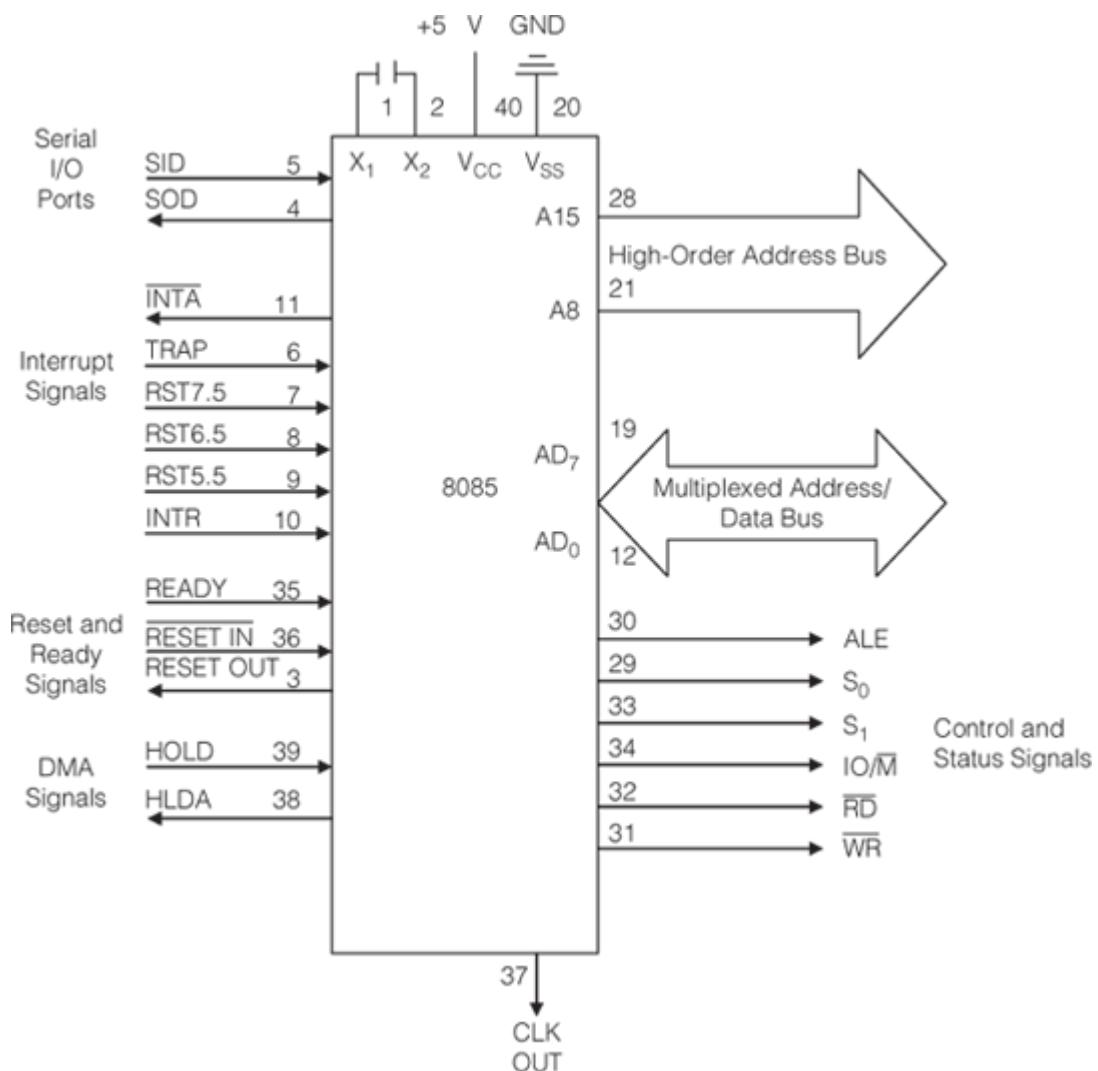


Figure 0.17 | Functional block diagram of 8085

0.9.4.4 / Reset and Ready Pins

There are two pins pertaining to reset. RESTIN is an input signal to the processor. When this is pulled low, the processor resets and the program

counter is cleared. Program execution starts at the address 0000, which is the address corresponding to the value of the PC when the system is reset. While this is happening, RESET OUT can be used to reset other devices in the system concurrently.

Ready

This is a signal sent to the processor by a peripheral. Usually, peripherals are slower than processors. The time for accessing any peripheral or memory location is specified in terms of a number of clock cycles. For slower peripherals, more clock cycles will be needed. When that is the case, the processor looks at the READY pin that is controlled by the peripheral. As long as the READY pin is low, the access cycle does not end, and thus, more time is obtained for reading or writing. It must be arranged that the READY pin goes high once the required delay is obtained.

0.9.4.5 / Interrupt

An important feature of any processor is the capability to be interrupted. However, before discussing the interrupt pins, let us try to understand what is meant by an interrupt.

Say a processor is performing a particular task, executing instructions corresponding to that task. If another more important task is required to be done, the current task will have to be stopped temporarily and the other more urgent one must be taken up. Once the urgent task is done with and finished, the old one can be resumed. This is the scenario regarding interrupts. This means that the processor will have to be ‘interrupted’. Who gives the interrupt request? It could be a peripheral requesting urgent service. In this case, the peripheral places its interrupt request on a hardware pin of the 8085. Another case is when the processor is interrupted by software. Here, an instruction is written to interrupt the execution of the current task.

In either case, the sequence of events that follow the interrupt are the same. The processor completes the current instruction, saves the program counter and processor status and takes up the new task by going to a new location in memory. This new memory location is called the ‘interrupt vector’ corresponding to the specific interrupt. What is there at the interrupt vector? It is at this address that the Interrupt Service Routine (ISR) is stored. The ISR is the program that the interrupting device wants to get executed. Thus, for each interrupting device, there should be an ISR at an ‘interrupt vector’.

Interrupt Pins of the 8085

This processor has five hardware interrupt pins on which peripherals can place their interrupt requests. The first one is the INTR (Interrupt) pin. A peripheral should place a high on this pin for a minimum specified time, which causes the processor to get interrupted. The processor responds by lowering the INTA line, which is an ‘Interrupt Acknowledge’ signal. The peripheral on getting this tells the processor where the ISR for this task is placed – we say that the peripheral gives the ‘interrupt vector’ for its task. The processor goes to this address, executes that task, and after that goes back to the task that had been interrupted.

This is the case for an interrupt on the INTR pin. How about the other interrupt pins? They are TRAP, RST 7.5, RST 6.5 and RST 5.5. The specialty of these pins is they are ‘vectored’ interrupt pins. This means that when a peripheral places its request on any of these pins, the peripheral does not have to give the address of the ISR. For each of these interrupts, there is a specific pre-defined address to which control automatically goes – the hardware of the processor takes care of this. The interrupt vectors for these pins are as in Table 0.6. There is also another issue regarding interrupts. Suppose a number of peripherals interrupt simultaneously on the many interrupt pins of the 8085, obviously, only one of the requests can be serviced. Which one? There is a pre-defined priority for the interrupt pins – TRAP is the highest priority pin followed by RST 7.5, 6.5, 5.5 in that order and last, the INTR pin.

Table 0.6 | Interrupts Pins and Their Vectors

Interrupt pins	Interrupt vector
TRAP	0024H
RST7.5	003CH
RST6.5	0034H
RST5.5	002CH

0.9.4.6 / Hold and Hlda

See pins 38 and 39; they correspond to the HOLD and HLDA (hold acknowledge) pins. These are the pins that co-ordinate DMA activities of the processor. Now, what is DMA? Normally, a data transfer between memory and an I/O device always takes place with the processor acting as an intermediary. If data from memory is to be sent to an output device, it first has to be sent to the processor’s accumulator, and from there, it is

transferred to the peripheral. This is okay if the data involved is small. However, for bulk data transfer, this becomes cumbersome. So, the processor allows data to be transferred directly between a peripheral and memory while the processor stands aside after just initiating the process.

If, say, a peripheral wants DMA service, it sends a request to the processor. The DMA request is placed on the HOLD pin. Since DMA is a high priority issue, the processor stops whatever it is doing and sends the acknowledge signal HLDA to the peripheral. With this, the processor's buses are tri stated such that the processor effectively gets cut off from the system. Thus, a direct path of data transfer is established between the peripheral and memory. Once this data transfer is over, the HOLD request is removed by the peripheral and the processor gets back its status as the master of the system.

0.9.4.7 / SID and SOD

These are two pins that most other microprocessors do not have. They are the Serial Input Data (SID) and Serial Output Data (SOD) pins. Using SID, data can be sent in serially, converted to parallel form (internally) and saved in a register. Similarly, parallel data can be converted to serial form and sent out as one bit at a time, using the SOD. Of course, there are special instructions to aid in the serial transmission process.

Conclusion

We have discussed all pins of the 8085 processor, except the status pins S_0 and S_1 . These are pins which carry some status information, which is not very important for small systems. With this, we conclude our review of the 8085 and will go on to the task of understanding the x86 microprocessors, starting from the 8086. That will be from Chapter 1 onwards.

KEY POINTS OF THIS CHAPTER

- The first PC which made a great impact in the world of computing was the IBM PC with Intel's 8088 microprocessor being its CPU. This PC was launched in August 1981.
- Gradually, the x86 family of microprocessors became the *de facto* standard for PCs the world over.
- A computer system consists of a CPU, memory and I/O, which communicate with one another through the system bus.
- The system bus comprises the data bus, address bus and the control bus.
- A processor's activities are restricted to fetching, decoding and executing instructions.

- For reading and writing from/to memory, a number of clock cycles of time are required. The time expended for this is called the memory access time.
- When comparing assembly language programming with high level language programming, we conclude that the former is faster in execution and more efficient and compact but is more difficult to learn and master.
- RISC and CISC are two different philosophies in computer design, and even though a lot of controversy still rages around which is better, the two seem to have merged more or less.
- Computers do all the computations in binary, but for entering data through the keyboard and for displaying it on the monitor, ASCII codes are used.
- Negative numbers are represented in two's complement form by all computers.
- The 8085 microprocessor is still a very popular chip even though it is not used in PCs.
- The 8085 has 16 address lines and 8 data lines.
- The 8085 is one of the processors of Intel which has the clock generator inside the chip.

QUESTIONS

1. Which is the first noteworthy microprocessor developed by Intel?
2. Name two 8-bit microprocessors of Intel which are not part of the x86 family.
3. Which microprocessor was used in the first IBM PC?
4. What is IBM's contribution to the PC revolution?
5. What is meant by the term 'x86 family' of microprocessors?
6. Have you heard of the term 'bus contention'? What does it mean in the context of a computer system?
7. If the data bus width of a processor is 64 bits, what would you say about its complexity and capability?
8. If the address bus of a processor is 64 bits, what is its address space?
9. What could be a 'multi processing' system?
10. What is the first step in the execution cycle of a processor?
11. How does the system clock frequency influence the speed of processing?
12. If a system uses a 1.5-GHz clock, what is its clock period?
13. What is meant by the word 'system bus'?
14. Why should a computer have an I/O controller?
15. What are the difficulties involved in learning and using assembly language programming?
16. Name one distinguishing feature each of RISC and CISC computers.
17. How are the hexadecimal and binary number systems related?
18. When two signed positive numbers are added and the sum exceeds 127, what is the problem that arises?
19. What is the range of signed numbers that can be represented in 12 bits?
20. Why is the 8085 called an 'accumulator based' processor?

21. Distinguish between the LDA and LDAX instruction.
22. List the interrupt lines of 8085.
23. What is meant by DMA?

EXERCISES

1. Write briefly the history of the x86 family of microprocessors.
 2. Explain RISC and CISC processors distinguishing between the two, elaborating the advantages and disadvantages of each type. Name a few processors of each type.
 3. Find out about the 'Apple Mac' – its history, its features and why and how it is considered a 'great' PC.
 4. Write the decimal equivalent of the following numbers:
 1. 31.3H
 2. 1100.101B
 3. A32.3H
 4. 100101B
 5. Convert the following numbers to binary form:
0. 34
1. 200
 2. 90
6. Convert to hexadecimal format.
0. 3454
1. 4523
 2. 789
7. Write the binary values for:
0. 34ADH
1. 78FH
 2. 407BH
8. Write the hexadecimal values of:
0. 11000101010110001B
1. 1001111100001010B
9. Find the packed BCD representation of the following decimal numbers:
0. 45
1. 4678
 2. 802345
10. Represent the packed BCD of the following numbers in hex:
0. 235
1. 9123
11. What is the ASCII of each of the following?
0. 7
1. 8
 2. 0

- 3. A
- 4. Z
- 5. y
- 6. d
- 7. *
- 8. &

12. Find the two's complement representation of the following numbers in 8 bits:

- 0. -45
 - 1. -90
 - 2. -12
 - 3. -34

13. Represent the following negative numbers using 16 bits:

- 0. -267
 - 1. -4
 - 2. -5676
 - 3. -675

14. Perform binary addition for the following numbers:

- 0. 34 and 56
 - 1. -52 and -70
 - 2. -47 and +120

15. Convert to packed BCD and add:

- 0. 46 and 23
 - 1. 55 and 67
 - 2. 34 and 49
 - 3. 99 and 44

16. Subtract after converting to binary form:

- 0. -20 from -75
 - 1. +49 from +97
 - 2. E5H from A4H

17. Add the following signed numbers:

- 0. F3H and 3245H
 - 1. AH and F45H
 - 2. B2H and 123EH

18. How many bytes constitute the following?

- 0. 5 MB
 - 1. 4 KB
 - 2. 32 MB
 - 3. 32 KB
 - 4. 8 GB

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

- [Recommended](#)
- [Playlists](#)
- [History](#)
- [Topics](#)
- [Tutorials](#)
- [Settings](#)
- [Support](#)
- [Get the App](#)
- [Sign Out](#)

© 2018 [Safari](#). [Terms of Service](#) / [Privacy Policy](#)

1

The Architecture of 8086

IN THIS CHAPTER, YOU WILL LEARN

- The internal architecture of the 8086.
- The registers of 8086 and their specific functions.
- The operation of the flags of 8086.
- The memory segmentation technique used by the x86 family.
- How to perform memory address calculations.
- The different addressing modes of 8086.

Introduction

We will start our learning of the x86 family of microprocessors by understanding the architecture of the 8086 on which is based the architecture of the whole family. The 8086 is a processor catering to 16-bit data (D0 to D15) and having a 20-bit address (A0 to A19).

1.1 | Internal Block Diagram of the 8086

The 8086 processor is a 16-bit processor internally as well as externally—which means that its data bus width as well as the bit size of its internal data registers is 16. Thus, it can access an external source 16 bits at a time through its data bus. However, it also has the capability to access and work on 8-bit (byte) data.

As the first step in understanding the architecture of the 8086, let us examine the internal block diagram of the 8086. Figure 1.1a shows the simplified block diagram and Fig. 1.1b shows it in more detail. Let us examine the simplified diagram first. Though not specifically shown in the block diagram, it is to be noted that all actions in the processor are synchronized by a system clock, which provides the basic timing. There is a control unit which provides the control signals for the overall functioning of the processor.

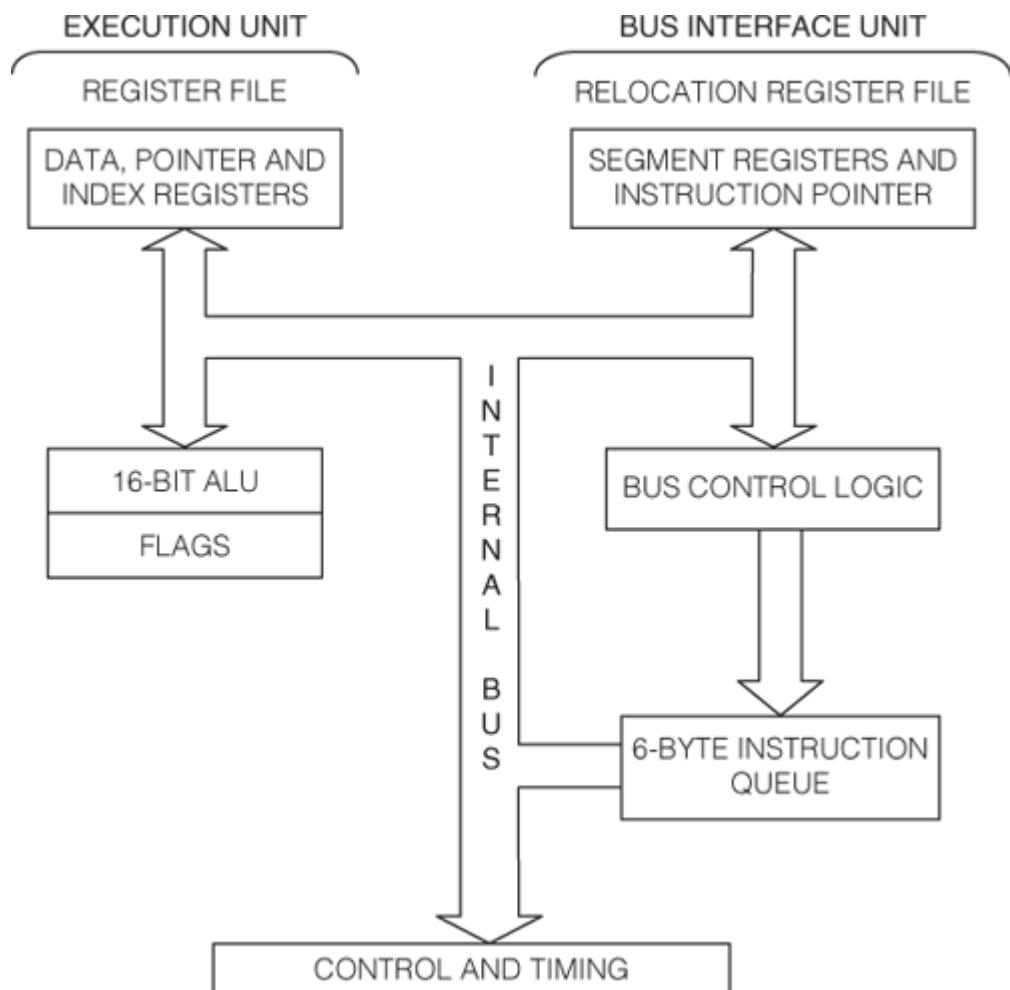


Figure 1.1a | Internal block diagram of the 8086 (simplified model)

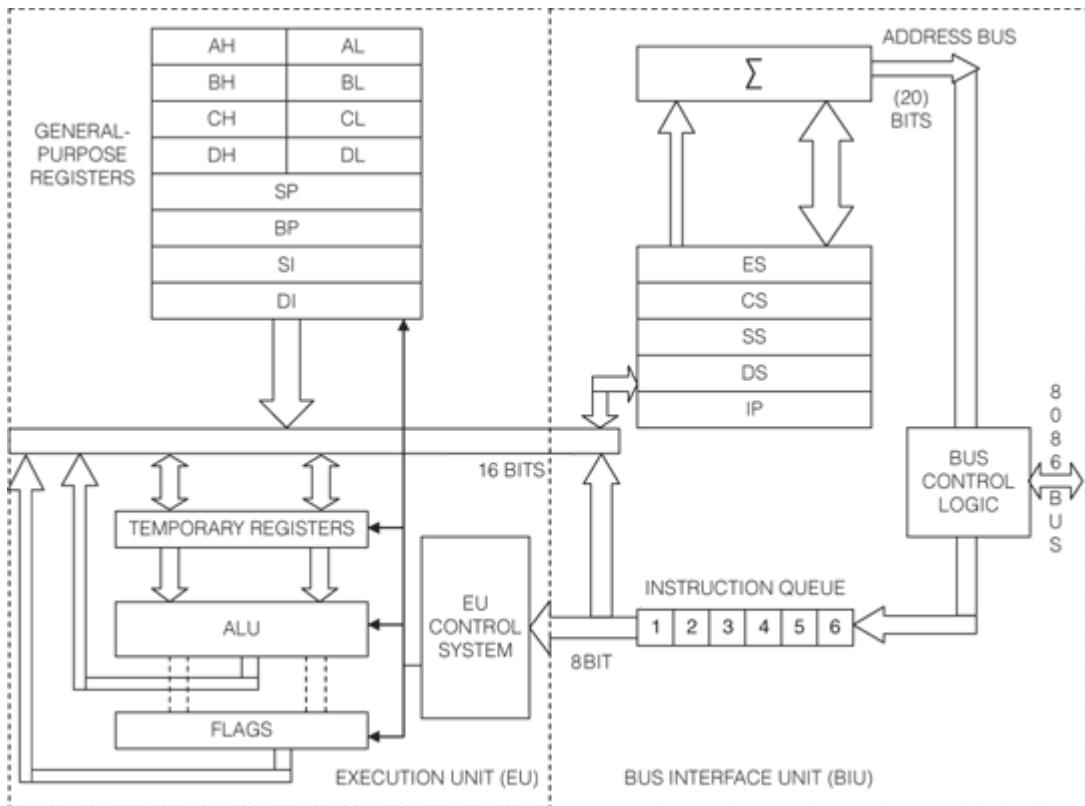


Figure 1.1b | Internal block diagram of the 8086 (detailed)

We see that the internal block diagram has been partitioned into two logical units, namely, the Bus Interface Unit (BIU) and the Execution Unit (EU). They interact directly with each other through the *internal bus* but perform separately as two units with well-defined functions for each unit. Let us explore the internal configuration of these units one by one.

1.2 | Execution Unit

It contains the arithmetic and logic unit, the control unit, an internal bus, plus a few registers. Let us get a bird's eye view of the EU starting with the register set.

Why Registers?

All processors have internal registers. An internal register is actually an on-chip RAM. Some are program visible, while others are not directly visible to programmers. The more the number of program visible registers, the better it is for the programmer. This is because internal registers are used as temporary storage for operands, and if the operand is already in a register, it takes less time for execution of the associated

instruction. Such registers are called general-purpose or scratchpad registers.

However, registers constitute on-chip RAM, which is costly. Also, more the number of registers, more are the number of bits required to identify and address a particular register. This can cause an increase in the length of *opcodes* in modes involving registers. Thus, the number of registers a processor can have is a trade-off between a number of conflicting factors.

Register Set of 8086

Let us see how the register set of 8086 has been organized. The 8086 has data registers, address registers, segment registers and also a flag register. We will first examine the registers in the Execution Unit.

1.2.1 | Scratchpad Registers

They are called so because they are used for temporary storage, just as we jot down a few things temporarily in a scratchpad. As shown in Fig. 1.2, the 8086 has four 16-bit general-purpose registers labeled as AX, BX, CX and DX. Each of these registers can also be used as two separate and exclusive 8-bit registers also, i.e., AX has two parts AH and AL, where H and L stands for the high and low portions, respectively. When used as 8-bit registers, AH and AL are separate, but they can be concatenated to form the 16-bit register AX. Similar is the case for the registers BX, CX and DX. Each register, while being *general-purpose*, has specialized uses as well. As such, let us understand these features clearly.

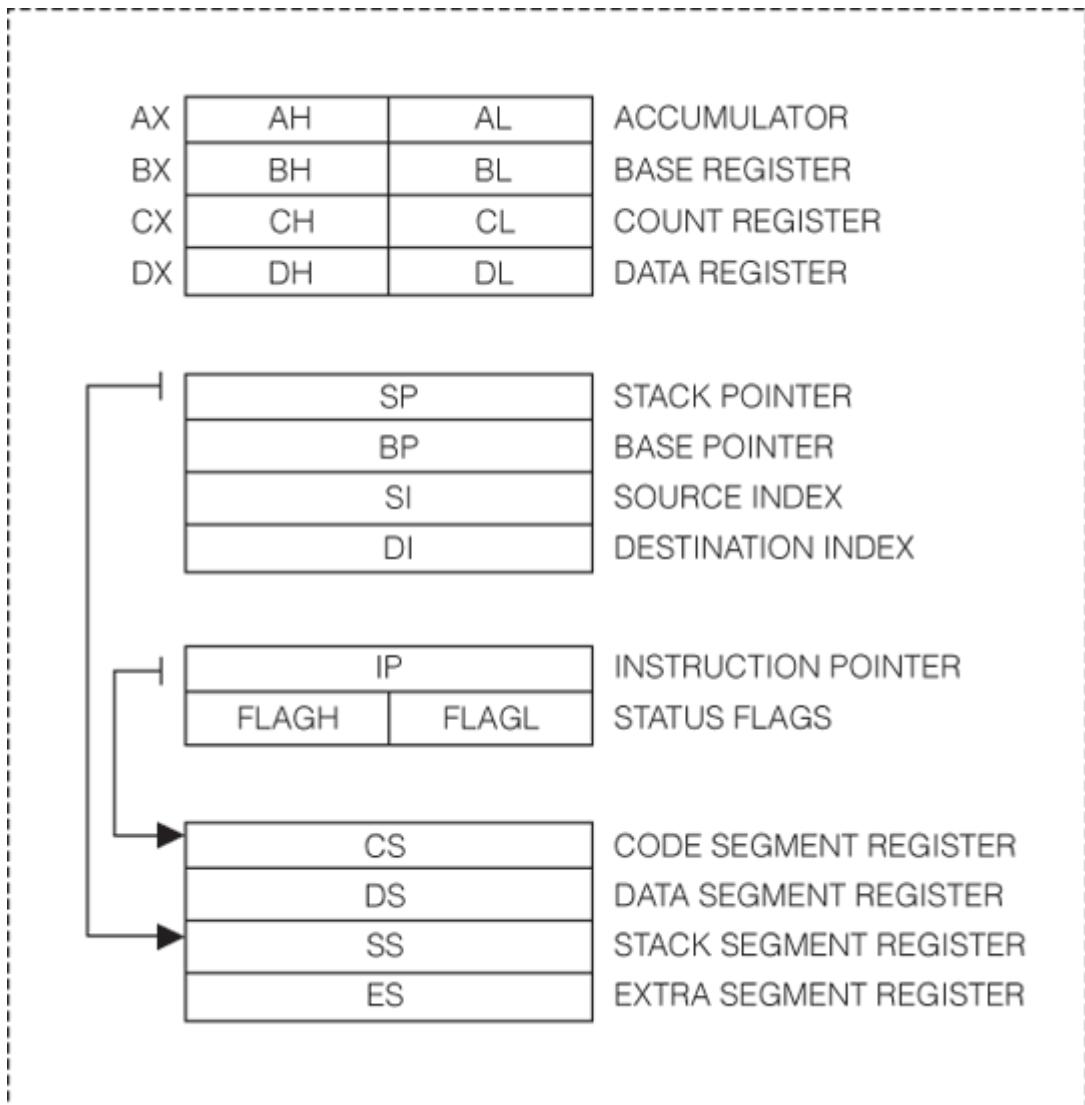


Figure 1.2 | Register model of the 8086

AX

This is a 16-bit data register, but it can also be used as two unrelated 8-bit registers AH and AL. AL /AX is sometimes called the *accumulator*, but the relevance of the accumulator is less for 8086 compared to the earlier 8085 in which one operand is implied to be in the A register for many instructions. However, for the 8086, only for certain specialized instructions is the AL or AX register *implied* to contain one of the operands. In most other instructions, the source and destination must be specifically mentioned in the instruction. (The destination is the register or memory location in which the result of any operation is to be put.)

BX, CX and DX

These are the other working registers of the 8086, which means that temporary data storage, arithmetic calculations and data manipulation can be done with these registers. They also can be used as 8-bit or 16-bit registers as mentioned earlier. Still, each of them has special functions.

Base register BX is frequently used as an address register in many based addressing modes. *Counting register* CX is used as a counter in many instructions. *Data register* DX is used in I/O instructions as a pointer to data by storing the address of the I/O port. Also, multiply and divide operations involving large numbers assume DX and AX as a pair.

Pointer and Index Registers

SP, BP, SI and DI are address registers and can be used only as 16-bit registers. They are very important in programming, as they facilitate the use of various addressing modes.

BP and SP

They are the Base pointer and Stack pointer, respectively. SP always points to the top of the stack, while BP can point to any location in the stack. (We will discuss the concept of the stack shortly.) Thus normally, whenever we speak about data referred to by BP and SP, it is implied that we are talking about data in the *stack*.

SI and DI

These are Index registers labeled as Source Index and Destination Index, respectively. They function as address registers in various addressing modes, but they have a specialized function as being the default registers in *string instructions*.

1.2.2 | Flag Register

It is a 16-bit register (Fig. 1.3), of which 7 bits are unused. Out of the remaining bits, 6 bits are used as conditional flags. The others are control flags. A conditional flag is a single-bit flip-flop which is set or reset according to the result of an arithmetic or logic operation.

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

U = unused

CONDITIONAL FLAGS

OF = Overflow flag
SF = Sign Flag
ZF = Zero Flag
AF = Auxiliary Carry Flag
PF = Parity Flag
CF = Carry Flag

CONTROL FLAGS

DF = Direction Flag
IF = Interrupt Flag
TF = Trap Flag

Figure 1.3 | 8086 Flag register

The conditional flags available are the Carry (CF), Zero (ZF), Parity (PF), Overflow (OF) and Sign Flag (SF). Since conditional flags are an important ingredient in computation using assembly language, we will not defer our discussion on them.

Carry flag (CF)

The carry flag (CF) gets set if there is a carry out from the most significant bit (MSB) during a calculation. For example, when 8-bit addition causes the result to be greater than 8 bits, there is a carry out from the MSB (D7), which causes this flag to be set. For 16-bit operations, the carry will be from bit D15, and CF will be set.

Zero Flag (ZF)

When the result of an arithmetic or logic operation is zero, the zero flag gets set. For example, if we keep on decrementing the contents of a register (say the CX register, which is used as a count), it will finally become zero. At this instant, the zero flag gets set, i.e., ZF = 1. Another case is when two numbers are compared. Comparison is achieved by subtraction. If the numbers compared are equal, the zero flag is set (ZF = 1), indicating equality of the operands.

Parity Flag (PF)

The setting of this flag indicates the presence of an even number of '1' bits in the least significant byte (lower 8 bits) of the destination. For example, after a particular arithmetic or logic operation, if the destination contains the number 11100111, the parity flag is set to indicate even parity. Even if the operation is on 16-bit data, only the

parity of the lower 8 bits is verified. This flag does not find much applications in current programming practices.

Sign Flag (SF)

After an arithmetic or logic operation, if the result contains a negative number, the sign flag is set. In essence, it contains the MSB of the result, which can be interpreted as the sign bit in signed arithmetic operations.

Auxiliary Carry Flag (AF)

This flag functions similar to the Carry flag, except that the overflow is from bit D3 into D4. It thus indicates a carry out from the lower 4 bits. The need for this flag is for the Decimal Adjust instruction, which is important in BCD number calculations. There are no other instructions that directly test the state of this flag and no conditional branching is associated with this flag.

Overflow Flag (OF)

This flag is set under one of the following conditions:

1. there is an overflow into the MSB (8th or 16th bit) from the bit of lower significance, but no carry out from the MSB,
2. there is a carry out from the MSB, but no carry into the MSB.

This flag indicates that the result of a *signed* number operation is too large, causing the higher order bit to overflow into the sign bit, thus changing the sign bit.

Note: To understand the use of flags, we will use a few simple instructions with adequate explanation. But the instruction set will be dealt in detail only from Chapter 2 onwards.

Example 1.1

Find the status of the flags CF, SF and AF after the following instructions are executed.

MOV AL, 35H
ADD AL, 0CEH

Solution

The format of a MOV instruction is MOV destination, source.

In the above two instructions, a number 35H is first moved to AL. In the next line, oCEH is added to AL. The sum will be in AL, the destination register.

$$\begin{array}{r}
 35H \quad 0011\ 0101 \quad +
 \\ + \quad \underline{\text{CEH}} \quad \underline{1100\ 1110} \\
 \underline{103H} \quad \underline{1\ 0000\ 0011}
 \end{array}$$

After computation, AL will contain 0000 0011 and

CF = 1 since there is a carry out from D7.

SF = 0 since the sign bit (MSB) of the 8-bit destination is 0.

AF = 1 since there is an overflow from D3 to D4.

Example 1.2

Show the effect of the following instructions on the CF, ZF and OF bits of the flag register.

MOV BX, 45ECH

ADD BX, 7723H

Solution

This is the case of 16-bit addition

$$\begin{array}{r}
 45ECH \quad 0100\ 0101\ 1110\ 1100 \\
 + \quad \underline{7723H} \quad \underline{0111\ 0111\ 0010\ 0011} \\
 \underline{\text{BD0FH}} \quad \underline{1011\ 1101\ 0000\ 1111}
 \end{array}$$

The sum will be in BX, which is the destination register.

CF = 0 since there is no carry out from D15.

ZF = 0 since the destination is not zero.

OF = 1 since there is an overflow into the MSB D15.

Example 1.3

Assuming we are adding *signed* 8-bit numbers, how is the result of the following addition to be interpreted?

```
MOV AL, 125  
ADD AL, 75
```

Solution

Both operands here are decimal numbers.

This program causes AL to be loaded with +125 and then +75 to be added to AL. The result is to be in AL.

Signed numbers consider the MSB as the sign bit. With 8 bits, the maximum range of signed numbers is -128 to +127.

When adding the given signed numbers,

$$\begin{array}{r} + \quad 125 \\ + \quad 75 \\ + \quad \hline 200 \end{array} \qquad \begin{array}{r} 0111 \ 1101 \\ 0100 \ 1011 \\ \hline 1100 \ 1000 \end{array}$$

There has been an overflow from D6 to D7 (i.e., to the MSB). In this case, we find OF = 1. The sum is interpreted as a negative number with value of the sign bit = 1.

The setting of the OF flag indicates that the magnitude of the result has caused an inversion of the sign bit, indicating a mistake. This interpretation is done, because the maximum range of *signed positive* numbers is only +127 for 8-bit representation.

Incidentally, the Sign Flag is also set (SF = 1).

Control Flags

Besides the conditional flags, there are three control flags, namely, the Trap flag, Direction flag and Interrupt flag. The control flags have to be deliberately set or reset according to the requirements of the program. The Trap flag (TF) is set to perform step-by-step execution during debugging. The Interrupt flag (IF) is set to enable interrupts. The Direction flag (DF) is used in string operations. Details of the functioning of these flags will be discussed in later chapters.

Arithmetic Logic Unit

It is the part of a computer that performs all arithmetic and logic computations. The ALU is the most important unit of the processor.

Instructions that are fetched and decoded are executed in the ALU. Thus, the ALU has direct access to the general-purpose registers and flags.

1.3 | Bus Interface Unit

This unit is responsible for address calculations, pre-fetching instructions for the queue and sequencing instructions one by one. Let us examine each function.

1.3.1 | The Instruction Queue

Instructions are found in memory, from where they are fetched and decoded as and when they need to be executed. However, in 8086, there is a queue which fetches instructions ahead of the execution time and places them in a 6-byte first-in-first-out (FIFO) queue. This pre-fetching is done when the buses are free, i.e., not being used for the execution of the current instruction.

The advantage of pre-fetching is that when a particular instruction is to be executed, there is a good chance of finding it in the queue (which is on-chip), rather than having to go to memory to fetch it. However, the queue will have to be emptied in the event of a branch instruction, i.e., when the next instruction to be executed is not the next one in the sequence. After this, the queue needs to be re-loaded from the new ‘branch’ address. This pre-fetching belongs to a class of ideas called pipelining, which means that both execution and fetching take place at the same time, i.e., while the execution of one instruction is going on, the fetching of another one can be done. Pipelining greatly speeds up processing.

1.3.2 | Memory Segmentation

The 8086 has a 20-bit address bus, but let us remember that we have not seen any 20-bit registers so far. It is understandable that the general-purpose registers are used as 16-bit or 8-bit registers, as this is the way data is organized. However, we see that all address registers also are 16-bit. An ingenuous idea called segmentation has been used to take care of this situation. A segment is just an area in memory. Thus, the total memory size can be considered to be divided into segments of various sizes. The idea of dividing memory this way is called segmentation.

In memory, data is stored as bytes. Each byte has a specific address. With 20 address lines, the memory that can be addressed is 2^{20} bytes. This will amount to 1,048,576 bytes (1 Megabyte). Thus, the 8086 can

access a physical memory with addresses ranging from 00000 to FFFFFH. A 20-bit physical address is to be placed on the address bus to access any location in memory.

Memory as a whole is considered to have four different *types of segments* labeled as data, code, stack and extra. One idea of segmentation is to keep data and code separately. Thus, the code segment contains code only. Data is stored in the data segment, and the extra segment is just another type of data segment used in a special way. The stack segment is an area of memory which functions and is accessed differently from the other three segments.

1.3.2.1 | Segment Registers

Each of these segments is addressed by an address stored in corresponding segment registers. These registers are all 16-bit in size. Each register stores the base address of the corresponding segment. A base address is the starting address of a segment. However, since the segment registers cannot store 20 bits (of the base address), they store only the upper 16 bits. The least significant nibble is implied to be zero.

How is a 20-bit physical address obtained if there are only 16-bit registers? The 20-bit address of a byte is called its *physical address*. However, it is specified as a *logical address* which is in the form of two 16-bit numbers in the format *base address: offset*. See Fig. 1.4, for a typical example.

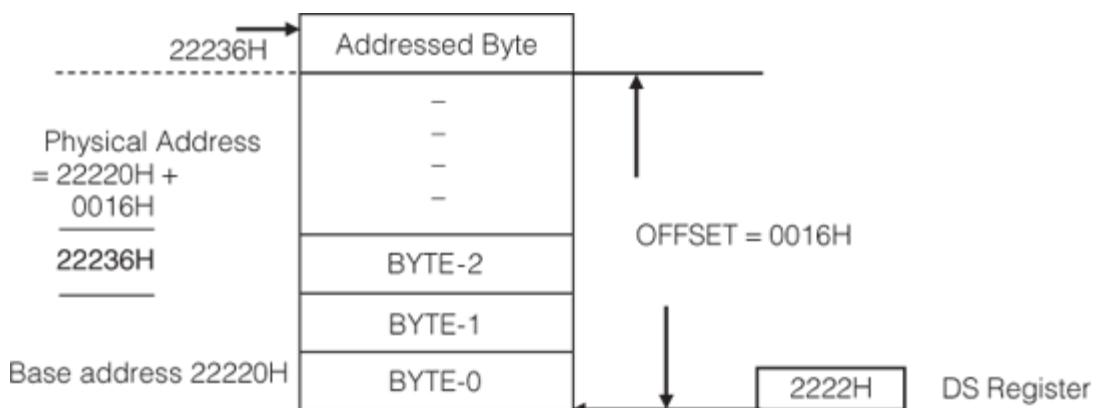


Figure 1.4 | Calculation of a physical address from the logical address for a data segment

We consider that a data byte is stored in a data segment, whose base address is 22220H. Then the data segment register (DS) will contain the number 2222H. The data at any location within this segment is referenced by an offset (displacement) with respect to the base address.

Thus, if a data at a location has a logical address specified as 2222H : 0016H, the number 0016H is the offset or displacement with respect to the base address. For physical address calculation, the BIU appends the segment register content with four binary zeros (or one hex zero) on the right. The physical address is thus calculated as

$$\begin{array}{r} 22220H \\ + \quad 0016H \\ \hline \underline{\underline{22236H}} \end{array}$$

Similar is the case for the other segments. All offsets are limited to 16 bits, which means that the maximum size possible for a segment is only 2^{16} , which is 65,536 bytes or 64K bytes. We can conclude that every memory reference by the 8086 will use one of the segment registers (i.e., DS, ES, SS, or CS) that is combined with an offset (usually given in the instruction) to determine the physical address being accessed. Figure 1.5 highlights this. A data byte has the logical address of the form DS: offset.

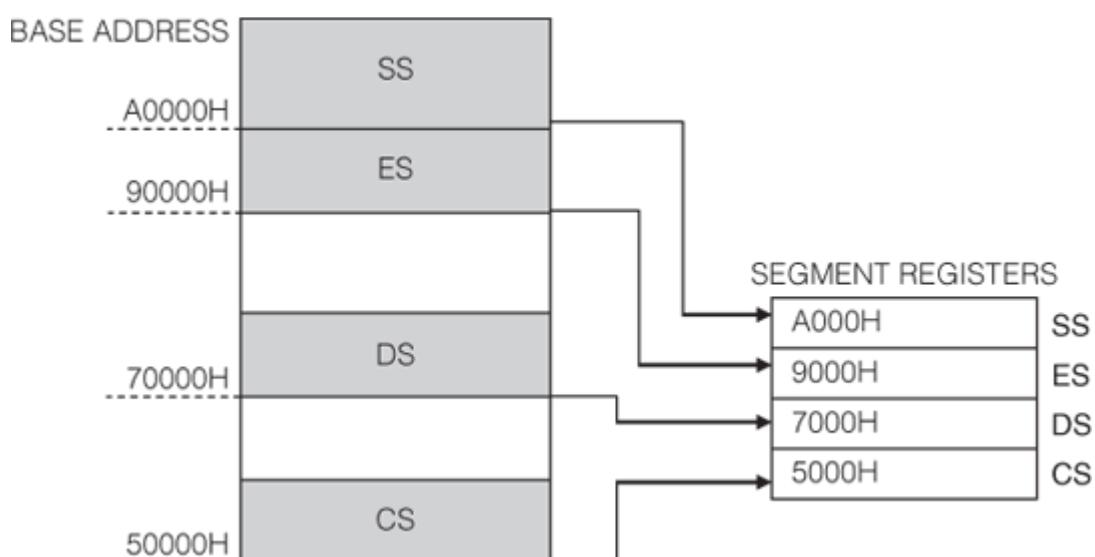


Figure 1.5 | Segment registers and corresponding segments

1.3.2.2 | The Code Segment and the Instruction Pointer

The code segment is the area of memory where code alone is stored. The offsets within the code segment are referenced using the Instruction Pointer (IP), which is a 16-bit register. The IP sequences the instructions and always points to the next instruction to be executed. Whenever an instruction byte has to be fetched from memory, the bus interface unit (BIU) performs the address calculation using the contents of CS register and the IP. This 20-bit address is then placed on the address bus and the instruction byte is fetched. Thus, the logical address for an instruction byte is of the form CS : IP.

1.3.2.3 | The Stack Segment and the Stack Pointer

The stack is an area of memory that is used in a special way. Data is generally pushed in and popped out of the stack. The stack of the 8086 is a last-in-first-out (LIFO) stack, which means that the last data that was pushed in is the first one that can be popped out. This also means that data can only be taken out or put in only from the top of the stack. However, the ‘stack top’ changes as data is put in or taken out.

There is a 16-bit register called Stack Pointer, which points to the top of the stack. The stack segment is like any other segment, and the upper 16 bits of its base address is available in the SS register. A stack address of the form 4466H : 0122H means that the SS register contains 4466H, the stack pointer (SP) contains the number 0122H, and the physical address of the top of the stack is $44660H + 0122H = 44782H$. This is shown in Fig. 1.6.

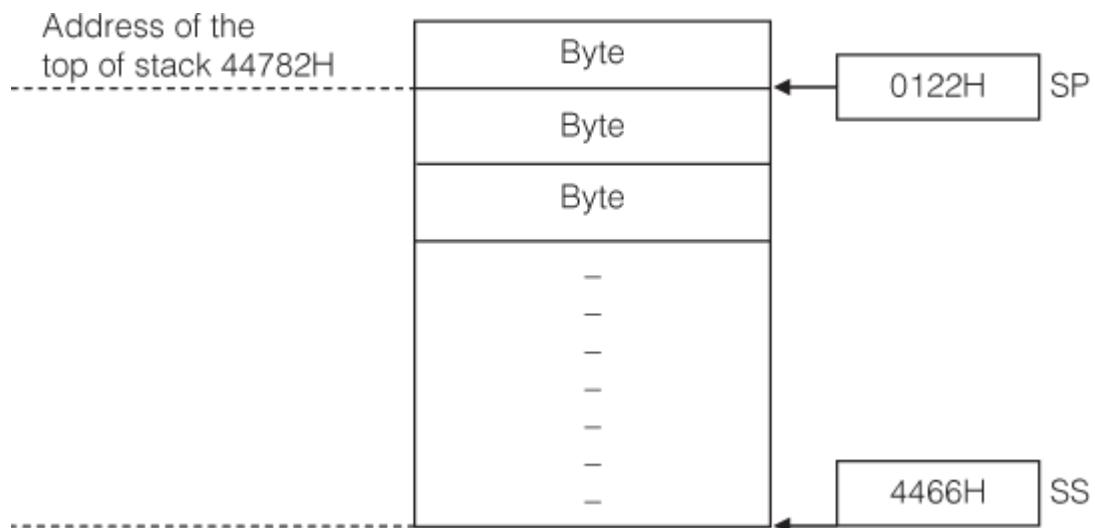


Figure 1.6 | Stack and physical address of the top of stack

The stack is used mostly to keep aside address and data temporarily, when a subprogram is called. This will be taken back when the subprogram ends. The 8086 has a stack which grows downwards (i.e., to lower memory addresses). This means that each push instruction causes the SP to be decremented (by two for 8086), and data is pushed deeper downwards into the stack segment. We will discuss the operation of the stack in more detail in Chapter 4.

Even though a stack is usually accessed only from the top, the 8086 has another register named BP (base pointer), which is used to reference data anywhere within the stack.

Thus, the format SS : BP is also a logical address. This has certain interesting applications.

1.3.2.4 | The Data Segment and Extra Segment

Both these segments store data, but in certain special cases (string instructions), it may be necessary to list them separately. There is an Extra Segment (ES) register to store the upper 16 bits of the base address of the extra segment. The offset within the data segment is also termed as an '*effective address*'. The effective address calculation depends on the mode of addressing. We will discuss this very soon. Table 1.1 shows the segment registers and the corresponding registers that can be used for offsets.

Table 1.1 | Segments and Corresponding Registers Used for Specifying Off sets

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP BP	Addresses in the stack
ES	BX, DI, SI	Address of destination data (for string instructions)

Example 1.4

The content of DS is 345BH. The amount of data that is to be stored in the data segment is 12K bytes. Where in memory will this segment be located?

Solution

DS contains the number 345BH.

This number corresponds to the upper 4 hex digits of the starting (base address) of the data segment.

The base address of the segment is 345BoH.

The last address will have an offset of 12K from the base address.

1K = 1024 bytes.

$$12K = 12 \times 1024 = 12288 = 3000H.$$

The last address of the segment = $345\text{BoH} + 3000\text{H} - 1 = 375\text{AFH}$.

Hence, the data segment occupies the memory addresses from 345BoH to 375AFH .

Example 1.5

The contents of the following segment registers are as given

$\text{CS} = 1111\text{H}$, $\text{DS} = 3333\text{H}$, $\text{SS} = 2526\text{H}$.

$\text{IP} = 1232\text{H}$, $\text{SP} = 1100\text{H}$, offset in data segment = 0020H .

Calculate the corresponding physical addresses for the addressed byte in
(a) CS, (b) SS and (c) DS.

Solution

1. The base address of the code segment is 11110H . The address of the next instruction to be executed is referenced by CS and IP, which is given by $11110\text{H} + 1232\text{H} = 12342\text{H}$.
2. The current top of the stack is referenced by SS and SP. The base address of the stack segment is 25260H . The corresponding physical address is $25260\text{H} + 1100\text{H} = 26360\text{H}$.
3. The data that needs to be accessed is given by DS and the offset. The base address of the data segment is 33330H . The physical address of this data is calculated as $33330\text{H} + 0020\text{H} = 33350\text{H}$.

Segmentation: General Concerns

1. Segments begin at paragraph boundaries, where a paragraph is 16 bytes. That is, Base addresses are divisible by 16 (because the lowest nibble of all base addresses is to be zero).
2. Segments can overlap with a new segment starting every 16 bytes. This also means that there can be more than one logical address for the same memory location (physical address). For example, $0000 : 0100$, $0001 : 00F0$, and $0010 : 0000$ all refer to physical address 00100H .
3. We can have many segments with the same base address, but the offsets within the segment will have to be taken care of to prevent overwriting of data. For example, in string operations, we might use the same segment as the data and extra segments. This is achieved by loading the same number in the ES and DS registers.
4. A segment can have a maximum size of 64K.
5. A program need not have all four segments, but it will have at least the code segment.

- The addresses used in instructions are called logical addresses and are translated by the BIU to physical addresses.

Advantages of Segmentation

- It allows all address registers to have the same size as the data registers (16-bit), while allowing the use of 20-bit physical addresses.
- All addresses in memory are *re-locatable*. This means that any program or data can be loaded in any address in memory. A re-locatable program is one which can be placed in any area of memory and executed without change. Data is also re-locatable. Changing the base address of the corresponding segment is the only action we need to perform in order to re-locate. All addresses within the program are relative to the base address, as they are of the form Base address: offset. All processors in the x86 family have this kind of segmentation. The higher order processors also have *protection* added to this structure.

Memory Organization

We have seen that data is stored in memory as bytes. Each byte location has an address. If a word (two bytes) is to be stored, two locations of consecutive addresses are needed. Consider that the number 34E6H is to be stored. Then, 34H is stored in one location and E6H in the next - location. Intel's processors use what is termed the *little endian format*, which means the lower byte is stored in the lower address and the upper byte in the higher address. Thus, if the above word has an address of 60088H, the storage is as shown in Fig. 1.7. This is in contrast to certain other processors (like Motorola) which uses the *big endian format*. (The origin of the word 'endian' is supposed to be from Gulliver's travels, referring to a controversy about whether an egg should be broken at the little end or the big end.)

ADDRESS	DATA
60088H	E6H
60089H	34H

Figure 1.7 | Little endian format

Since the data in this case is a word, two memory locations with two addresses have to be used to access it. The hardware connection between the processor and memory is designed in such a way that both these locations are accessed and a word (16 bits) access is obtained, even though only one logical address is mentioned in the instruction.

1.4 | Addressing Modes

For computations in assembly language, we need an opcode and operands. Opcode stands for *operation code* – i.e., the code which specifies the operation to be performed. Opcodes operate on data, which are then called the *operands*. In the 8086, an instruction can have either one or two operands (not more). For example, an addition operation needs two operands, while a shift operation operates on a single operand. The way in which operands are specified in an assembly language instruction is called its *addressing mode*. Let us attempt to get a clear understanding of the different addressing modes allowed for this processor. We will use the MOV instruction for understanding these modes. This has the format

MOV destination, source

which causes the source data to be copied into the destination.

The basic assumptions in this context are

1. the operands can be in registers, in memory, or may be in the instruction itself. However, the 8086 does not have an addressing mode in which both operands are in memory locations,
2. in the case of two operands, one of them can be in memory, but the other will have to be placed in a register,
3. data types should match, that is, the source, and destination should both be either bytes or words.

1) Register Addressing

Here, both the source and destination are registers. No memory access is involved. See the following instructions:

MOV AL, AH	;copy the content of AH to AL
MOV CH, BL	;copy the content of BL to CH
MOV SI, BX	;copy the content of BX to SI
MOV ES, AX	;copy the content of AX to ES

Note that the first two are *byte* operations, while the other two are word operations.

MOV AX, BL	;gives an error as AX is 16 bit and BL is 8 bit
MOV BL, AX	;gives an error for the same reasons

2) Immediate Addressing

In this mode, the source will be a constant data

MOV AL, 45H	;copy 45H to AL
MOV BX, 34E3H	;move the hex number 34E3H to BX
MOV CL, ‘Q’	;move the ASCII value of Q to CL
MOV PRICE, 40	;move the hex number 40 to the memory location with label PRICE
MOV NUMS, 0FC6H	;move the hex number 0FC6H to the memory location NUMS

Segment registers are not allowed to use this mode of addressing.

MOV DS, 2300H	;gives an error as DS is a segment register
---------------	---

3) Direct Addressing

Here, either the source or the destination will be a memory address.

MOV AX, [2345H]	;move the word in location 2345H into AX
MOV [1089H], AL	;the byte in AL is moved to location 1086H

It is to be remembered that the addresses in the instructions are offsets within the data segment (i.e., the logical address). We need to know the content of DS to calculate the physical address. The size of the registers indicates, the size of the operand. Hence, in the first instruction, a data *word* is referred to; while in the second, a data *byte* is moved. The square brackets are necessary to indicate that the number is an address and not data. However, we may use labels for addresses, and re-write the above two instructions.

MOV AX, PRICE
MOV COST, AL

We must then ensure that the referred addresses have been defined earlier with these labels. We will see this aspect later.

4) Register Indirect Addressing

In this mode, the address of the data is held in a register. The register acts as a pointer to the data. The registers must be enclosed in square brackets to indicate that they function as pointers. We also use the term *effective address* for the address of the operand. For this mode of addressing, the address registers allowed are BX, SI, and DI.

$EA = \{[BX]/[DI]/[SI]\}$	
MOV AL, [BX]	;move into AL the content of the location whose address is in BX
MOV [SI], CL	;move the content of CL to the address pointed by SI
MOV [DI], AX	;move the content of AX to the address pointed by DI

In the third instruction, AX contains two bytes. Hence, the content of AL will be moved to the address pointed by DI. The content of AH will be moved to the address $[DI + 1]$.

Example 1.6

Show the location of data in memory, after the execution of each of these instructions, if the content of registers are as given

- $DS = 1112H, AX = EE78H$ and $BX = 3400H$
- i) MOV [0422H], AL
 - ii) MOV [0424H], AX
 - iii) MOV [BX], AX

Solution

1. This is a case of direct addressing. The destination is a memory location, which is specified directly in the instruction. AL contains 78H

$$DS = 1112H$$

Data segment base address is 11120H
The offset in the instruction is 0422H
Physical address is 11120H +
0422H
11542H

After the instruction execution, the content of the addressed location is as shown

PHYSICAL ADDRESS	CONTENT
11542H	78H

2. This is also a case of direct addressing. The physical address is calculated as
11120H +
04224H
11544H

However, since AX contains two bytes of data, the lower byte (AL) is stored in the address 11544H and the higher byte (AH) in the next address (recollect the ‘little endian’ concept)

PHYSICAL ADDRESS	CONTENT
11544H	78H
11545H	EEH

3. This is a case of register indirect addressing. BX is a pointer to the address 3400H, which is an offset. The corresponding physical address is

$$\begin{array}{l} 11120H + \\ \underline{3400H} \\ \hline \underline{14520H} \end{array}$$

The content of AX is moved to this address. Hence, the corresponding memory will be as shown

PHYSICAL ADDRESS	CONTENT
14520H	78H
14521H	EEH

5) Register Relative Addressing

In *relative* addressing mode, a number or displacement is part of the effective address.

$EA = \{[BX]/[DI]/[SI]/[BP]\} + 8\text{-bit or } 16\text{-bit displacement}$
 MOV CL, 10[BX] ;move the content of the address specified by adding the content of BX and 10.

Thus, the effective address is [BX + 10]. Once the effective address is computed, the physical address is calculated as the sum of the segment base address and the effective address. The displacement can be a 16-bit signed/unsigned number or an 8-bit *sign extended* number. However, the displacement should not be so large as to make the effective address go beyond the range of the maximum size of a segment. The above instruction can also be written as

MOV CL, [BX + 10]
 MOV CL, [BX] + 10 or
 MOV CL, [BX][10] or
 MOV CL, PRICE [BX]

In the last case, PRICE has to be defined earlier as a displacement of 10.

6) Based Indexed Mode

In this mode, an index register and a base register together carry the effective address. The content of these two registers are added and called the effective address.

MOV AL, [BX][SI] ;move the content of the effective address pointed by [BX] and [SI] into AL

The effective address is obtained by adding the content of BX and SI. Since the destination register is an 8-bit register, this is a byte operation. The following is a *word* operation as the source register CX is 16-bit in size.

MOV [BX][DI], CX ;move the content of CX to the effective address pointed by [BX] and [SI]

7) Relative-based Indexed Mode

This is the case when the ‘effective address’ is specified with a base register, an index register as well as a displacement. The effective address is the sum of the two registers and the displacement. For example, the following use the relative-based indexed mode of addressing:

MOV DL, 5[BX][DI]	;EA = 5 + BX + DI
MOV 5[BP][SI], AX	;EA = 5 + BP + SI
MOV CL, COST[BX][SI]	;EA = COST + BX + SI ;COST has to be defined as a displacement earlier

Example 1.7

Find the address of physical memory for the following instructions if the content of the required registers are as given below.

SS = 2344H, DS = 4022H, BX = 0200H, BP = 1402H, and SI = 4442H

1. MOV CL, 1234H[SI]
2. MOV AL, 5[SI][BP]

Solution

- i) MOV CL, 1234H[SI]

This is a case of register relative addressing. The effective address is obtained from the instruction to be the sum of the displacement and SI

$$\begin{array}{r} \text{Effective address} = 1234H + \\ \underline{4442H} \\ \underline{\underline{5676H}} \end{array}$$

The segment base address is obtained from DS to be 40220H.

The physical address is the sum of the segment base address and the effective address.

$$\begin{array}{r} 40220H + \\ \underline{5676H} \\ \underline{\underline{45896H}} \end{array}$$

i) MOV AL, 5[SI][BP]

This is a case of relative-based indexed mode. The effective address is calculated as the sum of the displacement and the contents of the registers SI and BP.

$$\begin{array}{r} \text{Effective address} = 0005H + \\ \underline{4442H} \\ \underline{1402H} \\ \underline{\underline{5849H}} \end{array}$$

The physical address is the sum of the segment base address and the effective address. In this case, as *BP* is one of the address registers, the segment referred is the *stack segment*.

$$\begin{array}{l} \text{Physical address is } 23440H \quad (\text{base address of the stack segment}) \\ + \quad \underline{5849H} \\ \underline{\underline{28C89H}} \end{array}$$

Segment Override

Table 1.2 | Effective Address and Referred Segments for Various Memory-based Addressing Modes

SI No.	Addressing mode	Effective address	Segment
1	Direct	offset	DS
2	Register indirect	[BX] [SI] [DI]	DS DS DS
3	Register relative	Disp + [BX] Disp + [SI] Disp + [DI] Disp + [BP]	DS DS DS SS
6	Based Indexed	[BX] + [SI] [BX] + [DI] [BP] + [SI] [BP] + [DI]	DS DS SS SS
7	Relative-based Indexed	Disp + [BX][SI] Disp + [BX][DI] Disp + [BP][SI] Disp + [BP][DI]	DS DS SS SS

Note that BP cannot be used without a displacement. See it being used in 'register relative' mode but not in register indirect mode.

Table 1.2 shows the default segments to be used in the various memory-based addressing modes. However, this can be overridden by a segment override prefix. That is, we can use segments other than those specified in the table. The format of this *override prefix* is shown below. MOV AL, ES : [BX] overrides the fact that an instruction of this sort implies that the data segment is to be used. Here, it says that ES is to be used instead. Hence, physical address calculations use the content of the CS register as the base address. Other examples are

MOV DS : [BP + 7], BL ;DS to be used instead of SS
 MOV AX, CS : [BX] ;CS to be used instead of DS

KEY POINTS OF THIS CHAPTER

- The architecture of 8086 is representative of the architecture of all members of the x86 family
- The data bus width of 8086 is 16, and internal registers also have the same bit size
- The internal block diagram shows two units named the ‘execution unit’ and the ‘bus interface unit’ with well-defined and separate functions
- The BIU and EU communicate through the internal bus. Both have a common timing and control unit
- The execution unit takes care of computations, and thus, it contains the necessary registers and flags
- The conditional flags are set or reset according to the result of certain arithmetic/logic operations

- The 8086 has 20 address lines, which means it can access 1 MB of memory
- Memory is segmented into four types of segments
- Code and data are kept in separate segments
- The upper 16 bits of the base address of any segment are available in the corresponding segment registers
- The stack segment is a type of segment that grows downwards as data is pushed into it
- The BIU converts the logical addresses to physical addresses
- Segmentation causes data and code to be re-locatable
- The addressing mode dictates where operands are available
- 8086 follows the 'little endian' scheme of data storage in memory

QUESTIONS

1. Name the 8-bit registers of 8086.
2. What is meant by saying that the 8086 is a 16-bit processor?
3. Which are the first and last memory addresses that an 8086 can address?
4. Which unit (EU or BIU) is responsible for performing arithmetic calculations?
5. How will it be helpful for a processor if it has a large number of internal registers?
6. What are the minus points of the presence of too many scratchpad registers?
7. Which are the registers catering to string instructions?
8. What is meant by a LIFO stack?
9. What is the difference in operation between the carry flag and the overflow flag?
10. Which register is frequently used as a counter?
11. Where in memory is the program code located?
12. BP is a register which refers to the extra segment. True or False?
13. BP cannot be used without a displacement. True or False?
14. What is meant by the term 'program relocation'?
15. What happens to the pre-fetch queue when a branch instruction is encountered?
16. What is the format of a logical address?
17. If the physical address of a byte is 12345H, suggest three logical addresses that it can have.
18. Which is the default segment when IP is used?
19. Can we have segments of size less than 64K?
20. Write an example instruction that contains a segment override prefix.

EXERCISES

1. Find the content of the destination and the status of the carry flag (CF) and zero flag (ZF) after the execution of the following set of instructions:

 1. **MOV AL, 09H**
ADD AL, CFH
 2. **MOV BX, 0876H**
MOV AX, 0034H

ADD BX, AX

3. MOV AL, 0FFH
 ADD AL, 01

2. Add the numbers 100 and 89 using the following instructions:

 MOV AL, 100

 ADD AL, 89

 Which of the conditional flags are set and why?

3. Show the binary number that will be available in AL after the execution of each of the following instructions:

 o. MOV AL, 'C'

 1. MOV AL, 25

 2. MOV AL, 34H

4. Find the physical address of the top of the stack if SS = 0777H and SP = 1234H.

5. If IP contains the number 0034H and CS = 5555H, where will the next instruction be fetched from?

6. If CS contains the number OFC2H and the volume of code is 2400 bytes, which is the last address in this code segment?

7. Write the addressing mode of the following instructions:

- o. MOV [4560H], AX

 1. ADD BL, 89H

 2. ADD BX, [DI]

 3. MOV 6[BP][DI], AL

 4. MOV CX, [BP + 9]

 5. ADD CL, BL

8. Find the physical address of the memory locations referred to in the following instructions if DS = 0223H, DI = 0CCCH, and SI = 1234H:

- o. MOV [DI], AL

 1. MOV [SI][56H], BL

9. Which segments will be accessed in the following cases?

- o. MOV [6][BX], CX

 1. MOV [BP] + 4, CL

 2. PUSH AX

 3. MOV CS : [BX], DL

 4. MOV AL, ES : [45H]

10. The memory addresses starting from the logical address 0124H : 0056H store the numbers 0978H, CE45H, 45H and 7809H in consecutive locations. Draw a diagram showing the physical address and the corresponding data stored.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

2

Programming Concepts—I

IN THIS CHAPTER, YOU WILL LEARN

- The working principle of an assembler.
- The various assemblers available for x86.
- The features and use of a popular assembler.
- To write programs using simplified memory models.
- To write programs using the full segment definition.
- To understand the techniques of instruction design for 8086.

Assembly Language Programming

In chapter 0, we have seen why assembly language is very important. To code efficiently in assembly language for a particular processor, the prerequisites are a good knowledge of the internal architecture of the processor and addressing modes, which means that understanding the contents of chapter 1 is absolutely necessary.

When we want to write and execute an assembly language programming (ALP), we first type it using an editor. Then we assemble, link and run it. There are many assembly language program development tools available for each of these actions. We shall examine them one by one.

2.1 | The Assembly Process

The way an assembler is designed depends heavily on the internal organization of the processor for which it is used. Architectural features such as memory word size, number formats, internal character codes, index registers, and general-purpose registers affect the way assemblers are written and the way an assembler handles instructions and directives.

In ALP, coding is done in symbolic language (called mnemonics). The word *mnemonic* comes from a Greek word meaning *pertaining to memory*; it is a memory aid. The mnemonic specifies the operation to be done. We need an assembler which translates this symbolic language to

machine code which the processor can understand. As such, we guess that this ‘translation’ is indeed the main task of the assembler.

Let us first get an idea of the assembly process. An interesting definition for an assembler is that it is a translator that translates source instructions (in symbolic language) into target instructions (in machine language), on a **one-to-one** basis. The point is that each source - instruction is translated to exactly one target instruction. This is indeed the main task of the translator. However, an assembler is more than just a translator. It is an important program development tool; the quality of which can affect the ease of programming. Since ALP is more prone to errors (than HLLs); the assembler should give appropriate error messages to guide the programmer at every step during programming.

2.1.1 | Features of Assemblers

When code is written in assembly language, it is easy if we can use labels (names) for memory locations. All constants and variables are given names so that they can be referred by name. Managing and utilizing these labels (called symbols) efficiently reflects on the quality of an assembler.

Assemblers that support macros are called Macro Assemblers. A macro is a name given to a sequence of instruction lines. Once a macro is defined, its name may be used in place of this set of lines. In the assembly process, each macro is replaced by the text lines. Thus, a macro seems to the programmer to be like a mnemonic, and he can use it as frequently as he wants, without incurring any of the overheads that a procedure or function may cause. The use of macros can help to make programs structured.

2.1.2 | Instructions and Directives

An assembly language program contains pseudo-instructions (directives) along with instructions. Instructions get translated to machine codes, but directives do not i.e., instructions produce executable code, but directives do not. They direct the assembler to perform in certain ways that we would like it to. The directives perform as ‘help’ for the assembler for deciding certain other aspects of the assembly process. Thus, the assembler takes our **source code** (which is our program containing instructions and directives) and converts it into **object code** (which contains the machine code). The object file is the one that will be loaded into memory and executed after linking it with other necessary files.

2.1.3 | The Forward Reference Problem

A typical line in a program could be

```
BEGIN: MOV AX, COST ;move the content of the location COST to AX
```

In this example, BEGIN is a label, which corresponds to the address at which this line of code is stored. MOV is a mnemonic. This instruction means that the content of COST is to be moved to register AX. COST is a label (for a memory location) that acts as one operand in this two-operand instruction. A label when defined is called a symbol. A symbol can only be defined once but it can be used many times. There will be a memory address corresponding to all symbols. In this instruction, the other operand is AX. The text coming after ‘;’ is a comment.

All current assemblers allow the use of labels, rather than numeric addresses of memory locations. However, finally, the assembler must translate these labels to memory addresses. As such, an important issue that an assembler will have to sort out is what is called the ‘forward reference problem’.

See the code below

```
BEGIN: MOV AX, BX  
        JMP AGAIN
```

```
-----  
-----  
-  
-  
-
```

```
AGAIN: -----
```

As the assembler reads the second line of this program, an undefined label AGAIN is encountered. The assembler has no knowledge of it as yet, but the value of the symbol AGAIN is needed before it can be defined. This is also called the future symbol problem or the problem of unresolved references. This problem is solved in various ways depending on the type of the assembler.

2.1.4 | Two Pass Assemblers

There are many types of assemblers—one pass, two pass and multi-pass assemblers. Each pass optimizes the assembly process. Two pass assemblers are the commonest, so let us discuss those. A pass implies ‘a reading’ of the code from beginning to end. A two-pass assembler does

two passes over the source file (the second pass will be over a file generated in the first pass). In the first pass, it looks for label definitions and inserts them in the symbol table after assigning them addresses. Modern assemblers keep more information in their symbol table, which allows them to resolve addresses in a single pass. Known addresses (backward references) are immediately resolved. Unknown addresses (forward references) are “back-filled”, once they are resolved.

Memory is allocated sequentially, and a ‘location counter’ is incremented at each step. At the end of this pass, the symbol table should contain definitions of all the labels used in the program. This means that during the first pass, the only duty of the assembler is assigning memory addresses to labels.

In pass 2, the actual translation of assembly code to machine code is done. Errors are also reported after this. The assembly process produces a ‘re-locatable’ object file. This object file can be loaded anywhere in memory, when the program is to be executed. A ‘loader’ is a program which does this. Please note that loading can be done anywhere in memory and that it is not necessary to load it at the same location each time the program execution is done.

2.2 | Assemblers for x86

An assembly language is specific to a particular processor. As such, we should concern ourselves with assemblers for the x86 family of processors, which includes the whole set from 8086 to the latest x86 processor. A large set of assemblers is available, but the code running on one assembler will not run on another. Similarly, another matter to be taken care of is the OS under which it can run. A86 was the first assembler to appear on the scene. It could run only under DOS. Now, there are assemblers that can run under DOS, Windows and Linux. The names of a few popular assemblers are NASM, FASM, MASM, TASM and HLA. FASM and NASM can run under all the three operating systems mentioned.

Two popular Windows-based assemblers are TASM and MASM. TASM is Borland’s Turbo Assembler, whereas MASM is Microsoft’s Macro Assembler. It is found that the code written on TASM runs on MASM, although the reverse need not be true. Both of them run under DOS and Windows, which means that they can use 16-bit and 32-bit coding. Both of them are classified as high-level assemblers, which means that we can use high-level control structures (such as IF, WHILE, REPEAT...UNTIL) that make programming easy. HLA is another assembler that uses high-

level control structures, high-level data types and high-level procedure calls.

2.2.1 | MASM and NASM

For programs in this book, we choose MASM because it is one of the best assemblers and also because of its popularity. A popular assembler gives us the opportunity to enable easy access to references and material. Moreover, MASM has many constructs and features that make ALP easy and fun for beginners. Another reason is that the documentation available for it is very good. MASM was a commercial product for a very long time and is a sort of industry standard. Microsoft has written considerable documentation for this assembler and many third parties have written assembly language reference manuals for MASM. The versions of MASM 6.0 and above have a lot more features (aimed at simplification in writing code) than previous versions.

Incidentally, MASM is a proprietary product of Microsoft. We will now use the 16-bit version that runs under DOS, because we can use DOS and BIOS interrupts in this version. The 32-bit version runs under Windows. This is freely available on the net (the 32-bit version does not allow the use of DOS and BIOS interrupts). In Chapter15, we will use MASM32 for the 32-bit x86 processors. This approach gives a very good understanding of assembly language in general. The DOS version used for programs in this book is MASM 6.14. These programs will run in all the versions of MASM 6.0 and above. Now, MASM is also available for use in Microsoft's Visual Studio program.

The problem users now face is that the latest 64-bit Windows versions do not have DOS inbuilt into them. Thus, 16-bit MASM will not run in the 64-bit versions of Windows Vista, Windows 7, and Windows 8. The solution for this is to install DosBox and use MASM in it. DosBox is an emulator for a PC with built-in DOS. It simulates a PC with an older operating system, and hence, DOS programs can run on it. Refer to the online resources of this book for more detail on how to use 16-bit MASM in DOSBox.

NASM running on Linux is also very popular now. NASM is freely available on the internet at www.nasm.us. It can be downloaded as an installation package from the site and installed easily. For readers who need example programs designed for NASM under Linux, a tutorial (Courtesy: Mohammed Yazar) has been put up in the book website.

2.2.2 | Assembly Language Programming

Now that we have understood the assembly process and have decided the assembler to use, let us get into the nitty-gritty of programming. The steps involved are:

1. Write the code with the help of an editor. We can have an editor within MASM or can use notepad or wordpad. Save it as filename.asm in the BIN directory of MASM. Note that instructions, directives and labels are **not case sensitive**.
2. Open the DOS command window and go to the directory containing MASM. From this, go to the BIN directory.
3. Assemble the code. Syntax errors (if any) will be reported at this stage. This will generate the object file with the name filename.obj. Other files are also generated and we can opt to view them (list file, configuration file and so on). Details of these are given in Appendix B.
4. Link the object file. The linking involves combining separately assembled modules into one executable file (filename.exe).
5. Run the exe file. This will cause the program to be loaded in memory and executed.
6. The results of execution will be in memory or in registers. If we want to view it on our display unit, we will have to use special codes for it (this will be discussed later). Otherwise, we can examine the memory and register contents using the debugger. The debugger has all the commands needed to do this, and the full set of commands is detailed in Appendix B. DOS commands are also listed here.

2.3 | Memory Models

To write programs, we know we have to define segments, which means that the segment registers must be initialized. This will take us to the conventional **full segment model**. However, versions of MASM 6.0 and above have incorporated certain shortcuts which help to make programming simple. These are called the dot models. They have the format:

.MODEL MODEL NAME

The different models tell the assembler how to use segments and to provide sufficient space for the object code. The simplest of these are the tiny model and the small model. We will start with these models. The tiny model is used when code, data and stack will all fit into one segment with a maximum size of 64 K. The small model can have one data segment and one code segment each of which has a maximum size of 64 K.

2.3.1 | The Tiny Model

Let us now write a program using the tiny model.

Example 2.1

```
.MODEL TINY          ;choose single segment model
.CODE               ;start of code segment
.STARTUP            ;start of program
MOV AL, 67H          ;move 67H to AL
MOV BL, 45H          ;move 45H to BL
ADD AL, BL           ;add BL to AL
MOV DL, AL           ;copy AL to DL
.EXIT               ;exit to DOS
END                 ;program end
```

Let us enter this code using an editor and save it as **tinym.asm** in the BIN directory of MASM6.14. Open the DOS command window. Go to the directory MASM 6.14/BIN. In the BIN directory, assemble and link the file using the command **ml tinym.asm**. Example 2.2 shows the DOS command window corresponding to this. The ml command causes assembling and linking to be done at a go. Now to get the list file, type the command **ml/FI tinym.asm**. The list file will be saved in the BIN directory as **tinym.lst** (more commands and options are available in Appendix B). Earlier versions of MASM required two steps for the same actions. To assemble, the command was **masm filename.asm**. Linking required the command **link filename.obj**. The list file was obtained by using **masm/l filename.asm**. Examples 2.2 shows the DOS command window as we do assembling and linking. Example 2.3 shows the list file.

Note If you think this part is taxing, rush through these topics fast and come back to it after you have learned more aspects of programming and are ready to do programming using this assembler.

Example 2.2

```
C:\masm6.14\BIN>ml tinym.asm
```

```
Microsoft (R) Macro Assembler Version 6.14.8444
```

```
Copyright (C) Microsoft Corp 1981–1997. All rights reserved.
```

```
Assembling: tinym.asm
```

Microsoft (R) Segmented Executable Linker Version 5.60.339 Dec 5
1994

Copyright (C) Microsoft Corp 1984–1993. All rights reserved.

Object Modules [.obj]: tinym.obj /t

Run File [tinym.com]: “tinym.com”

List File [nul.map]: NUL

Libraries [.lib]:

Definitions File [nul.def]:

Example 2.3

```
Microsoft (R) Macro Assembler Version 6.14.8444
tinym.asm
      .MODEL TINY      ;choose single segment model
0000      .CODE        ;start of code segment
          .STARTUP    ;start of program
0100 B0 67      MOV AL, 67H    ;move 67H to AL
0102 B3 45      MOV BL, 45H    ;move 45H to BL
0104 02 C3      ADD AL, BL   ;add BL to AL
0106 8A D0      MOV DL, AL   ;copy AL to DL
          .EXIT        ;exit to DOS
          END         ;assembler to stop reading
```

Example 2.3 shows the listing corresponding to our tinym program. (Only a part of the list file is shown here. See Appendix B for the additional details.) A lot of information can be obtained just by looking at the list file. The list file is one of the outputs obtained in the assembly process. Examine Example 2.3.

We first see the details of the version of MASM that is used. The complete assembly language program along with the comments that we had typed in, is seen next. On the left hand side, we see the offsets within the code segment in which the code is saved. These offsets have been generated by the assembler. We also find the opcodes corresponding to each instruction. For example in the first line:

0100 B0 67 MOV AL, 67H ;move 67H to AL

0100H is the offset in the code segment where the first instruction is stored, and B0 67 is the opcode of MOV AL, 67H. If we add a directive.listall, the list file is changed as shown in Example 2.4.

Example 2.4

```
.MODEL TINY
0000          .CODE
              .STARTUP
0100          * @Startup:
                  .listall
0100  B0 67      MOV AL, 67H      ; move 67H to AL
0102  B3 45      MOV BL, 45H      ; move 45H to BL
0104  02 C3      ADD AL, BL      ; add BL to AL
0106  8A D0      MOV DL, AL      ; copy AL to DL
                  .EXIT          ; return to DOS
0108  B4 4C      *     mov     ah, 04Ch
010A  CD 21      *     int     021h
                  END
```

.Listall has listed the instructions corresponding to .EXIT

.EXIT is a shortcut, which is actually transformed to two instructions:

```
MOV AH, 4CH
INT 21H
```

These two instructions terminate program execution and bring control back to DOS. END is a directive. It tells the assembler to stop reading, as there are no instructions beyond this. The .STARTUP command is available only in the versions of MASM 6.0 and above. It is a shortcut, whereby segment registers get initialized automatically, which means that programmers need not write instructions for that.

Now let us use the debugger to get more information about these aspects. The debugger allows us to view memory and registers, along with various other functions (see Appendix B). To enter the debugger, type **debug tinym.com**. We get an underscore as the prompt. On typing ‘r’, we can see the contents of the registers before execution of the program. Now type ‘u’, which is the command for unassembling or disassembling. Disassembling is the reverse of assembling. It gets the mnemonics back from the machine codes. Example 2.5 gives the result of these two commands.

Example 2.5

```

C:\masm6.14\BIN>debug tinyM.com
-r
AX=0000 BX=0000 CX=010C DX=0000 SP=0000 BP=0000 SI=0000
DI = 0000 DS = 13AD ES = 13AD SS = 13BD CS = 13BD IP = 010
0 NV UP EI PL NZ NA PO NC
13BD:0100 B067      MOV     AL, 67

-u
13BD:0100 B067      MOV     AL, 67
13BD:0102 B345      MOV     BL, 45
13BD:0104 02C3      ADD     AL, BL
13BD:0106 8ADO       MOV     DL, AL
13BD:0108 B44C       MOV     AH, 4C
13BD:010A CD21       INT     21

```

The r command gives us the content of the registers before the execution of the instruction MOV AL, 67H. We find that the content of the CS register is CS = 13BD. On typing the ‘u’ command, we find that the logical address of the first instruction is 13BD:0100. The value of CS need not be the same every time the program is loaded into memory. This is because the object file generated by the assembler is **re-locatable**.

2.3.2 | COM and EXE Files

For this program, there is only one segment, which is the code segment. The run file generated for the tiny model is a Command (. com) file, rather than an Executable (. exe) file. This can be seen in Example 2.2. in the line:

Run File [tinyM.com]: “tinyM.com”

Later, it will be seen that run files will be obtained as ‘executable’ with .exe as the extension. However, the tiny model always assembles to be a com file. This is a special case, when we need a compacted form of data and code, all fitting into just 64 K size for memory efficiency. Other models, which we will see later, all assemble to .exe files. Quantifying the features of COM files:

1. Size is limited to 64 K.
2. Only one segment, which is the code segment.
3. Data is defined in this code segment.
4. Code starts at offset 0100 H, just after the PSP (program prefix segment) of DOS.
5. Smaller file compared to exe files, because it does not have the 512-byte header block.

From Examples 2.2 to 2.5, we have seen the tactics of running and analyzing a single segment assembly language program running on MASM. Now, see the listing corresponding to another program which uses the tiny model.

Example 2.6

```
0000      .MODEL TINY      ;choose single segment model
          .CODE           ;start of code segment
          .STARTUP        ;start of program

0100 8B C8  MOV    CX, AX   ;copy contents of AX to CX
0102 8B C2  MOV    AX, DX   ;copy contents of DX to AX
0104 8A C3  MOV    AL, BL   ;copy contents of BL to AL
0106 8A E1  MOV    AH, CL   ;copy contents of CL to AH
0108 8A F2  MOV    DH, DL   ;copy contents of DL to DH
010A 8E DA  MOV    DS, DX   ;copy contents of DX to DS
010C 8E D3  MOV    SS, BX   ;copy contents of BX to SS
          .EXIT          ;exit to DOS
          END            ;end of program
```

Example 2.6 shows a sequence of instructions that copy various data between 16- and 8-bit registers. The act of moving data from one register to another changes only the destination register, never the source.

Example 2.7

```
0000      .MODEL TINY      ;choose single segment model
          .CODE           ;start of code segment
          .STARTUP        ;start of program

0100 B8 0000  MOV  AX, 0    ;place 0000 into AX
0103 B0 43    MOV  AL, 'C'   ;place 'C' into AL
0105 B9 5674  MOV  CX, 05674H ;place 5674H into CX
0108 B2 2D    MOV  DL, 45    ;place 45 (decimal) into DL
010A B6 45    MOV  DH, 45H   ;place 45H into DH
          .EXIT          ;exit to DOS
          END            ;end of program
```

Example 2.7 is a listing and it shows various assembly language instructions that use immediate addressing. Only 8-bit data can be placed into an 8-bit register. AL, DL and DH are 8-bit registers. 16-bit data can be copied to 16-bit registers. Though in the editor, the second instruction was written as MOV AL, 'C', the assembler translates ASCII 'C' to its hex equivalent 43H. All data written in memory will be in the hex format (i.e., the compressed form of binary representation). Thus, the decimal 45 is found to 2DH in the listing.

Note A data (byte, word or longer) starting with the hex characters A, B, C, D, E or F must be preceded by a 0. Otherwise the assembler will give an error, i.e., that is, MOV AL, EFH gives an assembly error. It must be rewritten as MOV AL, 0EFH. Similarly, MOV BX, C456H is wrong. Rewrite it as MOV BX, 0C456H.

2.3.3 | Definition of Data Types

Before we go on to using a two-segment memory model, we need to understand a few directives of the assembler which define and describe different kinds of data. Data which is used in a program can be bytes or words or of greater length. We have to define data and assign labels to their corresponding addresses. Assigning labels eases the programmer's burden as he does not have to concern himself with numerical values. The location counter in the assembler keeps on incrementing as labels are encountered.

Defining data implies allocating space for data. Data is defined accordingly using directives. Thus, DB defines data byte, while DW defines data word. This is the traditional way of defining data. The newer versions of MASM (6.0 and higher) allow the use of the directive BYTE for DB, and WORD for DW. Table 2.1 shows the definitions of data of different lengths.

Table 2.1 | Data Definitions Used by MASM

Definition	Traditional directive	New directive
Byte	DB	BYTE
Word	DW	WORD
Double word	DD	DWORD
Quad word	DQ	QWORD
Ten bytes	DT	TBYTE

Example 2.8

```

        .MODEL TINY
0000          .CODE
0000 50      NUM1 DB    50H      ;place 50H into NUM1
0001 210A    NUM2 DW    210AH    ;place 210AH into NUM2
0003 0789    NUM3 DW    0789H    ;place 0789H into NUM3

        .STARTUP
0100 A0 0000 R    MOV AL, NUM1      ;move NUM1 to AL
0103 8B 1E 0001 R  MOV BX, NUM2      ;move NUM2 into BX
0107 8B C8       MOV CX, AX        ;copy AX to CX
0109 8B 1E 0003 R  MOV BX, NUM3      ;move NUM3 into BX
        .EXIT
        END

```

Example 2.8 is a listing, which shows data being placed in the code segment itself. In the tiny model, we can have data and code in the same segment, with the restriction that the size of the segment should not exceed 64 Kbytes. NUM1, NUM2, and NUM3 are locations which store data. NUM1 is a byte location, while NUM2 and NUM3 are word locations.

2.3.4 | The Small Model

Now we are in a position to write a program using the small model, which has one data segment and one code segment. Example 2.9 shows the same program as in Example 2.8, except that a separate data segment is used to store the data.

Example 2.9a

```

        .MODEL SMALL      ;choose small model
0000          .DATA      ;start data segment
0000 50      NUM1 DB    50H
0001 210A    NUM2 DW    210AH
0003 0789    NUM3 DW    0789H

        .CODE      ;start code segment
        .STARTUP   ;start of program

0017 A0 0000 R    MOV AL, NUM1
001A 8B 1E 0001 R  MOV BX, NUM2
001E 8B C8       MOV CX, AX
0020 8B 1E 0003 R  MOV BX, NUM3
        .EXIT
        END

```

The salient points of this program listing are:

1. The word R on code lines with addresses says that the addresses are **re-locatable**. This can be observed in the listing of Example 2.8 as well.
2. In the data segment, data locations have been given labels. Data will be stored at offsets as shown in Fig. 2.1 (recollect the **little endian** concept).

LABELS	OFFSET	DATA(H)
NUM3+1	0004	07
NUM3	0003	89
NUM2+1	0002	21
NUM2	0001	0A
NUM1	0000	50

Figure 2.1 | Data segment with labels and offsets corresponding to Example 2.9

Now let us see the same program listing with the .LISTALL directive inserted before the startup directive. This will show the conversion of .STARTUP and .EXIT to a number of instructions of 8086. What is to be noted is that the startup directive corresponds to instructions necessary to initialize the segment registers (except CS, which is done by the OS alone). Compare this with Example 2.12. However, if all this confuses you, leave it aside for the time being and come back to it when you become more proficient in ALP.

Example 2.9b

```

        .MODEL SMALL      ;choose small model
0000          .DATA          ;start data segment

0000 50          NUM1 DB   50H
0001 210A        NUM2 DW   210AH
0003 0789        NUM3 DW   0789H

        .LISTALL
0000          .CODE          ;start code segment
        .STARTUP      ;start of program

0000          *@Startup:
0000 BA ---- R    *  mov   dx,  DGROUP
0003 8E DA        *  mov   ds,  dx
0005 8C D3        *  mov   bx,  ss
0007 2B DA        *  sub   bx,  dx
0009 D1 E3        *  shl   bx,  001h
000B D1 E3        *  shl   bx,  001h
000D D1 E3        *  shl   bx,  001h
000F D1 E3        *  shl   bx,  001h
0011 FA           *  cli
0012 8E D2        *  mov   ss,  dx
0014 03 E3        *  add   sp,  bx
0016 FB           *  sti

0017 A0 0000 R      MOV AL, NUM1
001A 8B 1E 0001 R    MOV BX, NUM2
001E 8B C8          MOV CX, AX
0020 8B 1E 0003 R    MOV BX, NUM3

        .EXIT
0024 B4 4C          *  mov   ah,  04Ch
0026 CD 21          *  int   021h
        END

```

2.3.5 | The DUP Directive

This directive is used to replicate a given number of characters. For example, we may want to fill up a number of locations in the data segment with the same word or byte.

NUMS DB 10 DUP(0) fills up with os the 10-byte locations starting with the label NUMS.

STARS DB 5 DUP('#') fills up 5-byte locations starting at location STARS, with the ASCII value of the character #.

BLANK DB 10 DUP(?) reserves 10-byte spaces starting from location with the label BLANK, but these are not initialized, which implies that whatever data is there will remain. We can overwrite these locations with

new data when we choose to. WRDS DW 4 DUP(FFoFH) fills up 4 word locations with the word FFoFH.

2.3.6 | The EQU Directive

This directive allows us to equate names to constants. The assembler just replaces the names by the values mentioned. Examples are:

```
TEMP    EQU    34
PRICE   EQU    199
```

Example 2.10

```
.MODEL SMALL
0000          .DATA
0000 45        ONE DB 45H
0001 00        TWO DB ?
0000          .CODE
              .STARTUP
= 0025          FACTR EQU 25H
0017 A0 0000 R  MOV AL, ONE      ;copy ONE to AL
001A 04 25      ADD AL, FACTR    ;add FACTR to AL
001C A2 0001 R  MOV TWO, AL     ;move the sum to TWO
              .EXIT
              END
```

Example 2.10 shows the use of the EQU directive. From this list file, we see that the label FACTR is replaced by 25H when the program is being executed.

2.3.7 | ORG Directive

ORG is a directive, which means origin. In the context of assembly language programming, it can change the location of data storage or code in memory, i.e., the programmer gets the freedom to decide the offset of data or code when it is stored. The format is ORG d16, where d16 means a 16-bit displacement. Let us try out the usage of this directive.

Example 2.11a

```

.MODEL SMALL
.DATA

NUMS DB 20H, 40H
NUMS1 DW 8907H, 0FDH

.CODE
.STARTUP
MOV AL, NUMS           ;this is an instruction
MOV BL, NUMS+1          ;this is an instruction
MOV DX, NUMS1           ;this is an instruction

.EXIT
END

```

Example 2.11b

```

.MODEL SMALL
.DATA

ORG 0010H           ;origin to be at 0010H
NUMS DB 20H,40H
ORG 0030H           ;origin to be at 0030H
NUMS1 DW 8907H,0FDH

.CODE
.STARTUP

MOV AL,NUMS          ;this is an instruction
MOV BL,NUMS+1         ;this is an instruction
ORG 0040H            ;org is a directive
MOV DX,NUMS1          ;this is an instruction

.EXIT
END

```

If we compare Examples 2.11a and 2.11b, then we will see that the data at NUMS and NUMS1 in the second program is placed in different locations offset by the amount suggested by the ORG directive. The same applies to the **code line** MOV DX, NUMS1 as well. The new locations can be verified using the debugger.

2.3.8 | Other Models

These are other models available to be used according to our requirements. Table 2.2 will give an idea of those models and their specifications. There is another model called the flat model, which uses 32-bit addressing and un-segmented memory and is used in the protected mode. From the above, it is clear that all the models are not for 8086 alone, but are the features of MASM, which caters to the whole x86 family.

Table 2.2 | The Different Memory Models Used by MASM

SI No	Name	How many code segments?	How many data segments?
1	Tiny	1	Nil
2	Small	1, max size 64K	1, max size 64K
3	Medium	Any number, any size	1, max size 64K
4	Compact	1, max size 64K	Any number, any size
5	Large	Any number, any size	Any number, any size
6	Huge	Any number, any size	Any number, any size

2.3.9 | Full Segment Definition

Now let us see the traditional model of MASM. In the simplified memory model, it is left to the loader software to initialize the segment registers. In the traditional model, we use directives to define segments, and instructions to initialize the segment registers. This is called the **Full Segment** definition. The CS and SS registers are automatically initialized by the loader, but the DS and ES (if the extra segment is being used) will have to be initialized by the programmer. Let us re-write Example 2.9 using the full segment definition.

Example 2.12

```

DAT           SEGMENT          ;start the data segment
NUM1         DB    50H
NUM2         DW    210AH
NUM3         DW    0789H
DAT           ENDS            ;end the data segment

COD           SEGMENT          ;start the code segment
ASSUME        CS:COD,DS:DAT
MOV AX, DAT      ;move DAT to AX
MOV DS, AX       ;copy AX to DS
MOV AL, NUM1
MOV BX, NUM2
MOV CX, AX
MOV BX, NUM3
MOV AH, 4CH
INT 21H

COD           ENDS            ;end the code segment
END             ;end of program

```

The salient features of this model are:

1. The data segment has been given a name DAT. The data within the segment is enclosed between the SEGMENT and ENDS directives which are similar to parentheses for a segment.
2. Similarly, the code segment is named COD and the contents of this segment also have been enclosed between the same directives (segments can have any name).
3. The DS register has been initialized by the first two instructions. DS cannot use immediate addressing. Hence, DAT, which corresponds to a number, has to be loaded to AX (or any other 16-bit register) and then transferred to DS.
4. The value of DAT is loaded into DS. This means that DAT corresponds to the base address of the data segment. However, only when the loader loads this program in memory, will the actual base address (i.e., the upper 16 bits of the base address) be loaded into the DS register. At this point, the assembler just recognizes the ‘reference’ to a segment named DAT.
5. The CS register need not be initialized in the program. This is done by the loader by default.
6. The program is terminated by using the INT 21H instruction with function number 4CH (DOS function calls will be discussed in detail later).
7. ASSUME is a directive which relates each segment to its segment register, by specifying the name.

However, the simplified dot models will be used throughout this text book, except when special reasons call for the use of the full segment definition. In older versions of MASM, before the advent of the .startup directive, segment registers had to be initialized with the instructions as below:

`MOV AX, @DATA`

`MOV DS, AX`

where DATA is used at the start of the data segment.

2.3.10 | General Rules for Writing Assembly Language

Now, let us summarize the general format for an assembly language line, with an example.

Example 2.13

LABEL	OPCODE	OPERAND	COMMENT
NUM1	DB	56H	;define NUM1 as a byte of 56H
NUM2	DW	0557AH	;define NUM2 as a word of 0557H
START:	MOV	BL, NUM1	;copy NUM1 into BL
	MOV	AL, BL	;copy BL into AL

An assembly language line has four fields, namely, label, opcode, operand and comment. A label is positioned at the left of a line and is the symbol for the memory address, which stores that line of information. There are certain rules regarding labels that are allowed under MASM.

1. All labels should begin with a letter or the special characters @, \$, _, and ?
2. A label can have 1 to 31 characters, which may be digits, letters in upper or lower case or the special characters at (@), dollar (\$), underscore (_), dot(.) or question mark (?).
3. No reserved words of MASM may be used.
4. Each label must be unique.

The second field is the opcode or instruction field. The third is the operand field, and the last is the comment field which starts with a semicolon. The use of comments is advised for making programs more readable.

2.4 | Instruction Design

Until now, we were discussing assemblers, the best assemblers available and how to run our programs using assemblers and other program development tools. Now, let us go a bit deeper into the core of the processor and investigate the process of how machine codes have been designed. In doing so, we will realize why the assembly process is a ‘one-to-one’ process.

2.4.1 | Manual Coding

Another idea to think of is the possibility of doing manual or hand coding, as it is called—taking an assembly instruction, looking up or finding out its machine code, and feeding it directly to the processor. Why not? It looks quite feasible, doesn’t it? Actually it is, but very soon it becomes obvious that it is too cumbersome to think of programming as fun.

During the time that the 8085 was a popular processor, hand coding was the rule rather than the exception. This was because 8085 had a limited instruction set and a look up table was sufficient for hand coding.

The scenario has changed with the 8086, which has more registers and more instructions, but the most notable fact is that it has many more modes of addressing, which implies that each instruction can be expressed in too many modes to facilitate the use of a look up table. With the more advanced of the x86 processors coming to the scene, the difficulty has multiplied many fold. As such, we will conclude that using a standard and good assembler is the best solution.

However, it will be very instructive and informative to understand how and why a particular mnemonic has been converted to a particular code. Let us understand the underlying principles of the Instruction Set Architecture of 8086.

2.4.2 | Instruction Set Architecture (ISA)

The following statement gives an almost appropriate definition of what is meant by ISA. An instruction set, or instruction set architecture (ISA), is defined as the part of computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), i.e., the native commands implemented by a particular CPU design.

It must be noted that processors with the same ISAs need not have the same internal designs (micro architectures)—they only need to share a common instruction set. That is why we find that AMD64 and P64 have the same ISA but they are not the same internally. The ISA is an interface between a higher level language and the machine language.

2.4.3 | Instruction Set Design of 8086

Now let us get to the 8086 processor, which is a CISC processor. One feature of CISC is that its instruction size varies—all instructions do not have the same size. 8086 has instruction size varying from one byte to five bytes. This makes the processes of assembly, disassembly and instruction decoding complicated because the instruction length needs to be calculated for each instruction. An instruction should have the complete information for fetching and executing an instruction. It should thus have the following information:

1. Opcode corresponding to the operation to be carried out.
2. Size of the operands.
3. Addressing mode of each operand mentioning one or more of the following:

1. General-purpose register.
2. Value of an immediate operand.
3. Address of operand.
4. Base register.
5. Index register.
6. Combination of a base and index register.
7. g) Displacement in combination with base/index/[base + index] registers.

See the general format of the first three bytes of an instruction (see Fig. 2.2).

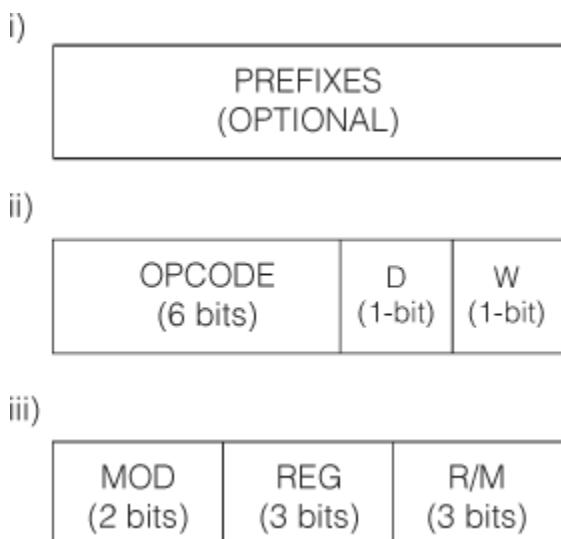


Figure 2.2 | Format of three bytes of an instruction

Prefix

This is an optional byte and need to be used only to change the operation, e.g. segment override prefix.

First byte

This is considered to be the first byte of an instruction. In this byte, there is the operation code (opcode), which is 6 bits long. This is the code which defines the operation to be carried out. The other two bits are the D and W bits.

W (1-bit)—operand size. W = 1, means word operand; W = 0, means byte operand.

D (1-bit)—Direction bit. D = 1, means register is destination; D = 0, means register is source.

Second byte

MOD (2-bit)—Register bits.

REG (3-bit)—the identifying code of the register used.

R/M (3 bits)—Specifying a register or memory operand.

The MOD and R/M bits together specify the **addressing mode** of the instruction.

Note All instructions need not have the W and D bits. In such cases, the size of the operand is implicit, and the direction is irrelevant.

2.4.4 | Designing a Code

The design of a code requires more information than has been presented so far. We need the identifying codes of the registers, obviously. Each 16-bit and 8-bit register has a unique code. These codes are given in Tables 2.3 and 2.4. In addition, a table showing the MOD and R/M bits corresponding to various combinations of addressing modes is also given. Using these, code design will be quite simple. We also need the Intel manual for the opcodes and formats of instructions. This is given in Appendix A.

Table 2.3 | Codes of General-Purpose Registers

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Table 2.4 | Codes of Segment Registers

Segment regs	Code
ES	00
CS	01
SS	10
DS	11

With the help of the above tables, let us try designing a code for a few simple instructions.

i) MOV AX, BX

Coming to our task, first look in the Intel manual for the opcode of MOV, with the specification ‘Register/Memory to/from Register’. It is seen to be 100010. This will correspond to the first 6 bits of the first byte. W = 1, as the operand is a word. The direction bit D has to be defined. In this case, both the source and destination are registers. Let us view it as a word being moved to AX, i.e., register AX is thus the destination. Hence, D = 1. Then we must use the code of AX in the register field of the second byte. The first byte is

OPCODE						D	W
1	0	0	0	1	0	1	1

Byte 1

To design the second byte, first look in Table 2.5 for the MOD and R/M bits. We find that the MOD bits are 11 (last column of the table) for register addressing, the reg bits are 011 for AX (from Table 2.3) and the R/M bits are treated as the ‘second register’ field. Here, the second register is BX, and the code of which is 011 (Table 2.5 4th row). Thus, the second byte is

MOD		REG			R/M		
1	1	0	0	0	0	1	1

Byte 2

Table 2.5 | Codes Pertaining to Addressing Modes – MOD and R/M Bit Patterns

R/M	MOD	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	(BP) + (DI) + d16	BL	BX
100		(SI)	(SI) + d8	(SI) + d16	AH	SP
101		(DI)	(DI) + d8	(DI) + d16	CH	BP
110		d16 (direct address)	(BP) + d8	(BP) + d16	DH	SI
111		(BX)	(BX) + d8	(BX) + d16	BH	DI
		MEMORY MODE			REGISTER MODE	

Note d8 and d16 are 8-bit and 16-bit displacement, respectively.

There is no other information needed in coding this instruction. This is a two-byte instruction with the code 8B C3 H.

ii) ADD BL, CL

OPCODE						D	W
0	0	0	0	0	0	1	0

Byte 1

MOD		REG			R/M		
1	1	0	1	1	0	0	1

Byte 2

The opcode of ADD with the specification of 'Register/Memory with Register to either' is 00000. W = 0, since it is a byte transfer. D = 1 considering BL as the destination register. For the second byte, MOD = 11. Register code for BL is 011, R/M bits corresponding to CL is 001. The code thus is 02 D9 H.

iii) MOV CX, 0213H

This is an instruction in the immediate mode. Two bytes are necessary for the immediate data, which is 16-bit. Now, let us look at Intel's manuals on the format of MOV in the immediate mode 'to register'. It is shown as in Fig. 2.3.

1011 W	Reg	data	data if W = 1
--------	-----	------	---------------

Figure 2.3 | Format of a MOV instruction in the immediate mode

Here W = 1, Reg (CX) = 001. Hence, the coding of this instruction is 1011 1001 as the first byte followed by the word data in the little endian scheme as 13 02 H. The complete instruction is B9 13 02H.

iv) MOV DX, [23CDH]

OPCODE						D	W
1	0	0	0	1	0	1	1

Byte 1

MOD		REG			R/M		
0	0	0	1	0	1	1	0

Byte 2

The first byte is 8BH. The second byte has MOD = 00 (Column 2 in Table 2.5) and R/M bits are 110 (corresponding to direct addressing in Table 2.5). Thus, the second byte is 16H. This is followed by the direct address in little endian form. The 4-byte machine code is 8B 16 CD 23 H.

OPCODE						D	W
1	0	0	0	1	0	0	0

Byte 1

MOD		REG			R/M		
0	0	1	1	0	1	1	1

Byte 2

For the second byte, MOD = 00 corresponding to [BX]. The register bits are 110 for DH and R/M = 111 from Table 2.5. Thus, the instruction code is 88 37H.

vi) MOV AL, [SI][BX]

OPCODE						D	W
1	0	0	0	1	0	1	0

Byte 1

MOD		REG			R/M		
0	0	0	0	0	0	0	0

Byte 2

By following the same procedure as discussed for the previous instructions, the code for this is 8A 00 H.

OPCODE						D	W
0	0	0	0	0	0	1	1

Byte 1

MOD	REG			R/M		
0	1	0	0	0	0	0

Byte 2

The third byte of the instruction is 53H, being the 8-bit displacement given in the instruction. Thus, the code is 03 40 53 H.

Segment Override Prefix

viii) MOV CS: [BX], DH

The segment override prefix is a byte with the format 001XX110. In place of XX, the code of the segment register which overrides the default segment has to be inserted. For the above instruction, the prefix byte is 00101110, using the code of CS which is 01 (Table 2.4). We have already coded the instruction MOV [BX], DH. Appending the prefix to it, the three-byte code is now 2E 88 37 H.

Now, to confirm if the ‘hand coding’ we have done is correct, let us put it all in a program and examine the list file. Example 2.14a is the program and 2.14b is the unassembled file from the debugger, which contains the assembly code as well as the machine code. On checking it, we find that our hand coding indeed agrees with the assembler’s coding.

Example 2.14a

```

    .MODEL SMALL
    .DATA
ORG 23CDH
DAT DW 0000
.CODE
.STARTUP
MOV AX, BX
ADD BL, CL
MOV CX, 0213H
MOV DX, DAT
MOV [BX], DH
MOV AL, [SI][BX]
ADD AX, 53H[SI][BX]
MOV CS:[BX], DH
MOV AX, CS:[BX]
.EXIT
END

```

Example 2.14b

0B56:0017 8BC3	MOV	AX, BX
0B56:0019 02D9	ADD	BL, CL
0B56:001B B91302	MOV	CX, 0213
0B56:001E 8B16CD23	MOV	DX, [23CD]
0B56:0022 8837	MOV	[BX], DH
0B56:0024 8A00	MOV	AL, [BX + SI]
0B56:0026 034053	ADD	AX, [BX + SI + 53]
0B56:0029 2E	CS:	
0B56:002A 8837	MOV	[BX], DH

In the debugger, the unassembled instructions are shown above, with the addresses (offsets within the code segments) of the instructions also shown. For example, in the line
0B56:0019 02D9 ADD BL, CL

0B56 is the CS value, 0019 is the offset of the instruction, and 02D9H is the machine code of the instruction ADD BL, CL.

Notice how the address of the next instruction depends on the size (number of bytes) of the current instruction.

Now that we have the procedure of coding for a few instructions, it can be concluded that manual coding is an alternative in the absence of an assembler, but it is not a good practice to rely on it. Using a good assembler is recommended for fast and efficient use of assembly code. However, knowing the methodology of coding does help us to get an idea of the way in which data, registers and instructions are handled by the processor.

Note

All the instructions we have coded in this section are at least two bytes in length. However, there are some instructions that have only one byte—examples are STI (set interrupt flag) and CLD (clear direction flag). Try to find out why they are only one byte long.

KEY POINTS OF THIS CHAPTER

- An assembler is a software that translates assembly language into machine code on a one to one basis
- Assemblers convert labels to memory addresses
- An assembly language program contains instructions and directives—the former produces executable code but the latter only gives directions to the assembler
- For an assembler, having to know the memory address corresponding to a label that has not yet been defined is called the ‘forward reference’ problem
- Assemblers can be one pass, two pass or multi-pass. Each pass refines the optimality of the assembly process
- A number of assemblers are available for x86, of which MASM is currently one of the most popular ones
- MASM 6.14 and MASM 32 are used in this book
- The tiny model is a memory model which is compact and allows only one segment. The run file generated herein is a com file
- The small model needs a data segment as well. It generates an exe file as the run file
- Data types have to be defined as bytes, words and so on, before labels are used for data
- Programs can also be written using the conventional ‘full segment definition’
- Hand coding of instructions is possible if all the information pertaining to instruction formats, mode, register codes and opcodes are available

QUESTIONS

1. The translation from assembly code to machine code is a one-to-one process, but the translation from a high level language to machine code is not so. Explain why.
2. List the steps involved in converting source code to executable code, using an assembler and other program development tools.
3. What are the roles of the linker and loader in the generation of the run file?
4. How is a com file different from an exe file?
5. ‘The assembled code is re-locatable’. What is the implication of this statement?
6. Why does a .com program have the origin at 0100H?
7. Is MASM a high level assembler (HLA)? Comment.

8. How does a debugger help in the programming process?
9. What exactly does the .EXIT statement do?
10. List the rules regarding labels that can be used in MASM.

EXERCISES

1. Using the tiny model, write a program that does the following.
 1. Moves 1020H to AX, 056H to BL and C76H to CX.
 2. Copies the content of AX to DS, the content of BH to AH, and the content of CX to DX.
 2. Write a program to copy 25 to AL, 'S' to BL and '*' to CL. Then move these to AH, BH and CH, respectively.
 3. Using the small model, store the bytes 56H and FCH, and words 0978H and CFD4H in the data segment.
 4. Store three bytes and four words in the data segment. Copy one byte and one word of these to appropriate registers.
 5. Using the small model, write a program which incorporates the following addressing modes.
- o. register mode
1. immediate mode
 2. register indirect mode.
6. Write a program which adds a constant correction factor to a temperature that is stored in the data segment. The corrected temperature should be stored in a new location in memory.
 7. Using the DUP directive allocate space for 10 data words, 5 double words and 2 quad words in the data segment. Also fill the next 12 spaces with 0, the next 10 spaces with '\$' and the next space with 'S'
 8. Using the full segment definition, write a program to add the content of two words which are in memory, and store the sum in a third address.
 9. Design the machine codes for the following instructions.
- o. XOR AL, AL
1. AND AX, BX
 2. OR AL, BL
 3. MOV DS, AX
10. Find the machine codes for the instructions
- o. SUB AL, [5676H]
1. CMP AX, 0CDE2H
 2. CMP 34H [DI][BX], DL
 3. ADD CX, [SI]

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

3

Programming Concepts—II

IN THIS CHAPTER, YOU WILL LEARN

- Good programming practices.
- How to use a few important DOS functions in programs.
- A part of the instruction set of 8086.
- To use data transfer instructions in different modes of addressing.
- The concept and importance of branch instructions.
- To use unconditional and conditional branch instructions.
- To use the arithmetic instructions catering to unsigned numbers.
- To use the multiply and divide instructions for interesting applications.
- The use of the logical, shift and rotate instructions of the 8086.

3.1 | Approaches to Programming

We are now in a position to feel that we have understood to a small extent the tools needed for programming (in this case assembly programming). However, before getting down to the actual coding process, it might be worthwhile to spend some time trying to understand the various approaches to be used in programming. These ideas apply to all levels of programming, i.e., whether we use high level languages or assembly level languages.

One common argument is whether programming is an art or a skill. To say that it is an art implies that good programmers are born, not made. This may be partly true, because it is seen that some people have a natural flair for programming. However, this applies to all abilities we have. Even if programming skill is an inborn ability, one has to polish it and practice this craft to become good at it. There is no other way but hard work to become an expert.

Coming to the other side of the argument that programming is a skill – then also, the same rules apply. The craft of programming has to be learned with sound basics – we should develop our thought processes to become good problem solvers. For every problem, there is more than one way of solving it. Finding the optimal solution depends to a great extent

on how much we have learned to build castles in the air – where these castles are the solutions to our problems.

Let us conclude that just anyone can become a reasonably good programmer by getting a good grasp of the basics and following it up with keen studying and continuous practice.

There are various approaches to solve a programming problem. One is called the **top down** approach. This approach tends to look at the problem as one whole functional block. The solution to the problem is planned after the whole problem is thoroughly understood. This functional block is then broken down to smaller blocks, whose functions are to be defined along the way.

The other approach is the **bottoms up** approach. Here, the basic functional blocks which are available or can be clearly defined are joined together to get the bigger module. Thus, small function blocks are integrated to form the top level entity. Both these approaches have their advantages and disadvantages. Depending on the type of problem at hand, the methodology has to be decided.

Now, a Tip and Advice Before Getting Started

Something that is to be avoided is to jump to ‘coding’ as soon as a problem is presented. We must first build the solution to our problem, optionally in our mind – but the best way is to write a pseudo code or draw a flow chart before writing the actual code. In this way, our thought processes lead us to the algorithm to use. As we become more familiar with programming, we may gradually avoid pseudo codes or flow charts for small programs but use them for the more complex ones. In this book, only the coding part is done – it is left to the student to visualize or write the algorithm as he chooses.

Assembly Programming Tips

From the previous chapter, you might have got an insight into assembly programming using MASM. Now, we are in a position to get a better idea of the instruction set of 8086, in a phased manner.

In chapter 2, a few programs were run, but we did not have any mechanism to display the outputs. There was also no mechanism to take in data through the keyboard; i.e., interactive programming could not be done. There we used the debugger to access memory and view registers in order to verify if our results are as they should be. For entering data,

we were supposed to store it in the data part of the segment being used. For a programmer, this approach seems cumbersome. Interactive programming is definitely more fun and instructive. Let us see what it is.

What is DOS?

We are now in the computer age when our learning of computers starts with using the latest version of Windows. However, that was not so in the beginning of computer evolution. A number of operating systems were tried, tested and discarded before the graphical interface that Windows came to common use. Earlier, DOS was a very popular OS. It is still used and is very useful.

DOS stands for Disk Operating System which could be an acronym for any OS, but it is most often used as shorthand for MS-DOS (Microsoft disk operating system). Originally developed by Microsoft for IBM, MS-DOS was the standard operating system for IBM-compatible personal computers.

The initial versions of DOS were very simple and resembled another operating system called CP/M. Subsequent versions have become increasingly sophisticated as they incorporated features of minicomputer operating systems. However, DOS is still a 16-bit operating system and does not support multiple users or multitasking.

DOS is considered insufficient for many of the graphical operations that we use our computers for, but all Windows have a strong dependency on DOS. So, newer versions of DOS are inbuilt in all Windows OS. DOS is considered to control hardware directly. It is this aspect of DOS which will prompt us to use DOS and BIOS interrupts in this book. A few basic commands of DOS are given in Appendix B. They will help you to get started on MASM 6.14.

3.1.1 | BIOS and DOS Function Calls

Until now, we were viewing the 8086 processor as being connected only to memory, for example, the physical memory – RAM. However, in reality, the processor is also connected to various input and output devices. For example, the PC has the keyboard and mouse as input devices. The video monitor is an output device. As programmers, we would like to input data into the processor using the keyboard and get the results displayed on the video monitor. To do this, we need to have some way of accessing these I/O devices, which contain a lot of extra hardware. Luckily, there is software available to handle this problem,

and they come in the form of functions written to access Input and Output devices. The word BIOS may seem familiar. BIOS (Basic Input Output System) consists of a set of functions written for this purpose. Besides this, there is another set of functions called DOS functions or DOS interrupts. They are part of the DOS operating system. They are functions written for accessing input and output devices, which are called in the form of software interrupts. We have seen one such DOS - interrupt which is the one that the .EXIT statement gets translated to:

```
MOV AH, 4CH  
INT 21H
```

In this example, 4CH is the function number which has to be loaded into the AH register before calling the interrupt with type number 21H. This function caused exiting the program and returning to the DOS prompt.

In Chapter 8, we will see the logic of interrupts and their way of functioning. However, in this chapter, we will use a few DOS interrupts to aid us in the process of programming.

Using these simple DOS interrupts will help us get the feel of programming as we are able to use the available input and output devices interactively. We will see that assembly language programming can be fun. In this chapter, a few DOS function calls will be introduced. BIOS interrupts will be used in later chapters.

3.1.2 | Using DOS Function Calls

We will start with four important function calls, all of which are of interrupt type 21H.

i) Read the keyboard with echo

```
MOV AH, 01  
INT 21H
```

This call exits with the ASCII value of the key pressed being available in AL. The key pressed is also **echoed** on the screen.

ii) Read keyboard without echo

```
MOV AH, 08  
INT 21H
```

This call exits with the ASCII value of the key pressed being available in AL. The key pressed is **not echoed** on the screen.

iii) Write a character to the standard display unit

For this, the ASCII value of the character to be displayed should be in DL.

```
MOV DL, S'  
MOV AH, 02  
INT 21H
```

With this, the character S is displayed on the video screen.

iv) Display a character string on the standard display unit

The logical address DS : DX should point to the beginning of the string. This is to be followed by the following instructions:

```
MOV AH, 09  
INT 21H
```

3.1.3 | Instruction Set of 8086

The instruction set of the 8086 is reasonably large. Besides, each instruction can be used in various addressing modes making the whole set larger. Knowing all the instructions, their format and their features is a key factor in becoming an expert programmer. Let us list out the general features of instructions which will become clearer as they are explicitly earned.

1. Most instructions have two operands, i.e., the destination and the source. These can be registers or memory operands. However, both operands cannot be memory operands. Some instructions can also use immediate data as the source operand.
2. Certain instructions have only one operand mentioned which is either the destination or the source. These can be register or memory operands but not immediate data.
3. There are instructions that have no operands explicitly mentioned in the instructions. In that case, some register is implicit in the usage of these instructions.

To learn the instruction set, the approach will be to list instructions based on their functionality, learn their format and function, and then write a few programs using them.

3.2 | Data Transfer Instructions

Table 3.1 gives all the data transfer instructions of 8086, except the input and output instructions, which will be introduced in Chapter 5. A few of these instructions will be discussed in detail and used in programs here. The rest will be used in later chapters.

Table 3.1 | List of 8086 Data Transfer Instructions with Format and Function

SI No.	Instruction format	Function performed	Flags affected
1	MOV dest, src (Move)	Copy the contents of the source to the destination	None
2	LEA reg16, memory (Load Effective Address)	Load the offset of specified memory location in the (16-bit) register	None
3	XCHG dest, src (Exchange)	Exchange the contents of the destination and source	None
4	PUSH source (Push)	Transfer one word from source to the stack top (SS : SP).	None
5	POP destination (Pop)	Transfer word at the current stack top (SS : SP) to the destination	None
6	LDS reg16, memory (Load to DS and register)	Load far pointer to DS and register	None
7	LES reg16, memory (Loads to ES and register)	Load far pointer to ES and register	None
8	LAHF (Load AH with flags)	Copy bits 0–7 of the flag register into AH	None
9	SAHF (Store AH with flags)	Transfer bits 0–7 of AH into the flag register	None
10	PUSHF (Push flags)	Push the flag register onto the stack	None
11	POPF (Pop flags)	Pop word from stack into the flag register	None
12	XLAT (Translate)	Replaces the byte in AL with byte from a user table addressed by BX	None

3.2.1 | MOV – Move

Usage: MOV destination, source

This instruction causes the source content to be copied to the destination (copying does not change the contents of the source). Remember that the

data types of source and destination should match – both should be either bytes or words.

```
MOV AX, CX           ;copy CX to AX – this is a word operation  
MOV AL, AH          ;copy AH to AL – this is a byte operation  
MOV AX, [BX] ;copy into AX the data in the address pointed by BX  
  
MOV COST, DX ;copy the word in DX to the location labeled COST
```

The use of the MOV instruction in the register and immediate addressing modes was covered in Chapter 2. Hence, we will use MOV in other modes of addressing now.

Example 3.1

```
.MODEL SMALL           ;select small model  
.DATA                 ;start data segment  
COSTP    DB   67H       ;store cost price  
SELLP    DB   ?         ;space for selling  
price  
                    .CODE             ;start code segment  
                    .STARTUP         ;start program  
  
PROFIT EQU 25H         ;define the profit  
                    MOV AL,COSTP      ;move COSTP to AL  
                    ADD AL, PROFIT    ;add PROFIT to AL  
                    MOV SELLP, AL      ;store the sum in SELLP  
                    .EXIT             ;exit to DOS  
                    END               ;end of program
```

Example 3.1 shows the use of the MOV instruction in the direct mode of addressing. This mode uses a memory location as one operand. Here two memory locations labeled COSTP and SELLP have been defined in the data segment. COSTP has a data byte stored in it, but SELLP is only allocated space for later storage. The first MOV instruction gets the data byte in COSTP into AL. After adding a predefined constant PROFIT to it, the sum in AL is stored in SELLP. Both MOV instructions use **direct addressing**. At this stage, you need to remember that the assembler will do the conversion of the labels to memory addresses.

Example 3.2

```

        .MODEL SMALL      ;select small model
0000          .DATA      ;start data segment
0000 10 20 30 40 50 ARRAY DB    10H, 20H, 30H, 40H, 50H
                                ;setup array
0000          .CODE      ;start code segment
                    .STARTUP ;start program
0017 BF 0000      MOV DI,0      ;load DI with 0
001A 8A 85 0000 R  MOV AL,ARRAY[DI] ;getfirstelement in AL
001E 04 07        ADD AL,07H    ;add 07 to it
0020 83 C705      ADD DI,05H    ;add 5 to DI
0023 88 85 0000 R  MOV ARRAY[DI], AL ;move sum to memory
                    .EXIT      ;exit to DOS
                    END       ;end of program

```

Example 3.2 shows the list file of a program which uses the MOV instruction in the **register relative addressing mode**. Let us analyze the program. The data segment is first initialized with 5 bytes. The address of the first location is labeled as ARRAY. DI is used to address the element numbers. With DI = 0, the first element is accessed, and with DI = 5, the sixth element is accessed. Then the effective address (EA) is [ARRAY + DI]. When DI = 0, the instruction MOV AL, ARRAY[DI], transfers the data in the first memory location to AL. To this, 07 is added. With DI = 5, ARRAY[DI] becomes the effective address of the sixth location in memory. Then MOV ARRAY[DI], AL stores the sum in memory. The displacement for the relative mode is the value of the label ARRAY. From the list file, it is seen the offset corresponding to ARRAY is 0000. Since ARRAY is the address of the first location in the data segment, the data segment will look like this, after the program is executed. The sum will be in the location ARRAY + 5.

EA	CONTENT
ARRAY + 5	17H
ARRAY + 4	50H
ARRAY + 3	40H
ARRAY + 2	30H
ARRAY + 1	20H
ARRAY + 0	10H

Example 3.3

```

        .MODEL SMALL      ;select small
0000          .DATA       ;start data
                           ;segment
0000 34 87 56 05 07 ARRAY  DB 34H, 87H, 56H, 05H, 07H
                           ;setup array
0000          .CODE       ;start code
                           ;segment
        .STARTUP      ;start program

0017 BB 0000 R      MOV BX, OFFSET ARRAY ;address ARRAY
001A BF 0000      MOV DI, 0           ;load DI with 0
001D 8A 01        MOV AL, [BX + DI]   ;get first data
001F 04 35        ADD AL, 35H       ;add 35H to it
0021 BF 0005      MOV DI, 05        ;address new
                           ;location
0024 88 01        MOV [BX + DI], AL  ;move sum here
                           ;exit to DOS
        .EXIT        ;end program
        END

```

Example 3.3 is the list file of a program that performs the same function as the previous example but uses the **based indexed mode** of addressing; i.e., the effective address is the sum of a base register and an index register. The instruction MOV BX, OFFSET ARRAY causes the offset of ARRAY (which is 0000 as seen from the list file) to be loaded into BX. Thus, BX now is a pointer to the location labeled as ARRAY. However, to point to each element one by one, another register DI is also used. With DI = 0, the instruction MOV AL, [BX + DI] causes the data in the first location to be copied to AL. With DI = 5, [BX + DI] accesses the sixth location in the data segment.

3.2.2 | LEA – Load Effective Address

Usage: LEA reg16, memory

This instruction causes the offset of the specified memory location to be loaded in the indicated register.

```

LEA SI, COSTP    ;load the offset of memory location COSTP in SI
LEA BX, ARRAY    ;load the offset of memory location ARRAY in BX

```

You may have noticed that the above is a replacement for the instruction MOV BX, OFFSET ARRAY used in Example 3.3.

Now let us use this instruction in an interesting application. We would like to display two character strings on two lines on the video monitor. It

was discussed in Section 3.1.2 that DOS function calls are needed for this. The following is the list file for the program which does this.

Example 3.4a

```
.MODEL SMALL
0000          .DATA
0000 48 45 4C 4C 4F 24    MESG1    DB    "HELLO$"
0006 0A 0D 49 20 41 4D    MESG2    DB    0AH, 0DH, "I AM SAM$"
                           20 53 41 4D 24
0000          .CODE
               .STARTUP
0017 8D 16 0000 R    LEA DX, MESG1 ;load offset of MESG1 in DX
001B B4 09           MOV AH, 09H   ;DOS function number for
001D CD 21           INT 21H     ;displaying a string
001F 8D 16 0006 R    LEA DX, MESG2 ;load offset of MESG2 in DX
0023 B4 09           MOV AH, 09H
0025 CD 21           INT 21H
               .EXIT
               END
```

Example 3.4b

HELLO

I AM SAM

C:\masm6.14\BIN>

Now let us analyze the program that does this, the listing of which is shown in Example 3.4(a). In the data segment, two messages are stored at offsets with labels MESG1 and MESG2. The character strings are enclosed in double quotes and terminated by the \$ sign. This sign is mandatory to end a character string which is to be printed on the display device. Notice that the listing in Example 3.4a shows the corresponding ASCII values for each character of the string. There is an ASCII value for ‘space’ which comes between each word of the second message, i.e., I AM SAM. It is seen to be ‘20H’. Note that ‘24H’ is the ASCII value of \$ and it is seen at the end of both strings.

On the second line, we also see two characters oAH and oDH. These are the ASCII values for line feed (which advances the cursor to the next line) and carriage return (sets cursor to the left position of screen). These

are necessary for printing to occur on the second line, starting from the left.

To print the first message, DX points to the offset MESG1. This is done by using the LEA instruction, which loads the offset of MESG1 in DX. Then the DOS function with number 09 is called. Similarly, for the second message, DX should point to the offset of MESG2. The two messages as they are printed on the command window are seen in Example 3.4(b). The .EXIT command causes control to return to DOS prompt

3.2.3 | XLAT – Translate a Byte in AL

Usage: XLAT

This instruction translates a byte in AL with a byte from a look-up table in memory. XLAT is an instruction which can handle look up tables and access data depending on the value of the pointing number. There should be a one-to-one relationship between the number and the data in the table. Before using XLAT, the look up table must be in memory and the offset of the table must be in BX.

Requirements

Load into BX the offset of the look-up table.

Then, use XLAT

Then, into AL, load the number which will extract the required data. Refer Table 3.2. For example, if you want to get the ASCII value of 5, load AL = 5. It will get the sixth entry from the table. See the look up table which relates a decimal number to its ASCII value. The ASCII values such as ‘0’, ‘1’, ‘2’ have numerical values 30H, 31H, 32H. Refer to the ASCII table in Section 0.6.5.

Table 3.2 | Look-Up Table for Decimal to ASCII Conversion

Number	ASCII
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'

Example 3.5

The following program gets into the memory location, starting from VALUES, the ASCII values of 1, 3, 5, 4, 0 ... in that order. The look-up table is stored in the array named ASC.

```
.MODEL SMALL
.DATA
ASC DB  '0', '1', '2', '3', '4', '5', '6', '7'
NUM DB  1, 3, 5, 4, 0
VAL DB  ?
.CODE
.STARTUP
    LEA BX, ASC      ;let BX point to offset of the table
    LEA SI, NUM      ;let SI point to NUM
    LEA DI, VAL      ;let DI point to VAL
    MOV CX, 5         ;CX = 5
AGN: MOV AL, [SI]      ;move the content of NUM to AL
    XLAT              ;translate
    MOV [DI], AL       ;move translated value from AL to VAL
    INC SI             ;increment the pointer to NUM
    INC DI             ;increment the pointer to VAL
    LOOP AGN          ;repeat until CX = 0
.EXIT
END
```

XLAT is equivalent to the following instructions:

```
MOV AX, 0
MOV SI, 0
MOV AL, [BX + SI]
```

SI or DI will be okay for showing the equivalence.

Try using XLAT for various look up tables; for example:

1. An integer and its square.
2. An integer and its cube.
3. A BCD number and its seven segment code.
4. A key and its ASCII or seven segment code.

3.2.4 | PUSH and POP

Usage: PUSH source

POP destination

For the 8086, only 16-bit data can be pushed and popped.

Note CS is not a valid destination for POP.

PUSH BX	;save the contents of BX to stack
PUSH [BX]	;save to stack the contents of the word pointed by BX
PUSH DS	;save to stack the contents of the segment register DS
PUSH COSTP	;save to stack the contents of the word location COSTP
POP CX	;load the contents of the stack top to CX
POP [SI]	;load the contents of the stack top to the memory location pointed by SI

The PUSH and POP instructions are relevant only for the stack segment. **So, scroll down and read about the operation of the stack before attempting to understand the PUSH and POP instructions.**

The stack is a region of temporary storage. When a subprogram is called, the register contents of the main program at that point of time and the current value of IP and CS are pushed onto the stack and retrieved on returning from the subprogram. These things are done automatically as part of a CALL instruction. Besides this, a programmer can opt to store in stack temporarily, the contents of some registers so that he can use those registers for other computations. He can get back the previous contents of these registers after he completes his current computation. These operations are accomplished by the PUSH and POP instructions.

3.2.4.1 | Operation of the Stack

The 8086, as mentioned in Chapter 1, uses a Last-In First-Out (LIFO) stack. A stack is an area of memory in which the lowest address is the base address of the segment. The upper 16 bits of this base address will be in the SS register. The highest address will be in the SP register—or rather, the SP will contain the offset of the highest address with respect to the base address. This is why it is called the TOP of STACK. For example, if the stack is defined from addresses 20000H to 203E8H, SS = 2000H and SP = 03E8H, this stack then has a size of 3E8H bytes (1000 bytes). In the 8086, only a word (16 bits) can be pushed on to stack. A push thus causes two memory locations to be accessed. This is shown in Fig. 3.1.

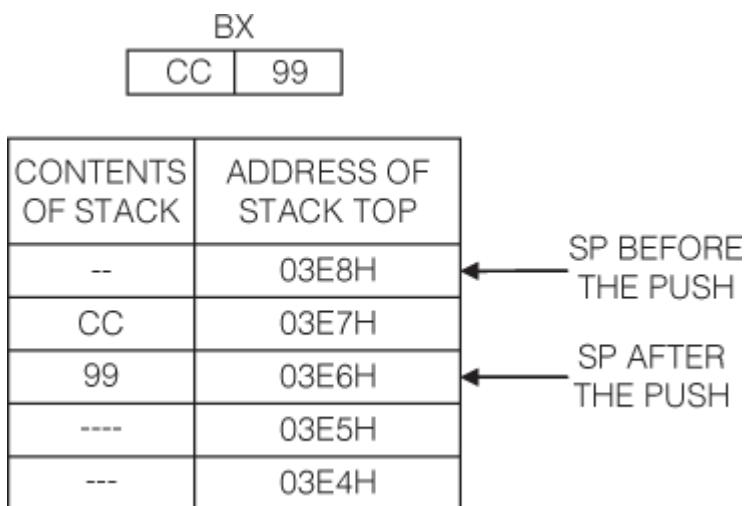


Figure 3.1 | PUSH operation in a stack

The operation of the stack is different from normal memory operations. It is a push-down memory. For example, the value of SP mentioned above, let us say, the PUSH BX instruction is to be executed. Let BX = CC99H. Recollect that SP = 03E8H. The operation of pushing is as follows. SP is first decremented by two. The new data will be put in addresses (offsets) 03E7H (SP-1) and 03E6H (SP-2), adhering to the ‘little endian’ concept; i.e., the higher byte of BX will be the higher address (03E7H) and the lower byte in the lower address (03E6H). The new value of SP will be 03E6H. This shows that new data to be pushed onto the stack will be pushed in the lower two addresses with respect to the current stack top. Always, the stack top (SP) has the offset corresponding to the last byte that was pushed in.

The reverse will be the case after a POP operation. Assume a POP CX operation at a time when the SP value is 0312H. The content of the stack will first be loaded into the CX register, and SP is incremented by two. See Fig. 3.2.

CONTENTS OF STACK	ADDRESS OF STACK TOP
--	0314H
0E	0313H
67	0312H
---	0311H
---	03E10H



Figure 3.2 | POP operation in a stack

3.2.4.2 | Defining a Stack

Now, how will we use the stack in programming? Obviously, we have to define elements including the stack size, base address of segment and the stack top. If we are using the tiny segment, this is done for by DOS – but in other memory models, we will have to do it ourselves. In case we do not, a warning ‘No stack segment’ will appear on assembling. This can be ignored if the stack size is less than 128 bytes. Otherwise, the stack that gets defined by DOS may erase the PSP (program segment prefix) and cause the system to crash.

Example 3.6a

```
.MODEL SMALL
.STACK 400H ; specify stack size
```

Example 3.6b

```
STACKSEG SEGMENT STACK
DW 200H
STACKSEG ENDS
```

Example 3.6 shows two ways of defining a stack. Example 3.6a uses the simplified memory model and it specifies the size of the stack as 400 bytes. Example 3.6b uses the full segment definition. 200 data words are to be set aside as stack, as per this definition. Here, both definitions create stacks of the same size.

Example 3.7

Explain what is done in this program. Assume SP = 0310H when the stack was initialized.

```
.MODEL SMALL
.STACK 300H
.CODE
.STARTUP
MOV AX, 4567H
MOV BX, 0ACEH
PUSH AX
PUSH BX
POP AX
POP BX
.EXIT
END
```

Solution

This program defines a stack of 300 bytes by the use of the stack directive. The program first loads two numbers in AX and BX. These numbers are pushed into stack using the PUSH operation, followed by POP operations. What we need to remember is that popping occurs in the reverse direction. What is pushed in last will be popped out first. The last PUSH was PUSH BX. It is obvious then that by the POP AX operation, the content of AX is replaced by the content of BX. Similarly, the second POP causes BX to get the content of AX. Thus, this program exchanges the content of these two registers, using the stack as a temporary data storage area. After execution, the contents of the registers are: BX = 4567H and AX = 0ACEH.

Note Exchanging register content is not really an application for the stack. We can exchange these register contents easily by using the instruction XCHG AX, BX. However, this program shows the relationship between POP and PUSH operations.

3.3 | Branch Instructions

Branch instructions are very important, because they carry on them the full power of a computer. Branching can be conditional or unconditional. Unconditional branching becomes necessary when looping back or forward is done infinitely, like when we use software to generate a square wave continuously. However, most of the time, conditional branching is what may be more important. Taking decisions based on the result of a computation is what gives computers their power and versatility. Most instructions that we will use subsequently become meaningful only when

conditional branching is used. In this chapter, we will discuss only the jump and loop instructions. Other branching instructions like call and return will be discussed in Chapter 4. We will start with the jump instruction.

3.3.1 | *JMP – Jump*

Usage: JMP destination

The destination is a memory location, which can be expressed in different ways as will be discussed presently. The jump destination is frequently referred to as ‘target’ as well. A jump instruction breaks the normal sequence of program execution and takes control to a different location in the code segment. There are many ways and restrictions in specifying the target value, and based on this, we can have different types of jumps. Figure 3.3 could be a typical scenario. When the ADD instruction is being executed, IP (Instruction Pointer) points to the next instruction which is an **unconditional** JMP instruction. On decoding the JMP instruction, the EU (Execution Unit) realizes that the next instruction should be taken from the location with label AGAIN. It is obvious, then, that the IP content must be changed in such a way that a new sequence is to be followed. The salient points regarding jump are:

1. The jump location may be in the same code segment, in which case it is **near** jump. For a near jump, the value of CS remains the same, but the value of IP will change. This is also called an intrasegment (within segment) jump.
2. The jump location may be in a different code segment – then it is called a **far** jump. In this case, both CS and IP will have to take on new values. This is an intersegment (between segments) jump.
3. The label of the destination address should be followed by a colon.
4. Jumping can be backward or forward in the program sequence.
5. Jumping can be unconditional. Then, the format of the jump instruction is JMP label, where label is the address of the destination.
6. Besides unconditional jumps, there are a number of conditional jumps available. The conditions are based on the state of the flags which get affected by arithmetic or logic instructions.

```

MOV.....  

ADD,-----  

JMP AGAIN  

.....  

-----  

-  

-----  

AGAIN: .....

```

Figure 3.3 | Control flow using the jump instruction

3.3.1.1 | The Near Jump

In the **direct mode** of addressing, the near jump has the format JMP label where the label is the address of a memory location. The destination can be a 16-bit signed number, which means that the range of jumping is within $+/-32K$ bytes. In direct jump, the jump location is a signed displacement from the current value of IP to the destination address (as shown in Fig. 3.4); i.e., the assembler calculates this displacement, and adds it to the current IP value. Thus, the jump address is relative or rather, relocatable. For a forward jump, this displacement will be positive, while for a backward jump, this will be negative. It uses three bytes as instruction size – one byte for the opcode and two bytes for the 16-bit destination offset.



NEW IP=CURRENT IP+OFFSET (16-bit)

Figure 3.4 | Format of the near jump instruction

3.3.1.2 | The Short Jump

The short jump is a special case of a near jump with the format JMP destination. See Fig. 3.5. For short jumps, the displacement must be an 8-bit signed number, which means that the jump destination can only be -128 or $+127$ bytes displaced from the current value of IP.



NEW IP=CURRENT IP+OFFSET (8-bit)

Figure 3.5 | Format of the short jump instruction

A directive SHORT can be used if we are sure that our jump destination is within the above specified range. This instruction is two bytes long – one-byte for the opcode and one-byte for the destination offset. Now let us see an address calculation for a short jump instruction.

Example 3.8

```
0000 B4 01      BEGIN:      MOV AH, 01
0002 CD 21          INT 21H
0004 8A D0          MOV DL, AL
0006 EB 07          JMP SHORT THERE
-----
-----, skipped instructions>-----
-----
000F 80 C2 02      THERE:      ADD DL, 02
0012 B4 02          MOV AH, 02
0014 CD 21          INT 21H
0016 EB E8          JMP SHORT BEGIN
```

Example 3.8 shows how ‘short’ jump instructions pass control from one part of the program to another.

0006 EB 07 JMP SHORT THERE

See the above line which is taken from the list file of the corresponding program. It shows the address (0006) at which this instruction is stored. The opcode of this direct JUMP instruction is EB and the ‘displacement’ is 07.

During the time that this instruction is decoded, the IP value is 0008 (address of the next instruction). At the beginning of the first pass, the assembler reserves one-byte space in memory for the address of THERE. The assembler knows that only one-byte is required for THERE, because of the directive SHORT. Otherwise it would have reserved two bytes space, as for a general near jump. If the address of the next instruction (0008) is added to the sign extended displacement (0007) of the jump instruction (this displacement is calculated by the assembler), the address of THERE is at location $0008H + 0007H = 000FH$. Thus, control branches to location 000FH.

Now, see the other direct jump instruction in the program

0016 EB E8 JMP SHORT BEGIN

The opcode is EB and the displacement is E8. JMP BEGIN is seen to be a backward jump as BEGIN is a label appearing much before this instruction line. The assembler calculates this displacement (which is negative) and then adds this displacement to the value of IP at that point of the instruction sequence. Then, it jumps to the destination BEGIN. In this case, the negative displacement is E8H. When JMP SHORT BEGIN

is executed, the IP content will be 0018H. For jumping to BEGIN, the address calculation is $0018 + FFE8 = 0000$, which is the address of the instruction with the BEGIN label. **Note that the negative number E8H, when used in 16-bit calculations, has to be sign extended. So, E8H is FFE8H in 16-bit sign extended form.**

Now let us write and run a fun program using an unconditional jump instruction. Example 3.9a is a simple program named UPLow which uses DOS interrupts to enter a character from a keyboard. If this character is an upper case alphabet, it converts it to lower case. The ASCII character table shows that a lower case character can be obtained by adding 20H to the uppercase code, i.e., the ASCII of 'A' is 41H and that of 'a' is 61H, and so on.

Example 3.9a

```
.MODEL TINY
.CODE
.STARTUP
START: MOV AH, 01H          ;AH = 01
        INT 21H           ;enter a key
        ADD AL, 20H         ;add 20H to it
        MOV DL, AL           ;move the sum to DL
        MOV AH, 02H          ;AH = 02
        INT 21H           ;use this function to display
        MOV DL, 0AH          ;0AH is the ASCII for newline
        MOV AH, 02H          ;display a new line
        INT 21H
        JMP START           ;go to start to repeat
.EXIT
END
```

Example 3.9b

```
C:\masm6.14\BIN>UPLow
Aa
Bb
Cc
Dd
Ee
Ff
Gg
Hh
^C
C:\masm6.14\BIN>
```

INT 21H with function number 01 is used to enter a key. Then, 20H is added to the ASCII value of the key, and this is displayed in the same line. To get further displays on separate lines, the ASCII character 0AH is displayed. Remember that 0AH corresponds to ‘newline’. The JMP START instruction is used to repeat this whole sequence indefinitely. Since there is no stopping condition for this loop, we have to use the ‘ctrl c’ combination of the keyboard to terminate the program. The displayed output is shown in Example 3.9b. Note that for any key input other than an uppercase character, the output is some random character.

3.3.2 | Other Forms of the Unconditional Jump Instruction

There are other modes of addressing for the near jump.

i) JMP reg16

This is called an indirect jump. The 16-bit register contains the destination address (offset). This jump is not a relative jump. IP gets replaced by the value in the register.

JMP BX	;jump to the destination address in BX. Make IP = BX
JMP SI	;jump to the destination address in SI. Make IP = SI

ii) JMP [reg 16]

This is like a double indirect jump. The register points to an address which contains the jump location.

JMP [SI]	;jump to the address which is stored in the memory location pointed by SI
----------	--

If SI = 0670H, the destination address is taken from the data segment at locations 0670H and 0671H. IP is replaced by the value at these addresses.

3.3.2.1 | Conditional Jumps

Conditional jumps are the best part of the idea of control transfer. They change the sequence of program execution based on the result of a computation which causes flag bits to be set or reset. However, there is one case (JCXZ) where the register content is checked.

Note All conditional jumps are short jumps.

Usage: J(condition) destination

Table 3.3 contains the jumps used for unsigned data, and those directly testing flags and registers.

Table 3.3 | List of Conditional Jump Instructions which Cater to Unsigned Arithmetic and which Directly Address Flags or Registers

SI No.	Mnemonic	What it means	Condition tested
1	JA	Jump if Above	CF = 0 and ZF = 0
2	JAE	Jump if Above or Equal	CF = 0
3	JB	Jump if Below	CF = 1
4	JBE	Jump if Below or Equal	CF = 1 or ZF = 1
5	JC	Jump if Carry	CF = 1
6	JCXZ	Jump if CX Zero	CX = 0
7	JE	Jump if Equal	ZF = 1
8	JNA	Jump if Not Above	CF = 1 or ZF = 1
9	JNAE	Jump if Not Above or Equal	CF = 1
10	JNB	Jump if Not Below	CF = 0
11	JNBE	Jump if Not Below nor Equal	CF = 0 and ZF = 0
12	JNC	Jump if No Carry	CF = 0
13	JNE	Jump if Not Equal	ZF = 0
14	JNP	Jump if No Parity	PF = 0
15	JNZ	Jump if Not Zero	ZF = 0
16	JP	Jump if Parity	PF = 1
17	JPE	Jump if Parity Even	PF = 1
18	JPO	Jump if Parity Odd	PF = 0
19	JZ	Jump if Zero	ZF = 1

Notes

1. Certain mnemonics are equivalent; for example, JZ and JE mean the same as they test the same flags. So do JC and JB, JP and JPE and so on. More can be identified from the list.
2. JCXZ tests the CX register but other Jump instructions test conditional flags only.
3. There is another set of conditional jump instructions which are specially meant for signed arithmetic. The list of those instructions and their usage will be dealt with in Chapter 4. We will use a few of the conditional jumps in the forthcoming examples.

3.3.2.2 | Far Jump

A far jump is an intersegment jump, which means that the destination address is in a different code segment. This will be a 5-byte instruction

(as shown in Fig. 3.6), the first byte being the opcode, the second and third the new values of IP, and the fourth and fifth the new values of CS. To specify that a jump is to a different code segment, a **far pointer** can be used in the format.

OPCODE	IP LOW	IP HIGH	CS LOW	CS HIGH
--------	--------	---------	--------	---------

Figure 3.6 | Format of the far jump instruction

JMP FAR PTR AGAIN ;PTR stands for ‘pointer’

Defining an address as external and then using a jump to this address is also possible. These ideas will be discussed in Chapter 5.

3.3.3 | The LOOP Instruction

Usage: LOOP label

There is a very useful and frequently used instruction called LOOP. This combines jump with a counter. The register CX is assigned to decrement every time LOOP executes. When CX = 0, the looping is exited.

```

MOV CX, N
MORE: .....
.....
.....
.....
LOOP MORE
.....
.....
```

Initially CX is loaded with a number, which we call ‘count’. See the above program sequence. What is to be understood here is that, the sequence of instructions up to the line containing the LOOP instruction, will execute once. Then, when the LOOP instruction is encountered, CX is decremented and then tested. If CX is found to be equal to zero, looping is exited and the next instruction in the sequence is taken up. If CX! = 0, control returns to the label MORE.

LOOP can be combined with other conditions to make it a conditional instruction. LOOPNE / LOOPNZ and LOOPE / LOOPZ can be used. These instructions test the zero flag as well as the value of CX. Examples 3.10a and b show the use of an unconditional LOOP instruction. In this program, three arrays have been declared in the data segment. Corresponding bytes from the first two arrays are to be added and the sum is to be saved in the third array. To point to the data items in the

three arrays, three registers are used. Data is accessed using register - indirect addressing. After one round of these actions, the pointer registers are incremented using the INC instruction. This instruction adds 1 to the registers. This sequence of operations is repeated six times by using the LOOP instruction. Once CX = 0, the loop is exited and the program ends.

Example 3.10a

```
.MODEL SMALL
.DATA
NUMS1 DB 45H, 67H, 89H, 65H, 34H, 23H
NUMS2 DB 09, 09, 12H, 13H, 08, 02
NUMS3 DB 6 DUP(0)
.CODE
.STARTUP
LEA BX, NUMS1 ;use BX to point to NUMS1
LEA SI, NUMS2 ;use SI to point to NUMS2
LEA DI, NUMS3 ;use DI to point to NUMS3
MOV CX, 06 ;CX = 6
REPEA: MOV AL, [BX] ;take into AL data from the array
                  NUMS1
        ADD AL, [SI] ;add to AL the data in array NUMS2
        MOV [DI], AL ;save the sum in the third array
        INC BX ;increment pointer
        INC SI ;increment pointer
        INC DI ;increment pointer
        LOOP REPEA ;repeat the sequence until CX = 0
.EXIT
END
```

This problem can be done with less number of address registers if the ‘register relative mode’ of addressing is used. See Example 3.10b. Here, only the register BX is used. Its content is added to different addresses so as to access the three different arrays.

Example 3.10b

```

        .MODEL SMALL
        .DATA
NUMS1  DB    45H, 67H, 89H, 65H, 34H, 23H
NUMS2  DB    09, 09, 12H, 13H, 08, 02
NUMS3  DB    6 DUP(0)
        .CODE
        .STARTUP
        MOV CX, 6
        MOV BX, 0
REPEA:  MOV AL, NUMS1[BX] ;point to the first array
        ADD AL, NUMS2[BX] ;add numbers of the second array
        MOV NUMS3[BX], AL ;save in the third array
        INC BX
        LOOP REPEA          ;repeat until CX = 0
        .EXIT
        END

```

Now, we will digress to the discussion of arithmetic instructions, but you can see that branch instructions in various forms will continue to be used.

3.4 | Arithmetic Instructions

The complete list of arithmetic instructions is given in Table 3.4. Instructions from 1 to 9 will be discussed in detail and used in this chapter. The rest will be dealt with in Chapter 4 for signed number arithmetic.

Table 3.4 | Full Set of Arithmetic Instructions

SI No.	Instruction format	Function performed	Flags affected
1	ADD dest, src	Add dest and src	AF CF OF PF SF ZF
2	ADC dest, src	Add dest, src and CF	AF CF OF PF SF ZF
3	INC dest	Add 1 to the destination	AF — OF PF SF ZF
4	SUB dest, src	Subtract src from dest	AF CF OF PF SF ZF
5	SBB dest, src	Subtract source and CF from dest	AF CF OF PF SF ZF
6	DEC dest	Subtract 1 from dest	AF — OF PF SF ZF
7	CMP dest, src	Subtracts source from destination and updates the flags but does not save result. Only flags are affected	AF CF OF PF SF ZF
8	MUL src	Unsigned Multiply	— CF OF — — —
9	DIV src	Unsigned binary divide	— — — — —
10	CBW	Convert byte to word	None
11	CWD	Convert word to double word	None
12	DAA	Decimal Adjust for Addition	AF CF — PF SF ZF
13	DAS	Decimal Adjust for Subtraction	AF CF — PF SF ZF
14	NEG dest	Two's Complement Negation	AF CF OF PF SF ZF
15	IMUL src	Signed Multiply	— CF OF — — —
16	IDIV src	Signed Integer Division	— — — — —
17	AAA	ASCII Adjust for Addition	AF CF — — — —
18	AAS	ASCII Adjust for Subtraction	AF CF — — — —
19	AAM	ASCII Adjust for Multiplication	— — — PF SF ZF
20	AAD	ASCII Adjust for Division	SF — — PF — ZF

Notes

- i) In the list, dest means destination and src means source. After an operation, the result is available in the destination.
- ii) As mentioned earlier, the destination and source may be both registers or a memory location and a register. The source can be immediate data, unless explicitly mentioned (for certain instructions) that immediate mode of addressing is not allowed.
- iii) Flags are affected by most of the arithmetic instructions. In certain cases, all flags are affected, but some flags are undefined. The — notation for the corresponding flags is used in such cases meaning 'undefined' (in the table).
- iv) In certain cases, flags are not affected. This is specified by 'none'.

3.4.1 | Flag Control Instructions

There are certain instructions which can be used to set/reset flags. This is very relevant in the case of control flags like direction flag, interrupt flag and so on. However, there is one conditional flag which can be set/reset using instructions, and that is the carry flag. The relevant instructions for it are:

1. CLC – Clear carry.
2. STC – Set carry.
3. CMC – Complement carry.

Now, we will see the format and usage of some of the important and commonly used arithmetic instructions.

3.4.2 | Addition Instructions

ADD – Add.

Usage: ADD destination, source.

This instruction adds the destination and source and puts the sum in the destination. All conditional flags get affected.

ADD AH, AL	;add AL to AH, sum in AH
ADD AL, COSTP	;add the byte in COSTP to AL, sum in AL
ADD BX, 0987H	;add the number 987H to BX, sum in BX
ADD CX, [BX]	;add the word in the location pointed by BX to CX sum in CX

ADC – Add with carry.

Usage: ADC destination, source.

This instruction adds CF and source to the destination, and puts the sum in the destination. There are three operands, of which the third is the carry. All conditional flags get affected.

ADC AH, 0	$AH = AH + 0 + CF$
ADC [BX], AL	;add the byte pointed by BX with AL and CF, put sum in the location pointed by BX
ADC AX, [BX][SI]	;add to AX, CF and the word with EA = BX + SI sum in AX

INC – Increment.

Usage: INC destination.

This instruction adds 1 to the destination. All conditional flags, **except the carry flag**, get affected.

INC BX	;add 1 to the content of BX
INC [BX]	;add 1 to the content of the memory location pointed by BX
INC AH	;add 1 to the content of AH

The PTR directive

This is a convenient point to introduce the PTR (pointer) directive. When the size of the operand is not implicit in the instruction, a pointer is used to indicate whether the operand is a byte, a word, or a double word. For example, see the instruction.

INC [BX]

Here, BX is a pointer to a location whose content is to be incremented. It is not clear whether the data being pointed to is a byte or a word or a double word. To make it clear, it will be helpful to modify the instruction as:

	INC BYTE PTR [BX]	;byte pointer
or	INC WORD PTR [BX]	;word pointer
or	INC DWORD PTR [BX]	;double word pointer

as the case may be. We will encounter various instances where this pointer is used.

Example 3.11

The following is a program which adds two bytes stored in memory. The sum of the bytes is likely to be greater than FFH. Hence, a word space needs to be allocated for the sum.

```
.MODEL SMALL
.DATA
NUMS DB 95H, 0FCH      ;store the two bytes
SUM DW ?                ;allocate a word space for the sum
.CODE
.STARTUP
MOV AX, 0                ;AX = 0
CLC                      ;clear the carry flag
MOV AL, NUMS              ;move the first number to AL
ADD AL, NUMS + 1          ;add the second number to AL
ADC AH, 0                 ;add AH and CF and 0
MOV SUM, AX                ;the sum in AX is moved to memory
.EXIT
END
```

First, AX is loaded with 0. Observe how a byte addition causes a result to be a word. The sum of two bytes which are added to AL is too large to fit in AL. Hence, it generates a carry. The next instruction adds CF, 0 and AH which contains a 0. So, we find that the carry bit is accommodated in AH. Thus, AH-AL, i.e., AX contains the sum which is a word.

Example 3.12 adds 10 bytes stored in memory. The sum of these ten bytes will not fit into a byte location. So, a word location is allocated for the sum. It must also be noted that the sum will definitely fit into a word location. [If there are ten bytes, the maximum number possible for the

sum of ten bytes is only $255 \times 10 = 2550$, which is less than 65,536 the maximum decimal number that a word can hold.]

Example 3.12

```
.MODEL SMALL
.DATA
NUMS DB 245, 178, 190, 167, 56, 178, 250, 89, 150, 235
SUM DW 0
.CODE
.STARTUP
CLC           ;clear the carry flag
LEA BX, NUMS ;BX to point to the numbers
MOV CX, 10    ;move the count to CX
MOV AX, 0     ;make AX = 0
REPEA: ADD AL, [BX] ;add the numbers, sum in AX
ADC AH, 0     ;add the carries into AH
INC BX        ;increment the pointer
LOOP REPEA   ;repeat if CX is not equal to 0
MOV SUM, AX   ;store AX in SUM
.EXIT
END
```

The above is a simple program which gives a lot of information. For one thing, note that data is written in the decimal form and not as hexadecimal bytes. However, if the list file is checked, it will show that the data is converted to hexadecimal format (in the machine code part of the listing). The program uses the LOOP instruction to carry out cumulative addition. CX is used as a counter, which decrements with each addition. The first ADD instruction adds two bytes. It is possible that a carry is generated out of this addition. This carry is added to AH, the other operand of which is 0. Thus, the sum of the carries of the ten addition operations will be in AH, which will form the upper byte of the sum. This sum is thus available as a word in AX which is saved in the word location SUM. It is important to clear the carry flag before using the ADC instruction, to avoid the possibility of any residual carry leading to a wrong result.

Example 3.13 is an interesting program which adds two double words. A doubleword is two words long, i.e., 4 bytes long.

Example 3.13

```

        .MODEL SMALL
        .DATA
LONGNUM1 DD 0F8FC6768H ;first double word data
LONGNUM2 DD 0C6EF2109H ;second double word data
SUM      DD 0           ;sum
SUMC    DB 0           ;space for carry out
        .CODE
        .STARTUP
        LEA SI, LONGNUM1 ;SI as pointer to first operand
        LEA DI, LONGNUM2 ;DI as pointer to second operand
        LEA BX, SUM      ;DX as pointer to the sum
        CLC              ;clear carry
        MOV CX, 2         ;CX to count the number of words
REPEA:  MOV AX, [SI]   ;move to AX the first word
        ADC AX, [DI]    ;add the second operand and
                         ;carry
        MOV [BX], AX     ;the lower word of result stored
        INC SI           ;increment SI twice to point to
        INC SI           ;next word operand
        INC DI           ;increment DI twice to point to
        INC DI           ;second word operand
        INC BX           ;increment BX twice to point to
        INC BX           ;second word of the sum
        LOOP REPEA       ;go back if CX is not 0
        MOV DL, 0         ;make DL = 00
        ADC DL, 0         ;add the carry to DL
        MOV SUMC, DL     ;move the carry to SUMC
        .EXIT
        END

```

There are a number of notable features for this program. Let us list out these features.

1. The double words are stored in two DD locations named LONGNUM1 and LONGNUM2. They will be stored in the little endian format.
2. The sum of two double words can be more than a double word. This can be accommodated as a carry, added to a register and can be stored in memory as the uppermost byte of the sum. Three registers are used as pointers to each double word. However, addition can be done only as words, as the maximum size of an operand is only 16 bits (called a word), for the 8086. Hence, first the lower order words of the two double words are added. There is a possibility of a carry due to this addition, which may have to be added to the upper word. Since addition is looped, the ADC instruction is used so as to accommodate the addition of the carry.
3. The address pointers have to be incremented by 2 to point to the upper word of the double word. **The INC instruction is preferred over the ADD instruction, as it does not affect the carry flag.** It is important

to preserve the carry bit out of the second word operation. This final carry is added to DL (which is first ensured to be 0). The numbers used herein are two double words whose sum is a double word plus a byte (01 BFEB 8871H). Table 3.5 shows the relevant portion of the data segment after execution of this program.

Table 3.5 | Relevant Portion of the Data Segment After the Execution of Example 3.13

Offsets in memory	Data	
000C	01	} SUMC
000B	BF	
000A	EB	
0009	88	} SUM
0008	71	
0007	C6	
0006	EF	
0005	21	} LONGNUM 2
0004	09	
0003	F8	
0002	FC	
0001	67	} LONGNUM 1
0000	68	

4. The memory locations and the corresponding data, after program execution is as shown (assuming the origin of the data segment is 0000).
5. The three arrays in memory can be accessed by using ‘register relative addressing’ as in Example 3.10b. Try to see if it gives any advantage in terms of reducing the number of instructions/registers used.

3.4.3 | Subtraction

SUB – Subtract.

Usage: SUB destination, source.

This instruction subtracts the source from the destination. The result is in the destination. All conditional flags are affected.

SUB AX, BX	;subtract BX from AX
SUB AL, [BX]	;subtract the byte pointed by BX from AL
SUB COST[SI], CX	;subtract CX from the word with EA = COST + SI
SUB AX, 8956H	;subtract 8596H from AX
SUB CL, BYTE PTR[SI]	;subtract from CL the byte pointed by SI

SBB – Subtract with borrow.

Usage: SBB destination, source.

This instruction subtracts the source and the carry flag from the destination. All conditional flags are affected.

SBB CH, 7

;subtract from CH, 7 and CF – result in CH

SBB AX, [BP + 2]

;subtract from AX, the word pointed by [BP + 2]. Since BP is used, the data is taken from the stack segment. The result is put in AX

DEC – Decrement.

Usage: DEC destination.

This instruction subtracts 1 from the destination. All conditional flags, except the **carry flag**, are affected.

DEC CL

;subtract 1 from CL

DEC WORD PTR [SI]

;subtract 1 from the word pointed by SI

DEC BYTE PTR NUMB[BX]

;subtract 1 from the byte pointed by the effective address NUMB + BX

Example 13.14

Explain what occurs on execution of the following instructions:

MOV CL, 0B5H

SUB CL, 0FCH

Solution

$$\begin{array}{r} \underline{\text{B5H}} & \underline{\text{181}} & \underline{\text{1011 0101}} & - \\ - \underline{\text{FCH}} & - \underline{\text{252}} & \underline{\text{1111 1100}} & \\ \underline{\text{B9H}} & \underline{\text{-71}} & \underline{\text{1011 1001}} & \end{array}$$

This is a subtraction of a bigger number from a smaller number. Hence, the carry flag is set to indicate that borrowing was required in the subtraction. The difference is -71 (decimal), which is represented as B9H which is the two's complement representation of decimal -71. The carry flag is set to indicate that the subtraction has used a 'borrow'. The sign flag is set to indicate that the MSB of the result is 1, which in this case is interpreted to be a negative number.

Now let us use the SUB instruction in an interesting example. Division is an operation which can be achieved by repeated subtraction. Many early microprocessors (8085, for example) had no divide instruction. The 8086 has a divide instruction. However, here, let us use repeated subtraction to divide, say, 100 by 9. We know that the quotient is 11 (OBH) and the remainder is 1.

Example 3.15

```
.MODEL SMALL
.DATA
QUOTIENT DB 0
REMAINDER DB 0

.CODE
.STARTUP
MOV AL, 100      ;copy dividend to AL
MOV CL, 0        ;CL, the quotient register = 0
REPEA: SUB AL, 9  ;subtract 9 from 100
JC OVER          ;stop when CF = 1
INC CL           ;increment CL if CF is not set
JMP REPEA        ;repeat subtraction if CF is not set
OVER: ADD AL, 9   ;add 9 to AL to get remainder
MOV REMAINDER, AL ;store remainder
MOV QUOTIENT, CL ;store quotient
.EXIT
.END
```

The steps of the program are as follows:

1. Subtract 9 from 100 repeatedly until a negative number is obtained. In the program, start with AL = 100. When AL becomes negative, the carry flag will be set (due to borrow). This is the stopping condition for subtraction.
2. In this problem, after subtracting 9 from 100 eleven times, AL will contain 01. One more subtraction will cause the content of AL to become -8 (F8H in 2's complement form). The carry flag is found to be set, and further subtraction is stopped.
3. To get the remainder, add 9 (the divisor) to AL. We get 1. This is the remainder.
4. Every time a subtraction is performed unhindered, increment a count in CL. This will give the quotient.
5. The quotient and remainder are stored in their allocated space in the data segment.

The SBB instruction can be used in instances when multibyte subtraction is done. For example, to subtract two double words or quad words, SBB

will have to be used, just as ADC is used (Example 3.13) for multi-byte addition.

We have seen so far addition and subtraction operations using numbers in the hexadecimal (essentially binary) format. However, numbers are also represented in other formats such as BCD and ASCII. There are special instructions like DAA and DAS that cater to BCD numbers, and instructions like AAA, AAS which cater to ASCII operations. In Section 4.4.2, these instructions will be used in arithmetic calculations using such numbers.

3.4.4 | Compare Instruction

CMP – Compare.

Usage: CMP destination, source.

This instruction compares the two operands and causes the conditional flags to be affected, but neither the destination nor the source changes. Comparison is done by a subtraction operation, and the flags are set/reset according to the result of this. However, only two flags really matter – they are the Zero flag and the Carry flag.

Consider the instruction CMP destination, source. The condition of the flags will be as shown in Table 3.6. It is thus obvious that following up a compare instruction with a JNC/JC or JNZ/JZ will do the trick. If the instruction is, say, CMP AL, BL, the way to picture it is:

if AL > BL, CF is reset
if AL < BL, CF is set
if AL = BL, ZF is set

Table 3.6 | Flag Settings After a Compare Instruction

If	CF	ZF
destination > source	0	0
destination < source	1	0
destination = source	0	1

If two unsigned numbers are to be compared, it will be clearer if we use JA/JB as the mnemonic following the compare instruction, instead of JC or JNC. This will make it easier to comprehend the meaning of the result of the comparison. If we want to compare AX and BX, we could think of it as: Jump to target if AX is above BX. The instructions to use are:

```
CMP AX, BX  
JA Target
```

Similarly, when testing for equality, we could use JE/JNE instead of JZ/JNZ.

Note These tips are only for enhancing the readability of the program. Now, let us write a few programs illustrating the use of the compare instruction.

Example 3.16

Two unsigned words are stored in the data segment in locations WORD1 and WORD2. The program compares the two numbers and displays a message stating which number is bigger.

```
.MODEL SMALL
.DATA
WORD1 DW      -----          ;store the first number
WORD2 DW      -----          ;store the second number
MES1 DB      "WORD1 IS BIGGER$" ;first message string
MES2 DB      "WORD2 IS BIGGER$" ;second message string
.CODE
.STARTUP
MOV AX, WORD1      ;copy WORD1 to AX
CMP AX, WORD2      ;compare AX with WORD2
JB SECOND          ;If AX < WORD2, go to SECOND
LEA DX, MES1        ;point DX to MES1
JMP DISP            ;jump to DISP
SECOND: LEA DX, MES2    ;point DX to MES2
DISP:   MOV AH, 09        ;AH = 09 for string display
        INT 21H           ;DOS function INT21H is called
.EXIT
END
```

This is a very simple program in which the WORD1 is brought into AX. It is then compared with WORD2. The jump (JB) instruction following the comparison directs control to the appropriate message. Remember that DOS function call with AH = 09 will display a string on the console.

Note To run this program, actual data will have to be written in the data segment at locations WORD1 and WORD2.

Example 3.17 finds the biggest of 10 bytes stored in memory.

Example 3.17

```

        .MODEL SMALL
        .DATA
NUMS      DB  56H, 38H, 09H, 98H, 99H, 0C7H, 07H, 0BCH, 0CH,
0ECH
BIGGEST   DB  ?           ;store the biggest number
        .CODE
        .STARTUP

        LEA BX, NUMS      ;BX is the pointer to the array
        MOV CL, 0AH         ;CL acts as the counter
        MOV AL, 0            ;AL = 0
REPEA:    CMP AL, [BX]     ;compare each number with AL
        JAE AGAIN          ;if AL is above or equal, jump
        MOV AL, [BX]         ;if below, bigger no. to be in AL
AGAIN:    INC BX           ;increment pointer
        DEC CL             ;decrement counter
        JNZ REPEA          ;repeat the sequence if count! = 0
        MOV BIGGEST, AL     ;the biggest number will be in AL
        ;store it in the location BIGGEST
        .EXIT
        END

```

The logic of this program is as follows:

1. The numbers are stored as an array in memory named NUMS. BX is used as a pointer to this array.
2. CL contains the count of the numbers (here CL = 0AH).
3. Initially AL = 0.
4. AL is compared with each number. If AL is not above or equal to the number in the array, the bigger number is brought to AL.
5. If AL is above the number in the array, the content of AL is retained as it is.
6. This is one round of comparison. Then, the pointer (BX) is incremented and count (CL) is decremented.
7. When the count reaches zero, ZF is set and the looping is terminated because of using the JNZ mnemonic.
8. The biggest number is now in AL which is stored in memory in the space allocated for it.

Example 3.18 is another interesting program, which searches a character string for the presence of a particular character and displays appropriate messages.

Example 3.18

```

.MODEL SMALL
.DATA
STRIN DB "HOWAREYOUMYBOY"      ;string of characters
LEN    DW 0EH
MESG1 DB 0AH, 0DH, "CHARACTER FOUND$"
MESG2 DB 0AH, 0DH, "CHARACTER NOT FOUND$"

.CODE
.STARTUP
LEA BX, STRIN    ;BX to point to the string
MOV CX, LEN      ;copy the string length to CX

MOV AH, 01        ;DOS function call for
INT 21H          ;character input with echo
;the entered character is in AL

REPEA:   CMP [BX], AL    ;compare AL with string bytes
JE FOUND         ;if equality is found, display
;appropriate message

INC BX            ;point to the next byte in array
LOOP REPEA        ;if count is not 0, repeat
LEA DX, MESG2    ;character not found, go to MESG2
JMP DISP

FOUND:   LEA DX, MESG1  ;display MESG1
DISP:    MOV AH, 09    ;DOS function call for display
INT 21H          ;of string

.EXIT
.END

```

A character string is stored in location labeled STRIN. The length of this string is specified in a word location named LEN. The character whose presence (or absence) we want to check for is entered through the keyboard using the DOS function call with AH = 01. This character is compared with each of the characters in the string. Once equality is found, the comparing loop is exited and MES1 is displayed. Otherwise, MES2 is displayed, after all the characters in the array have been compared. Note that both the messages (MESG1 and MESG2) are preceded by the newline (0AH) and carriage return (0DH) characters. This is to ensure that the display is obtained in a different line after the input character (the one that is entered through the keyboard) is echoed on the screen.

3.4.5 | Unsigned Multiplication

MUL – Multiply.

Usage: MUL source.

This instruction multiplies a number in AL or AX by the source (where the source can be a register or a memory location **but not an immediate number**). All the conditional flags are affected, but only the CF and ZF are defined as meaningful for the result. The destination depends on the size of the operand. There are two ways of performing multiplication.

i) Byte by byte

In this, one of the operands must be in the AL register, and the source can be a byte in a register or memory location. The product (a word) will be in AX.

MUL BL	;multiply BL by AL – product in AX
MUL BYTE PTR[SI]	;multiply the byte pointed by SI, by AL – product in AX
MUL BIG	;multiply the content of BIG by AL – product in AX

ii) Word by word

In this, one of the operands must be in the AX register, and the source can be a word in a register or memory location. The product will be in DX and AX, with the upper word in DX.

MUL CX	;multiply CX by AX – product in DX and AX
MUL WORD PTR [DI]	;multiply the word pointed by DI with AX – product in DX and AX

iii) Word by byte

This is only a special case of the word \times word multiplication. The byte must be extended to be a word by making the upper byte to be 0. If the byte is in AL, extend it to be a word by making the content of AH to be zero. Thus, AX now contains one of the operands.

Flags Affected

For the multiply instruction, all conditional flags are affected, but only the carry (CF) and overflow (OF) flags have any significance. They will be set or reset according to the size of the product. Recall that a byte \times byte multiplication can produce a result of a word size. However, if the operands are small, the product itself is only a byte and needs only the AL register for the product. In that case, both the carry and overflow flags are found cleared, but if the product occupies a word size, both these flags are set (CF = 1, OF = 1). This concept can be extended to the word \times word multiplication as well. We can summarize that if the

product has a size equal to the size of the operands, both these flags are reset. i.e., CF = 0, OF = 0. However, if the product is large enough to occupy the registers assigned for it, these flags are set. i.e., CF = 1, ZF = 1. See this multiplication:

```
MOV AL 78H  
MOV CL, 0F9H  
MUL CL
```

This program segment will cause the product to be 74B8H. Thus, the result of the multiplication is AX = 74B8H, CF = 1 and OF = 1.

However, if the program segment is:

```
MOV AL, 13H  
MOV CL, 09H  
MUL CL
```

The result is AX = 00ABH, CF = 0 and OF = 0.

In example 3.19, two bytes stored in the data segment are multiplied. The result of multiplication is available in AX, which is then moved to the location PROD, a word location.

Example 3.19

```
.MODEL SMALL  
.DATA  
MULT  DB  0AH          ;multiplier  
MULP   DB  0F6H         ;multiplicand  
PROD   DW  ?            ;space allocated for product  
.CODE  
.STARTUP  
MOV AL, MULP           ;move the multiplicand to AL  
MUL MULT              ;multiply with the multiplicand  
MOV PROD, AX           ;product moved from AX to memory  
.EXIT  
END
```

Now, let us use the multiply instruction in a more complex application. Example 3.20 is a program to find the factorial of a number N. For 8086, the maximum size of an operand for multiplication is only a word. This places a limitation on the value of N that can be used. It can be verified that N is to be less than 9 for the program to give the correct result. In this program, the value of N should be entered from the keyboard, remembering that N is to be less than 9. Since, any value that is entered

from the keyboard is in ASCII form, 30H is subtracted from it to get the binary value of N to be used in the computation.

Example 3.20

```
.MODEL SMALL
.DATA
FACT DW 0      ;space allocated for the factorial
.CODE
.STARTUP
MOV AH, 01
INT 21H        ;enter N from the keyboard
SUB AL, 30H    ;convert ASCII to binary
MOV AH, 0       ;convert N in AL to a word in AX
MOV BX, AX     ;move it to BX
MOV AX, 1       ;AX = 1, to start the iteration
CMP BX, 0       ;compare BX (= N) to 0
JZ FINAL        ;if N = 0, jump to find 0!
REPEA: MUL BX  ;for N not 0, multiply with AX
DEC BX         ;decrement BX
CMP BX, 0       ;compare with 0
JNE REPEA      ;repeat if BX is not 0
FINAL: MOV FACT, AX ;AX = 1, hence 0! = 1
.EXIT
END
```

The computation is done iteratively. First, AX = 1. The value of N is copied to BX. For N = 0, the factorial is 1. For this case, the content of AX is transferred to the location FACT and corresponds to 0!. If N is not zero, then it is multiplied with AX. BX is decremented and cumulatively multiplied with the product in AX. As N becomes higher, N! increases steeply, and only for numbers up to 8, with the factorial fit into a word location. Further multiplication becomes impossible because one of the operands has a size greater than a word. The value of N! is finally stored in the memory location labeled FACT.

3.4.6 | Unsigned Division

DIV – Divide

Usage: DIV source

This instruction divides AX or DX – AX by the source, where the source can be a register or a memory location but not an immediate number. All the conditional flags are affected, but undefined; hence, they do not give any interpretation or information about the result. The destination

depends on the size of the operand. There are two ways of performing division:

i) Divide a word by a byte.

Here, the dividend must be a word placed in AX and the source must be a byte. The result of division causes AL to contain the quotient and AH to contain the remainder.

DIV BL	;divide AX by the byte in BL
DIV BYTE PTR [BX]	;divide the word in AX by the byte pointed by BX
DIV DIG	;divide the word in AX by the byte in DIG

See this division:

```
MOV AX, 0C678H  
MOV CL, 0F9H
```

This program segment will cause AH (remainder) = 0CH and AL (quotient) = CCH.

Dividing a byte by a byte.

This is just a special case of a division of a word by a byte. In this case, convert the dividend byte to a word by loading the dividend in AL and 0 in AH. Thus, AX will be the word that acts as the dividend.

ii) Divide a double word by a word.

In this case, the dividend has to be in AX and DX (the upper word in DX). The divisor should be a word. The result of this division causes the quotient to be in AX and the remainder to be in DX.

DIV BX	;divide the double word in DX-AX by the word in BX
DIV WORD PTR [SI]	;divide the double word in DX-AX by the word pointed by SI
DIV ANGLE	;divide the double word in DX-AX by the word in ANGLE

Dividing a Word by a Word.

Similar to the previous case, if we want a word by word division, extend the word in AX to be a double word by loading 0 in DX to get the dividend to be a double word in AX and DX.

Divide by Zero Error.

For division, if the divisor is zero, the quotient becomes undefined. In attempting such a division, the 8086 will exit from this program and generate an interrupt. This state is said to be a ‘divide by zero error’. An interrupt generated by an error, is termed an exception. However, division by zero is not the only condition to cause such an error. If the quotient register is too small to accommodate the quotient, then also this happens. For example, if the dividend is a large number and the divisor is very small, such a condition is possible.

```
MOV AX, 09876H
MOV CL, 25H
DIV CL
```

The above program segment will give a quotient of 41EH, which obviously cannot be accommodated in AL. When such an error occurs, program execution is aborted and the assembler displays the message ‘divide overflow error’. If this problem had been foreseen, the solution would be to convert the dividend to a double word and the divisor to a word. Then, the quotient will be in AX, which will be big enough for it.

Example 3.21

```
.MODEL SMALL
.DATA
FIRST    DB  89H
SECOND   DB  0CAH
AVG      DB  ?
.CODE
.STARTUP
MOV AL, FIRST    ;copy FIRST to AL
MOV AH, 0         ;zero-extend
MOV BL, SECOND   ;copy SECOND to BL
MOV BH, 0         ;zero-extend
ADD AX, BX        ;add the zero extended words
MOV CL, 02        ;load divisor to CL
DIV CL           ;divide AX by CL
MOV AVG, AL       ;save quotient to AVG
.EXIT
END
```

Example 3.21 gives a complete program for finding the average of two bytes stored in memory locations labeled FIRST and SECOND. These are copied to registers AL and BL and zero extended to convert them to words. Addition of the numbers is then done as words. This sum, which is the dividend, is in the AX register. The divisor 2 is copied to the CL register. After division, the quotient is available in AL and the remainder

in AH. For this problem, only the quotient is important, and this is saved to location AVG.

Now, we will see a very useful and interesting application of the division operation. This program converts a 16-bit hexadecimal number to decimal and displays the decimal number on the console after converting it to ASCII form. To see the logic of this program, let us take a number, say, 246.

1. Divide this by 10 to get a quotient of 24 and a remainder of 6. Push the remainder on to the stack.
2. Next, divide the quotient again by 10 to get 2 as the quotient and 4 as remainder. Push this remainder also on to the stack.
3. One more division causes the quotient to be zero. This is the stopping condition in the division loop.
4. The number of division operations done is counted along with all these steps. Now, pop out the remainders from the stack. This will be in the reverse order of the push operations. Thus, 2 will be popped out first and 6 last. This can be displayed in this order.
5. For displaying, each digit must be converted to its corresponding ASCII. This is done by adding 30H to each digit.

Now, examine Example 3.22. The program converts any 16-bit hexadecimal number to decimal and then displays it. Here, the hexadecimal number is placed in the data segment. Note that the division here is accomplished as a **double word by word** division.

Example 3.22

```
.MODEL SMALL  
.DATA
```

```

NUM DW 0CEF6H ;the hexadecimal number
COUNT DB 0 ;count of the number of divisions done
.CODE
.STARTUP
MOV AX, NUM ;get the hex number into AX
MOV DX, 0 ;DX = 0, thus DX-AX is the dividend
MOV CX, 10 ;the divisor 10 is loaded into CX
REPEA: DIV CX ;divide by CX
PUSH DX ;the remainder in DX is pushed to stack
MOV DX, 0 ;DX = 0 again to get the dividend in DX-AX
INC COUNT ;increment the count
CMP AX, 0 ;check if the quotient (in AL) is zero
JNE REPEA ;if AL is not zero, repeat the division
DISP: POP DX ;pop out the remainders for displaying
ADD DL, 30H ;add 30H to convert to ASCII
MOV AH, 02 ;DOS function call for displaying
INT 21H ;a character
DEC COUNT ;decrement the count
JNZ DISP ;if count = 0, it means all remainders
;have been popped out
.EXIT
END

```

3.5 | Logical Instructions

Table 3.7 gives the complete list of logical instructions and the function performed.

Table 3.7 | List of Logical Instructions and Functions Performed by them

SI No.	Instruction format	Function performed	Flags affected
1	AND dest, src	Logical AND of the two operands returning the result in the destination	CF OF PF SF ZF (AF undefined)
2	OR dest, src	Logical inclusive OR of the two operands returning the result in the destination	CF OF PF SF ZF (AF undefined)
3	XOR dest, src	Performs a bitwise exclusive OR of the operands returning the result in the destination	CF OF PF SF ZF (AF undefined)
4	NOT dest	Inverts the bits of the "dest" operand forming the one's complement	None
5	TEST dest, src	Performs a logical AND of the two operands updating the flags register without saving the result	CF OF PF SF ZF (AF undefined)

Example 3.23

Find the result and the state of the flags CF, ZF and OF due to the following instructions. Given AX = 008CH, BX = 345EH, CX = 67EBH

1. AND BL, CL
2. OR AH, BH
3. XOR AL, CH
4. TEST AH, BL

Solution

i) AND BL, AL

$$\begin{array}{ll}
 \text{BL} = & 5\text{EH} \quad 0101 \ 1110 \\
 \text{CL} = & \underline{\text{EBH}} \quad \underline{1110 \ 1011} \\
 & \underline{4\text{AH}} \quad \underline{0100 \ 1010}
 \end{array}$$

After this operation, BL contains 4AH. CF = 0, OF = 0, ZF = 0.

ii) OR AH, BH

$$\begin{array}{ll}
 \text{AH} = & 00 \quad 0000 \ 0000 \\
 \text{BH} = & \underline{34\text{H}} \quad \underline{0011 \ 0100} \\
 & \underline{34\text{H}} \quad \underline{0011 \ 0100}
 \end{array}$$

After this operation, AH = 34H. CF = o, OF = o, ZF = o.

iii) XOR AL, CH

AL =	8CH	1000 1100
CH =	67H	0110 0111
	<u>EBH</u>	<u>1110 1011</u>

After this operation, AL = EBH, CF = o, OF = o, ZF = o.

iv) TEST AH, BL

AH =	00	0000 0000
BL =	5EH	0101 1110

After this operation, the ZF = 1, OF = o, CF = o.

Masking

There is a word ‘masking’ associated with the AND operation. Masking is used to select the part of a word or a byte needed, while making the unwanted bits to be zero. For example, observe the value in AL after the following instructions are executed.

MOV AL, 78H
AND AL, 0FH

This gives AL = 08H in the AL register. The upper nibble of 78H has been masked. As another example, see below where 16-bit data is ANDed with 0FFFH, so as to mask the upper 4 bits. This is a particular case when only 12 bits of this data are needed.

MOV AX, 9876H
AND AX, 0FFFH

AX now has a content of 0876H.

Typical Applications of Logical Instructions

Now, let us use these logical instructions in some simple applications.

AND

To convert an ASCII number to a binary number, mask the number with 0FH.

```
MOV AH, 01          ;get in the number through the keyboard  
INT 21H  
AND AL, 0FH        ;mask the upper nibble
```

OR

To convert an 8-bit binary number (from 0 to 9) to ASCII, OR with 30H.

```
MOV AL, 9           ;AL = 09  
OR AL, 30H          ;AL = 39H
```

XOR

To clear a register, use XOR.

```
XOR BL, BL         ;BL = 0
```

TEST

To test whether a bit is set or not use the TEST instruction.

```
TEST BL, 01H
```

This tests whether bit D0 of BL is 1 or not. If the bit under test is reset, the AND operation corresponding to the TEST operations causes ZF to be set (ZF = 1).

```
TEST CX, 8000H
```

This tests whether D15 of CX is set or not. The result of this that ZF = 0 as D15 = 1.

3.6 | Shift and Rotate Instructions

Table 3.8 gives the complete list of the shift and rotate instructions.

Table 3.8 | List of the Shift and Rotate Instructions

SI No.	Instruction format	Function performed	Flags affected
1	SHL dest, count	Shift logical left by 'count' bits	CF OF PF SF ZF (AF undefined)
2	SAL dest, count	Shift arithmetic left by 'count' bits	CF OF PF SF ZF (AF undefined)
3	SHR dest, count	Shift logical right by 'count' bits.	CF OF PF SF ZF (AF undefined)
4	SAR dest, count	Shift arithmetic right by 'count' bits	CF OF PF SF ZF (AF undefined)
5	RCL dest, count	Rotate through carry left by 'count' bits.	CF OF
6	RCR dest, count	Rotate through carry right by 'count' bits	CF OF
7	ROL dest, count	Rotate left by 'count' bits	CF OF
8	ROR dest, count	Rotate right by 'count' bits	CF OF

3.6.1 | Shift

1. Shift instructions do arithmetic or logical shifting.
2. They also shift right or left.
3. Arithmetic shift is used for signed number operations, while logical shift caters to unsigned numbers.
4. Shifting right causes a divide by 2, for each bit position shifted, while shifting left corresponds to a multiplication by 2.
5. The count means the number of bit positions by which shifting is to be done.
6. If the count >1, load it in CL; otherwise use '1' in the immediate mode.

SAL/SHL – Shift Left Arithmetic/Shift Left Logical.

Usage: SAL/SHL dest, count.

This instruction shifts the destination left by 'count' bits, and zeroes are shifted in to fill the vacant positions on the right. The Carry flag contains the last bit shifted out, because '0' is shifted from the right side. **Left shift 'logical' and 'arithmetic' perform the same operation, and hence, the mnemonic SAL or SHL can be used interchangeably.**

Modifies Flags: CF OF PF SF ZF (AF undefined).

SHL BX, 1	;shift left logical by one position, the word in BX
SAL AL, CL	;shift left arithmetic, AL by the count specified in CL
SHL DATA2, CL	;shift left logical, the content of memory DATA2 by the count specified in CL
SHL BYTE PTR [BX][DI], 1	;shift left (logical) once, the byte with EA = BX + DI
SAL WORD PTR [DI], CL	;shift left (arithmetic) the word pointed by DI, by the count specified in CL

Example 3.24 shows a program segment for finding the weight of the modulo-2 sum of two numbers. Modulo-2 sum essentially is finding the positions in which the two numbers are different – which is an XOR operation. The weight of a number is the number of 1s in it.

In this problem, the modulo-2 sum is found by XORing the two numbers in AL and BL. The result of this is then shifted left. The bit shifted out will be in the carry flag. On checking the carry bit, if it is found to be 1, the register CH is incremented. Along with this, the register CL is decremented. CL contains the length of the operand. Here CL = 8. At the end of the program, we get the ‘weight’ of the modulo-2 sum of AL and BL in CH. For the numbers used in this program, CH = 6 is the result.

Example 3.24

```

.MODEL TINY
.CODE
.STARTUP
MOV AL, 89H ;move first number to AL
MOV BL, 0F7H ;move second number to BL
XOR AL, BL ;EX-OR the two numbers
MOV CL, 8 ;CL = 8
BACK: SHL AL, 1 ;shift left AL once
        JNC REPEA ;repeat if there is no carry
        INC CH ;increment CH if there is a carry
REPEA: DEC CL ;decrement count
        JNZ BACK ;repeat the shifting until CL = 0
.EXIT
.END

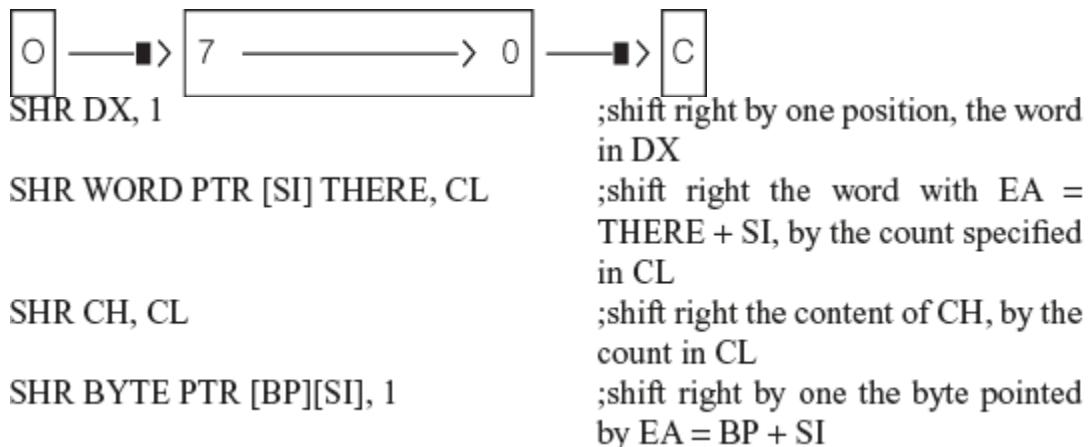
```

SHR – Shift Right Logical.

Usage: SHR dest, count.

This instruction shifts the destination right by ‘count’ bits and fills with zeroes the vacant positions on the left. The Carry flag contains the last bit shifted out.

Modifies Flags: CF OF PF SF ZF (AF undefined)



Example 3.25 shows an example of a shift operation. When a decimal number is represented as a 4-bit binary nibble, and these nibbles are packed in a byte, it is called packed BCD representation. For example, the packed BCD representation of 56 is 0101 0110. Unpacking it means separating it as two bytes 0000 0101 and 0000 0110.

Example 3.25

Convert a packed BCD byte to two unpacked bytes.

Solution

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, 95H      ;copy the packed BCD byte to AL
MOV BL, AL        ;copy the same byte to BL
AND AL, 0FH       ;mask the upper nibble of AL
AND BL, 0FOH      ;mask the lower nibble of BL
MOV CL, 04         ;move a count of 4 into CL
SHR BL, CL        ;shift BL right 4 times
.EXIT
END
```

In this problem, a packed BCD byte is in AL. It is to be unpacked and placed in two registers. The operand is in AL. Keep a copy of it in BL too.

For unpacking, first the upper nibble of AL is masked. This gives one unpacked BCD byte. This is now in AL. Next, the lower nibble of BL is masked, and the result is shifted right 4 times to bring the upper nibble data to the lower nibble position. Now, the two unpacked bytes are available in AL and BL.

Note The operation and practical use of the SAR instruction will be discussed in Section 4.7.3, where signed number arithmetic is dealt with.

3.6.2 | Rotate Instructions

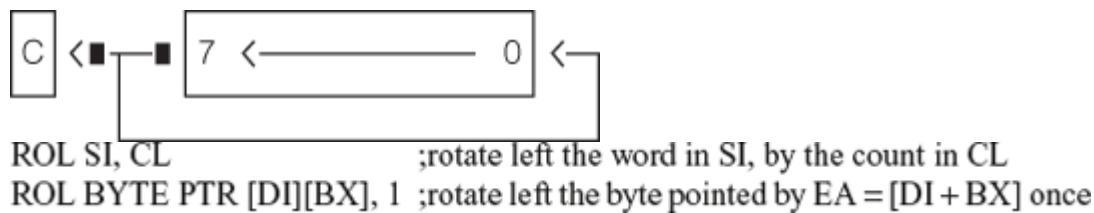
Rotate instructions have the same format as the shift instructions. There are two types of rotate – rotate through carry or without taking the carry bit as a part of the rotate act.

ROL – Rotate Left.

Usage: ROL dest, count.

This instruction rotates the bits in the destination to the left ‘count’ times with all data pushed out at the left re-entering on the right. The Carry flag will contain the value of the last bit rotated out.

Modifies Flags: CF OF



ROR – Rotate Right

Usage: ROR dest, count



This instruction rotates the bits in the destination to the right ‘count’ times with all data pushed out at the right side re-entering on the left. The Carry flag will contain the value of the last bit rotated out.

Modifies Flags: CF OF

ROR AL, 1	;rotate right once the byte in AL
ROR DX, CL	;rotate right the word in DX the count in CL

Let us examine what Example 3.26 will do.

Example 3.26

```
.MODEL TINY
.CODE
.STARTUP
MOV BX, 5634H
MOV CL, 8
ROL BX, CL
MOV AL, 0C6H
MOV CL, 4
ROR AL, CL
.EXIT
END
```

In the first instance, there is a 16-bit register BX which contains the value 5634H. By using ROL, and rotating 8 times, the value in BX changes to 3456H. Similarly with AL = C6H, using ROR with a count of 4, AL will have the new value of 6CH. Thus, we can switch exchange nibbles or bytes using the rotate instructions with the appropriate count in CL. Note that, for this application, it does not matter if ROL or ROR is used.

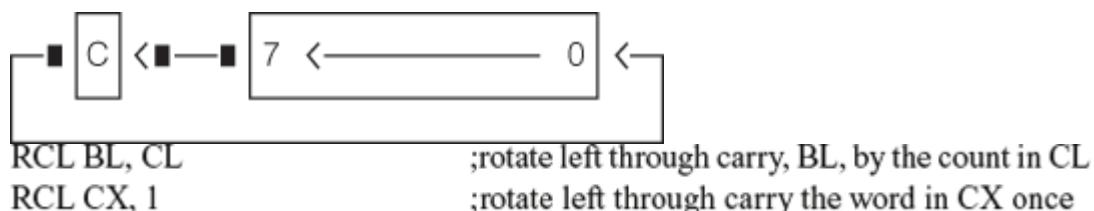
It is easy to check that Example 3.25 can be re-written replacing the SHR instruction with either the ROL or ROR instruction.

RCL – Rotate Through Carry Left.

Usage: RCL dest, count.

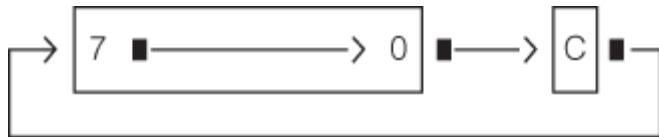
This instruction causes the left most bit (LSB) to enter the Carry flag, and the CF enters through the left end (MSB). Thus, due to one shift operation, the MSB enters the CF, and the CF gets into the LSB position.

Modifies Flags: CF OF



RCR – Rotate Through Carry Right.

Usage: RCR dest, count.



This instruction causes the right most bit (MSB) to enter the carry flag, and the CF enters through the left end (MSB). Thus, due to one shift operation, the LSB enters the CF, and the CF gets into the MSB position.

Modifies Flags: CF OF

RCR BYTE PTR [SI], 1

;rotate right through carry, the byte pointed by SI, once

RCR WORD PTR [DI][BX], CL

;rotate right through carry the word pointed by EA = SI + BX, by the count specified in CL

Example 3.27

Find the values in the destination for each line of this program segment.

```
STC
MOV AX, 5485H
RCR AL, 1
MOV CL, 03
RCL AX, CL
MOV CL, 05
ROR AX, CL
ROL AX, 1
```

Solution

Instructions

```
STC
MOV AX, 5485H
RCR AL, 1
MOV CL, 03
RCL AX, CL
MOV CL, 05
ROR AX, CL
ROL AX, 1
```

Result after execution

CF = 1
AX = 0101 0100 1000 0101
AL = 1100 0010
CL = 03
AX = 1010 0110 0001 0101
CL = 5
AX = 1010 1101 0011 0000
AX = 0101 1010 0110 0001

KEY POINTS OF THIS CHAPTER

- BIOS and DOS function calls facilitate the use of input and output devices in programming, making it interactive and more fun.

- The instruction set of 8086 can be divided into groups based on functionality.
- MOV is the most frequently used instruction and it is part of the data transfer group.
- LEA is an instruction that copies the offset of a memory address to a 16-bit register. It can be used to make the register be a pointer to an array of data items in memory.
- PUSH and POP can be used only with the stack segment.
- The SS register contains the upper 16 bits of the lowest (base) address in the stack segment. SP contains the offset (with respect to the base address) of the highest address in the stack.
- Branch instructions cause the program sequence to change.
- JUMP is an important branch instruction which can be far or near, and unconditional or conditional.
- Jumps using the direct mode of addressing are re-locatable and ‘relative’.
- LOOP is a jump instruction which causes branching as well as decrementing a count in CX. A LOOP instruction can be conditional as well.
- There are 20 arithmetic instructions for 8086, of which a few specifically cater to signed numbers, a few to BCD numbers, and a few to ASCII numbers.
- The ADC (Add with carry) and SBB (Subtract with borrow) instructions find use in multi-byte additions and subtractions.
- The INC and DEC instructions do not affect the Carry flag.
- The CMP (Compare) instruction subtracts the source from the destination and sets/resets flags, but neither the source nor the destination is altered.
- MUL is the mnemonic for unsigned multiplication. For a byte by byte multiplication, one operand is implied to be in AL and the product in AX. For a word by word multiplication, one operand is to be in AX and the product will be in DX-AX.
- DIV is the mnemonic for unsigned division. For a word by byte division, the dividend is to be in AX. After division, the quotient will be in AL and the remainder in AH. For a double word by word division, the dividend is to be in DX-AX. Then, the quotient will be in AX and the remainder in DX.
- If the quotient is too large to fit in the register assigned for it, a ‘divide by zero’ error will be generated.
- The logical instructions of 8086 are AND, OR, XOR, NOT and TEST.
- The TEST instruction does logical ANDing, but only the flags are affected.
- Shifting of a data can be done left or right and also by as many positions as required.
- Arithmetic shifting is meaningful for signed numbers, while logical shifting pertains to unsigned arithmetic.
- Rotation can be done left or right and can be direct or through the Carry flag.

QUESTIONS

1. Distinguish between the top-down and bottoms-up approach in programming.
2. What is the use of DOS function calls in assembly programming?
3. How many operands do each of the following instructions have?
 1. ADD
 2. ADC
 3. INC
4. Which conditional flag is not affected by the DEC instruction?
5. If AX = 5600H and BX = 0C07H, find the contents of CX and DX after the following three sets of instructions:
 - o. PUSH AX
 - PUSH BX
 - POP CX
 - POP DX
 1. PUSH AX
 - PUSH BX
 - POP DX
 - POP CX
 2. PUSH BX
 - PUSH AX
 - POP CX
 - POP DX
6. Why is the stack pointer called the TOP OF STACK?
7. Distinguish between a near and a far jump.
8. Specify two ways of specifying an unconditional jump in the indirect mode of addressing.
9. Why does the far jump have five bytes of instruction length?
10. Name a conditional jump instruction that does not test a conditional flag.
11. What are the conditions under which a LOOPZ instruction exits looping?
12. What does the instruction STC do?
13. Can the ADD instruction use CS as a destination?
14. The instruction INC [SI] seems ambiguous. Why? How can the ambiguity be corrected?
15. What does the instruction CMP AX, BX do?
16. What is the role of OF and CF in multiplication?
17. What happens if the DIV instruction is used for a dividend of 1938H and divisor of 05?
18. What is meant by masking? Give an example of how it is used.
19. How can multiplication and division be achieved by shift operations?
20. What is the difference between the instructions RCL and ROL?

EXERCISES

1. Write a program to print on two separate lines the messages ‘Hi, folks’ and ‘I am Samantha’.
2. Enter a character through the keyboard without echoing it on the video monitor. Compare it with a number stored in memory and display the message ‘EQUAL’ or ‘NOT EQUAL’ as the case may be.
3. Display each character of the word PRADO on a separate line.
4. Enter ‘N’ characters through the keyboard without echo. Save it in memory as an array. Then, display it as a character string.
5. Indicate what is wrong (if anything is wrong) with each of these instructions:
 1. MOV CX,DL
 2. ADD DATA1, 0978H
 3. MOV BYTE PTR [SI][BX], DX
 4. MOV 045FH, AX
 5. MOV [DX], AL
 6. MOV [SI][CX], AX
 7. MOV DS, 3453H
6. Write a program that adds two quad words and stores the result in memory.
7. Write a program that subtracts two double words and stores the result in memory.
8. Find the status of the CF and ZF flags after the execution of the following sets of instructions:
 - o. MOV AX, 9078H
 1. CMP AX,0C089H
 2. XOR AL, AL
 3. MOV AL, 29H
 4. CMP AL, 0
 9. Find the biggest number in an array of:
 - o. bytes
 1. words

Display the biggest number in each case, as a decimal number.

10. There are 10 unsigned bytes stored in memory. Arrange these bytes in the following order:

- o. ascending
 1. descending
11. Find the average of 20 bytes stored in memory. Display the average as a decimal number.
12. Add the sum of the first 20 natural numbers. Display the sum.
13. Write a program that counts the number of 1s in a binary number.
14. Write a program to add two $N \times N$ matrices.
15. Find the number of times a particular character is present in a character string stored in memory.

- [Copy](#)

- [Add Highlight](#)
- [Add Note](#)

4

Programming Concepts—III

IN THIS CHAPTER, YOU WILL LEARN

- To use string instructions for various applications.
- The concept and use of procedures and call instructions.
- To distinguish macros from procedures and learn to write and use macros.
- To write programs to convert between the commonly used number formats.
- Signed number arithmetic using 8086 instructions.
- To write programs using the high level language constructs of MASM.

4.1 | String Instructions

The 8086 has a set of instructions for handling blocks of data in the form of bytes or words. They are called ‘string’ instructions. A string is an array of data of the same type – for example, a character string or a byte string. Table 4.1 gives the list of string instructions/prefixes which are used in string manipulations. The usefulness of string instructions can be seen when in the memory data has to be moved, searched or compared in blocks.

Table 4.1 | List of Instructions/Prefixes Used in String Operations

SI No.	Instruction format	Function performed	Flags affected
1	MOVSB/MOVSW	Move byte or word string	None
2	CMPSB/CMPSW	Compare byte or word string	AF CF OF PF SF ZF
3	SCASB/SCASW	Scan byte or word string	AF CF OF PF SF ZF
4	LODSB/LODSW	Load byte or word string	None
5	STOSB/STOSW	Store byte or word string	None
6	CLD	Clear direction flag	DF
7	STD	Set direction flag	DF
8	REP (Prefix for a string instruction)	Repeat execution of string instructions while CX is not 0	None
9	REPE/REPZ (Prefix for a string instruction)	Repeat execution of string instructions while CX is not 0 and while Zero flag is set	None
10	REPNE/REPNZ (Prefix for a string instruction)	Repeat execution of string instructions while CX is not 0 and while Zero flag is not set	None

Note In string instructions, 'SB' stands for 'string byte' and 'SW' for 'string word'.

Consider the case of 100 (say) words or bytes in a particular memory area that is to be moved to another memory area. This can very well be done using pointer registers and looping using a counter. However, this whole process can be automated with lesser number of instructions if we use string instructions. Similarly, we may need to compare two blocks of data for equality or search a data block for a particular data. In all these cases, string instructions make our task easier and our code shorter. However, before using these instructions, we have to include a few initialization steps in our code.

Pre-requisites for Using String Instructions

1. Two segments are to be defined, i.e., the data segment and the extra segment. This means that the corresponding segment registers DS and ES have to be initialized and used. The data segment is the source segment and the extra segment is the destination segment.
2. The SI registers and DI registers should act as pointers to the data segment and extra segment, respectively. This means that, initially, SI should contain the address (offset) of the first location in the data segment. Similarly, DI should contain the address (offset) of the first location in the extra segment.

3. There is a control flag called the direction flag which is used exclusively for string operations. Its purpose is that in string operations, if the flag is set, the pointer registers get automatically decremented, and if reset, the reverse occurs. So, whenever string instructions are being used, the direction flag (DF) should be set or reset depending on the direction the addresses are to be modified after each operation.
4. The counter CX should be loaded with the count of the number of operations required.

4.1.1 | The MOVS Instruction

Example 4.1 shows the use of the string instruction MOVS with the necessary pre-requisites incorporated. Let us examine how this has been done. The problem in hand is to transfer 10 bytes (the character ‘*’) from an area of memory designated as DATA1 to an area named DATA2. Thus, the source has to be the data segment, and the destination the extra segment. How do we incorporate the extra segment?

If we use the full segment model, we can define two segments – the data segment (DAT) and the extra segment (EXTR). See how it is done. When both the segments registers are defined, the segments get defined. Then, the program instructions cause the data in the data segment to be copied to the extra segment. Example 4.1a uses the full segment model for this.

Example 4.1a

```

DAT           SEGMENT
DAT1 DB 10 DUP('*')          ;store '*' in 10 locations
DAT           ENDS

EXTR          SEGMENT
DAT2 DB 10 DUP(0)           ;allocate 10 locations
EXTR          ENDS

COD           SEGMENT
ASSUME CS:COD, DS:DAT, ES:EXTR
MOV AX, DAT
MOV DS, AX                 ;initialize the DS register
MOV AX, EXTR
MOV ES, AX                 ;initialize the ES register
LEA SI, DAT1              ;point SI to Source
LEA DI, DAT2              ;point DI to destination
MOV CX, 10                 ;load count in CX
CLD
REP MOVSB                ;move the byte string
MOV AH, 4CH                ;AH = 4CH to return to DOS
INT 21H

COD           ENDS
END

```

In Example 4.1a, two segments DAT and EXTR are defined and two segment registers DS and ES are initialized. The addresses SI and DI point to the source data and destination data addresses, and DF is cleared for auto incrementing the pointer registers.

Now let us do the same program using the simplified model. The small model can have only one data segment, so how can we have an ‘extra segment’? The solution is to have both data areas in the data segment itself, but after the data segment is defined, copy the value of DS into ES – this makes the assembler believe that the extra segment is available – though it is the same as the data segment. In this example, the destination data area is chosen to be at org 0200H. This is done just to space the source and destination data areas and is not mandatory. Thus, the first prerequisite of having both the data and extra segments is satisfied.

Next, the SI register is made to point to the address DATA1 and the DI register to DATA2. Then, the counter register is loaded with 10, the count of data transfer operations required. The instruction CLD is used to clear the direction flag (DF). This causes the value of SI and DI to be incremented after each move operation. Here, the data is in the form of bytes. Hence, SI and DI are incremented only by one each time. The

string instruction used here is MOVSB (prefixed by REP). This causes the data in the location pointed by SI to be moved to the location pointed by DI. After each such move operation, two actions occur. One is that CX decrements by 1 and the other is that the pointer registers are incremented. This sequence continues until CX = 0. Example 4.1a and 4.1b illustrate these steps.

Example 4.1b

.MODEL SMALL	;select small model
.DATA	;start data segment
DAT1 DB 10 DUP('*')	;store '*' in 10 locations
ORG 0200H	;define destination area
DAT2 DB 10 DUP(?)	;set aside 10 locations
.CODE	
.STARTUP	
MOV AX, DS	;move DS into AX
MOV ES, AX	;move AX to ES
LEA SI, DAT1	;point SI to Source
LEA DI, DAT2	;point DI to destination
MOV CX, 10	;load count in CX
CLD	;clear Direction flag
REP MOVSB	;move the byte string
.EXIT	
END	

For the above program, if the data involved is in the form of **words**, the only change would be to replace the line REP MOVSB with REP MOVSW. In this case, CLD will cause SI and DI to be incremented by 2.

4.1.2 | The CMPS Instruction

Now, let us use the next string instruction – CMPSB/CMPSW. This instruction is for string comparison, byte by byte or word by word, as the case may be. The conditional flags are modified according to the result of the comparison. String comparison has to be accompanied by the use of the conditional REPE/REPZ prefix. **Since string comparison checks only for equality, the Zero flag is made use of automatically.** Example 4.2 compares two character strings (of length six) which are saved in the data segment. One of two messages is to be displayed depending on whether WRD2 is the same as WRD1.

Example 4.2

```

.MODEL SMALL
.DATA
WRD1    DB "SAMSON"      ;store the first word
WRD2    DB "SAMRON"      ;store the second word
GUD     DB "SAME$"       ;store the message for equality
BAD     DB "NOT SAME$"   ;store the message for inequality

.CODE
.STARTUP
MOV AX, DS
MOV ES, AX          ;copy DS to ES
LEA SI, WRD1        ;point SI to source
LEA DI, WRD2        ;point DI to destination
MOV CX, 6
CLD                ;clear direction flag
REPE  CMPSB          ;repeat comparison if equal(ZF = 1)
JNZ MSG             ;if ZF is reset, go to MSG
LEA DX, GUD         ;point DX to the first message
JMP DISP
MSG:   LEA DX, BAD   ;point DX to second message
DISP:  MOV AH, 09    ;use the function for string display
        INT 21H          ;use DOS interrupt 21H
.EXIT
END

```

Here, SI and DI point to the two character strings, CX stores the count and the direction flag is cleared for auto incrementing the address pointers. REPE is used as the prefix for comparing the string bytes. REPE CMPSB means that the corresponding string bytes of the source and destination are compared as long as they are equal. Equality also implies that the Zero flag is set. The comparison is stopped as soon as an **inequality** is encountered. One of the following two conditions cause the comparison loop to be exited:

1. An inequality is seen i.e., the Zero flag gets reset.
2. Normal exiting when CX = 0.

In the example, the two character strings are shown to be different, which causes the message 'NOT SAME' to be printed. If WRD1 and WRD2 are the same, the message 'SAME' is printed.

4.1.3 | The SCAS Instruction

The SCAS instruction scans a byte or word string to ascertain the presence of a specific byte or word. This specific data is loaded into AL or AX. **The string which is to be scanned has to be in the extra segment and is to be pointed by DI. This is mandatory and**

cannot be overridden. Example 4.3 illustrates a typical scenario for the use of this instruction. Suppose that an ASCII string labeled LIST is stored in memory. The program searches the string for the presence of the ASCII character ‘S’. DI addresses the string to be searched. The string to be searched is placed in the Extra segment, by making the ES have the same value as DS. Thus, the extra segment is the same as the data segment. In this example, the SCASB instruction has an REPNE prefix (repeat while not equal) prefix. The REPNE prefix causes the SCASB instruction to repeat until either the CX register reaches 0 or until an equal condition is indicated ($ZF = 1$) as the outcome of the SCASB instruction’s comparison operation. The scanning of the string continues as long as ‘S’ is not found in the string. If ‘S’ is found, the search is terminated immediately.

Example 4.3

```

.MODEL SMALL
.DATA
LIST    DB "ANBHTSHFGTRIUUJEHNPBGDVBTH"
PR      DB "CHARACTER PRESENT$"
NOTP   DB "CHARACTER NOT PRESENT$"
.CODE
.STARTUP

MOV AX, DS          ;copy DS to AX
MOV ES, AX          ;copy DS to ES
LEA DI, LIST +25   ;point SI to the last character
STD               ;auto-decrement
MOV CX, 26          ;load count in CX
MOV AL, 'S'         ;load the character in AL
REPNE  SCASB        ;repeat if not equal
JZ MSG
LEA DX, NOTP
JMP DISP

MSG:   LEA DX, PR
DISP:  MOV AH, 09
       INT 21H
.EXIT
END

```

In the above, a variation (from the previous examples) has been used. DI has been made to point to the last character of the string, and the direction flag is set, such that the address auto decrements after each scanning operation. Refer to Example 3.18, where the same problem has been done without using string instructions.

4.1.4 | The STOS and LODS Instructions

i) STOS

The STOS instruction is the mnemonic for ‘**storing**’ a string in memory. As such, we need to define a memory area in which ‘**storing**’ is to be done. **This memory area is defined to be the extra segment, and it is addressed by DI as it is the destination segment.** The data to be stored is placed in the AL or AX register. An area in memory can be filled with the required data using this instruction. Example 4.4 illustrates the use of the STOS instruction. Here, AX is loaded with 0001, and this is moved to a location named AREA, which has been defined as an array of 50 words. This array is filled with the word 0001 by the STOSB instruction, using the REP prefix and the counter CX initialized with 50. Since STOS is specified for the extra segment, ES register has to be initialized. Notice that, as done earlier, a data segment is first defined (by the .DATA directive) and the DS register content is then moved to ES. Note also that the memory AREA is addressed using the DI register. **SI register cannot be used.**

Example 4.4

```
.MODEL SMALL           ;select small model
.DATA                 ;start data segment
AREA DW 50 DUP(?)    ;define AREA
.CODE                ;start code segment
.STARTUP             ;start program
MOV AX, DS            ;copy DS to AX
MOV ES, AX            ;copy AX to ES
LEA DI, AREA          ;address AREA with DI
MOV CX, 50             ;load count in CX
CLD                  ;clear direction flag
MOV AX, 0001           ;load AX with 0
REP STOSW             ;repeat until CX = 0
.EXIT                ;exit to DOS
END                  ;end of program
```

It is obvious that the STOS instruction is used to fill up an area in memory with the same data.

ii) Lods

The last of the string instructions is LODS. This is an instruction for ‘**loading**’. Loading always means the act of taking data from memory and putting it into a register. Here, the source memory is the data segment

and the pointer to it is SI. The data segment is the source segment and the destination register is the AL or AX register. There is no sense in using the REP prefix for the LODS instruction as data can be loaded to AL/AX only once.

4.2 | Procedures

In high level languages (i.e., C, C++), you might have come across ‘functions’. A function is a program which does a specific task. When this task is to be done repeatedly, the function is used again and again. When a ‘main’ program considers this as a subsidiary task, the function (or subroutine) is ‘called’ whenever its service is required. This is applicable to assembly language programming also. Borrowing Intel’s terminology, we shall, however, call it a ‘procedure’.

Figure 4.1 shows two cases. One is the case of a main program calling many different procedures. The second is a program calling the same procedure repeatedly. Thus, there is a main program which can call a procedure anywhere in its body. The former is the ‘calling program’ and the latter is the ‘called program’. The main program is in a particular code segment and the procedure may be written in the same code segment (in which case it is a ‘near’ procedure), or it may be in a different segment (‘far’ procedure).

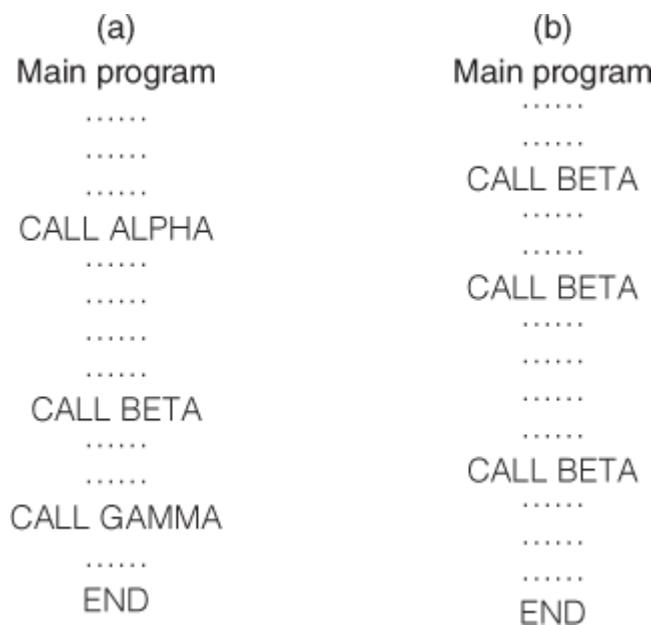


Figure 4.1 | **a** Main program calling different procedures **b** Main program calling the same procedure repeatedly

Let us now see the sequence of actions taken by a processor when using a procedure. For now, let us assume that the procedure is a ‘near’

procedure. In the course of the action of execution of the main program, a CALL instruction is encountered. This signals that a procedure written elsewhere has to be executed. This is essentially a branching operation, as the normal program execution sequence is disturbed. At the time the CALL instruction is being executed, the IP (instruction pointer) will be pointing to the next instruction in the main program. The steps taken by the processor automatically are:

1. It saves the current IP content on the stack (this is the ‘return’ address for coming back to the main program after executing the procedure).
2. The CALL destination (specified in the CALL instruction) will be the address of the procedure. The IP is now loaded with this address and execution proceeds from that location.
3. The procedure is executed until a RET (return) instruction in the procedure is encountered.
4. Then, the old value of the IP is retrieved from the stack and control returns to the main program at the return address.

Figure 4.2 shows a CALL encountered in a main program. Then, the procedure MULT is taken up for execution. At the end of the procedure, there is the RET instruction, executing which causes control to return to the main program. The instruction after the CALL instruction will now be taken up.

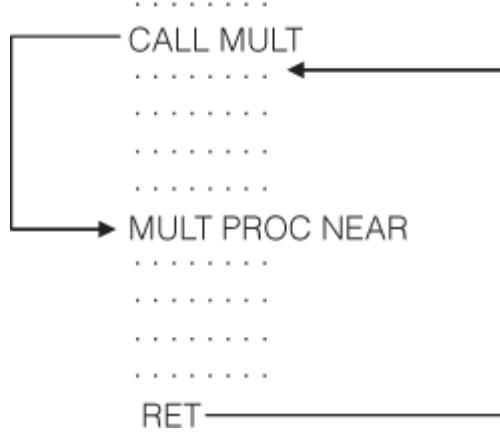


Figure 4.2 | Call and return

4.2.1 | Writing a Procedure

Like any assembler, MASM also has specifications for writing a procedure. The procedure should begin with the procedure name followed by the directive PROC. We also use the directives NEAR or FAR which specify the ‘type’ of the procedure. The procedure should end with the procedure name and ENDP. Example 4.5 shows a program which enters 10 single digit numbers through the keyboard, finds their squares

and stores the squares in memory. Only byte locations have been allocated in memory for the squares as the square of the highest single digit number is 81, which will fit into a byte space. In the procedure, the register assigned for the product is AX. However, the product will occupy only AL. Hence, only the value in AL is moved to memory.

Now, let us examine the salient features of this program.

1. A procedure named SQUARE is used to calculate the squares.
2. Entering data from the keyboard and storing the squares in memory is done in the main program.
3. The procedure is in the same code segment. Hence, it is designated as near.
4. The procedure ends with a RET (return) instruction which takes control back to the main program.
5. The .EXIT statement is the last statement in the main program. We know that it gets converted to the instructions for returning control to DOS. Thus, it is the last instruction which gets executed.
6. The last line in the code is the END directive which tells the assembler to stop reading.
7. We know that the stack segment is necessary for the working of a procedure. For now, we are not defining any stack segment but using the stack defined automatically by DOS. When we need more stack size, we will define our own stack.

Example 4.5

```

.MODEL SMALL
.DATA
NUMBERS DB 10 DUP(0) ;allocate space for the squares
.CODE
.STARTUP
MOV CX, 10           ;load count in CX
LEA SI, NUMBERS    ;let SI point to NUMBERS
REPEA: MOV AH, 01      ;key board input
INT 21H
SUB AL, 30H          ;convert from ASCII to binary
CALL SQUARE          ;call the procedure for squaring

        MOV [SI], AL      ;store the square in memory
        INC SI             ;increment pointer
        LOOP REPEA         ;decrement CX and repeat if CX!=0
        .EXIT              ;return to DOS prompt
SQUARE  PROC NEAR       ;define the procedure
        MOV BL, AL          ;move multiplicand to BL
        MUL BL              ;multiply
        RET                 ;return to main program
SQUARE  ENDP              ;define the end of the procedure
END

```

In Example 4.5, the square of the numbers has been calculated and saved in memory. It may be interesting to display them on the monitor. This can be done by referring to Example 3.22. The program for converting any hexadecimal number to binary and displaying can be made another procedure and called here.

4.2.2 | Call and Return Instructions

Now that we have seen the general format and structure of a program with procedures, let us examine the CALL instruction in greater detail. This instruction is obviously a ‘branch’ instruction because it causes control to move to a different address (called the target or destination), which may be in the same or a different code segment.

4.2.2.1 | Intrasegment or ‘Near’ Call

i) Direct CALL

Usage: CALL label (see Fig. 4.3).

OPCODE	OFFSET LOW	OFFSET HIGH
--------	------------	-------------

$$\text{NEW IP} = \text{CURRENT IP} + \text{OFFSET} \text{ (16 bit)}$$

Figure 4.3 | Format of the direct near CALL instruction

The direct call is like a direct jump instruction and is three bytes long. It is relative and the destination can be -32,768 bytes to +32,767 bytes from the address of the instruction following the call (this will be the current content of the IP). This means that the offset can be a 16-bit signed number. When this call is executed, the new value of IP = old IP + offset, where the second and third bytes of the instruction give the offset. The assembler must calculate the offset during its first pass. Example 4.5 uses this type of the call instruction – the direct near call.

CALL NEW_WAY ;using the label, the assembler calculates the offset

CALL MULTI ;calls a procedure named MULTI

ii) Indirect CALL

Usage: CALL reg16, CALL [reg16]

In this case, the destination is specified in a 16-bit register or in a memory location pointed by a register. This is not a ‘relative’ call. The content of the referred register or memory location is loaded into IP for using the procedure.

CALL BX ;the procedure's address is in BX
CALL WORD PTR[BX] ;the address of the procedure is in memory and
;is pointed by BX

The following program illustrates the use of an indirect near call. Example 4.6 illustrates the use of the CALL register instruction to call procedures that begin at offset addresses ENT and DISP. (These calls could also call the procedures directly as CALL ENT and CALL DISP.) The offset address ENT is placed in BX and DISP in SI, and then the CALL BX and CALL SI instructions call these procedures. The ENT procedure allows keyboard entry with echo. If an upper case letter is entered by ENT, the calling program converts into lowercase and the DISP procedure displays the corresponding lowercase letter.

Example 4.6

```

.MODEL TINY      ;select tiny model
.CODE           ;start code segment
.STARTUP        ;start program
LEA BX, ENT    ;offset of ENT in BX
LEA SI, DISP   ;offset of DISP in SI
CALL BX         ;call the ENT procedure
ADD AL, 20H     ;upper case to lower case
MOV DL, AL       ;transfer lower case to DL
CALL SI         ;call the DISP procedure
.EXIT

ENT PROC NEAR
    MOV AH, 01      ;procedure for entering key
    INT 21H
    RET
ENT ENDP

DISP PROC NEAR
    MOV AH, 02      ;procedure for displaying
    INT 21H
    RET
DISP ENDP
END

```

4.2.2.2 | Intersegment or Far Call

Direct Far Call

A far call is an intersegment call, which means that the destination address is in a different code segment. This will be a 5-byte instruction, the first byte being the opcode, the second and third bytes being the new value of IP and the fourth and fifth being the new values of CS. This is not a relative call. When the procedure is called, the IP and CS values are replaced by the corresponding values in the call instruction as shown in the Fig. 4.4.

OPCODE	IP LOW	IP HIGH	CS LOW	CS HIGH
--------	--------	---------	--------	---------

Figure 4.4 | Format of the far jump instruction

Indirect Far Call

For an indirect call, the destination address is not in the instruction – rather, it is in a register or memory. For a far call, four bytes are needed to specify a destination. Obviously, a register cannot specify it. Hence, the four bytes needed to specify a destination are stored in memory and pointed by a register. As an example, CALL DWORD PTR [SI] can be a

far call instruction. [SI] and [SI + 1] gives the new value of IP, and [SI + 2] and [SI + 3] gives the new value of CS.

Far calls and procedures are specified using the ‘far’ directive when defining the procedure. In Chapter 5, we will see how the EXTRN directive is used in this context.

4.2.3 | The RET Instruction

i) Usage: RET

When a procedure is called, the current value of IP is pushed on to the stack. This naturally means that when the procedure has ended, control should go back to the main program at the point where it had branched off. This means that the value of IP (and CS for a ‘far’ call) will have to be retrieved. This is done by the ‘return’ instruction which is the last instruction in the procedure. The execution of RET causes the return address to be popped from the stack to IP or IP and CS. (Whenever a procedure is defined, it is known whether it is a far or near procedure. From this, it is determined whether the stack has saved just the old IP value or both IP and CS.)

ii) Usage: RET n

This is another form of the RET instruction. This adds the number ‘n’ to the stack pointer (SP) after the return address has been popped off the stack on return from a procedure. We will see the use of this in Example 4.10.

4.2.4 | The Use of the Stack in Procedure Calls

The stack is used in procedure calls for saving the return address. There is another context when a stack is necessary. The main program may be using a number of general-purpose registers. Since the number of registers is limited, what happens if the procedure too needs a few of those registers? If the procedure uses the same registers, obviously their content will be changed. To avoid this problem, the contents of these registers (including the flag register) may be pushed on to the stack before calling the procedure and popped back from stack on returning from the procedure. This allows both the main program as well as the procedure the use of the same registers without losing their content. It is up to the programmer to write these push and pop instructions (either in the main program or in the procedure). The only point to remember would be to push and pop in reverse order. However, now MASM 6.x has

a construct ‘USES’ which automatically pushes the specified registers onto the stack. The popping will be automatic on returning from the procedure. Example 4.7a illustrates this.

Example 4.7a

```
.MODEL SMALL
.DATA
-----
.CODE
.STARTUP
.
.
.
CALL MULTI
.
.
.

MULTI    PROC NEAR USES DX CX
        MUL BL
        MOV DX, 4509H
        ADD AX, DX
        MOV CX, 0608H
        ADD AX, CX
        RET
MULTI    ENDP
END
```

Now see Example 4.7b. The main program is not shown. What is illustrated is that it uses registers DX and CX. These registers are used in the procedure MULTI as well. The USES construct at the start of the procedure will be converted to PUSH DX and PUSH CX when the procedure is called and to POP CX and POP DX when returning from the procedure. This can be verified by ‘debugging’ the program or by using the directive.list all. Then, the list file will show the listing of the procedure as shown below. Note the push and pop DX and CX which we have not written specifically in the program but have been caused by the USES construct.

Example 4.7b

```

002B          MULTI  PROC NEAR USES DX CX
002B  52      *      push   dx
002C  51      *      push   cx
002D  F6 E3           MUL BL
002F  BA 4509          MOV DX, 4509H

0032  03 C2           ADD AX, DX
0034  B9 0608          MOV CX, 0608H
0037  03 C1           ADD AX, CX
                           RET
0039  59      *      pop    cx
003A  5A      *      pop    dx
003B  C3      *      ret    00000h
003C          MULTI  ENDP
                           END

```

4.2.5 | Passing Parameters To and From Procedures

When procedures are called, they have to be given data on which to work on, and then these procedures will have to return results to the calling program. There are various ways of doing this. One way would be to place it in registers which can be accessed by the main program as well as the procedure. Another way would be to place data in memory and access it by name or by using pointers. Still another method is to use the stack for this. Data or addresses made available to both the calling and called programs are called ‘parameters’ and now we will discuss various ways of ‘passing parameters’ to and from procedures.

4.2.5.1 | Passing Parameters Through Registers

In this case, data is placed in registers by the main program, and these registers are used by the procedure. Example 4.8 is a program which calculates and places in memory the Nth term of an arithmetic progression. The formula for the Nth term of an A.P. is $A + (N - 1) D$, where A is the first term and D is the common difference. Here, the values of A, N and D are placed in the data segment from where the main program takes them, loads them into registers and passes the values to the procedure. The procedure does the computation and passes the result to the main program through register BX.

Example 4.8

```

.MODEL SMALL
.DATA
A DB 4          ;A = 4
D DB 5          ;D = 5
N DB 20         ;N = 20
NTH DW?

.CODE
.STARTUP
MOV BL, A        ;copy A to BL
MOV BH, 0        ;BX = 0 to have A in BX as a word
MOV AL, D        ;copy D to AL
MOV CL, N        ;Copy N to CL
DEC CL           ;decrement CL to get N-1
CALL NTH_TERM_AP ;call the procedure

MOV NTH, BX      ;save the nth term in memory
.EXIT

NTH_TERM_AP PROC NEAR
    MUL CL          ;multiply (N-1) x D, product in AX
    ADD BX, AX      ;Nth term = A + (N-1)D will be in BX now
    RET
NTH_TERM_AP ENDP
END

```

4.2.5.2 | Passing Parameters Through Memory

In Example 4.9, the data which is used by the procedure is accessed from memory through the pointer register BX, and data is put back in memory in the same way. Example 4.9 is a program for arranging a set of numbers (stored in memory) in descending order. The method involves comparing numbers pair-wise and repeating this $N - 1$ times if there are N numbers to be sorted. The steps are:

1. Suppose the numbers are 6, 8, 34, 0 and 67. The first two numbers are compared. If the first number is greater than the second, no change is made to the ordering. Otherwise, exchange them such that the bigger number comes first in the pair. For our set, now the ordering becomes 8, 6, 34, 0 and 67.
2. Next, the second and third numbers (i.e., 6 and 34) are compared. The new ordering is 8, 34, 6, 0 and 67.
3. This is repeated $N - 1$ times, so that due to pair-wise ordering, the bigger number comes first in the new ordering of pairs. In Example 4.9, the procedure one_set does this.
4. This sequence of steps is repeated $N - 1$ times, such that we get all the numbers sorted in descending order, replacing the unsorted array.

Example 4.9

```

.MODEL SMALL
.DATA

N      WORD 0009H
NUMS BYTE 10H, 08, 89H, 78H, 67H, 07, 2, 99H, 0FEH

.CODE
.STARTUP
    LEA BX, NUMS ;BX to point to the number array
    MOV CX, N      ;move the array size to CX
    DEC CX         ;CX must contain N-1
    MOV SI, BX     ;SI to have copy of starting address
AG:   MOV DX, CX     ;DX = N-1
STRT: CALL ONE_SET ;the procedure for pair-wise ordering
      DEC DX        ;keep a count of the numbers of calls
      JNZ STRT      ;repeat until DX = 0

      MOV BX, SI     ;put the starting address back in BX
      LOOP AG        ;loop back until CX = 0
      .EXIT

ONE_SET PROC NEAR
    MOV AL, [BX]    ;get the first number in AL
    INC BX          ;increment the pointer
REPEA: MOV AH, [BX]    ;get the second number in AH
        CMP AL, AH    ;compare the numbers
        JB EXCH       ;jump if first number is below
BACK:  RET           ;otherwise return
EXCH:  MOV [BX], AL   ;put AL in second address
        MOV [BX-1], AH ;put AH in first address
        JMP BACK      ;then return
ONE_SET ENDP          ;end the procedure
END

```

In the above program, nine numbers (bytes) are stored in the data segment and sorted. Any number of bytes can be sorted this way, limited only by the maximum value of N. For sorting in ascending order, only one instruction in the procedure need be changed (which one?). After sorting is done, if we write a procedure for converting hexadecimal numbers to ASCII, the sorted numbers can be displayed as well.

4.2.5.3 | Passing Parameters Through the Stack

There may be reasons where the stack, which is an area in the memory, can be used to store data. The procedure can take data from the stack for its computation. Example 4.10 is a simple example which also elaborates how the register BP is used to access data in the stack. Here, four words which are in the data segment are pushed on to the stack by the main program. The procedure accesses the data using the indexing features of BP. Remember that BP is a register associated (by default) with the stack

segment and that BP cannot be used without an offset. Example 4.10 calculates $(A + B) - (E + D)$ where A, B, E and D are words stored in data memory. The result is also to be saved in the data segment.

Note We do not use C as a label because 'C' is a reserved word in MASM.

Example 4.10

```
.MODEL SMALL
.STACK 100H
.DATA
A DW 0987H           ;data values in the data segment
B DW 678H
E DW 0A5EH
D DW 0034H
RESULT DW?
.CODE

.STARTUP
PUSH A             ;push to stack
PUSH B
PUSH E
PUSH D
CALL COMPUTE       ;call the procedure
MOV RESULT, AX     ;save AX in memory
.EXIT

COMPUTE PROC NEAR
MOV BP,SP          ;copy SP to BP
MOV AX, [BP + 8]   ;move the top most stack value to AX
MOV BX, [BP + 6]   ;move next word in stack to BX
ADD AX, BX
MOV CX, [BP + 4]   ;move the next word to CX
MOV DX, [BP + 2]   ;move the next word to DX
ADD CX, DX
SUB AX, CX
RET 8              ;return and add 8 to SP
COMPUTE ENDP
END
```

Notes The salient features of the program are:

1. A stack of 100 bytes has been defined in the beginning.
2. The push instructions are in the main program.
3. In the procedure, the SP value is copied to BP.
4. The data in the stack is accessed by adding offsets to BP.
5. The return instruction is RET 8. This causes SP to have the value it had before the four PUSH operations are executed. Otherwise, what would happen is that each time this program is run, the stack pointer decrements

by 8 bytes (for the four PUSH operations). Thus, the stack size gets reduced by that much for each procedure call which will finally cause the stack size to reduce to zero, thus causing system crash.

6. RET 8 adds 8 to SP after returning from the procedure. In effect, it erases the data that had been pushed on to the stack.

Refer to Fig. 4.5. Let us say that before the first PUSH, the value of SP = 0120H. This figure shows the content of the stack after the four PUSH operations. The address of SP now is 0118H. When the CALL is executed, SP = 0116 H, because the content of IP is pushed on the stack. Before returning from the procedure, SP = 0116 H. RET 8 causes IP to be popped. So, SP = 0118 H. Adding 8 to this value of SP causes SP to become 0120 H again. It is important to be very clear about stack operations if you plan to use the stack. Any wrong alteration of the stack can cause a system crash.

Stack address	Stack content
011FH	09
011EH	87
011DH	06
011CH	78
011BH	0A
011AH	5E
0119H	00
0118H	34

A B C D E

Figure 4.5 | Stack operation for the PUSH and RET 8 instructions

Stack Overflow and Underflow

SP can have a maximum value of FFFFH. For each PUSH operation, the SP value decrements by 2, and in the limiting case, it can go to SP = 0000.

Any PUSH operation beyond this will cause a ‘stack overflow’. This creates a condition when there is no space in the stack for new data.

Stack underflow is the other case when POP operations cause SP to have values beyond the defined top of stack.

4.3 | Macros

The name of our assembler is ‘Macro assembler’. As such, we see that it is expected to be able to handle an item called ‘macros’. What is a macro? It is like an opcode – when used, it executes. A macro, when called by

name, executes the instructions that are listed under that name. Thus, essentially, a macro is a short hand notation for a number of instruction lines. It essentially makes assembly language coding more readable and helps to avoid repetitive coding.

Usually, a macro, like a procedure, is defined for performing a specific function. However, the ‘overheads’ involved in invoking a procedure are not incurred here. A procedure call causes pushing and popping of addresses/data in stack. A macro when invoked just expands the code by putting in all the instructions corresponding to the called macro. It does not have to ‘call’ and ‘return’. Thus, when we write our code with macros, it may look small, but when assembling the code, each macro is replaced by the full set of instructions it consists of. Thus, we can say that macros execute faster but the assembled code takes more memory. This is because a procedure is written only once in memory, but the macro statements are written as part of the code every time the macro is invoked.

4.3.1 | Writing a Macro

Whenever we have a set of code lines that we may use frequently, it could be convenient to make a macro out of it. A macro has the following format:

```
MACRO NAME MACRO [parameter list]
    Instructions (body of the macro)
    ENDM
```

Let us think of a simple task that could be defined as a macro. If we frequently want to enter data through the keyboard, it could be written as a macro.

```
ENTR MACRO ;the macro for entering a character with echo
MOV AH, 01H
INT 21H
ENDM
```

This macro does not have parameters to be passed to it. Another similar macro is for displaying a character as shown below.

```
DISP MACRO          ;the macro for displaying a character
    MOV AH, 02H
    INT 21H
ENDM
```

When writing programs using macros, make sure the macros are defined before they are used. Example 4.11 is a simple illustration of these principles.

Example 4.11

```
.MODEL TINY
.CODE
.STARTUP
ENTR MACRO      ;define the macro for entering a key
MOV AH, 01H
INT 21H
ENDM

DISP MACRO      ;define the macro for displaying a character
MOV AH, 02H
INT 21H
ENDM

ENTR            ;invoke macro ENTR
MOV DL, AL       ;copy ASCII code of key from AL to DL
DISP            ;invoke macro DISP
EXIT
END
```

When we run the above program, we get the pressed key displayed twice. Why?

Now, let us define and use a macro using parameters.

```
DISPLAY  MACRO  STRINGDAT
        MOV     AH,09
        LEA     DX,STRINGDAT
        INT     21H
```

The above is a macro which displays the string whose address is loaded into DX. The parameter is 'STRINGDAT'. It is also called a dummy variable. When the macro is invoked, the actual name of the string to be displayed may be used. Example 4.12 illustrates these ideas.

Example 4.12

```
.MODEL SMALL
.DATA
MESG1 DB "MY NAME IS ANTONY GONSALVES$"
MESG2 DB "I AM SAMANTHA$"
.CODE
.STARTUP
DISPLAY MACRO STRING ;define a macro for string display
    MOV AH, 09
    LEA DX, STRING
    INT 21H
    MOV DL, 0AH ;DL = ASCII data for newline
    MOV AH, 02
    INT 21H
    MOV DL, 0DH ;DL = ASCII data for carriage return
    MOV AH, 02
    INT 21H
    ENDM
DISPLAY MESG1 ;invoke macro for displaying MESG1
DISPLAY MESG2 ;invoke macro for displaying MESG1
.EXIT
END
```

In the above program, the DISPLAY macro is first defined. Then, the macro is invoked twice with two different parameters MESG1 and MESG2. Note that the macro not only displays a string but also has instructions for displaying new line and carriage return. This ensures that after displaying one line the cursor moves to the left side of the next line. Thus, in Example 4.12, the two messages are displayed on separate lines.

4.3.2 | Using the ‘Local’ Directive in Macros

Whenever macros use labels, they should be declared as local to the macro. Otherwise, if the same macro is used many times, they will create assembly errors, as the label will correspond to different addresses each time. So, just after the macro name is declared, the labels must be - written preceded by the word ‘LOCAL’. Any number of labels can be declared in this way.

It would be a good idea for you to examine the list file of the following program to see how the label ‘AGAIN’ is being managed.

Example 4.13

```

.MODEL TINY
.CODE
.STARTUP

STAR MACRO N      ;define macro STAR with parameter N
LOCAL AGAIN        ;declare AGAIN to be a local label
MOV CL, N          ;move count to CL
MOV DL, '*'        ;move '*' to DL
AGAIN: MOV AH, 02   ;AH = 2 for displaying a character
    INT 21H
    DEC CL           ;decrement the count
    JNZ AGAIN        ;repeat the displaying until CL = 0
    ENDM             ;end the macro

NEW    MACRO         ;define a macro for moving to next line
    MOV DL, 0AH
    MOV AH, 02
    INT 21H
    MOV DL, 0DH
    MOV AH, 02
    INT 21H
    ENDM

    STAR 5          ;invoke macro STAR with N = 5
    NEW             ;invoke macro NEW
    STAR 4          ;invoke macro STAR with N = 4
    NEW
    STAR 3
    NEW
    STAR 2
    NEW
    STAR 1
    NEW

.EXIT
END

```

Example 4.13 shows such a case. Two macros have been defined – STAR and NEW. The former contains a label AGAIN which has been declared as LOCAL. This macro causes the character '*' to be printed on a line N times. N is a parameter whose value is passed to this macro, when called. The macro NEW just moves the cursor to the left side of the next line. The output of Example 4.13 is as shown below.

```

*****
 ****
 ***
 **
 *
C: \masm6.14\BIN>

```

We know that computers do all their calculations using binary arithmetic, but we are accustomed to doing calculations in decimal form. We see numbers printed out in the decimal form and enter data using the keyboard as decimal numbers. Our conclusion obviously is that though arithmetic data is processed mostly in binary form by computers, there are methods to convert binary data to forms better suited for display and understanding. Also, if we want to process numeric data in a format other than binary, that should also be possible.

As such, let us examine some of the number formats frequently used. Two of the most widely used formats are BCD and ASCII. BCD is ‘binary coded decimal’ – two versions of which are ‘packed BCD’ and unpacked BCD. As an example of the former, a decimal number 56 (say) is written as four bit binary packed in a single byte as 0101 0110. Unpacked BCD means using a whole byte to represent a decimal number. For example, 56 is written as two bytes – 0000 0101(5) 0000 0110(6). Another popular and very important number format is the ASCII format. It represents a decimal digit in a byte. Thus, 6 is 0011 0110 or 36H. The decimal digits from 0 to 9 have their ASCII representation as 30H, 31H ... 39H.

Some useful conversions are between the following different number formats:

1. Packed BCD to unpacked BCD.
2. Unpacked BCD to packed BCD.
3. Unpacked BCD to ASCII.
4. Binary to ASCII.
5. ASCII to binary.

Note Number format conversions have been discussed in detail in Chapter 0. If you have doubts regarding such conversions, refer to Section 0.6.

4.4.1 | Packed BCD to Unpacked BCD Conversion

This has been done in Example 3.25 for a 2-digit packed BCD number. The exact method of conversion is detailed there along with the example. We will now see how an eight-byte packed BCD is unpacked, converted to ASCII and displayed.

In Example 4.14, we see that an eight-digit decimal number is represented in packed BCD, such that each decimal digit has a four-bit representation. This is stored as a double word (four bytes). However,

when accessing it for processing, one byte each of this double word is moved to AL and processed. The steps are:

1. A procedure BCD_TO converts packed BCD to unpacked BCD.
2. This procedure saves CX and AX as they are used in the procedure.
3. To convert to ASCII, two unpacked BCD bytes are put in a word and logically ORed with 3030H.
4. After the conversion of the 8 bytes is over, we find that the ASCII values are in memory with the most significant digit in the highest memory pointed by BX. The instructions from the label STRT onwards are for displaying the ASCII digits.

Example 4.14

```

.MODEL SMALL
.DATA
BCD DD 67450823H ;store the 4-byte packed BCD number
ASC DB 8 DUP(?) ;allocate space for the 8-byte
;ASCII number

.CODE
.STARTUP
LEA SI, BCD ;load the offset of BCD in SI
LEA BX, ASC ;load the offset of ASC in BX
MOV CX, 4 ;count of BCD bytes
RPT: CALL BCD_TO ;call the conversion procedure
INC SI ;increment pointer
INC BX ;increment pointer
LOOP RPT ;continue until CX = 0
;with the above loop, the
;conversion is
;done and only displaying is left
MOV CX, 08 ;count of ASCII bytes
DEC BX ;to point to address of last
;ASCII digit
STRT: MOV DL, [BX] ;move ASCII value to DL for
displaying
    MOV AH, 02 ;function to display a character
    INT 21H
    DEC BX ;point to next ASCII digit
    LOOP STRT ;repeat until CX = 0
.EXIT

BCD_TO PROC NEAR USES CX AX
    MOV AL, [SI] ;move first BCD byte to AL
    MOV AH, AL ;copy it to AH as well
    AND AL, 0FH ;mask the upper nibble of AL
    AND AH, 0FOH ;mask the lower nibble of AH
    MOV CL, 04 ;count of shift operations
    SHR AH, CL ;shift AH 4 times
    OR AX, 3030H ;convert AX to two ASCII bytes
    MOV [BX], AL ;move one ASCII value to memory
    INC BX ;BX to point to next locations
    MOV [BX], AH ;store the next digit
    RET
BCD_TO ENDP
END

```

Converting from ASCII to packed BCD is just a reverse of all these processes.

4.4.2 | BCD Calculations

i) Addition

DAA – Decimal adjust AL after addition.

Usage: DAA

For BCD, none of the digits should have a value greater than 9, but we know that a nibble can contain a value of OFH. This causes problems when BCD numbers are added. To sort out this problem, there is a specific instruction DAA which stands for ‘Decimal Adjust Accumulator’. **This assumes that the sum of the BCD addition is in AL.**

The need for this instruction is because direct addition of BCD numbers causes errors, and a correction is usually needed. The details of the correction done are discussed in Section 0.7.2. Now, see the following example which adds two multi byte BCD numbers to get the result in BCD itself. The two BCD numbers are stored as double words (DD).

Note When they are stored thus, the lowest byte of the number will be in the lowest address. Now, BX and SI point to the BCD numbers to be added. For adding, only one byte each of the numbers is accessed and added. The sum is stored in the byte location SUM. Here too, the LSB of the sum will be in the lowest address.

Example 4.15

```

.MODEL SMALL
.DATA
BCD1 DD 45679834H ;store the first BCD byte as a double word
BCD2 DD 93870989H ;store the second BCD byte as double word
SUM DB 5 DUP(0) ;allocate 5 byte memory for the sum

.CODE
.STARTUP
MOV AH, 0 ;clear AH
CLC ;clear carry flag
LEA BX, BCD1 ;BX to point to first BCD byte
LEA SI, BCD2 ;SI to point to second BCD byte
LEA DI, SUM ;DI to point to SUM location
MOV CX, 04 ;count for the number of bytes
RPT: MOV AL, [BX] ;copy the bytes of the first number to AL
      MOV DL, [SI] ;copy the bytes of the first number to AL
      ADC AL, DL ;add with carry AL and DL
      DAA ;do the BCD correction
      MOV [DI], AL ;store the result of each byte addition
      INC BX ;increment pointer BX
      INC SI ;increment pointer SI
      INC DI ;increment pointer DI
      LOOP RPT ;loop until CX = 0

      ADC AH, 0 ;add carry, AH and 0;sum in AH
      MOV [DI], AH ;store AH in memory
.EXIT
END

```

On adding 4 BCD bytes, there is a possibility of a carry, which corresponds to the 5th BCD byte of the sum. To preserve this final carry, AH register is used as shown. The sum of the given two numbers is 0139559823.

ii) Subtraction

DAS – Decimal adjust AL after subtraction.

Usage: DAS

For BCD subtraction, there is an instruction DAS which operates on the data in AL. Refer Section 0.7.5 for the intricacies of BCD subtraction. DAS is used (like DAA) to make the necessary corrections. The use of this instruction is left to you.

The processor has a number of instructions relating to the processing of ASCII numbers.

4.5.1 | Ascii Addition

AAA – ASCII Adjust AL after addition

This instruction is used after addition of two ASCII numbers. It checks the lower nibble of AL for one of the two conditions:

1. whether it is within A and F.
2. whether the auxiliary carry flag (AF) is set.

Depending on this, it performs the required corrections. See the three cases shown.

Add ‘5’ and ‘4’, get the sum in AL.

i) **Case 1**

35H

34H

69H

In this case, the lower nibble is not within A and F, and the AC flag is not set, i.e., AF = 0. Then, the instruction just resorts to making the upper nibble of AL to 0. Thus, if the AAA instruction is used after the addition, the sum becomes 09, i.e., 0000 1001. To convert it back to ASCII, OR it with 30H.

MOV AL, ‘5’	;AL = 35H
ADD AL, ‘4’	;AL = 35H + 34H = 69H
AAA	;AL = 09H
OR AL, 30H	;AL = 39H, the ASCII value of 9

We get 39H in AL, which is the ASCII code of 9. Also, AH = 0.

Case 2

Add ‘7’ and ‘6’, get the sum in AL.

37H +
36H
6DH

In this case, the right most nibble is greater than 9 – within A and F – and it is no longer representing a decimal digit. Hence, 6 is added to the right most nibble (like in BCD correction, the number 6 is added because that is the difference between decimal and hexadecimal); it also clears the upper nibble of AL, sets CF and AF, and adds 1 to AH. Thus, if the AAA instruction is used after the addition, the sum in AL is now 03 and AH = 01 (if AH had been cleared earlier).

MOV AL, '7'	;AL = 37H
ADD AL, '6'	;AL = 37H + 36H = 6DH
AAA	;AL = 03, AH = 01, CF = 1
OR AX, 3030H	;AX = 3133

The AX register will contain 3133H after this, which is the ASCII value of 13. These facts must be taken into consideration when adding multibyte ASCII numbers.

Case 3

Add '9'and '8', get the sum in AL.

39H	+
38H	
<u>71H</u>	

Here, the right-most nibble is not within A and F, but AF = 1. Hence, 6 is added to the right most nibble, and it also clears the upper nibble of AL, sets CF and AF, and adds 1 to AH. Thus, if the AAA instruction is used after the addition, AL = 7, AH = 1, CF = 1 and AF = 1. In all cases, AAA clears the upper nibble of AL. These facts must be taken into consideration when adding multibyte ASCII numbers.

4.5.2 | Ascii Subtraction

AAS – ASCII adjust AL after subtraction.

Usage: AAS

This function is similar to the AAA instruction. After the subtraction of two ASCII numbers, AAS checks the lower nibble of AL. If this number has a value between A and F, or if the AF flag is set, 6 is subtracted from AL. Also, 1 is subtracted from AH, and AF and CF are set. In all cases, the upper nibble of AL is cleared.

Case 1

Subtract ‘5’ from ‘8’.

$$\begin{array}{r} 38H \\ - \\ 35H \\ \hline 03 \end{array}$$

If AAS is used after this subtraction, the content AL is 03.

Case 2

Subtract ‘8’ from ‘5’.

$$\begin{array}{r} 35H \\ - \\ 38H \\ \hline FDH \end{array}$$

After AAS, AH = FFH, AL = 07.

Actually, the answer should be -3 but we get FF07, which is the ten’s complement $10 - 3 = 7$.

Let us subtract ‘4’ from ‘15’. We get the ASCII value of ‘11’ in the AX register with the following program.

MOV AX, ‘15’	;AX = 3135H
SUB AX, ‘4’	;AX = 3101H
AAS	;AX = 3101H
OR AX, 0030H	;AX = 3131H

Now, it is up to you to find out what happens when ‘9’ is subtracted from ‘4’. Also, note that the instructions AAA and AAS can be used after adding or subtracting unpacked BCD numbers as well.

4.5.3 | Multiplication and Division

AAM – ASCII adjust AX after multiplication.

Usage: AAM

For multiplication and division of ASCII numbers, we need to convert them to unpacked BCD. Let us deal with ASCII multiplication first. In the following program, the numbers to be multiplied are 6 and 9. The product has to be 54. The ASCII numbers are first unpacked and multiplied. After that, the AAM instruction is used. This causes the product to be in the unpacked BCD format. If the result of multiplication

(MUL BL) is compared with the result of AAM, it is obvious that AAM accomplishes the conversion by dividing AX by 10. Following this, if ORing with 3030H is done, the ASCII value of the product is available in AX. In the program, this value is copied to CX and displayed using the DISP macro.

Note The most significant digit is displayed first.

Example 4.16

```
.MODEL TINY
.CODE
.STARTUP
DISP MACRO           ;define macro for display
MOV AH, 02
INT 21H
ENDM                 ;end of macro

MOV AL, '6'           ;AL = 36H
AND AL, 0FH           ;AL = 06H
MOV BL, '9'           ;BL = 39H
AND BL, 0FH           ;BL = 09H
MUL BL               ;AX = 0036H;decimal value is 54
AAM                  ;AX = 0504H, the unpacked BCD of 54
OR AX, 3030H          ;AX = 3534H, the ASCII value of 54
MOV CX, AX            ;copy AX to CX
MOV DL, CH            ;DL = CH to display the MSD
DISP                 ;call macro for display
MOV DL, CL            ;DL = CL to display the LSD
DISP                 ;call macro for display

.EXIT
END
```

AAD – ASCII adjust AX before division.

Usage: AAD

This instruction works by converting the unpacked BCD in AX to binary (hex) before division. It multiplies AH by 10, adds the product to AL and clears AH. This is the hex representation of the BCD number in AX. After division, we find the quotient in AL and the remainder in AH. See the following program segment.

MOV AX, '82'	;AX = 3832H
AND AX, 0F0FH	;AX = 0802H
AAD	;AX = 0052H, the hex value of the dividend
MOV BL, '9'	;BL = 39H, the ASCII value of 9
AND BL, 0FH	;BL = 09H
DIV BL	;AX = 0109H, the quotient in AL, remainder in AH
OR AX, 3030H	;AX = 3139H, the corresponding ASCII values

4.6 | Conversions for Computations and Display/Entry

We have seen various number formats so far, but by now it should be obvious that using BCD and ASCII numbers for computation is cumbersome, especially when large numbers are involved. A more convenient format for all arithmetic calculations is the binary format, which is compactly written in hexadecimal. Another point is that, once the computation is done, the natural format for display is the ASCII format. For displaying, we convert the binary number to unpacked BCD and then to ASCII. When entering data through the keyboard also, the ASCII number format is used. All this implies the necessity of converting binary numbers to ASCII and vice versa.

4.6.1 | Converting Ascii Numbers to Binary Form

Let us see how we convert a decimal number (say 6754) to binary form.

$$\begin{array}{r}
 6754 = 6 \times 1000 + 7 \times 100 + 5 \times 10 + 4 \times 1 \\
 4 \times 1 = 4 \quad + \\
 5 \times 10 = 50 \quad + \\
 7 \times 100 = 700 \quad + \\
 6 \times 1000 = \underline{\underline{6000}} \\
 \hline
 \underline{\underline{6754}}
 \end{array}$$

This is the algorithm to be used for conversion. Let us write a program which does this. Here, the multiplicands 1, 10, 100 and 1000 are stored in the data segment. The binary value also is finally stored in data memory.

Example 4.17

```

.MODEL SMALL
.DATA
ASCDAT DB '6754'           ;ASCII value of 6754
MULTI  DW 01,0AH,64H,3E8H
BINAR  DW 0

.CODE
.STARTUP
DIG EQU 4                  ;DIG specifies the number of digits
MOV CX, DIG                ;move into CX the number of digits
MOV DI, DIG-1              ;DI to point to LSD of the ASCII no.
LEA SI, MULTI              ;use SI to point to the multiplicands
RPT: MOV AL, ASCDAT[DI]     ;bring first ASCII data to AL
    AND AX, 000FH            ;mask AX
    MOV BX, WORD PTR[SI]     ;get the multiplicand to BX
    MUL BX                  ;multiply AX by BX
    ADD BINAR,AX             ;add to binary value
    DEC DI                  ;decrement byte pointer
    INC SI                  ;increment word pointer
    INC SI                  ;two increments required
    LOOP RPT                ;repeat until CX = 0
.EXIT
END

```

The above program is a direct application of the algorithm mentioned. We get the converted binary value in the word location BINAR. We can extend this for more number of digits too.

4.6.2 | Converting Binary Numbers to ascii Form

The method for this is repetitive division. This has been done in Example 3.22 for a 16-bit binary number.

4.7 | Signed Number Arithmetic

In all our discussions and problems, whenever we dealt with numbers, we assumed the numbers to be unsigned. Now, let us bring signed numbers also into our domain.

Negative numbers are represented in the two's complement format. When the data length is one byte, the maximum value of decimal numbers it can represent is -128 to $+127$. With 16 bits, this is from $-32,768$ to $+32,767$. If the result of any operation exceeds this word size, the overflow flag is set. This indicates that our result has to be re-

interpreted and corrected. The conditions under which the overflow flag is set can be stated as:

1. When there is a carry from D6 to D7, or a carry from D7 outwards, but not both. This is for byte operations.
2. For word operations, when there is a carry from bit D14 to D15, or a carry from D15 outwards, but not both.

Let us see a few cases of signed number operations.

Example 4.18

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, +34
ADD AL, -23
.EXIT
END
```

For the above program in which +34 (22H) and -23 (E9H) are decimal numbers, the calculation in binary form:

$$\begin{array}{r} 0010\ 0010 \quad + \\ 1110\ 1001 \\ \hline 1\ 0000\ 1011 \end{array}$$

Thus, the sum is 0BH (11 in decimal), which is the right sum. CF = 1, OF = 0, SF = 0.

Example 4.19

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, -28 ;AL = E4H
ADD AL, -78 ;AL = B2H
.EXIT
END
```

Computation:

$$\begin{array}{r} 1110\ 0100 \quad + \\ 1011\ 0010 \\ \hline 1\ 1001\ 0110 \end{array} \quad \text{OF} = 0, \text{SF} = 1, \text{CF} = 1$$

The answer of this computation is 96H (-106), which is correct.

In both the above problems, the overflow flag is found cleared, which indicates that the signed number computation has produced the right result. Now, see the next program.

Example 4.20

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, +100
ADD AL, +75
.EXIT
END
```

Computation:

0110 0100 (64H)	
0100 1011 (4BH)	
<u>1010 1110 (AFH)</u>	OF = 1, SF = 1, CF = 0

In the above case, the overflow flag will be set, because there has been an overflow from bit D6 to D7, and the sign bit is set. Even though two positive numbers have been added, the sign bit indicates a negative sum. Thus, AFH has to be thought of as a negative number, which is not correct. The problem is that the sum is +175, which is too large to fit in an 8-bit register. Since the overflow flag is set, it indicates that the register AL has **overflowed the size allotted for signed operations in 8-bit format**.

Understanding that the problem that has caused the error is one of inadequacy of the size of the destination, it can be solved by extending the bit size of the operands. This can be done by sign extending a byte to a word and a word to a double word. The 8086 has two specific instructions for it.

i) CBW – convert byte to word.

This instruction works on the data in AL. It copies the sign bit (D7) of AL to all the bits of AH, Thus, the byte in AL is extended to a word in AX.

ii) CWD – convert word to double word.

This instruction works on the data in AX. It copies the sign bit (D15) of AX to all the bits of DX. Thus, DX - AX is the new double word. Let us now use these instructions to avoid overflow in the case of signed number operations.

Example 4.21

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, +100
CBW
ADD AX, +75
.EXIT
END
```

Computation:

$$\begin{array}{r} 0000\ 0000\ 0110\ 0100 \quad + \\ 0000\ 0000\ 0100\ 1010 \\ \hline 0000\ 0000\ 1010\ 1110 \end{array} \quad OF = 0, SF = 0, CF = 0$$

In this case, the overflow flag is not set and hence the sum in AX is considered to be correct. The sign bit also indicates a positive number.

Note In a general case, we are not sure if the result of adding/subtracting signed numbers will exceed the register size allowed for it. To accommodate for a correction as in Example 4.21, the instruction JO (Jump on overflow) can be used. The program can be written in a way that after the addition or subtraction (as the case may be) operation, the JO instruction tests the overflow flag and the CBW or CWD instruction is used only if the overflow flag is found to be set. Then, re-computation using the extended word or double word can be done.

4.7.1 | Comparison of Signed Numbers

Comparison of two unsigned numbers is based on their numerical value only, and the sign does not come into the picture. For example, 7 is greater than 3 numerically. However, when these numbers are signed, the picture changes. Think of the numbers -7 and -3 . We now say that -3 is greater than -7 . With this picture, we use the terms ‘above’ and ‘below’ for unsigned numbers, and ‘greater than’ and ‘less than’ for signed numbers. The mnemonics used and the flags used also are different.

In Chapter 3, Table 3.2 lists out a number of conditional jump instructions catering to unsigned number operations. Now, let us have a look at the jump instructions for signed number operations. See Table 4.2.

Table 4.2 | Conditional Jump Instructions Catering to Signed Numbers

SI No.	Mnemonic	What it means	Conditions tested
1	JE	Jump if Equal	ZF = 1
2	JG	Jump if Greater	ZF = 0 and SF = OF
3	JGE	Jump if Greater or Equal	SF = OF or ZF = 1
4	JL	Jump if Less	SF != OF
5	JLE	Jump if Less or Equal	ZF = 1 or SF != OF
6	JNG	Jump if Not Greater	ZF = 1 or SF != OF
7	JNGE	Jump if Not Greater Nor Equal	SF != OF
8	JNL	Jump if Not Less	SF = OF
9	JNLE	Jump if Not Less Nor Equal	ZF = 0 and SF = OF
10	JNO	Jump if No Overflow	OF = 0
11	JO	Jump if Overflow	OF = 1
12	JNS	Jump if Not Signed	SF = 0
13	JS	Jump if Signed	SF = 1

Notes i) The mnemonics JE/JZ and JNE/JNZ are applicable for both signed and unsigned numbers, and the condition of the Zero flag being set or reset accordingly applies here as well.
ii) != is the symbol for 'not equal to'.

The list in Table 4.2 shows the conditions of the sign, overflow and zero flags and undoubtedly causes a certain amount of confusion. Let us test a program for the various situations possible and check the condition of these flags.

Example 4.22

```
.MODEL TINY
.CODE
.STARTUP
MOV AL, +40
CMP AL, -60      ;after this, OF = 0, SF = 0
CMP AL, +70      ;after this, OF = 0, SF = 1
MOV BL, -20      ;BL = -20
CMP BL, -70      ;after this, OF = 0, SF = 0
CMP BL, -10      ;after this, OF = 0, SF = 1
CMP BL, +10      ;after this, OF = 0, SF = 1
CMP BL, -20      ;after this, OF = 0, SF = 0, ZF = 1
.EXIT
END
```

The result of each comparison of the above program will help to clear the confusion.

1. When the destination is greater than the source, OF = SF.
2. When the destination is less than the source, OF != OF.
3. When the destination is equal to the source, ZF = 1.

Example 4.23 finds the smallest of three signed numbers and stores it in a memory location. It compares the first number with the second number. If the first number is less, it is compared with the third number. Otherwise, the second number is compared with the third number and finally the smallest number is identified and stored in memory. In this case, the negative number -90 is found to be the smallest number.

Example 4.23

```

.MODEL SMALL
.DATA
NUMS DB +45, -76, -90
SMALLEST DB 0
.CODE
.STARTUP
LEA BX, NUMS      ;use BX as pointer to the numbers
MOV AL, [BX]        ;get the first number in AL
INC BX             ;increment the pointer
CMP AL, [BX]        ;compare first no. with the second
JLE NEXT           ;if AL is less than or equal, jump
MOV AL, [BX]        ;else, load second no. in AL
NEXT: INC BX        ;increment the pointer
CMP AL, [BX]        ;compare second no. with third
JLE FINAL          ;if the first no. is less, jump
MOV AL, [BX]        ;move the smallest number to AL
FINAL: MOV SMALLEST, AL ;store the smallest no. in memory
.EXIT
END

```

4.7.2 | Signed Multiplication and Division

There are two instructions to be used for signed multiplication and division.

i) IMUL

All that was discussed about the MUL instruction (Section 3.4.4) holds true in the case of IMUL too. The only difference is that since the operands are signed, the product also is signed. Another point is that when the product is not large enough to use all the registers allocated for it, the sign bit is copied to the upper register (AH or DX as the case may be) and CF = OF = 0.

ii) IDIV

The signed division operation is similar to unsigned division, as discussed in Section 3.4.5, except that the operands and result are signed. Because of this, if the quotient register is AL, the value of the quotient is to be between -128 and +127. If the quotient register is to be AX, this value is to be between -32,768 and +32,767. If this is violated, a ‘divide overflow’ error message is displayed on the screen.

Now, let us use the signed multiply and divide instructions in a program. Let us compute $(XY-Z^2)/L$ and store the quotient alone. The

values of X, Y, Z and L are put in the data segment as shown in the program. For multiplication, a procedure named MULT is defined and used.

Example 4.24

```
.MODEL SMALL
.DATA
X DB -6
Y DB 25
Z DB 19

L DB -9
QUO DB?

.CODE
.STARTUP

MOV AL, X      ;copy X to AL
MOV BL, Y      ;copy Y to BL
CALL MULT      ;call the procedure to multiply X and Y
MOV CX, AX      ;copy the product to CX
MOV AL, Z      ;copy Z to AL
MOV BL, AL      ;copy Z to BL as well
CALL MULT      ;call the procedure to get Z2
SUB CX, AX      ;compute XY-Z2
XCHG CX, AX      ;exchange CX and AX, to get dividend in AX
IDIV L          ;signed division by L
MOV QUO, AL      ;copy the quotient to memory
.EXIT

MULT PROC NEAR ;define the MULT procedure
IMUL BL          ;multiply AL by BL, product in AX
RET              ;return to calling program
MULT ENDP        ;end the procedure
END
```

In this program, the operands are positive as well as negative numbers. Let us make a few observations.

1. With the values of X, Y, Z and L as given in the program, the product is -511 (FE01H) and the quotient is $+56$, which is found in AL as 38H.
2. If the divisor is now changed to $+9$, the quotient is -56 , which is found in AL as C8H.
3. If the divisor is changed to -3 , we get a message **divide overflow error** and the program execution is terminated. This is because the quotient ($+ 170$) is too large to fit in AL. Remember that the largest positive number that AL can hold is $+127$ only.

4.7.3 | Arithmetic Shift

A shift operation that makes a difference in the result depending on the sign of the operand is arithmetic shifting. There are two instructions pertaining to this.

i) SAR destination, count – Shift right arithmetic.

This is similar to the shift right logical (SHR) discussion in Section 3.5.2, but here, when shifting right, the sign bit is filled into the MSB. This instruction can be used to divide a signed number by 2 for one shifting each.

```
MOV BL, -24      ;AL = 1110 1000  
SAR BL,1        ; AL = 1111 0100, which is -12
```

ii) SAL destination, count – Shift left arithmetic.

This is the same as shift left logical (SHL). As shifting left does not involve any sign bit, logical and arithmetic shifting are the same. The two mnemonics (SAL and SHL) mean the same and can be used interchangeably.

4.8 | Programming Using High Level Language Constructs

MASM has become easier and very convenient to use ever since it was appended with a number of high level language constructs. Such constructs are available only for MASM 6.0 and higher versions. They are designated as ‘dot commands’ as they are preceded by a dot. These constructs make MASM a ‘high level assembler’. A few of these constructs are listed below:

1. IF, ELSEIF and ELSE ... ENDIF
2. REPEAT ... UNTIL
3. WHILE ... ENDW
4. BREAK
5. CONTINUE
6. REPEAT ... UNTILCXZ

Except the last one, these constructs function as they do in a high level language (for example, C and C++). For the last construct, the loop repeats until register CX = 0.

Now, we will endeavor to use these constructs in a few interesting programs. Example 4.25 is a simple program which takes in data through the keyboard (without echo) and verifies three conditions.

1. Is the value of the number between 0 and 5?
2. Is the value of the number between 6 and 9?
3. Is a key pressed ‘not a number’?

These three questions are resolved using IF, ELSEIF and ELSE statements and appropriate messages are displayed. The symbol ‘&&’ is used for ‘and’ just as in a high level language.

Example 4.25a

```
.MODEL SMALL
.DATA
FIRST DB "THE VALUE IS WITHIN 0 AND 5$"
SECOND DB "THE VALUE IS WITHIN 6 AND 9$"
OTHER DB "NOT A NUMBER$"

.CODE
.STARTUP

    MOV AH, 08          ;read in key without echo
    INT 21H

    .IF AL> = '0' && AL< = '5'      ;first IF condition
        LEA DX, FIRST           ;display the first message
        MOV AH, 09
        INT 21H
    .ELSEIF AL > = '6' && AL< = '9';second IF condition
        LEA DX, SECOND          ;display the second message
        MOV AH, 09
        INT 21H
    .ELSE                      ;ELSE
        LEA DX, OTHER           ;display the third message
        MOV AH, 09
        INT 21H
    .ENDIF
    .EXIT
END
```

Let us see what exactly these high level language constructs are. Obviously, they get converted to assembly language statements. We use the .LISTALL directive to find out. Example 4.25b shows the listing for the code segment alone up to the .ENDIF statement.

We find the equivalent assembly code for the high level language constructs, which in this case are compare and jump instructions.

Example 4.25b

```

0017 B4 08           MOV AH, 08
0019 CD 21           INT 21H
                    .IF AL> = '0' && AL< = '5'
001B 3C 30           * cmp al, '0'
001D 72 0E           * jb @C0001
001F 3C 35           * cmp al, '5'
0021 77 0A           * ja @C0001
0023 8D 16 0000 R    LEA DX, FIRST
0027 B4 09           MOV AH, 09
0029 CD 21           INT 21H
                    .ELSEIF AL > = '6' && AL< = '9'
002B EB 1A           * jmp @C0004
002D                 *@C0001:
002D 3C 36           * cmp al, '6'
002F 72 0E           * jb @C0005
0031 3C 39           * cmp al, '9'
0033 77 0A           * ja @C0005
0035 8D 16 001D R    LEA DX, SECOND
0039 B4 09           MOV AH, 09
003B CD 21           INT 21H
                    .ELSE
003D EB 08           * jmp @C0008
003F                 *@C0005:
003F 8D 16 003A R    LEA DX, OTHER
0043 B4 09           MOV AH, 09
0045 CD 21           INT 21H
.ENDIF

```

Next, we will do a program using the WHILE ... ENDW construct. Example 4.26 shows how the WHILE statement is used to repeat execution of a loop while a condition is satisfied and to exit the loop when the condition no longer holds.

As long as CX is not equal to zero, the loop repeats. The end of the loop is specified with a .ENDW construct. In this program, a string consisting of all the letters of the alphabet, stored in location starting from DAT, is to be reversed and displayed. After the reversed string is stored in location DAT1 onwards, it is appended with a '\$' so that it can be displayed with the INT 21H function number 9. The message REVERSED STRING: is displayed before the reversed string is displayed.

Example 4.26

```

.MODEL SMALL
.DATA

DAT DB 'ABCDEFGHIJKLM NOPQRSTUVWXYZ$'
MES1 DB 0DH,0AH,' REVERSED STRING: $'
DAT1 DB 27 DUP(?)

.CODE
.STARTUP

MOV BX,OFFSET DAT +25      ;address end of DAT
MOV SI,OFFSET DAT1         ;address DAT1
MOV CX,26                  ;load count in CX
.WHILE CX! = 0              ;loop while CX! = 0
    MOV AL,BYTE PTR[BX]      ;move byte to AL
    MOV BYTE PTR[SI],AL       ;save AL in DAT1
    INC SI
    DEC BX
    DEC CX
.ENDW                      ;end of while loop
    MOV BYTE PTR[SI + 1],'$'   ;append string with $
    MOV DX,OFFSET MES1
    MOV AH,09
    INT 21H
    MOV DX,OFFSET DAT1        ;display the reversed string
    MOV AH,09
    INT 21H

.EXIT
END

```

The output of the above program is as shown below.

REVERSED STRING: ZYXWVUTSRQPONMLKJIHG FEDCBA

C:\masm6.14\BIN>

For both the above problems, the high level language constructs are written indented to the left. This is done only to improve the readability of the program.

The above problem can be worked out using the REPEAT ... UNTIL construct. The loop is repeated until CX = 0. CX becoming zero is the stopping condition for the loop.

There is also a UNTILCXZ construct available that can be used in this example. The UNTILCXZ instruction uses the CX register as a counter to repeat a loop a fixed number of times. This example can be rewritten using a REPEAT UNTILCXZ loop also.

Now, let us see a case of nested loops. Example 4.27 shows the case of an IF loop nested within a repeat-until loop. In this, a character is to be searched within a string and the number of occurrences of that character in the string is to be found out. To make the program interactive for the user, messages asking for the character and the string are displayed using DOS INT 21H function 9. Both the character and the string are entered through the keyboard. The string is terminated when the ‘enter’ key is pressed. This corresponds to the ‘carriage return’ character oDH and this is the termination condition of the repeat until loop.

Example 4.27

```

.MODEL SMALL
.DATA
MES1 DB 0DH, 0AH, 'ENTER THE CHARACTER: $'
MES2 DB 0DH, 0AH, 'ENTER THE STRING: $'
MES3 DB 0DH, 0AH, 'NUMBER OF OCCURENCES: $'
STRING DB 50 DUP ('?')
.CODE
.STARTUP
MOV DX, OFFSET MES1      ;address MES1
MOV AH, 09                 ;display message
INT 21H                    ;for the character
MOV AH, 01
INT 21H                    ;enter character
MOV BL, AL                  ;copy character to BL
MOV DX, OFFSET MES2        ;address MES2
MOV AH, 09                 ;display message
INT 21H                    ;for entering string
MOV CL, 0                   ;initialize counter
.REPEAT
    MOV AH, 01               ;enter string, one character
    INT 21H                 ;at a time
    .IF (AL == BL)           ;compare with reference
        INC CL                ;if matching, increment CL
    .ENDIF
    .UNTIL AL == 0DH          ;exit when 'enter' is pressed
    ADD CL, 30H               ;convert to ASCII
    MOV DX, OFFSET MES3       ;address MES3 location
    MOV AH, 09                 ;display message
    INT 21H                    ;move count to DL
    MOV AH, 06                 ;display the count
    INT 21H
.EXIT
END

```

The output of the Example 4.27 for a string and the character ‘t’, entered through the keyboard, is shown below.

ENTER THE CHARACTER: t

ENTER THE STRING: tyu8iolhtttfeerttt

NUMBER OF OCCURENCES: 8

C:\masm6.14\BIN>

Now that we have seen the use of the high level language constructs, it is obvious that many of our previous programs can be re-written using these constructs.

KEY POINTS OF THIS CHAPTER

- String instructions are used to simplify coding when bulk data is moved, scanned and compared.
- Then, SI and DI registers are used as pointers to the data segment and extra segment, and the direction flag is used for auto increment/decrement.
- The conditional/unconditional REP prefix is used for repetitive string operations.
- Procedures are subprograms which are ‘called’ by the main program for performing specialized operations.
- Procedures may be intra-segment or inter-segment and this is clarified when defining a procedure by using the ‘near’ or ‘far’ prefix.
- Parameters to and from procedures may be passed via registers, memory or stack.
- Macros are functionally similar to procedures but act in a different way.
- Number format conversions are necessary as BCD, ASCII and binary numbers are used frequently.
- 8086 has specialized instructions for handling ASCII and BCD number operations.
- Signed number arithmetic involves the use of special instructions.
- The overflow flag is very important in signed operations.
- The high level language constructs of MASM have made assembly language programming easier.

QUESTIONS

1. What is the role of the direction flag in string instructions?
2. What is the difference in the functioning of the instructions STD and CLD?
3. How does the instruction CLD function in the case of the following two instructions?
 1. REPMOVSB
 2. REP MOVSW
4. Which is the pointer register to be used when using the extra segment in string operations?

5. Why does the LODS instruction not use the REP prefix?
 6. Do the following instructions function differently? In what way?
- o. REPNE CMPSB
1. REPE CMPSB
7. For REPNZ CMPSB, what are the conditions under which the loop is exited?
 8. Distinguish between a near and a far call.
 9. What is the difference in the functioning of the RET and RET n instructions?
 10. What is meant by the term ‘parameter passing’ in the context of procedures?
 11. What is the benefit of using macros?
 12. Compare procedures and macros.
 13. How are local variables in a macro taken care of?
 14. What is a dummy argument?
 15. Explain how the DAA instruction functions.
 16. Explain the action of the following instructions:
- o. AAM
1. AAS
 2. AAD
 3. AAM
17. Write down the steps required for the following conversions:
- o. binary to ASCII,
1. ASCII to binary.
 18. When is the CWD instruction used?
 19. Is it justified to call MASM a high level assembler?

EXERCISES

1. Write a program to move 100 words from ‘FROM’ to ‘TO’ which are two areas in the data segment separated by a 200-byte space.
2. Write a program to scan the name SAHABUDDIN and replace S by B and B by H.
3. Store a password in memory. Enter another password through the keyboard and verify if it matches the stored password. The password entered should not be displayed as such, but each letter should be displayed as ‘*’.
4. Write a string program for an application which requires the pointer registers to be auto decremented.
5. Write a program to search for a word in a block of N words.
6. Write a program which contains the following macros:
 1. for calculating the Fibonacci series for N,
 2. for entering the value of N,
 3. for displaying the numbers.
7. Do the above problem using procedures.

8. Display the factorial of three numbers. Solve the problem using macros as well as procedures.
9. Obtain the list file of the above program in both cases and observe the differences.
10. Enter a string of characters through the keyboard. Save it in memory. Verify if the string is a palindrome.
11. Using macros/procedures, write a program to add 2 BCD numbers entered through the keyboard and display their sum.
12. Write a program where nested macros are used.
13. There are 16 ASCII numbers stored in consecutive locations in memory.
Convert this to an eight-bit packed BCD number.
14. Write a program to add two ten-byte BCD numbers. The sum is to be displayed.
15. Two BCD numbers 4093H and 2986H are stored in memory. Subtract the second number from the first. Use the DAS instruction.
16. Write a program to add and display two ASCII numbers ‘5678’ and ‘3498’.
17. Divide ‘90’ by ‘9’. Use AAD and necessary programming steps.
18. Divide ‘3476’ by ‘5’. Display the quotient alone.
19. Use AAM to help in converting a 16-bit binary number in AX to a four-digit ASCII character. Hint Divide the content of AX-DX by 100, then use AAM.
20. Write a program which checks the number of ASCII digits present in the given number, and converts it to the corresponding binary equivalent.
21. Write a program to convert binary numbers to ASCII format for display.
The program should have the following features:
 - o. Display the number.
 1. Convert the ASCII number and store it in memory.
 2. A procedure, DISPLAY should access the memory to display the numbers.
22. There are 10 signed bytes stored in memory, some of which are negative and some of which are positive. Find the largest number of the lot.
23. Re-write Example 4.26 using REPEAT ... UNTIL and REPEAT ... UNTILCXZ constructs.
24. Re-write Example 4.27 using .CONTINUE and .BREAK constructs along with .IF and .REPEAT constructs.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

8

The Interrupt Structure of 8086

IN THIS CHAPTER, YOU WILL LEARN

- The concept of an interrupt.
- The interrupt response of the 8086.
- The way the interrupt vector table is organized for 8086.
- The idea of software interrupts.
- The operation of hardware interrupts of 8086.
- The significance of DOS and BIOS interrupts.
- How to access the video display by the use of BIOS interrupts.
- How to access the video display by direct writing/reading to the video RAM.
- How the PC's keyboard and its controller are organized.
- How to use BIOS interrupts for keyboard access.
- The method of 'hooking' interrupts.
- How to write TSR programs.

Introduction

The word ‘interrupt’ brings to our mind many situations. It could be that we are at home doing something rather important and interesting and suddenly the doorbell rings. There is no option but to go to the door and attend to the person who has interrupted us – it may be that the visitor needs only a few minutes of our time, but it may also happen that the visitor needs us to go out and do some errand for him. In either case, the task we were at is suspended temporarily. We spend some time attending to the task requested by the visitor, after which we get back to the suspended task. This is a real life situation that happens quite frequently, and being social beings, none of us ever get to do any of our tasks uninterruptedly.

This is exactly the case with the processor too. Since the processor is connected to various peripherals and also because it has various programs stored in its memory, there is very little chance of it being able to perform a single task without interruption. Besides, performing a task only when ‘requested’ is the most efficient way of using the capabilities of the processor. Let us look at it this way. The keyboard is a peripheral connected to the processor – this means that the processor expects a key

to be pressed at any time. One option is for the processor to do ‘polling’; i.e., the processor continually keeps checking if a key has been pressed. When a key is pressed, the processor goes on to the task of identifying the key. However, if the processor is ‘polling’ for a key press, it is obviously waiting and doing nothing else. This is just a waste of processor time. A better way of organizing the setup would be to let the processor do some task and ‘interrupt’ it only when a key is pressed. When a ‘key press’ interrupt is received, the processor performs the ‘interrupt service routine’ (ISR), which is the process of identifying the key that was pressed. This, in essence, is the philosophy of interrupts and it helps utilize the processor with maximum efficiency.

8.1 | Interrupts of 8086

Now that we have accepted that interrupts are a part of the normal working of a processor, let us see the interrupt processing mechanism of 8086. 8086 has hardware interrupts, software interrupts and error generated interrupts. For the three cases mentioned here, the interrupting mechanism is different, but the way the processor responds is similar. So, let us first have a look at the way 8086 responds to any interrupt. The point to note is that after an interrupt request is processed, the processor has to come back to its previous task, which was left unfinished. Thus, an interrupt is not very different from a CALL instruction in its philosophy, but there are some differences in the way it is handled and processed.

8.1.1 | Interrupt Response of 8086

After every instruction cycle, the processor checks if any interrupt is awaiting service. If it finds an interrupt request and decides to acknowledge and service it, the response is the following sequence of steps:

1. The flag register is pushed on to the stack.
2. The interrupt flag is disabled ($IF = 0$).
3. The trap flag is disabled ($TF = 0$).
4. The CS register is pushed on to the stack.
5. The IP register is pushed on to the stack.
6. Control is transferred to the location in which the corresponding ‘Interrupt Service Routine’ (ISR) is stored. This, in effect, would be a far jump.
7. The program corresponding to the ISR is executed. The last instruction in the ISR will be IRET.
8. Then, IP is popped off the stack.
9. CS is popped off the stack.

10. The Flag register is popped off the stack.
11. Control returns to the point at which it had left off.

Steps i to vi constitute the actions before getting to the **interrupt service routine**, and steps viii to xi occur on returning from the interrupt.

Let us go into the details of the steps in the above list. See Fig. 8.1.

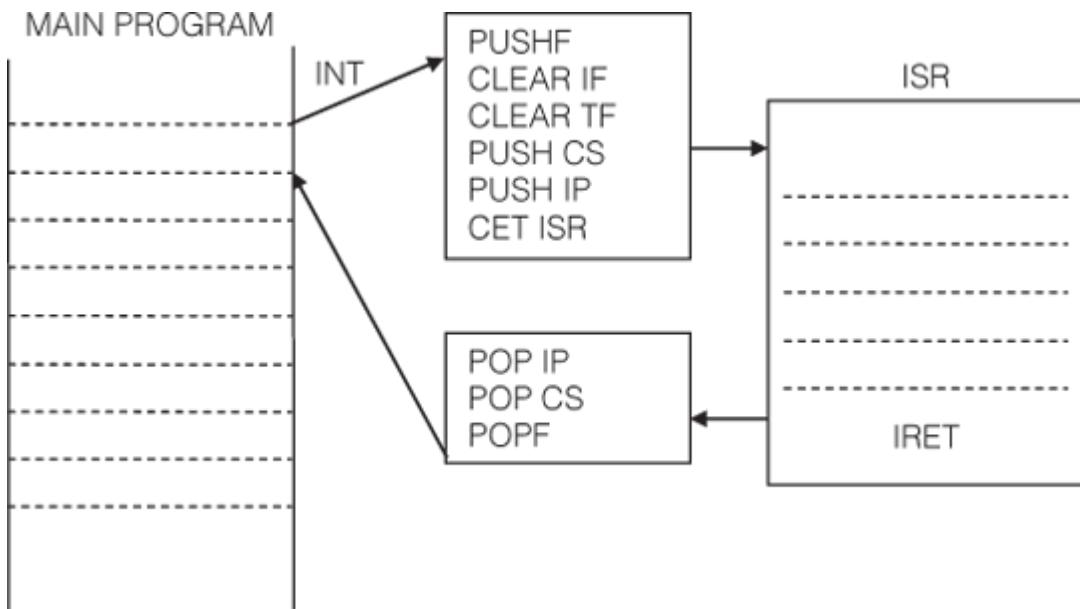


Figure 8.1 | Steps in processing an interrupt request

1. The first step is part of the action of saving the ‘context’ of the current program. First, the flag status is saved in stack for later retrieval. This is because the ISR may change the status of flags.
2. The second step is for ensuring that this interrupt routine does not get interrupted by other interrupts received on the INTR line. (We will see later that for interrupts on the INTR line to get acknowledged, it is mandatory for the interrupt flag to be set.)
3. The third step is for ensuring that the program taken up does not stop after the execution of each instruction (this happens if the Trap flag is set).
4. The fourth and fifth steps are for saving on stack the address of the next instruction in the current sequence, just as is done in the case of a far CALL. They will be taken back from the stack after the interrupt processing is over. Both IP as well as CS are saved, because the new program taken up will in a different code segment.
5. The sixth step is that of control being transferred to a location in a different code segment. That is why it is designated as a ‘far jump’.
6. The interrupt service routine, which is the program at which the jump has occurred, will now be taken up for execution. The last instruction of an ISR is an IRET instruction, which will cause the reversal of the previous actions.

7. In the eighth and ninth steps, the IP first and then CS are popped from stack. Thus, the address of the next instruction in the normal sequence is back in the required registers.
8. In the tenth step, the flag register is popped off, and thus the original status of all flags is restored.
9. Finally, instruction execution resumes from where it had been diverted.

8.1.2 | Interrupt Service Routine and Interrupt Vector

Now, what is an interrupt service routine? When an interrupt occurs, the processor suspends the execution of its current task and takes on another task as required by the interrupting source. This program, or routine as it may be called, is designated as an ‘interrupt service routine’. This routine corresponds to the request of a particular source of interrupt and it is also called its ‘interrupt handler’. This means that for any interrupt that occurs, there is a particular interrupt service routine (ISR).

Now, where is this ISR available? It is to be available in memory and must be accessed on the occurrence of the specific interrupt. For that, the address of the ISR must be obtained. The address of an ISR is called its **‘interrupt vector’**. For an 8086 based system, any address of code is in the form CS:IP. Thus, the interrupt vector for any interrupt has 4 bytes – two for the CS value and two for the IP value. Thus, if the interrupt vector for a particular interrupt is obtained, control can be transferred to the new location by using the new values of CS and IP specified as the ‘interrupt vector’.

8.1.3 | Interrupt Vector Table

It is obvious that the number of interrupt vectors in a system is the same as the number of interrupts that the system can process. The 8086 has 256 interrupt vectors, and since each vector is specified by 4 bytes, it implies that $256 \times 4 = 1024$ bytes (1K) of memory are allocated to store the interrupt vectors. These 256 vectors are stored in a table called the ‘Interrupt Vector Table’ (IVT) in system RAM from locations 00000 to 003FFH, i.e., up to 0000: 03FF. See Fig. 8.2a. Each interrupt vector is numbered from 0 to 255, and as such, each interrupt is also numbered in the same way. These numbers thus turn out to be what can be called as the ‘type numbers’ of the interrupts. Thus, there are interrupts designated as Int 0, Int 1, Int 2.....Int 255. As is obvious from Fig. 8.2a, these numbers show the position of the corresponding interrupt vector in the interrupt vector table. Thus, the vector of INT 0 is the 1st entry in the IVT, that of INT 1 is the second entry ... that of INT 6 is the 7th entry

and so on. Since each interrupt vector is 4 bytes long, the memory location corresponding to the vector of INT n is obtained as $n \times 4$; i.e., for INT 1, the address of the first byte of its vector is $1 \times 4 = 0004$, for INT 0 it is 0000 and so on, as seen in Fig. 8.2a. In fact, this is how the 8086 reacts to an interrupt with a type number. **It multiplies the type number by 4, gets the address of its interrupt vector and loads IP and CS with these new values and starts executing the ISR it has located.**

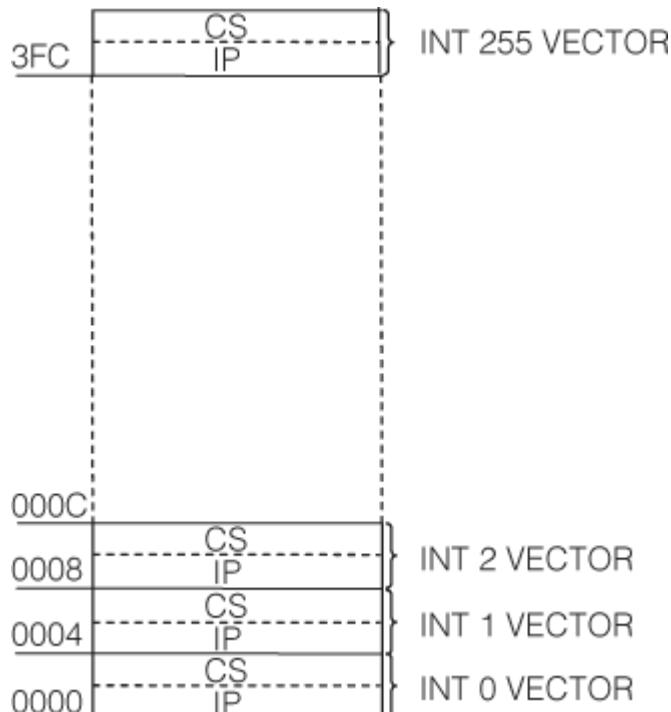


Figure 8.2a | Interrupt vector table of 8086

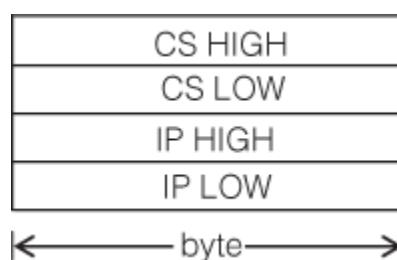


Figure 8.2b | A typical interrupt vector (4 bytes)

Example 8.1

Find the address (in the IVT) of the interrupt vector of INT 61H. Find the physical address of the ISR corresponding to this interrupt if the vector is 0Foo:9872.

Solution

The type number of the interrupt is 61H = 97 in decimal.

The address of the interrupt vector is $97 \times 4 = 388 = 184H$.

Thus, the interrupt vector is to be stored in the IVT in location $0000:0184$ onwards.

For the ISR, the CS value is $0F00H$ and the IP value is $9872H$. The part of the interrupt vector table which has these vectors stored is shown.

	OFH
0186H	00
	98H
0184H	72H
Address	

Now, How Does the Interrupt Vector Table Get These Entries?

It was mentioned in Chapter 6 that after ‘power on’, the processor wakes up at the address $FFFF0H$. This is the entry point to BIOS in ROM (Ref Table 7.2). BIOS contains a set of routines that provide device support. These routines check, verify and initialize various devices and also establish two data areas, one of which is the interrupt vector table from locations 00000 to $003FFH$.

How is an interrupt different from a ‘call’?

An interrupt seems to perform actions similar to a far CALL instruction. It differs in a few points, however. An interrupt causes the flag register to be pushed on the stack in addition to CS and IP values. It clears the trap and interrupt flags. The action of clearing the flags is not associated with a CALL but is part of the automatic response due to an interrupt. Also, note that the ISR ends with IRET rather than a RET instruction. The IRET instruction ensures that the flag register is also popped off the stack rather than just the CS and IP.

8.2 | Dedicated Interrupt Types

Intel has dedicated certain interrupt types for specific applications directly related to CPU operations. These are listed below.

8.2.1 | INT o (Divide by Zero Error)

The interrupt with type number 0 is dedicated to the ‘divide by zero’ error. This interrupt is an ‘error generated’ interrupt (also called an ‘exception’). On division, if the quotient register is not large enough to contain the quotient, this interrupt is generated automatically.

Dedicating Type 0 for this case means that the corresponding interrupt vector in the interrupt vector table is available at 0000:0000 (see Fig. 8.2a). Thus, the ISR for this error generated interrupt is written in the address specified in the IVT. What could be a possible ISR for this condition? One possibility is just to display a message indicating ‘divide overflow’ and expect the programmer to correct his data/program.

Another possibility is to write as the ISR a program which increases the size of the quotient register, so that the problem is corrected without user intervention. It is up to the system designer to decide how the error handler is to be written.

8.2.2 | Int 1 (Single Stepping)

This type number is dedicated for ‘single stepping’ or ‘trace’. Single stepping is an important idea in debugging. During logical debugging of our programs, we would like to stop after the execution of each instruction and check the contents of registers, memory and so on. We usually perform the action of ‘trace’ this way. Intel has provided the ‘Trap’ flag for this, and this flag has to be set to let this happen. The ISR for viewing the register and memory contents will be pointed by the vector of interrupt type 1. However, an important issue in this is how to set the trap flag. No such instruction has been encountered so far. Recollect the flag register configuration.

D ₁₅	D ₁₄	D ₁₃	D ₁₂	D ₁₁	D ₁₀	D ₉	D ₈	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

The Trap flag is the D₈ bit of the flag register.

PUSHF	:push the flag register to the stack
POP AX	:pop it to AX
OR AX, 000000010000000B	:OR it to set bit D ₈ i.e., TF
PUSH AX	:push AX to stack
POPF	:pop it to the flag register

This program segment sets the trap flag which causes INT 1 to occur.

Note The Trap flag will be reset as part of the interrupt response, so the ISR can execute without stopping after each program line. However, on going back to the mainline program, when the flags are popped back, the

set condition of the Trap flag will be retrieved, and therefore, single stepping can be continued for the mainline program until the Trap flag is reset again.

8.2.3 | Int 2 (Non Maskable Interrupt)

This interrupt corresponds to the vector (pointer) of the hardware interrupt NMI. When an interrupt is received on the pin NMI (Non Maskable Interrupt) of the processor, a type 2 interrupt occurs – this means that the ISR for NMI must be written in the address pointed by the corresponding IVT content. We will deal with the NMI interrupt in detail in Section 8.4.1.

8.2.4 | Int 3 (Breakpoint Interrupt)

This is the breakpoint interrupt, which is useful for de-bugging. We will need to set breakpoints (stop after executing a group of instructions) and check the content of registers and memory after executing instructions up to the breakpoint. Since setting breakpoints is an important aspect of debugging, the breakpoint interrupt is special in that it is a single byte instruction with the code CCH. When we set breakpoints in debugging, what is really happening is that this code (CCH) is inserted and getting executed at the position in the program where the breakpoint is set. Just like in single stepping, the ISR will be written to dump the register and memory contents on the video display.

8.2.5 | Int 4 (Overflow Interrupt)

This interrupt corresponds to the overflow flag. If the overflow flag is set, this interrupt occurs but not automatically. An instruction INTO (interrupt on overflow) must be written after the program segment which is likely to cause the overflow flag (OF) to be set.

```
MOV AL, NUM1  
ADD AL, NUM2  
INTO           ;interrupt on overflow
```

The last line of this program segment can pass control to the ISR written for INT 3, if the overflow flag is set by the result of the addition. Otherwise, INTO acts as a NOP instruction.

8.2.6 | Allocation of Interrupt Type Numbers

Thus, we see that interrupts with type numbers 0 to 4 have been allotted pre-defined actions related to the CPU. See Table 8.1. Interrupts with

type numbers 5 to 31 are reserved by Intel for special uses and for future processors. As a matter of fact, a few of them are used for various hardware and I/O devices and some for error conditions of the higher processors in the x86 family. The rest of the interrupts can be used by the user. However, in the PC, many interrupt types have been used by BIOS and DOS. We will see the list of those interrupts and also use some of them later.

Table 8.1 | Interrupt Vector Table Allocation

INT type no.	Location in IVT (HEX)	Application
0	0000:0000	Divide by zero error
1	0000:0004	Single step interrupt
2	0000:0008	Non maskable interrupt
3	0000:000C	Breakpoint
4	0000:0010	Overflow
5 to 31		Reserved by Intel
32 to 255		Available to user

8.3 | Software Interrupts

What is meant by the term ‘software interrupt’? When an interrupt is initiated by an instruction, it is called a software interrupt. The format of this instruction is:

INT type number

The type numbers can vary from 0 to 255. This is a very important and interesting way of using procedures whose vectors are inserted into the IVT. Thus, the user can write any procedure as an ISR, store it in some address in memory and call it by using the instruction INT n. Before using it this way, the user must however insert the CS and IP values of his ISR in the IVT in the location corresponding to INT n.

Hardware interrupt service routines can be tested using software interrupts. Take the case of NMI, which is a hardware interrupt. This is vectored to location 0000:0008; i.e., it corresponds to type number 2. The routine corresponding to this hardware interrupt can be tested by using the instruction INT 2. In effect, a hardware interrupt on the NMI pin and the software instruction INT 2 takes us to the same absolute address. Similarly, the effect of a ‘divide by zero’ error can be simulated by writing the instruction INT 0, which takes control to the ISR which has been written for the ‘divide by zero’ error.

What Is the Size of an Interrupt Instruction?

The format of an interrupt instruction is INT n where n can only be a byte. The opcode for INT is ‘CD’, again a byte – for example INT 5 has the opcode CD 05. Thus, an interrupt instruction is 2 bytes long, except the breakpoint interrupt INT 3, which has the code of just CC. This feature is given to this interrupt because setting of breakpoints is very frequently done in debugging and having one byte only means a lot of memory space saving.

8.3.1 | DOS and BIOS Interrupt Routines

We have used software interrupts in the form of the DOS function INT 21H. DOS has a number of functions (procedures) to access input/output devices, and we can use these functions by knowing the interrupt type number. The absolute address of the function need not be known to us.

Besides DOS interrupts, there is another set of functions for I/O access and this is supplied by the system BIOS. BIOS is an acronym for ‘Basic Input output System’. A collection of routines, i.e., procedures, are available here too to access I/O. These functions are accessed by - interrupts with type numbers specified for a particular peripheral (Refer Table 8.2). Thus, DOS and BIOS interrupts belong to the class of software interrupts and will be discussed in greater detail soon.

Table 8.2 | Interrupt Allocation for Type Numbers 0 to 21 in the IBM PC

Interrupt type no. (HEX)	Function
0	CPU – Divide by zero (exception interrupt)
1	CPU – Single step
2	CPU – Non maskable interrupt (NMI)
3	CPU – Break point instruction
4	CPU – Overflow
5	BIOS – Print screen
6	CPU – Invalid op-code
7	CPU – Math coprocessor not present
8	System timer interrupt (IRQ 0)
9	Keyboard data ready (IRQ 1)
A	Reserved (IRQ 2) cascade from slave interrupt controller
B, C	Hardware interrupt for serial communication
D	Parallel port hardware interrupt (IRQ 5, LPT 2)
E	Diskette controller hardware interrupt (IRQ 6)
F	Printer hardware interrupt (IRQ 7, LPT 1)
10	BIOS – Software interrupt to use video
11	BIOS – Equipment check call
12	BIOS – Memory check call
13	BIOS – Software interrupt to use hard drive
14	BIOS – Software interrupt to use serial port
15	Not used
16	BIOS – Keyboard software interrupt
17	BIOS – Printer software interrupt
18	BIOS – ROM BASIC loader
19	BIOS – Bootstrap loader
1A	BIOS – System and real time clock software interrupt
1B	BIOS – Control-break handler
1C	BIOS – System timer tick handler
1D	BIOS – Video initialization parameter table pointer
1E	BIOS – Diskette initialization parameter table pointer
1F	BIOS – Graphics display character bitmap table
20	DOS – Program terminate
21	DOS – Function request services

8.4 | Hardware Interrupts

Now, we will examine the hardware interrupt pins of 8086. There are two pins on which interrupt requests can be received – they are the INTR pin and the NMI pin.

8.4.1 | NMI

This is a positive edge triggered interrupt, but it is also required to have a duration in the high state of more than two clock cycles. Any high going transition of NMI is latched on-chip and will be serviced at the end of the current instruction. This is a non maskable interrupt, in that it does not depend on the setting of the interrupt flag (IF). It cannot be masked or prevented from being activated. It is a Type 2 interrupt, meaning that its

vector is obtained from the corresponding location in the interrupt vector table. NMI caters to applications of the highest priority, like power failure.

8.4.2 | INTR

This is the non-vectored interrupt pin of the 8086, which means that when an interrupt request is received on this pin, it does not get automatically directed towards any particular entry in the interrupt vector table. Another feature of this interrupt is that the interrupt flag (IF) is required to be set for an interrupt request on the INTR line to be honored. When the processor is reset, all flags are found to be cleared and so is the IF. It is to be ensured that the IF is set by the instruction STI if interrupts on the INTR lines are to be acknowledged.

INTR is a high level triggered interrupt. To be responded to, the line should be high during the clock period preceding the end of the current instruction. During any interrupt response, further interrupts on the INTR line are blocked, because clearing the IF is part of the processor's interrupt response. Now, let us see the sequence of actions following an interrupt request on this line. On being interrupted, the processor goes into an 'interrupt acknowledge' state which causes two 'interrupt acknowledge machine' cycles to occur. During the first machine cycle, the address/data buses are floated and the *INTA* (interrupt acknowledge) pin goes low in T_2 and remains low till T_4 . This is an indication for the interrupting device that its interrupt request has been accepted by the processor. In the second machine cycle, once again the line *INTA* is low from T_2 to T_4 , and then the interrupting device places the 'type number' of the interrupt on the lower data lines (D_0 to D_7), which comes out of the float condition. The processor accepts this number, multiplies it by 4, and accesses the interrupt vector table at that location.

In Fig. 8.3, the ALE signal is also shown. Actually, this signal has no role in the INTA cycle but is included just to show that in the first T state of any machine cycle, the ALE signal is activated.

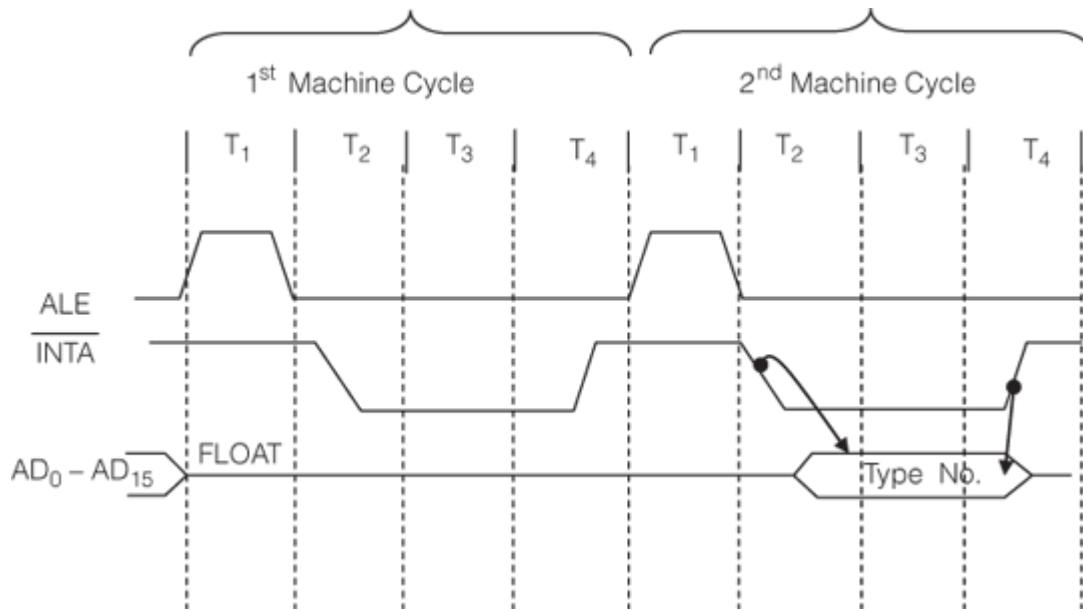


Figure 8.3 | Interrupt acknowledge machine cycle of the 8086

8.4.2.1 | Generation of a Type Number

Figure 8.4 shows a simple circuit which is capable of sending a ‘type number’ to the processor. Consider an interrupting device interrupting the 8086 on its INTR pin. The processor responds by lowering the INTA line. This signal is connected to the enable pins of a tri-state buffer (74LS244). Eight switches are connected to the input pins of the TSB. When a switch is closed, the line is grounded corresponding to a ‘0’ level on the pin. When a switch is open, it gets Vcc on it and hence is in the ‘1’ state. In the figure, the switch configuration corresponds to 1000 0001 or 81H. This is because the first and last switches are open, and the rest are closed. When the enable pins (OE) of the TSB are low (i.e., when INTA becomes low), the data at the input of the TSB is transferred to the output of the TSB chip.

During the first Interrupt acknowledge machine cycle, the address/data lines of the processor are floated. This ‘interrupt acknowledge’ machine cycle signals the interrupting device that its interrupt request has been acknowledged, and that it is to get ready. In this cycle, even though the type number appears on the output of the TSB, the bus of the 8086 is in the high impedance state and so nothing happens. However, during the second machine cycle, when the INTA line is lowered, the lines D₀ to D₇ of the processor are enabled, and the switch settings of the TSB are transferred to the data lines D₀ to D₇ of the 8086 constituting the type number. Thus, it is understood that this INTR interrupt is now vectored to type 81H (for fig. 8.4).

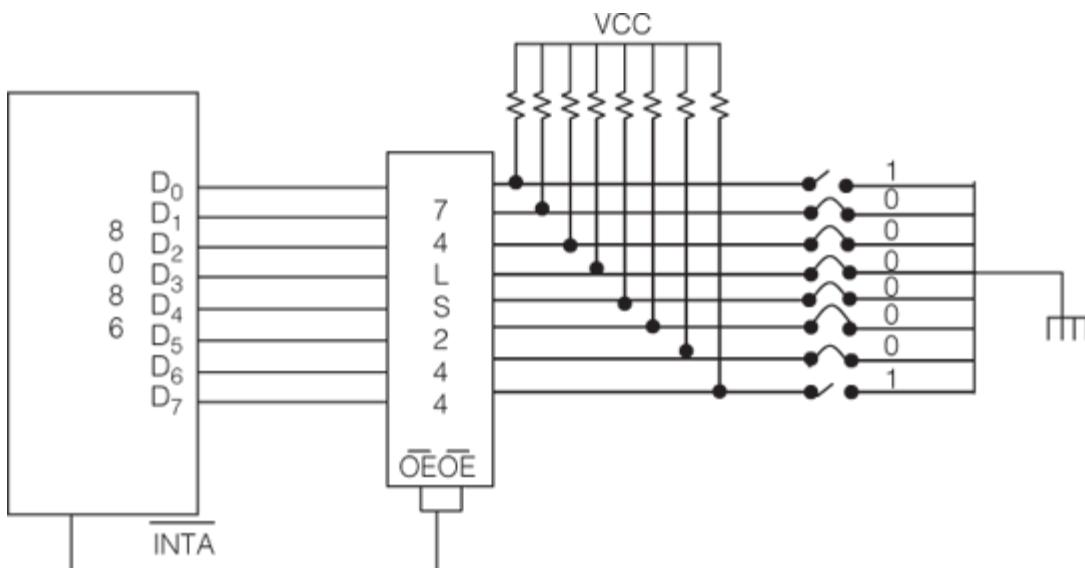


Figure 8.4 | Generation of a type number during the INTA cycle

Thus, INTR is an interrupt pin using which an interrupt can be funneled to any location in the IVT and thus facilitate flexibility for use in any application. It is also possible that more than one device can place interrupt requests on this line, and each of them may have to be associated with different type numbers. All these necessitate the need of extra hardware to manage the possible interrupts arriving on this pin from various peripherals. This is done by a peripheral chip named Programmable Interrupt Controller (PIC) 8259, the details of which will be presented in Chapter 10. The INTR pin along with the PIC is used in PCs for managing various hardware devices.

Can We Consider the Reset Pin as an Interrupt Pin?

When the reset pin is activated, the processor resets and control branches to the absolute address FFFF0H. Thus, reset has a particular pointer or vector – which is just the way an interrupt functions. Thus, RESET is sometimes included in the list of hardware interrupts along with NMI and INTR.

For a PC, the process of resetting also occurs when the system is powered on. Let us note some points regarding ‘power on’. In fact, the address FFFF0H is designated as the ‘power on reset vector’. This address (FFFF0H) is in ROM and contains a far jump instruction to the beginning of the BIOS chip ‘power on reset code’. To find this vector, go to command prompt C drive, type ‘debug’ and get the contents of the reset location. This is shown below.

C:\>debug

-ffff:0ooo

FFFF:0ooo EA 5B Eo oo Fo 30 36 2F-32 30 2F 30 35 oo FC oo.
[...06/20/05...]

The first 5 bytes, constitute the assembly code for the jump instruction, i.e., JMP (EA) Fooo:E05B. The next eight bytes were originally called the 'RELEASE MARKER' by IBM, as they always contain the release date of the code, i.e., BIOS (which is 20th June 2005 for this PC). See the ASCII code of this marker:

30 36 2F 32 30 2F 30 35, i.e., 06/20/05 in the format mm/dd/yy.

8.5 | Priority of Interrupts

When many interrupts occur at the same time, which source gets its request honored first? the processor decides the priority. The order of priority is set in the following manner:

1. Internal interrupts and Software interrupts get the highest priority.
2. NMI.
3. INTR gets the lowest priority.

Consider a case when the INTO, NMI and INTR occur simultaneously. Because of the inbuilt priority mechanism, the internal interrupt (INTO) will be taken up for servicing first. One step in the interrupt response would be to disable IF. Hence, INTR will not be taken up now. However, the NMI can interrupt the current ISR, as its functioning does not depend on the setting of the IF. Thus, the NMI will interrupt the INTO ISR. The end result is that effectively NMI gets serviced before the internal interrupt. This is fine, because NMI caters to highest priority. After the NMI ISR is processed, the INTO ISR is taken up, and after it is done with, the request on the INTR is dealt with.

8.6 | Interrupt type allocation for Current pcs

The allocation of type numbers for the PC is shown in Table 8.2. This will be our reference when discussing video and keyboard interrupts.

8.6.1 | BIOS and DOS Interrupts

We have talked about these interrupts before and know that they correspond to routines already written by system designers. Both these classes of interrupts cater to dealing with input and output devices. DOS

interrupts come along with the operating system (MS-DOS). BIOS routines are stored in ROM and thus are also called ROM BIOS. Since ROM access is rather slow, the current practice is of having the BIOS copied into RAM (which is faster) as well. This is called shadow memory, which is write protected to prevent tampering of the BIOS routines. In the beginning of the PC development era, there were problems due to non-standard BIOS but now standardization has been more or less agreed upon. Table 8.2 shows the type numbers of interrupts catering to specific devices. Even for a particular interrupt type, the routines that have been written have various functions and we can use them appropriately by passing parameters to designated registers. In comparing BIOS and DOS routines, it will be noticed that BIOS gives a greater degree of control than DOS routines and are thus closer to the actual hardware. DOS routines are more ‘high level’ in comparison, and they use BIOS routines for peripheral access. Now, we will use a few BIOS interrupts, the first of which is INT 10H, which is for video applications, mostly pertaining to the video display.

8.6.2 | BIOS 10H Functions

To be able to use these functions effectively, we first have to make a study of the video display. We need to have an overall idea (not all the intricate details) of its history such as components and standards. So, let us now embark on the study of the video system of a typical PC.

8.6.3 | Video Adapter

The video display along with the associated hardware and software is one of the most complex parts of the PC. To be able to get a certain amount of accessibility to it, we need to develop a basic understanding of the video system of a PC. In any PC, there is the display unit and a ‘video adapter’. This adapter is the hardware that provides the system its display capabilities. It is also designated by names such as video card and graphics card. It is the hardware that works between the processor and monitor. It relays the information received from programs and applications running on the system to the monitor and provides visual representation of the results.

Previously, the video adapter was plugged into a slot in the motherboard – now it is in the motherboard. Also, there is usually an additional video card with an inbuilt processor and RAM for improving the graphic capability of the PC. This is called a graphic accelerator.

Now, let us see the basic concepts of video adapters and displays. The video display usually has a ‘controller’ which takes care of the complexities of the display mechanism. Besides that, it needs RAM to store the data to be displayed. The IBM PC memory map (Table 7.3.) shows that locations from A0000H to BFFFFH (128K) is dedicated to video, and this area of memory is called video display RAM (VDR) or video memory or VRAM.

8.6.4 | History

In 1981, IBM released the MDA (monochrome display adapter), which could only give a black and white display. Subsequently, CGA (color graphics adapter) appeared, which could process color too.

Improvements over the CGA for color display are the EGA, VGA and XGA. These higher level display systems can also ‘emulate’ the lower ones. Thus, if we have VGA in our PC, it can also be made to work in a mode corresponding to MDA or CGA. For color displays, CGA is the lowest common denominator, and here, we will use the mode corresponding to this standard even if the display we have on our PC is VGA or XGA. The mode setting can be done using BIOS interrupts with specified function numbers. See what each of these display designations denote.

CGA: Color graphics adapter

EGA: Extended graphics adapter

VGA: Video graphics adapter

SVGA: Super VGA

8.6.5 | Text and Graphics

The most basic classification of the display mode is ‘text or graphics’. In text mode, a monitor can display only ASCII characters. In graphics mode, a monitor can display any bit-mapped image. The emphasis of our discussions here will be the text mode. In addition to the text and graphics modes, video adapters offer different modes of screen resolution and varying number of colors used.

CGA: This being the basic mode of color displays, we will concentrate on the CGA. Table 8.3 shows the text mode details for CGA. Only the text modes have been shown, and it corresponds to the CGA standard. VRAM shown in Table 7.2 specifies a memory space in RAM of 128 KB from

location A0000H to BFFFFH. A part of this is used to store the data to be displayed. All the color modes in CGA have the video memory to start from B8000H. See the third row, which corresponds to mode 2. It shows a screen to be composed of 25 rows and 80 columns, which means that 2000 elements can be displayed on the screen. Each element of text display is to have a data of 2 bytes – one byte for the ASCII character and one byte for the color attribute. This will take 4 KB for one page of display. 16K memory is totally allotted for this, and thus 4 pages can be in use at any time. Each display begins on a 4K boundary, at address B800:0000, B800:1000, B800:2000 and so on. Only one page can be displayed at a time, which is called the active page, but it is possible to switch to other pages reasonably fast. Note that when the screen resolution is less, more pages are possible (25×40 of modes 0 and 1 have 8 pages). For monochrome display, the video memory allocation starts from B000:0000. In our discussions ahead, we will use the color screen with size 25×80 .

Table 8.3 | Text Mode Details of CGA

Mode	Type	Rows × Columns	Pages	Starting address in VDR	No. of colors
00	Color	25×40	8	B800:0000	16
01	Color	25×40	8	B800:0000	16
02	Color	25×80	4	B800:0000	16
03	Color	25×80	4	B800:0000	16
07	Monochrome	25×80	1	B000:0000	2

8.6.6 | Character Display

In video memory, for any character, the ASCII value is in the even address and the attribute byte is in the odd address. The attribute byte decides the color of the background and text. The order of storing the data is that in the first word location starting from B800:0000, all the 80 character words of the first row are stored, then the data of the second row, and so on; in other words, data is stored in rows. For 25 rows and 80 columns, the co-ordinates of the first and last character elements are (0, 0) and (24, 79). This is shown in Fig. 8.5.

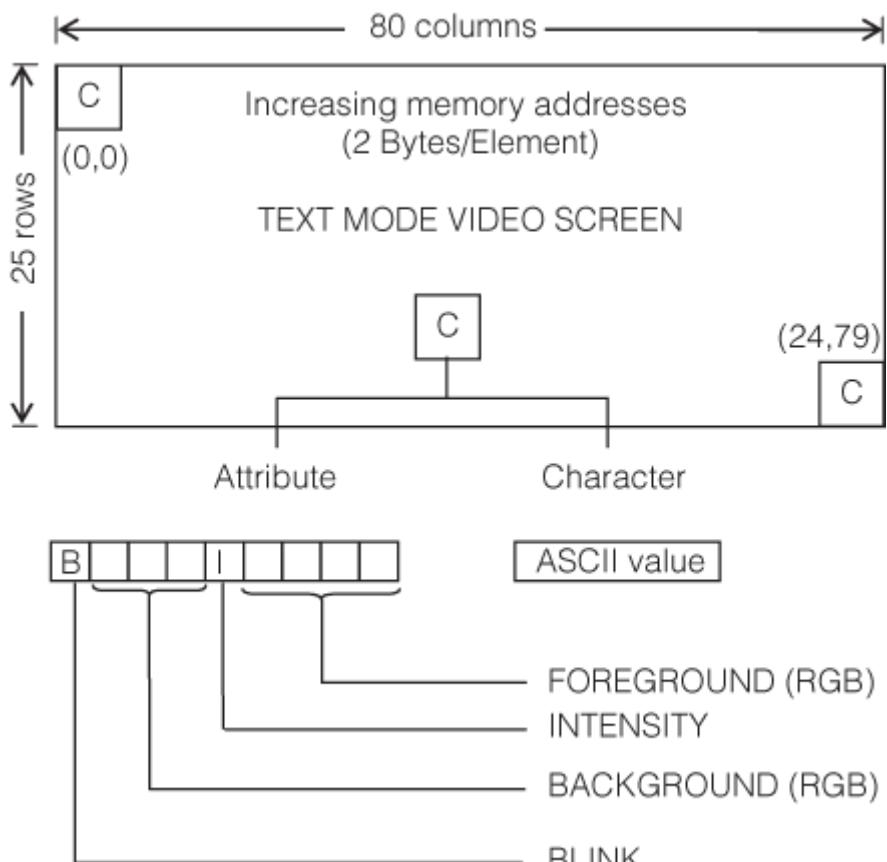


Figure 8.5 | Text mode video screen

8.6.7 | Color Configuration

The bit configurations for the various possible colors are given in Tables 8.5 and 8.6. All this is for the attribute byte which has the format shown in Table 8.4. This byte defines the background and foreground colors. Foreground refers to the color of the character displayed. Background is the color that surrounds a character. It is well known that adding the primary colors red, green and blue (RGB) in suitable proportions can give a resultant color. By adding intensity (brightness) to this color, we get a lighter color.

Table 8.4 | Format of the Attribute Byte

D ₇ B	D ₆ R	D ₅ G	D ₄ B	D ₃ I	D ₂ R	D ₁ G	D ₀ B
BLINK = 1 NORMAL = 0		BACKGROUND		FOREGROUND			

Table 8.5 | The 16 Colors of the Foreground–Bit Assignment

I	R	G	B	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	White
1	0	0	0	Gray
1	0	0	1	Light blue
1	0	1	0	Light green
1	0	1	1	Light cyan
1	1	0	0	Light red
1	1	0	1	Light magenta
1	1	1	0	Yellow
1	1	1	1	Bright white

Table 8.6 | The 8 Colors of the Background-Bit Assignment

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Brown
1	1	1	White

Now, refer to Table 8.5. This shows the 16 possible colors that can be obtained for the foreground. Adding R, G and B gives only 8 colors, but when intensity is another parameter, 16 colors are obtained. With intensity added ($I = 1$) and when intensity is not added ($I = 0$), two different shades of the same color are obtained. For example, in the fifth row of the table, the color specified is red with the I bit equal to 0. For the same value of RGB, when $I = 1$, the color becomes light red. For the background, only 8 colors are possible and this is given in Table 8.6.

Referring to Table 8.4, in the attribute byte, ‘I’ and ‘B’ apply to the foreground (text) only. ‘I’ means intensity and ‘B’ means blinking. What is blinking? Blinking happens when the background attribute is substituted for the foreground every two seconds. Thus, in the attribute

byte, if MSB = 1, it corresponds to ‘blinking’. Also, if the background and foreground colors are the same, the displayed character is invisible.

See the bit assignment of the attribute byte in Table 8.4.

Example 8.2

Find the attribute byte for the following cases of text mode display:

1. Black background with bright white text (foreground)
2. White background, red text (foreground)
3. Blinking with cyan background with brown text (foreground)

Solution

Look at Tables 8.4, 8.5 and 8.6.

1. For black background, the RGB values are 000, and since blinking is not mentioned, it is ‘normal’. Hence, the nibble corresponding to the background is 0000. This is to be the bits D₇ to D₄ of the attribute byte.

The lower four bits of the attribute is 0111.

Thus, the attribute byte is 0000 0111, i.e., 07.

2. No blinking. Hence, D₇ = 0.

White background needs 111.

Red foreground (text) needs 0100.

Thus, the attribute byte is 0111 0100, i.e., 74H.

3. Blinking and cyan background needs 1011.

Brown foreground (text) needs 0110.

Thus, the attribute byte is 1011 0110, i.e., B6H.

8.7 | BIOS 10H Functions

In Table 8.2, of interrupt vector allocation, it can be seen that the type number 10H has been allocated for video display applications. In this category, a number of actions are possible using different function numbers to be loaded in AH. In addition, each function requires that various information bytes be loaded in specified registers. Appendix D gives the list of functions of INT 10H. In this section, we shall use some of them (but not all) for various applications.

i) BIOS 10H Function 0 – Set video mode.

This function is for setting the ‘video mode’ – the function number is placed in AH and the video mode in AL. This function is for setting the video mode. It also clears the screen.

Example 8.3

```
.MODEL TINY
.CODE
.STARTUP

MOV AH, 0          ;set function number
MOV AL, 3          ;mode 3, standard text mode
INT 10H            ;the screen is found to be cleared
.EXIT
.END
```

This program clears the screen and sets the display for mode 3 (CGA).

ii) BIOS 10H Function 02 – Set cursor position.

This is for positioning the cursor by specifying the row and column co-ordinates. (The row co-ordinate varies from 0 to 24 and the column from 0 to 79.) The following registers should contain the relevant information. BH = page number (default page is 0), DH = row, DL = column.

Example 8.4

```
.MODEL TINY
.CODE
.STARTUP

MOV AH, 0          ;set function number
MOV AL, 3          ;mode 3, standard text mode
INT 10H            ;the screen is cleared with this.

MOV AH, 02         ;function number for setting cursor position
MOV BH, 0           ;specify page (which is the default page
MOV DH, 09          ;row co-ordinate
MOV DL, 35          ;column co-ordinate
INT 10H
.EXIT
.END
```

iii) Function 06 – Scroll up screen.

This function scrolls up the screen by the specified number of lines and blank lines appear at the bottom. The registers involved and the values to be stored in them are:

AL = number of lines to be scrolled up (0 for a full blank screen),

BH = attribute value or pixel value, CH:CL = starting row: column,

DH:DL = ending row: column

See the following program segment.

```
MOV AH, 06      ;function number
MOV AL, 00      ;number of lines to be scrolled up
MOV BH, 30H     ;cyan background, black foreground
MOV CH, 0       ;starting row co-ordinate = 0
MOV CL, 0       ;starting column co-ordinate = 0
MOV DH, 24      ;starting row co-ordinate = 24
MOV DL, 79      ;starting column co-ordinate = 79
INT 10H
```

This scrolls up 0 lines, and a full blank screen of cyan color is seen. Any text written on the screen will be of black color. The starting point is 0:0 (left hand corner of the screen), and the end point is the right corner, i.e., the point (24, 79).

This function number can also be used to create a window of a desired size given the co-ordinates of the window.

```
MOV AH, 06      ;function number
MOV AL, 9       ;number of lines to be scrolled up
MOV BH, 61H     ;brown background, blue foreground
MOV CH, 15      ;starting row co-ordinate = 15
MOV CL, 20      ;starting column co-ordinate = 20
MOV DH, 23      ;ending row co-ordinate = 23
MOV DL, 60      ;ending column co-ordinate = 60
```

The above program segment creates a window with the given co-ordinates.

Now, if two windows are to be created, repeat the above program segment with a different set of co-ordinates and a different color (for clarity of understanding). Because the attribute of a window remains unchanged until another operation changes it, we can set many different windows at the same time.

There is another function ‘Scroll Down screen’, with function number 07. This works similar to function 06, except that scrolling down the

screen causes the bottom lines to scroll off and blank lines to appear at the top.

iv) INT 10 H Function 09 – Display character with attribute at cursor position.

This function displays a character at the cursor position.

AL = ASCII character, BH = page number, BL = attribute byte, CX = count

```
MOV AH, 09      ;function number
MOV AL, 'S'     ;display 'S' at the cursor position
MOV BH, 0       ;page 0
MOV BL, 16H     ;blue background, brown foreground
MOV CX, 9       ;display the character 9 times
INT 10H
```

The above program segment displays ‘S’ nine times starting at the position of the cursor.

Note This function does not advance the cursor. So, if we try to display the word ‘Hello’ at the cursor position, the characters get displayed one on top of the other, unless we arrange to advance the cursor after the display of each character.

Example 8.5

The following program does the following:

1. Sets the video mode and clears the screen.
2. Makes a window of a specified size and color.
3. Sets the cursor at a specified position within the window.
4. Displays 10 times the character ‘*’ at the cursor position.

Solution

```
.MODEL TINY
.CODE
.STARTUP

MOV AH, 0      ;set video mode
MOV AL, 3      ;mode 3, normal text mode
INT 10H        ;clear screen
```

```

MOV AH, 06      ;function number for scrolling up
MOV AL, 10      ;no of lines = 10
MOV BH, 61H     ;brown background, blue foreground
MOV CH, 5       ;starting row co-ordinate = 5
MOV CL, 20      ;starting column co-ordinate = 20
MOV DH, 14      ;ending row co-ordinate = 14
MOV DL, 60      ;ending column co-ordinate = 60
INT 10H         ;create the scroll window

MOV AH, 02      ;function number to set the cursor
MOV BH, 0        ;page number = 0
MOV DH, 09      ;row co-ordinate of cursor
MOV DL, 39      ;column co-ordinate of cursor
INT 10H         ;cursor at (9, 39)

MOV AH, 09      ;function to display character
MOV AL, '*'     ;character to be displayed = '*'
MOV BH, 0        ;page number
MOV BL, 16H     ;blue with brown text
MOV CX, 10      ;count = 10
INT 10H         ;display the character 10 times

.EXIT
END

```

You may note that this program uses all the previously discussed functions of BIOS 10H for the purposes of clearing the screen (AH = 0), creating a scroll window (AH = 06), setting the cursor position (AH = 02) and then displaying the character '*' in the cursor position (AH = 09) 10 times.

Remember that the cursor position has not advanced by this. So, how do we write a string of different characters? We must fix up an initial value for the cursor position and then advance the cursor for each new character to be displayed.

Example 8.6

Write a program to display ‘HELLO’ vertically downwards at the center of the screen.

Solution

Here, we are to display a string vertically downwards. For that, we will do the following:

1. Fix up the initial cursor position. Write the first character there.
2. Next, increment the row co-ordinate of the cursor, and display the next character there. Repeat this for the whole string.
3. To avoid writing a long program, let us use two macros – one for setting the cursor (SET_C) at different row co-ordinates (the column co-ordinate is kept constant at 38) and another for displaying the different characters at each cursor position (DISP).
4. The output of this program is a vertical display of HELLO, with blue text blinking on a brown background. The brown and blinking together will cause the background to appear yellowish. The cursor position is at (15, 38) after the display.

```

.MODEL SMALL
.DATA
COL DB 38      ;specify the column co-ordinate of cursor
.CODE
.STARTUP

SET_C MACRO ROW      ;macro for positioning cursor
    MOV AH, 02      ;function for setting the cursor
    MOV BH, 0       ;page 0
    MOV DH, ROW     ;specify the row co-ordinate
    MOV DL, COL     ;specify the column co-ordinate
    INT 10H

ENDM

DISP MACRO CH      ;macro for displaying character
    MOV AH, 09      ;function for display
    MOV BH, 0       ;page 0
    MOV BL, 0E9H    ;blinking, brown background, blue text
    MOV AL, CH     ;the character to be displayed
    MOV CX, 1       ;display the character only once
    INT 10H        ;clear screen

ENDM

    MOV AH, 0          ;set video mode and clear screen
    MOV AL, 3          ;mode 3
    INT 10H
    SET_C 11         ;set cursor to row 11
    DISP 'H'          ;display 'H'
    SET_C 12         ;set cursor to row 12
    DISP 'E'          ;display 'E'
    SET_C 13         ;set cursor to row 13
    DISP 'L'          ;display 'L'
    SET_C 14         ;set cursor to row 14
    DISP 'L'          ;display 'L'
    SET_C 15         ;set cursor to row 15
    DISP 'O'          ;display 'O'

.EXIT
END

```

Example 8.7

Next, let us write a simple program to make the screen green in color on pressing the 'G' key.

Solution

```
.MODEL TINY
.CODE
.STARTUP

MOV AH, 01      ;DOS interrupt for keyboard entry
INT 21H         ;the code of the key pressed is in AL
CMP AL, 'G'     ;compare AL with the code of 'G' key
JNE LAST        ;if not equal, jump to LAST

MOV AH, 0        ;set video mode
MOV AL, 02       ;mode = 2
INT 10H         ;clear screen after setting the mode
MOV AH, 09       ;display function
MOV BH, 0        ;page 0
MOV BL, 22H      ;green background with green foreground
MOV CX, 2000    ;number of characters = 2000
INT 10H

LAST:
.EXIT
.END
```

In Example 8.7, the DOS interrupt 21H with AH = 01 is used for keyboard entry. The ASCII value of the character of the key pressed will be obtained in AL, which is compared with 'G'. In the display function, we have not loaded any character in AL to be displayed. This is because the attribute byte in BH is defined with the same foreground and background. Thus, whatever the character in AL, it will not be visible. We give a count of CX = 2000, which is the number of characters on the screen; i.e., 25×80 (2000) characters can be displayed.

8.8 | Addressing Video Memory Directly

Now, let us try to address video RAM, which is just like any other RAM. We know that the starting address of video memory for CGA display is B800:0000. We will attempt to display our data without using BIOS; rather, we write data directly into video RAM, which causes displaying of that information. Let us try to make the whole of page 0 display appear green. For this, a 16-bit number with the attribute byte and ASCII character is loaded in AX. This word is written into the 2000 locations

(25×80 characters) starting from the address B800H. Using string instructions, this display can be easily accomplished.

Example 8.8

```
.MODEL TINY
.CODE
.STARTUP
MOV AH, 0
MOV AL, 3
INT 10H      ;this is used to clear the screen

CLD          ;clear direction flag
MOV AX, 0B800H ;starting address of video memory
MOV ES, AX    ;make video memory the extra segment
MOV DI, 0     ;to point to first location in ES
MOV CX, 25*80 ;characters on the screen (2000)
MOV AH, 29H   ;attribute for green background
MOV AL, 20H   ;ASCII code of space bar key
REP STOSW    ;store this data in the 2000 locations

.EXIT
.END
```

Let us discuss the scheme of Example 8.8.

1. First, the whole screen is cleared using the INT 10H function 00.
2. Next, we use string instructions to address video memory. The ES register is loaded with the starting address of video memory.
3. DI is loaded with the offset 0.
4. CLD is for auto-incrementing this pointer.
5. The STOSW (store string word) is repeated 2000 times; i.e., the content of AX is stored in all the locations of page 0 of the video memory. For using the STOSW instruction, the destination memory should be defined to be the ‘extra segment’ and should be pointed by DI. This has been done here.
6. The content of AX is the attribute byte (AH) and the space bar character (AL). The space bar character display is a ‘blank’ display. Hence, the screen has only the background color, i.e., green.

In Example 8.9, the message ‘HELLO WORLD’ is displayed at the center of the screen. The background color is set to green with blinking (signified by the number A9H). Because of ‘blinking’, the green of this screen can be observed to be different from the green of Example 8.10. 20H corresponds to the ASCII code of the space bar key (no character). First, the whole screen is made green by setting the background color to be green and the foreground to be blank by filling it with the space bar code. Next, the message to be displayed is displayed starting in the

middle of the screen by locating the 2000th point (center of the 25×80 screen). Remember that each character takes two bytes of storage.

Example 8.9

```
.MODEL TINY
.CODE
.STARTUP
MESG DB 'HELLO WORLD'

MOV AH, 0          ;set video mode
MOV AL, 03         ;mode = 2
INT 10H            ;clear screen after setting the above

CLD                ;clear direction flag
MOV AX, 0B800H    ;starting address of video memory
MOV ES, AX         ;make video memory the extra segment
MOV DI, 0          ;point to first location in ES
MOV CX, 25*80      ;characters on the screen (2000)

MOV AH, 0A9H        ;green background, blinking blue text
MOV AL, 20H         ;ASCII code of space bar key
REP STOSW          ;store this in the 2000 locations
                   ;the screen is now green in color

MOV AX, 0B800H    ;starting address of video memory
MOV ES, AX         ;make video memory the ES
MOV DI, 2000        ;point to the center of the screen
MOV SI, OFFSET MESG ;SI to point to message
MOV AH, 0A9H        ;green background, blinking blue text
MOV CX, 11          ;characters in the message = 11

START:
LODSB              ;load each character in AL
STOSW              ;store each word in video memory
LOOP START          ;repeat until CX = 11

.EXIT
.END
```

The salient features of this program:

1. The foreground and background of the screen are made green and blank just as was done in Example 8.10. This is the first part of the program.
2. Then, each character is brought to AL using the instruction LODSB. For using this instruction, the data segment should contain the source data (the message) and SI should point to it. This has been done here.
3. The attribute byte is already in AH. The character to be displayed along with the attribute (in AH) is now in AX.

4. The content in AX is now stored in the video memory using STOSW. DI should point to the 2000th location of the video memory, since that is where the storing of the content of AX should start.
5. The LODSB and STOSW instructions are executed until CX = 0. Recollect that for using the LODSB instruction, the message should be in the data segment and should be pointed by SI. Here, the data segment is the same as the code segment as we are using the tiny model.
6. Thus, we find the string HELLO WORLD displayed on a green screen.

8.9 | Keyboard Interfacing

Next, let us use BIOS interrupts for the computer keyboard. For that, we first have to understand the computer keyboard as well as the hardware associated with the keyboard.

8.9.1 | Computer Keyboard

All of us are familiar with the PC keyboard, which is more or less standard. However, the keyboard, which is one of the most important (but taken for granted) part of the PC, did not always look like this. In the first IBM PC and PC-XT, there were 83 keys. Later, the PC-AT keyboard had the same number of keys, but the arrangement of the keys was different. After taking into consideration various design ideas, the current PC keyboard has been designed to have 101 keys, including the function keys F1 to F12. This is called the ‘enhanced keyboard’. We are all very much used to the keyboard of the PC, but not everyone really knows the functions of each and every key. So, let us start with a discussion on the keys of the keyboard.

Character keys

We have, first, the character keys a to z, and above it there are the number keys. All these keys have dual function, as they are pressed along with or without the shift key. These are the keys most commonly used and most familiar to us.

Function Keys

These keys marked F0 to F12 and located on the topmost row of the keyboard were widely used with many older DOS programs. They are still very popular as keyboard shortcuts. Sometimes, they are used in combination with other keys like Ctrl, Alt or Shift keys. The help menu of any program will be seen to contain shortcuts with such combinations.

Enter (Return) Key

This key is used to enter commands or to move the cursor to the beginning of the next line. If we make a choice out of a set, this key specifies the choice.

Escape Key

It is used to escape from a program or exit to a previous screen. It is helpful also to get out of difficult situations as many of us have experienced.

ALT (Alternate) Key

This key is used in combination with other keys like character keys or function keys and is, as defined, used in various programs. For example, in the TC++ compiler, Alt +F9 is the command for compile.

Ctrl (control) Key

This key is also used in combination with other keys, and it is defined for certain programs. For example, in MS Word, Ctrl+c copies data and Ctrl+v pastes the copied data. This is not necessarily valid for all programs – for example, not for the TC++ editor, anyway.

Caps Lock Key

The Caps Lock key is a toggle key. Pressing it once turns it on. Pressing it again turns it off. Some computer keyboards have a light or indicator that shows when the Caps Lock is on. When Caps Lock is on, every letter that is typed will be a capital letter. The Caps Lock key on a computer keyboard affects only letters. It has no effect on the number or symbol keys.

Num Lock and Numeric Keypad

Most desktop computers have a numeric keypad. It is on this numeric keypad that the page Down, page Up and other arrow keys are seen. The Num Lock is a toggle key, which when turned on allows numbers to be typed in. When this key is toggled off, the other options of each key will be activated, for example Pg Dn and Pg Up. Note that there are other keys also on the numeric keypad.

Windows Key

Most windows based computers have a ‘Windows’ key. It is marked with a Microsoft windows symbol and is in the bottom row of the keypad, and there may be two such keys on either side of the space bar key. Pressing this key will open the start menu. It can be used in combination with other keys for some short cuts. For example, Windows+d will minimize all open windows and one more such key press will maximize them.

Application Key

Another key, which most of us might not have noticed at all, is a key which has a symbol of a page with a number of lines. This is called the application key and it works like a right mouse click. Try using it in the middle of typing, using this key is much easier than having to get to the mouse and click it.

8.9.2 | Keyboard Hardware

Now, let us discuss the hardware associated with the keyboard of a modern PC. The keyboard along with the hardware involved is quite a complicated part, but here, the attempt will be to present it in a simple form with important details covered and more details left to be referred from other sources.

The earlier PCs had a keyboard of 83 keys only, but now, the standard keyboard has 101 keys including the alphanumeric characters. The keyboard we are assuming is the PS/2 keyboard and not the ones which use the USB connector (there are some differences in this case). Let us summarize some of the points regarding the keyboard.

8.9.2.1 | Keyboard Controller

The keyboard which is connected to the motherboard through a cable has a microcontroller in it, which belongs to the 8042 family. In the motherboard, there is another similar microcontroller (or its emulation, as in present times) and they communicate through a bidirectional serial communication protocol.

8.9.2.2 | Scan Code

The microcontroller inside the keyboard is programmed to detect whether a key has been pressed and identify the key in case of a key being pressed. When a key is pressed, the microcontroller therein sends to the motherboard a unique ‘scan code’ for each key. A key press is generally followed by a key release and this ‘release’ causes another scan

code to be sent, which is **different** from that of the key press of the same key. Key press and release are also called ‘make’ and ‘break’. The make and break scan codes are different for a key. They differ by 80H. For example, if the make scan code for a character is 9, its break scan code will be 89H. **Incidentally, the scan code has nothing to do with the ASCII or other values of any key but is a unique code defined by IBM.** (Tables 8.7 to 8.9, at the end of this chapter, give the scan codes for the 101 keys of a PC).

Keyboard Status Bytes

On checking this list of scan codes, it will be seen that the scan code of a key is the same whether the key is a lowercase or uppercase alphabet. Also, it is the same whether the shift key has been pressed or not or if the Caps lock is on or off. See the list of scan codes in Tables 8.7 to 8.10. The scan code of * and 8 which are on the same key is ‘09’. The scan code of g and G is 22H. How is that accounted for?

The answer is that there are two keyboard status bytes which are held in the BIOS data areas – 40:17 and 40:18. The first status byte maintains the current status of the modifier keys on the keyboard. The bits have the following meanings as shown in Fig. 8.6a. If a toggle key is on, the corresponding bit is 1. For example, for a caps lock key being toggled on, the corresponding bit is 1; otherwise, it is zero. For the shift, alt and ctrl keys, ‘1’ indicates that the corresponding keys have been pressed. The second keyboard status byte specifies if the corresponding keys are ‘currently’ down.

D ₇	Insert key toggled
D ₆	Caps lock key toggled
D ₅	Num lock key toggled
D ₄	Scroll lock key toggled
D ₃	Alt pressed
D ₂	Ctrl pressed
D ₁	Left shift key pressed
D ₀	Rightshift key pressed

Note The corresponding bit is ‘1’ for the keys pressed or toggled.

Figure 8.6a | First keyboard status byte

D ₇	Insert key pressed
D ₆	Caps lock key pressed
D ₅	Num lock key pressed
D ₄	Scroll lock key pressed
D ₃	Ctrl Num lock (pause) pressed
D ₂	Sys Req pressed
D ₁	Left Alt key pressed
D ₀	Left Ctrl key pressed

Note The corresponding bit is '1' for the keys kept pressed.

Figure 8.6b | Second keyboard status byte

8.9.3 | Keyboard Interrupt Type 09

In Table 8.2, we see that the interrupt with type number 09 has been allocated to the keyboard on IRQ1 line. Thus, this is a hardware interrupt. The IRQ line stands for 'interrupt request line 1' of the 8259 PIC/(Refer Chapter 10), which is vectored to type 9 of the interrupt vector table. Thus, when a key is pressed, its scan code is sent to the motherboard in serial form and therein converted to parallel form, and this 8-bit scan code is presented to port A of the 8255 (refer Chapter 9) with I/O address 60H, and along with this, the IRQ1 line is activated, which has been vectored to INT 9.

8.9.3.1 | BIOS INT 09 Routine

What is the INT 09 routine?

1. The ISR reads the scan code from Port 60H and identifies the key pressed. There are three possibilities for this.
 1. The key may correspond to special keys like shift, alt, control etc. In that case, the corresponding bit in the **status byte (Fig. 8.6a)** will be set.
 2. The second case is that the key may correspond to an ASCII character, and then the 8-bit scan code and the ASCII value are stored in the keyboard buffer (which is a

32-byte area in the BIOS data area starting from 0040:001E).

3. Another case is when there is no ASCII code for the key – this is for function keys F1 to F12. In this case, the ISR saves the scan code in the buffer and also puts 00 in place of the ASCII code.
2. When the key is released, it sends a different scan code, and this is verified by the ISR and confirmed to be a key release.
3. There is a possibility that the key continues to be kept pressed. If it is found so for more than half a second, the ISR identifies it as a new key press and sends its scan code accordingly. Repeating the same key is referred to as ‘typematic’ in IBM literature.

Next, let us use some software interrupts related to the keyboard. We have already used a few DOS interrupts for keyboard access. Now, we will get acquainted with a few BIOS interrupts for keyboard functions. The software interrupt type 16H has been allocated for the keyboard.

8.9.4 | BIOS 16H Functions

A few BIOS interrupts for keyboard control are presented here. Some of the early BIOS functions catered to the old 83-key keyboard only. Later, some more BIOS functions were added for the enhanced keyboard.

1. AH = 0 or AH = 10H. The first function number is for the older keyboard. The second one is the equivalent one for the enhanced keyboard. This function checks for a character in the keyboard buffer. If available, the scan code is returned in AH and ASCII value in AL. For function keys (F0 to F12) which have no ASCII values, AL = 0. If no character is available in the keyboard buffer, the function waits for a key press.
2. AH = 01 or AH = 11H. The first function number is for the older keyboard. The second one is the equivalent one for the enhanced keyboard. This function is similar to the previous one, except that if no character is available in the keyboard buffer, it does not wait for a key press. It simply sets ZF (ZF = 1) and returns.
3. AH = 02 or AH = 12H. The first function number is for the older keyboard. The second one is the equivalent one for the enhanced keyboard. This function returns the first keyboard status byte in the AL register. This status byte is also available in the BIOS data area 0040:0017.

Example 8.10

Explain what the following program does.

```

.MODEL TINY
.CODE
.STARTUP
MOV AH, 10H
INT 16H
MOV AH, 12H
INT 16H
.EXIT
END

```

Solution

There are two interrupt functions in the above program.

1. The function with AH = 10H waits for a key press. If the key pressed is 8, then AL = 38H and AH = 09. The value 09 is the scan code of 8 and 38H is its ASCII value of 8.

The key 8 also represents the '*' character which is displayed when the shift key is also pressed. Then, AH = 09 and AL = 2AH, which is the ASCII value of '*'. Thus, the scan code is the same for both the characters, but their ASCII values are different.

If the key pressed is F5, the content of AL = 0, because the function keys F0 to F12 have no ASCII values. The content of AH = 3FH, which is the scan code of the F5 key.

2. The function with AH = 12H provides the keyboard status byte in AL.

With no shift, alt or control keys pressed, AL = 0.

With CAPSLOCK on, NUMSLOCK on, and the left shift key kept pressed, the value in AL = 62H, i.e., 0110 0010B.

Keep the left shift key pressed, and check the content of the BIOS data area 0040:0017.

-d0040:0017
0040:0010 62-00

We find the first keyboard status flag saved there.

Example 8.11

The following program fills up the screen with the key pressed. If no key is pressed, it keeps waiting for a key press.

```

.MODEL TINY
.CODE
.STARTUP
NO_KEY: MOV AH, 11H ;function for waiting for key
        INT 16H
        JZ NO_KEY ;if ZF = 1, no key press, keep waiting
KEY:   MOV AH, 10H ;function for taking into AL the key
        INT 16H
        MOV CX, 2000 ;number of characters to fill the screen
DISP:   MOV DL, AL ;move the ASCII character to DL
        MOV AH, 02 ;function number to display DL content
        INT 21H ;DOS interrupt 21H
        LOOP DISP ;display until CX = 0
.EXIT
END

```

The program of Example 8.11 uses INT 16H function 11H. This function checks whether there is a character in the keyboard buffer. If there is no character and no key press either, it sets the Zero flag. The state of the Zero flag is used to wait for a key press. When a key is pressed, this loop is exited, and it takes into AL the ASCII value of the key pressed (using BIOS INT 16H function 10H). Next, the content of AL is moved to DL and DOS INT 21H function is used to display it 2000 times, so as to fill the whole screen.

Now that we have had a reasonably good exposure to the intricacies associated with the video display, keyboard and the BIOS interrupts associated with them, we can use them to manipulate the interrupt vectors of the PC.

8.10 | Hooking an Interrupt

A term associated with interrupts is ‘hooking an interrupt’. This just means the mechanism of installing an interrupt vector in the IVT. The tricky part of this is that the type number we plan to use may already be in use for another interrupt routine. This ISR is temporarily kept aside when we install a new vector corresponding to the same type number. To install an interrupt vector, the assembler needs to address absolute memory, and the easiest way to do it is using certain DOS interrupt functions. Now, we will go through the whole process of hooking interrupts.

8.10.1 | Terminate and Stay Resident

DOS uses a part of the 640K of RAM in the lower area of memory, and the exact amount of this varies from version to version. Whenever an

application program runs, a portion of RAM is allocated to the application. After this, the program is abandoned and the allocated memory is freed and is marked as available for other programs. If this is not done, the amount of free memory keeps on getting reduced. However, there are programs which are kept ‘resident’ in memory even after it is run and abandoned. So, when it is to be run again, it need not be brought from the disk again. This is quite comfortable for the specific program, but the disadvantage is that free memory is eaten up by such programs. Such programs are titled ‘Terminate and stay resident (TSR) programs’. There is a specific interrupt 21H function which makes a program resident. Most of these TSR programs are invoked by interrupts, and let us see how it is done.

8.10.2 | DOS Functions for Interrupt Hooks

There are three functions which we will use to hook an interrupt vector.

i) INT 21H, Function 35H – Get Interrupt Vector.

Get the segment-offset value of an interrupt vector.

Input: AL = interrupt number. Output: ES:BX = address of the interrupt handler.

When we attempt to get our new ISR pointed by a particular interrupt vector, there is the possibility that there was already a previous ISR using that type number. So, we have to save the value of CS:IP of that interrupt handler. This will be necessary for restoring the previous state of the interrupt vector table. This is done by Function 35H.

ii) INT 21H, Function 25H – Set Interrupt Vector.

Set an entry in the Interrupt Vector Table to a new address. Input: DS:DX points to the interrupt-handling routine that will be inserted in the table; AL = the interrupt number

We then have to get our ISR to be pointed by the interrupt vector we have decided upon. This is done by function 25H.

iii) INT 21H, Function 31H – Terminate and stay resident.

Terminate the current program or process but make it resident in memory and attempt to set the current memory allocation to the number of paragraphs specified in DX. Input: AL = return code and DX = requested number of paragraphs.

The function 31H makes our ISR resident in memory. The number of paragraphs (16-byte chunks) that our program occupies is to be specified in DX.

The program for installation of an interrupt hook is expected to be a com program and hence we proceed as follows.

Example 8.12

Here, the new ISR named ‘NEW60’ is to be installed as a type 60H interrupt. The ISR is a program to make the full screen cyan in color with black foreground.

```
.MODEL TINY
.CODE
.STARTUP
JMP START
OLD DD?
NEW60 PROC FAR
    MOV AH, 06      ;function number
    MOV AL, 00      ;no character
    MOV BH, 30H     ;cyan background, black foreground
    MOV CH, 0       ;row co-ordinate = 0
    MOV CL, 0       ;column co-ordinate = 0
    MOV DH, 24      ;row co-ordinate = 24
    MOV DL, 79      ;column co-ordinate = 79
    INT 10H
    IRET
NEW60 ENDP
START: MOV AX, 3560H   ;get interrupt vector of type 60H
       INT 21H
       MOV WORD PTR OLD, BX    ;save old IP in OLD
       MOV WORD PTR OLD + 2, ES ;save old CS in OLD + 2
       MOV DX, OFFSET NEW60    ;ISR pointed by 60H vector
       MOV AX, 2560H           ;AL = 60H, AH = 25H
       INT 21H
       MOV DX, OFFSET START    ;find number of paragraphs
       MOV CL, 4                ;divide by 16, by shifting 4 times
       SHR DX, CL               ;to the right
       INC DX                 ;add 1 to it
       MOV AX, 3100H           ;exit with AH = 31H rather than 4CH
       INT 21H                 ;this makes NEW60 resident
END
```

Let us examine the salient features of Example 8.12.

1. This program installs a new ISR corresponding to type 60H vector. It is at offset NEW6o. This means that the CS and IP of this program (ISR) are to be installed in the corresponding position in the interrupt vector table.
2. The new ISR is a program named NEW6o. It does the job of making the screen cyan in color. It ends with the IRET instruction and is defined as a far procedure.
3. The program for saving the old CS and IP and installing the new CS and IP begins at the label START.
4. The function AH = 35H gets the old IP and CS in BX and ES. These are saved in 'OLD', which is a double word location.
5. The function AH = 25H installs the CS and IP of NEW6o into the IVT.
6. The function AH = 31H makes the program NEW6o resident.
7. Note that the program does not end with AH = 4CH but has to end with AH = 31H.

The next issue is how to invoke this program which we have now installed. Since we have installed it at the type 60H location of the IVT, we can invoke it by the instruction INT 60H as in Example 8.15.

Example 8.13

```
.MODEL TINY
.CODE
.STARTUP
MOV AH, 0      ;function number for setting video mode
MOV AL, 3      ;mode 3
INT 10H        ;this is for clearing the screen
INT 60H        ;run the ISR now installed as Type 60H
.EXIT
END
```

Example 8.13 will make the screen cyan in color, because INT 60H gets invoked.

Notes

1. One thing to note is that the installation of the new ISR will be lost if the system is closed or re-booted. If you exit the command window, you will have to re-install it again; i.e., run the program of Example 8.14 again if you want to use the program in Example 8.15.
2. Never use DOS INT 21H functions in a TSR program, because DOS is **not reentrant**, and its usage in TSR programs will cause havoc.

What is a Reentrant Program?

Re-entrancy is a useful, memory-saving technique for multi-programmed timesharing systems. A Reentrant Procedure is one in which multiple users can share a single copy of a program during the same period. A programmer writes a reentrant program by making sure that no instructions modify the contents of variable values in other instructions within the program. Each time the program is entered for a user, a data area is obtained in which to keep all the variable values for that user. The data area is in another part of memory from the program itself. When the program is interrupted to give another user a turn to use the program, information about the data area associated with that user is saved. When the interrupted user of the program is once again given control of the program, information in the saved data area is recovered and the program can be reentered without concern that the previous user has changed some instruction within the program.

8.10.3 | Hooking Into Hardware Interrupts

Now, let us do the same with hardware interrupts. Let us choose the hardware interrupt with type number 09, which has been allocated to the keyboard on IRQ1 line. We will replace the normal keyboard sequence associated with this hardware interrupt with another program and that program will be activated with a hotkey.

What is a hot key? It is defined as a user-defined key sequence that executes a command or causes the operating system to switch to another program. We are going to use a hot key in Example 8.14, most of which is similar to Example 8.14. The important points are:

1. It replaces the normal routine associated with INT 9.
2. The hot key used is Alt+F10. To verify this combination, first the INT 16H routine with AH = 12H is used to check for the ALT key. This tests for the D₃ bit of the data in AL (first keyboard status byte), which is to be '1' if the Alt key has been pressed. Then, port A of the 8255 (with address 60H) is tested for the scan code of F10 (which is 44H). If both these conditions are true, the new ISR is taken up for execution.
3. If it is found that the hot key pressed is not the right one, control goes to OVER, which retrieves the old value of CS and IP for the type 9 interrupt. This is very important, because the keyboard is locked (until a reboot) if the normal keyboard routine is not restored.
4. After our TSR is executed, then also control goes back to OVER for the same reason as stated above.
5. The hot key activated ISR is similar to Example 8.5, which brings up a video window.

Note The scan codes of keys are listed in Tables 8.7 to 8.9.

Table 8.7 | PC Scan Codes for 83 key PC/XT Keyboard

Hex	Key	Hex	Key	Hex	Key	Hex	Key
01	Esc	15	Y and y	29	~ and '	3D	F3
02	! and 1	16	U and u	2A	LeftShift	3E	F4
03	@ and 2	17	I and i	2B	and \	3F	F5
04	# and 3	18	O and o	2C	Z and z	40	F6
05	\$ and 4	19	P and p	2D	X and x	41	F7
06	% and 5	1A	{ and [2E	C and c	42	F8
07	^ and 6	1B	L and l	2F	V and v	43	F9
08	& and 7	1C	Enter	30	B and b	44	F10
09	* and 8	1D	Ctrl	31	N and n	45	NumLock
0A	(and 9	1E	A and a	32	M and m	46	ScrolLock
0B) and 0	1F	S and s	33	< and ,	47	7 and Home
0C	_ and -	20	D and d	34	> and .	48	8 and UpArrow
0D	+ and =	21	F and f	35	? and /	49	9 and PgUp
0E	back space	22	G and g	36	RightShift	4A	-(gray)
0F	tab	23	H and h	37	PrtSc and *	4B	4 and LeftArrow
10	Q and q	24	J and j	38	Alt	4C	5 (keypad)
11	W and w	25	K and k	39	Spacebar	4D	6 and RightArrow
12	E and e	26	L and l	3A	CapsLock	4E	+(gray)
13	R and r	27	: and ;	3B	F1	4F	1 and End
14	T and t	28	" and '	3C	F2	50	2 and DownArrow
						51	3 and PgDn
						52	0 and Ind
						53	. and Del

Table 8.8 | Combination Key Scan Codes

Hex	Keys	Hex	Keys	Hex	Keys	Hex	Keys
54	Shift F1	60	Ctrl F3	6C	Alt F5	78	Alt 1
55	Shift F2	61	Ctrl F4	6D	Alt F6	79	Alt 2
56	Shift F3	62	Ctrl F5	6E	Alt F7	7A	Alt 3
57	Shift F4	63	Ctrl F6	6F	Alt F8	7B	Alt 4
58	Shift F5	64	Ctrl F7	70	Alt F9	7C	Alt 5
59	Shift F6	65	Ctrl F8	71	Alt F10	7D	Alt 6
5A	Shift F7	66	Ctrl F9	72	Ctrl PrtSc	7E	Alt 7
5B	Shift F8	67	Ctrl F10	73	Ctrl LeftArrow	7F	Alt 8
5C	Shift F9	68	Alt F1	74	Ctrl RightArrow	80	Alt 9
5D	Shift F10	69	Alt F2	75	Ctrl End	81	Alt 10
5E	Ctrl F1	6A	Alt F3	76	Ctrl PgDn		
5F	Ctrl F2	6B	Alt F4	77	Ctrl Home		

Table 8.9 | Extended Keyboard Scan Codes

Hex	Keys	Hex	Keys	Hex	Keys	Hex	Keys
85	F11	8E	Ctrl -	97	Alt Home	A0	Alt DownArrow
86	F12	8F	Ctrl 5	98	Alt UpArrow	A1	Alt PgDn
87	Shift F11	90	Ctrl +	99	Alt PgUp	A2	Alt Insert
88	Shift F12	91	Ctrl DownArrow	9A		A3	Alt Delete
89	Ctrl F11	92	Ctrl Insert	9B	Alt LeftArrow	A4	Alt /
8A	Ctrl F12	93	Ctrl Delete	9C		A5	Alt Tab
8B	Alt F11	94	Ctrl Tab	9D	Alt RightArrow	A6	Alt Enter
8C	Alt F12	95	Ctrl /	9E			
8D	Ctrl UpArrow	96	Ctrl *	9F	Alt End		

Example 8.14

```

.MODEL TINY
.CODE
.STARTUP
    JMP START
    OLD DD?
NEWINT9     PROC FAR           ;the ISR to be installed
    PUSH AX
    MOV AH, 12H          ;function AH = 12H, KB s/w interrupt
    INT 16H              ;gets into AL the kb status byte
    TEST AL, 08          ;check whether D3 is high
    JZ OVER              ;if not, go to OVER
    IN AL, 60H            ;read parallel port
    CMP AL, 44H          ;compare AL with scan code of F10
    JNE OVER             ;if not equal, go to OVER
    MOV AH, 0              ;the following is the pop up program
    MOV AL, 3
    INT 10H              ;set video mode and clear screen

    MOV AH, 06          ;function for scroll window
    MOV AL, 07          ;no of lines = 7
    MOV BH, 66H          ;attribute byte
    MOV CH, 0CH
    MOV CL, 19H
    MOV DH, 12H

```

```

        MOV DL, 36H
        INT 10H           ;this program sets a video window

OVER:   POP AX
        JMP CS:OLD

NEWINT9 ENDP

START:  MOV AH, 35H      ;save old interrupt vector
        MOV AL, 09       ;the type number = 9
        INT 21H          ;DOS interrupt for getting vector

        MOV WORD PTR OLD, BX ;save old IP
        MOV WORD PTR OLD+2, ES ;save old CS
        MOV AH, 25H         ;install new vector
        MOV AL, 09           ;at type number 9
        MOV DX, OFFSET NEWINT9 ;point DX to the new ISR
        INT 21H

        MOV DX, OFFSET START ;count the number of paragraphs
        MOV CL, 4
        SHR DX, CL
        INC DX
        MOV AX, 3100H        ;make the program resident
        INT 21H
        END

```

Example 8.15 is a similar program, except that the hot keys to be invoked are Ctrl+F3. The ISR for this hot key is copied from Example 8.8. It accesses the video memory directly and causes the full screen to be green in color.

Example 8.15

```

.MODEL TINY
.CODE
.STARTUP
JMP START
OLD DD?

NEWINT9 PROC FAR
    PUSH AX           ;hot key is     Ctrl+F3
    MOV AH, 12H
    INT 16H
    TEST AL, 04       ;test if Ctrl key is pressed
    JZ OVER
    IN AL, 60H
    CMP AL, 3DH       ;test for the scan code of F3
    JNE OVER
    MOV AH, 0           ;the new ISR for Int 9
    MOV AL, 3           ;this makes the screen green

```

```

INT 10H
CLD
MOV AX, 0B800H
MOV ES, AX
MOV DI, 0
MOV CX, 25*80
MOV AX, 2920H
REP STOSW

OVER: POP AX
      JMP CS:OLD
NEWINT9 ENDP

START: MOV AH, 35H
       MOV AL, 09
       INT 21H
       MOV WORD PTR OLD, BX
       MOV WORD PTR OLD+2, ES
       MOV AH, 25H
       MOV AL, 09
       MOV DX, OFFSET NEWINT9
       INT 21H
       MOV DX, OFFSET START
       MOV CL, 4
       SHR DX, CL
       INC DX
       MOV AX, 3100H
       INT 21H
END

```

The next is a TSR program activated by the hot key combination **ctrl+alt+i**. It prints the word ‘ANURAG’ in green color in a red window with size as specified in the program. Since DOS interrupts cannot be used, the string is not displayed using the DOS interrupt with function number 09. Instead, a macro is used with BIOS functions to print single characters. See Example 8.16.

Example 8.16

```

.MODEL TINY
.CODE
.STARTUP

JMP START
OLD DD?
CSET MACRO      ;macro for setting character position.
MOV AH, 02      ;function no to set the cursor

```

```

MOV BH, 0           ;page no = 0
INC DL
INT 10H
ENDM
DISP MACRO SYMB      ;macro for displaying a character
CSET
MOV AH, 09          ;to display character
                     ;at cursor position
MOV AL, SYMB        ;display SYMB at cursor position
MOV BH, 0           ;page = 0
MOV BL, 04AH        ;attribute byte
MOV CX, 1           ;display the character once
INT 10H
ENDM
NEWINT9 PROC FAR   ;the ISR to be installed.
PUSH AX
MOV AH, 12H          ;AH = 12H, kb s/w interrupt
INT 16H              ;gets into AL, kb status byte
TEST AL, 08h         ;check whether D3 is high - alt key
JZ OVER              ;if not, go to OVER
TEST AL, 04h         ;check if D2 is high - ctrl key
JZ OVER              ;if not, go to OVER
IN AL, 60H            ;read parallel port
CMP AL, 17H          ;compare AL with the scan code of i
JNE OVER              ;if not equal, go to OVER
                     ;the following is the pop up program

MOV AH, 0
MOV AL, 3
INT 10H              ;set video mode and clear screen
MOV AH, 06            ;function for a scroll window
MOV AL, 07            ;no of lines = 7
MOV BH, 4ah           ;attribute byte
MOV CH, 0
MOV CL, 18
MOV DH, 6
MOV DL, 61
INT 10H              ;video window from (0, 18) to (6, 61)
                     ;display values with cursor position
MOV DL, 20
MOV DH, 2
DISP 'A'
DISP 'N'

```

```

DISP 'U'
DISP 'R'
DISP 'A'
DISP 'G'
CSET

OVER: POP AX           ;popping AX
      JMP CS:OLD
      NEWINT9 ENDP      ;end of ISR
                           ;main program

START: MOV AH, 35H       ;save old interrupt vector
      MOV AL, 09          ;the type number = 9
      INT 21H             ;DOS interrupt for getting vector
      MOV WORD PTR OLD, BX ;save old IP
      MOV WORD PTR OLD+2, ES ;save old CS
      MOV AH, 25H          ;install new vector
      MOV AL, 09            ;at type number 9
      MOV DX, OFFSET NEWINT9 ;point DX to the new ISR
      INT 21H
      MOV DX, OFFSET START ;count number of paragraphs
      MOV CL, 4
      SHR DX, CL
      INC DX
      MOV AX, 3100H         ;make the program resident
      INT 21H
END

```

The advantage of TSR programs that work with a hot key is that as the program is resident in memory, it can be invoked while we are doing something else. The disadvantage of TSR programs in DOS is that it gives users direct access to hardware and peripherals, which may cause havoc.

8.10.3.1 | PC Scan Codes[†]

KEY POINTS OF THIS CHAPTER

- Whenever the processor is performing a task, it can be interrupted by other programs or peripherals requesting service.
- The 8086 responds to an interrupt in a standard way, irrespective of the source of the interrupt.
- Associated with any interrupt is an interrupt service routine and an interrupt vector.
- The 8086 has a table with size 1K for storing all its 256 interrupt vectors.
- Certain interrupt vectors are reserved by Intel for specific purposes.
- Many interrupt types are used by BIOS and DOS.
- BIOS 10H functions are commonly used for display functions.
- It is also possible to address video memory directly.

- For text mode display, each character needs two bytes, one for the ASCII code and one for the attribute.
- BIOS 16H functions are used for accessing the keyboard.
- The hardware interrupt 09 is used by the PC keyboard.
- It is possible to hook interrupts using some standard DOS functions.
- TSR programs are pop up programs that are resident in memory.
- TSR programs are useful, but since hardware and interrupt vector tables are accessed directly, they can create havoc if not used with caution.

QUESTIONS

- Why are the interrupt and trap flags cleared as part of the interrupt response?
- How many hardware and software interrupts can the 8086 support?
- Explain the terms ‘interrupt service routine’ and ‘interrupt vector’.
- The hardware interrupt INTR is called a non-vectored interrupt. Why?
- List the interrupts reserved by Intel.
- Explain the sequence of actions that occur in an interrupt acknowledge cycle.
- What happens if a number of interrupts occur at the same time?
- What is meant by the ‘attribute’ byte with reference to video display of text?
- List a few BIOS functions used for display activation.
- List a few BIOS functions used for keyboard testing.
- What does the word TSR mean?
- Why cannot DOS interrupts be used in TSR programs?

EXERCISES

1. Find the address in the IVT for the interrupts of types 102H and 200H.
 2. Find the attribute byte and character byte for the following cases:
 1. Displaying ‘&’ in a blue background and red foreground.
 2. Displaying ‘O’ with blinking red background and yellow text.
 3. Write a program to create display windows of three sizes and colors at the same time.
 4. Write a program to write a string ‘WELCOME HOME’ horizontally on a blue screen. Write the program using the following:
- o. BIOS functions.**
1. Accessing video memory directly.
 5. Modify the above program such that it is activated by pressing a specific key only.
 6. Use BIOS 16H function for the above program for checking for a specific key.

7. Write a program to print the names of three students on three rows with their roll numbers on the left of the names. This should be printed on a small red window on the top of the video screen.
8. Write TSR programs for getting the above display with different hot key combinations.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

9

Peripheral Interfacing—I

IN THIS CHAPTER, YOU WILL LEARN

- The necessity of using dedicated hardware for peripheral interfacing.
- The architecture and programming features of the chip 8255.
- The method of interfacing an ADC to the 8086.
- The methods of generating different waveforms using a DAC.
- The way in which an LCD can be used to display ASCII characters.
- The principle of the stepper motor and how it is controlled by the CPU.
- Hex keyboard interfacing to the 8086.
- The interfacing of dynamic multiplexed LEDs with the 8086.
- That in all the above cases the 8255 is used as the parallel interface controller.

Introduction

In Chapter 6, we made a study of the hardware features of 8086. In Chapter 7, the methods by which memory and ports are connected to the 8086 have been clarified. One point that should be clear by now is that the processor is connected to memory and I/O devices through the address/data and control lines. Remember that the processor is just a computing device. It computes what is expected of it and transfers it to the outside world; in essence, the outside world is external to the processor – memory and I/O devices are external to the processor. Memory management for 8086 is done by using the read and write

control signals, and I/O is also accessed using these signals. However, I/O devices are of different varieties, functions, features, specifications and so on. For example, a keyboard is very different from a video monitor. The point is that the 8086 does not have the requisite hardware within it to manage each of them according to their special requirements and specifications. Thus, it is clear that I/O devices need some extra hardware to let them be interfaced to the processor. It is in this context that various interfacing chips have been designed. All these chips are programmable in the sense that they can be made to work in different ways as required by ‘programming’ them using 8086 instructions. Thus, there are separate chips for managing parallel data transfer, serial data transfer, keyboard, interrupts and timers, for example. We will now go into the details of interfacing various I/O devices to the processor using these special purpose interfacing chips.

9.1 | Trainer Kit

Remember that current PCs do not use the 8086 processor, and therefore, we cannot use the PC for a first level understanding of interfacing. However, the PC uses many interfacing chips in its ‘chipset’. Now, we will try to understand the idea of ‘interfacing’ by the use of a trainer kit which has been designed for educational purposes. As the name indicates, it will have all the hardware and software components for ‘training’. The trainer kit used here has the following specifications (see Fig. 9.1).



Figure 9.1 | The trainer kit

Features

1. Intel 8086 CPU at 4.77 MHz clock speed.
2. 16 KB for monitor EPROM upgradable to 64 KB.
3. 16 KB RAM expandable to 64 KB.
4. Standard RS232C compatible serial port brought out to a 9-pin D-type male connector.
5. 3 channel 16-bit counter/timer using 8253 terminating in an I/O pin connector (one channel used for baud rate generation).
6. Fully buffered address, data and control signals terminated at a 50-pin header for interfacing.
7. 8 nos. of interrupt lines brought out to a 10-pin connector using 8259 programmable interrupt controller.
8. Serial communication package to link PC and the kit.

What is to be understood from these specifications is that the clock frequency of the processor used is 4.77 MHz, and that its data, address and control lines are buffered and available in a 50-pin connector. RAM and ROM are available on the board. The trainer also has a programmable interrupt controller (8259) and a timer chip 8253. It has an AT keyboard. The kit can be connected to the serial port of a PC. Thus, the most convenient way would be to write our program in the PC (using an editor), assemble and link it using MASM, and then download the resultant hex file to the RAM on the trainer board. Then, the processor on the trainer board will execute the program and give the results appropriately. The trainer also has what is called a ‘monitor program’ to manage the whole setup.

With this brief introduction, we will start our discussion on the various peripherals and the corresponding interfacing chips. The sequence followed will be to present a chip, its features and its programming and then use it for different applications.

9.2 | Programmable Peripheral Interface (PPI)-8255A

This chip is also called a parallel port chip and it eases out the problems and issues related to parallel data transfer. Parallel data transfer is what we have done all this time. We have transferred 8/16 bits at one go either to/from memory or I/O devices. The I/O devices we have discussed (Chapter 7) were quite simple and needed only a read/write control signal and a select pulse generated during the read/write cycle. However, when I/O devices are more versatile and have more features, a PPI will be very useful, especially when more than one I/O device is to be interfaced to the processor. When this chip is used, its functions are

sufficient to ensure that normally no other extra hardware is required to interface peripheral devices that perform parallel data transfer. The chip 8255A is a more advanced version of the original 8255, but since the original one is obsolete, we will simply refer to this chip as the parallel port chip 8255.

Using the 8255

Let us be clear about why this chip is necessary in the first place and how it can be used (see Fig. 9.2). The data bus of the 8086 is connected to the 8255. That means data transfer between the two chips is possible.

Consider first the case of data from the 8086 being sent to the 8255. Where does the 8255 keep the data? The answer is that it has registers called ‘ports’; there are three ports here – A, B and C, and these ports have pins connecting it to external devices. Thus, port A has 8 pins PA₀ to PA₇ – so also ports B and C as well. To these port pins, external devices like keyboards, displays and printers can be connected. Figure 9.2 shows a set of LEDs connected to Port C and a printer to Port B. This means that the data from the 8086 can be transferred to the output devices, routed through these ports of 8255. To do that, we must be able to ‘program’ Ports B and C as ‘output ports’. Port A has a keyboard connected to it. This means that Port A is to be programmed as an input port – then, the keyboard can send data to it, which can be received by the 8086 when it chooses to. Thus, with this simple setup, we see that the 8086 has been ‘interfaced’ to three I/O devices using the 8255 as a temporary storage space. Thus, the 8255 has three sets of ‘8-bit parallel ports’, which act as intermediary between a processor and a number of I/O devices. Note that in all cases the I/O devices used have only 8-bit data capability – so, only the lower data lines D₀-D₇ are connected to the 8255.

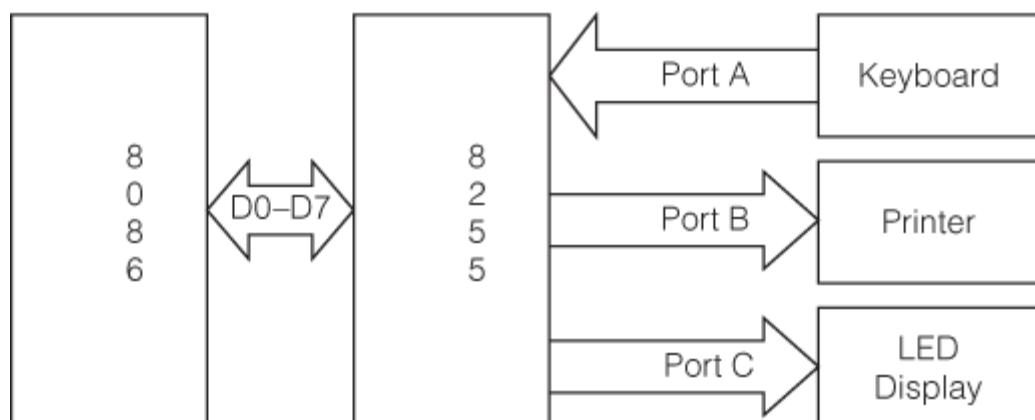


Figure 9.2 | The connections between an 8086, 8255 and three peripherals

However, it is a ‘programmable chip’ as are all the chips we will learn in the next two chapters as well. Keep in mind that these ‘interfacing chips’ do not have any ‘processing’ capability. They can only be made to act in the way we want to by ‘programming’ it using the instructions of 8086 (i.e., the processor to which it is connected). However, they have the hardware to do the special functions that are needed for each specific purpose. Programming interfacing chips entails writing of ‘control words’ in their control/command registers. We will learn this idea, starting with the 8255.

9.2.1 | Pin Configuration and Internal Block Diagram

Figure 9.3 shows the pin configuration of the DIP (dual-in-line) version of the chip. We see that it has 40 pins, which consist of three 8-bit ports named Port A (PA), Port B (PB) and Port C (PC), each of which can be programmed as input or output ports. Figure 9.4 shows the internal block diagram of the chip. It is seen to consist of various functional blocks, and let us take a quick look at each of the blocks.

1 PA3		PA4 40	
2 PA2		PA5 39	
3 PA1		PA6 38	
4 PA0		PA7 37	
5 \bar{RD}		\bar{WR} 36	
6 \bar{CS}		RESET 35	
7 GND		D0 34	
8 A_1	8	D1 33	
9 A_0	2	D2 32	
10 PC7	5	D3 31	
11 PC6	5	D4 30	
12 PC5		D5 29	
13 PC4		D6 28	
14 PC0		D7 27	
15 PC1		V_{CC} 26	
16 PC2		PB7 25	
17 PC3		PB6 24	
18 PB0		PB5 23	
19 PB1		PB4 22	
20 PB2		PB3 21	

Figure 9.3 | Pin diagram of 8255

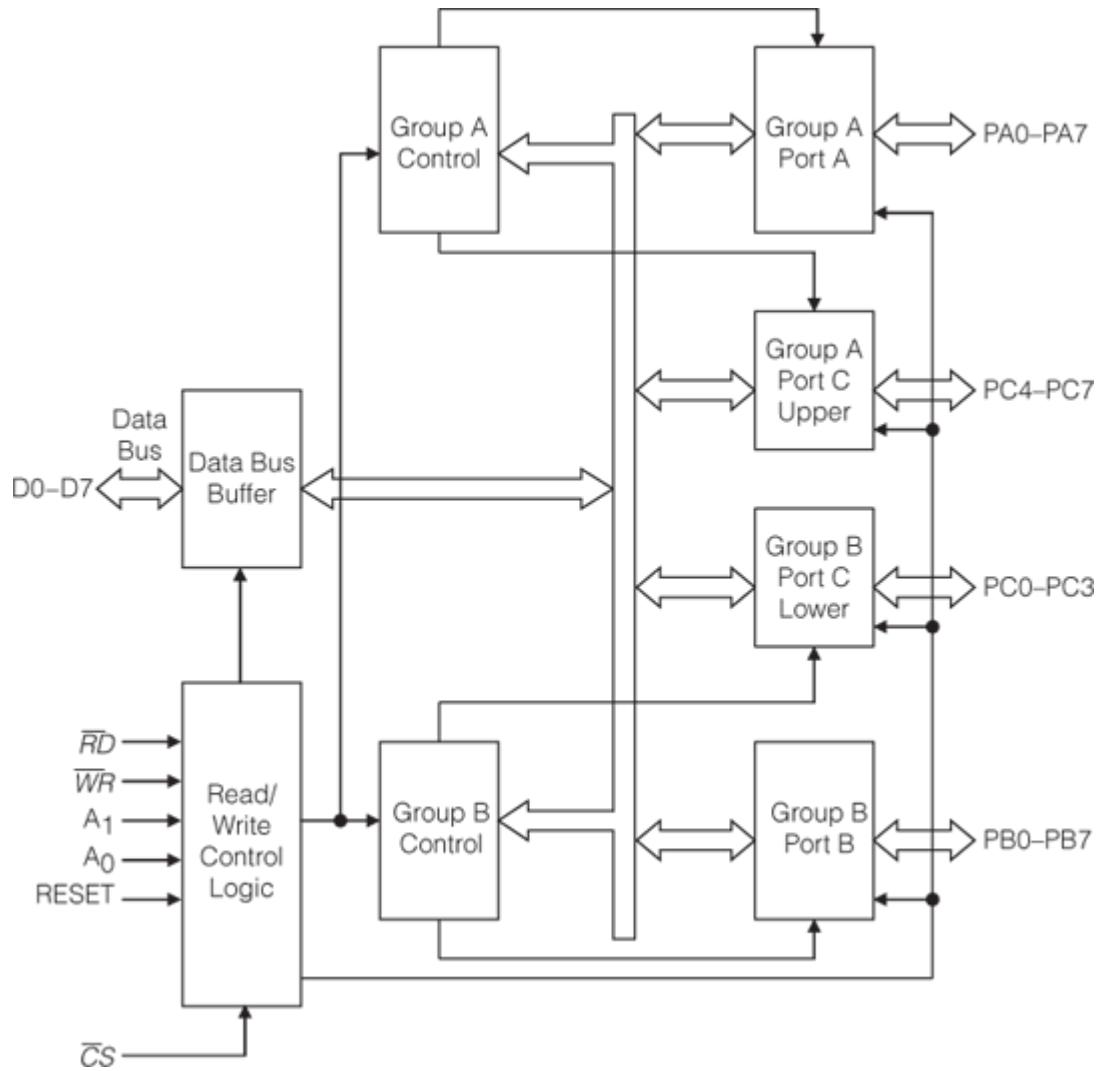


Figure 9.4 | Internal block diagram of the 8255

Data Bus Buffer

There is a three state bi-directional 8-bit buffer which is used to interface the chip to the data bus of the system. Upon execution of the processor's input or output instructions, data and control/status words are received or transmitted by the buffer.

Read/Write Control Logic

It manages all data transfer between the chip and the processor on accepting control signals from the control and address buses of the system.

Group A and B Controls

Functionally, this chip has been divided as Group A and B and they have their corresponding controls. We will soon see what this grouping is

intended for. The way the chip is to act is decided by a register called the control/status register, which can be written to and read from. Now see a typical connection between the 8086 and an 8255 in Fig. 9.5. The lower data lines of the chip can be connected to 8 bits of the data bus of the 8086. The *RD* and *WR* are connected to the *IORD* and *IOWR* generated from the 8086. If this chip (which is viewed as an I/O port by the processor) is to have only an eight-bit address, only the lower 8 lines of the address lines of the processor need to be involved in the address decoding process, which causes the *CS* line to be activated. Otherwise, 16 address lines may be used. (Recollect that no I/O port can have an address size greater than 16 bits.) Any two address pins of the processor are to be connected to the pins A_0 and A_1 of the 8255. This is because there are four separate entities associated with this chip, and each one needs a unique address. This is achieved by the four possible combinations of these two lines, as shown in Table 9.1.

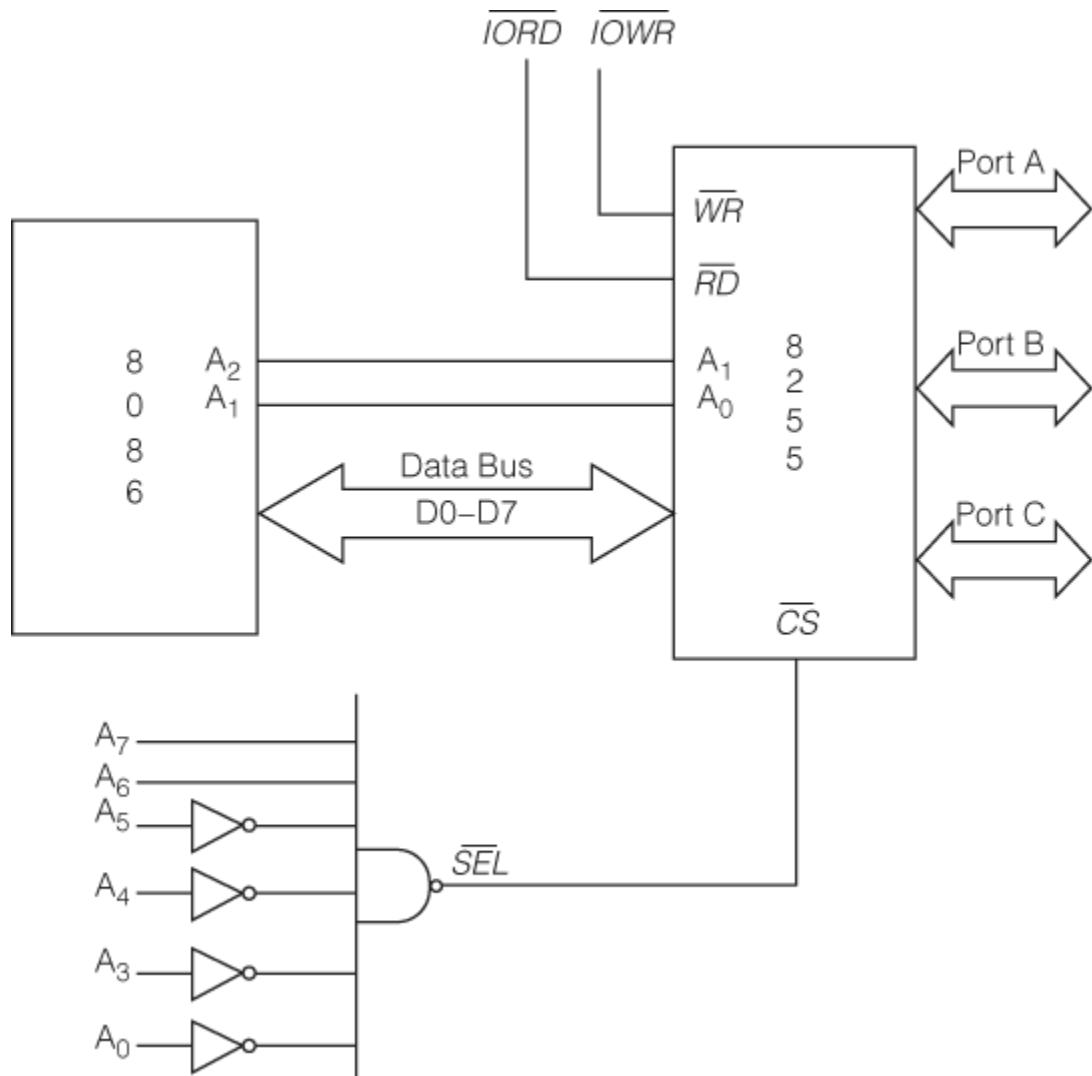


Figure 9.5 | Address decoding and connections between the 8086 and the 8255

Table 9.1 | Port Selection for the 8255 Using Pins A_o and A₁

CS	A ₁	A ₀	Selected entity
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control register

Example 9.1

Find the addresses of Port A, Port B, Port C and the control/status register of the 8255 whose address decoding circuitry is as shown in Fig. 9.5.

Solution

The NAND gate output is 0, when the address lines A₇ to A₃ and A_o have the following bit status, and that is when the chip gets selected/enabled.

A ₇	A ₆	A ₅	A ₄	A ₃	A ₀
1	1	0	0	0	0

When A₂ and A₁ are 00, then A₇ ... A₀ is 1100 0000, i.e., CoH. This is the address of Port A. Similarly (refer Table 9.1), the other ports have the address as shown in Table 9.2.

Table 9.2 | Address of the Ports of 8255 for the Connection in Fig. 9.5

Entity	Address (Hex)
Control register	C6
Port A	C0
Port B	C2
Port C	C4

9.2.2 | Programming the PPI

How is the chip programmed and what are the options available?

Programming the chip involves only the writing of a particular word to the control register. The control register is an 8-bit register which can be written into. The bits of this word (called the control word) will decide

the way the ports of the chip are to be configured. To understand this, let us have a look at the control word format (refer Fig. 9.6).

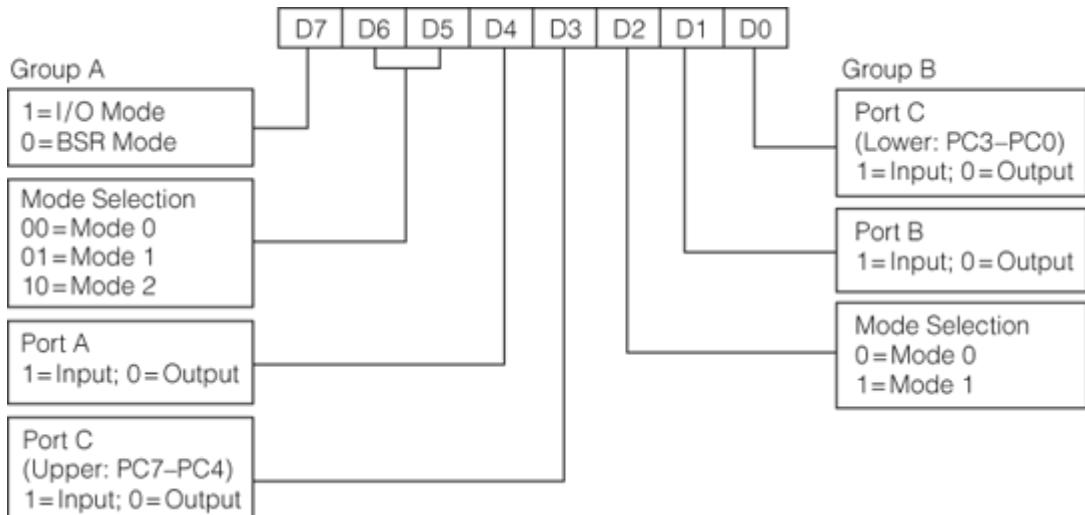


Figure 9.6 | Control word format of 8255

As seen in this and also in the block diagram, the ports A, B and C are grouped into two – Groups A and B. Group A consists of Port A and upper 4 bits (PC4–PC7) of Port C. Group B then obviously includes Port B and Port C lower (PC0–PC3). The division into groups give us only one piece of information – it is that Group A ports can have three operational modes (0, 1 and 2), but Group B ports have only two modes of operation.

Bit Assignment of the Control Word

First, look at D7 – it suggests the options of I/O mode or BSR mode. To begin with, let us use the chip in the I/O mode. This is the normal mode of operation (BSR mode is a special mode catering to Port C alone, which we will deal with later). The next thing to do in this is to decide the mode of operation of the two groups. It is acceptable for the two groups to be in different or same modes. After this is done, select the option of a particular port being either an input or output port. For i/p, the corresponding bit is to be 1, and for output, it is to be 0.

9.3 | Modes of Operation

The 8255 can be made to work in three modes as decided by the system designer. They are:

Mode 0: Basic input/output

Mode 1: Strobed input/output

Mode 2: Bi-directional bus

Mode 0

This is the simplest and most widely used mode. In this mode, the two 8-bit ports A and B and the 4-bit ports Port C upper and Port C lower may be used independently. Here, data is simply taken in from an input port or given to an output port.

Mode 1

This is the ‘handshaking’ mode. Handshake implies data transfer in which the communicating devices exchange request and acknowledge control signals with each other. For an 8255 operating in this mode, Ports A and B pins are used for data transfer, while 4 bits of each of Port C are used for generating the handshaking signals for each of the 8-bit ports.

Mode 2

This is the bidirectional mode. Only Group A can use this mode. Here, Port A is used for transmitting as well as receiving data. Handshaking signals generated by the upper 4 bits of Port C maintain bus discipline for proper flow of data in the required direction. Now, let us use the different modes.

9.4 | Mode 0

The specifications of this mode are:

1. Two 8-bit ports and two 4-bit ports.
2. Any port can be input or output.
3. Outputs are latched (ref Section 7.4.1 for the reason for this).
4. Inputs are not latched.
5. 16 different input/output combinations possible.

Example 9.2

Design the control word to configure the ports of an 8255 chip in mode 0, with Port B and Port C upper (PC_u) as inputs and Port A and Port C lower (PC_l) as outputs.

Solution

Refer Fig. 9.6.

Since we are using the I/O mode (rather than the BSR mode), $D_7 = 1$

Both groups are to be in mode 0. Hence, $D_6 D_5 = 00$ and $D_2 = 0$

For i/p, the corresponding bit is to be 1, and for output, it is to be 0.

Since Port A is to be an output port, $D_4 = 0$

Since Port C_U is to be an input port, $D_3 = 1$

Since Port B is to be an input port, $D_1 = 1$

Since Port C_L is to be an output port, $D_0 = 0$

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	1	0	1	0

The control word is thus 8AH.

Example 9.3

It is necessary to read the setting of 12 switches which are connected to the 8255 and display it on LEDs. Draw the setup and write the program for the same.

Solution

We can use the 8255 with the addresses as in Table 9.2 and the control word as in Example 9.2.

Figure 9.7 shows how eight switches are connected to Port B and Port C_U. When a switch is closed, a '0' is read in, and when it is open, a '1' is read into the port lines. To the pins of Port A and Port C_L, connect 12 LEDs along with their current limiting resistors. The LEDs glow when the corresponding pins have a high logic on them.

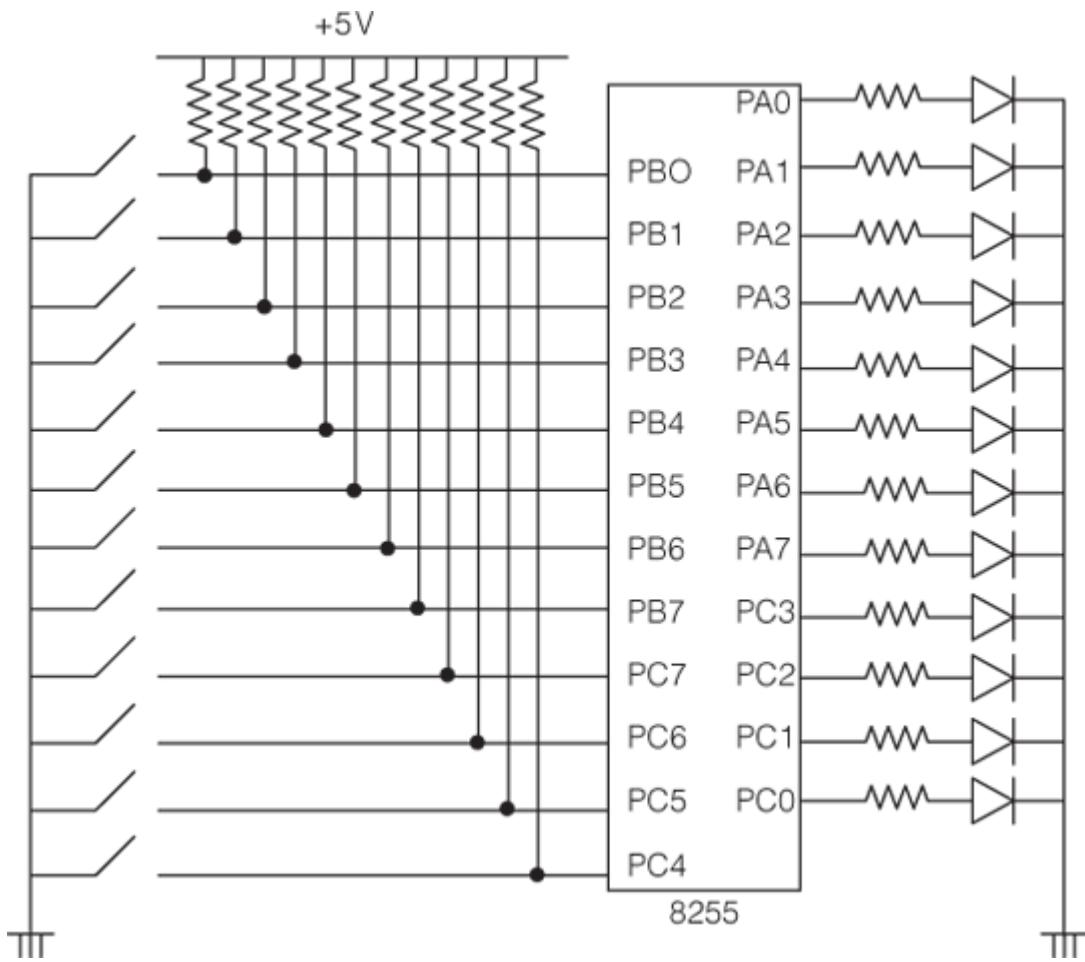


Figure 9.7 | Connection of switches and LEDs to an 8255

Now, let us discuss the steps in the program that will be run:

1. The ports and the control register addresses have been given labels and will be specified using the EQU directive. Since the port addresses are only 8 bits long, fixed port addressing can be used.
2. The switch settings at the input Port B can be read in directly and transferred to the output Port A.
3. The data at Port C_u can be read in by inputting Port C fully, but the switch settings will be available only at the upper 4 bits position and have to be shifted to the position of the lower 4 bits to output it to the LEDs.

The program is as follows:

```

CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A
PB EQU 0C2H      ;address of Port B
PC EQU 0C4H      ;address of Port C

MOV AL, 8AH       ;move control word to AL
OUT CR, AL        ;send it to the control register
IN AL, PB         ;get switch status from i/p port B
OUT PA, AL        ;output it to the LEDs at o/p port A
IN AL, PC         ;take in switch settings of Port C
MOV CL, 04         ;CL = 4
ROR AL, CL        ;rotate AL 4 times to the right
OUT PC, AL        ;the data is in the lower part of PC
                  ;output it to the LEDs there
END               ;end of the assembly file

```

Note In the programs on I/O interfacing, we will not use BIOS or DOS interrupts, because they are specific to PCs only. We are using trainer kit to run these programs. We will not specify any memory models also – that can be decided by the type of software supported by the trainer. Only the program lines will be written here.

9.4.1 | Interfacing 16 Bits I/O Ports to the 8255

All the above is for a port to which only 8-bit data is to be transferred from D₀ to D₇ of the data bus of the processor. Since our processor (8086) has a data bus width of 16 bits, it is quite possible that it may sometimes need to send/receive 16 bits through a parallel port when the port has a data bus width of 16 bits. This will necessitate the need for two chips of the PPI (see Fig. 9.8).

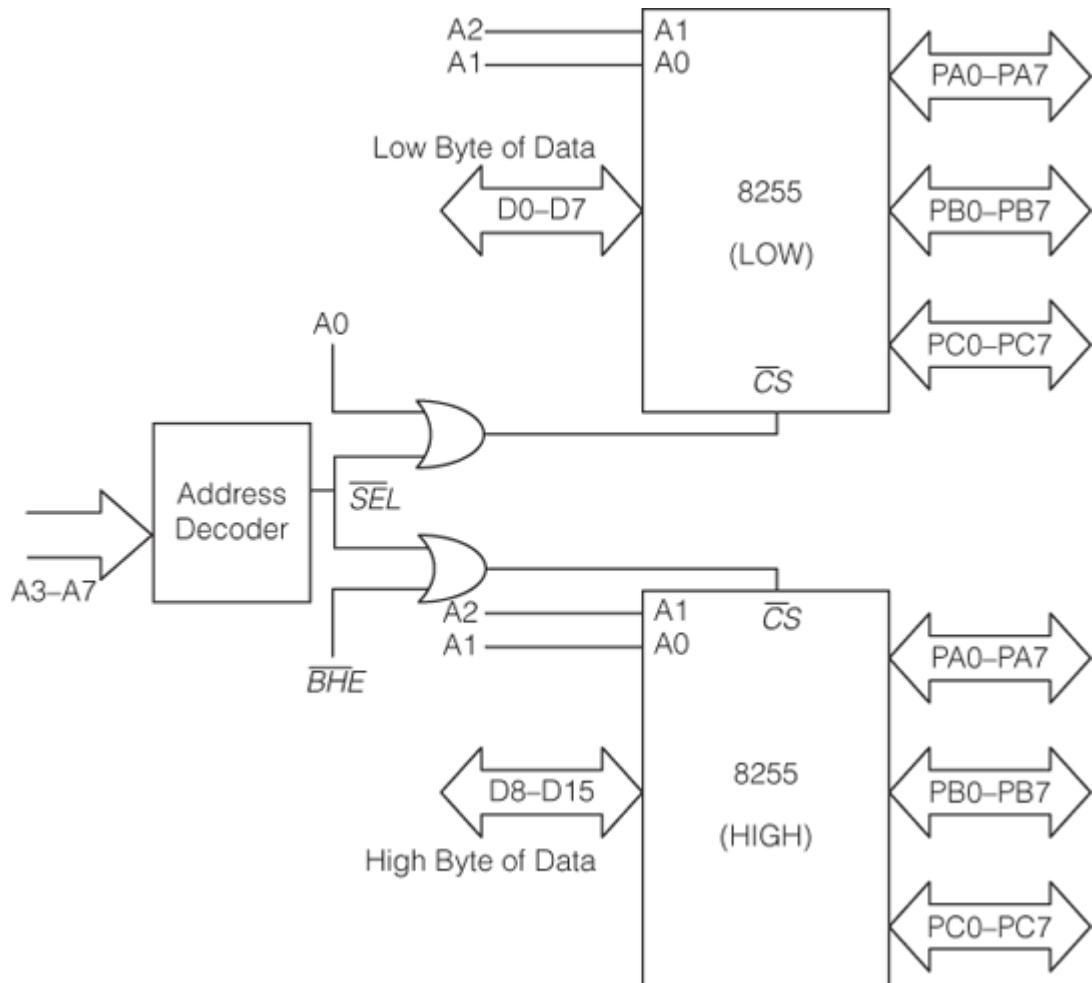


Figure 9.8 | 16-bit port interfacing using two PPIs

Recollect the discussion on memory banks and I/O banks. 16-bit data is to be arranged into two 8-bit banks, an upper bank and a lower bank. When data is to be accessed through the upper data lines, the processor makes arrangements to lower the *BHE* line. This is used to decode the address of the upper bank, as shown in the figure. Thus, to transfer a 16-bit data, both the PPIs will get enabled. With the same decoding logic as in Example 9.2, the addresses of the ports of each of the PPIs may be enumerated and is left for the interested reader. All the I/O devices we will discuss further are ones with only 8-bit data bus, and we will use Fig. 9.5 as our reference.

Example 9.4

Generate a square wave of 33% duty cycle at the lowest bit of Port A. The clock frequency of the processor is 4.77 MHz.

Solution

A 33% duty cycle means that the square wave should be high for one third of the cycle. Let us consider the ON time to be 1 msec. Then, the OFF period will be 2 msecs.

We can solve the problem this way – create a delay of 1 msecs and call the delay procedure once during the ON time and twice for the OFF time.

For delay calculations, refer to Example 6.2. The delay loop is:

```
MOV CX, N  
HERE:    NOP  
        LOOP HERE
```

Since the clock frequency is 4.77 MHz, one clock period = 0.21 μ secs

The delay is to be 1msec

Total delay time = 1 msec = $20 N \times 0.21 \mu$ secs

For 1 msec delay, the value of N = $[1 \text{ msec}/(20 \times 0.21 \mu\text{secs})] = 238$ or EEH

The complete program is as shown below. Note that we need the square wave only on the lowest pin of Port A, but it is available at all pins. This is because the pins of Port A are not ‘bit addressable’; i.e., we cannot manipulate Port A bits one at a time. For this case, it is okay to have the square wave generated from all the pins – but we need to use only one pin to observe the square wave. Connect a CRO probe to one pin and observe the waveform on the CRO. Also, for this application, Port A should be made an output port. All other ports are ‘don’t care’. The control word is written in mode 0, and bit D₄ = 0 specifically; i.e., 1000 0000 is the control word.

```

PA EQU 0C0H      ;address of Port A
CR EQU 0C6H      ;address of control register
MOV AL, 80H      ;control word for Port A to be output
OUT CR, AL       ;send it to control register
BACK: MOV AL, 0    ;AL = 0
        OUT PA, AL   ;send it to Port A
        CALL DELAY    ;call a delay for one msec
        CALL DELAY    ;call a delay for one msec
        MOV AL, 0FFH   ;AL = FFH
        OUT PA, AL   ;send it to Port A
        CALL DELAY    ;call a delay of 1 msec
        JMP BACK      ;repeat
DELAY PROC NEAR   ;delay program
        MOV CX, 0EEH   ;N = EEH
HERE: NOP
        LOOP HERE
        RET
        ENDP
        END

```

9.4.2 | Bit Set Reset Mode

It will be more useful to discuss the BSR mode before we go on to the discussion of modes 1 and 2. This is a special mode and is applicable only for the bits of Port C. In the control word format, if the MSB is made 0 ($D_7 = 0$), the bit set/reset (BSR) mode takes effect. In this mode, any bit of Port C can be set or reset by specifying the bit which has to be set or reset. However, at a time, only one bit can be addressed – and that bit is to be either set or reset. The corresponding control word has to be decided **and moved to the control register**. This is because even though we are manipulating the bits of Port C, it is a ‘control word’ that is being written. See the format of the BSR control word in Fig. 9.9. D_3 to D_1 specifies the bit to be set/reset.

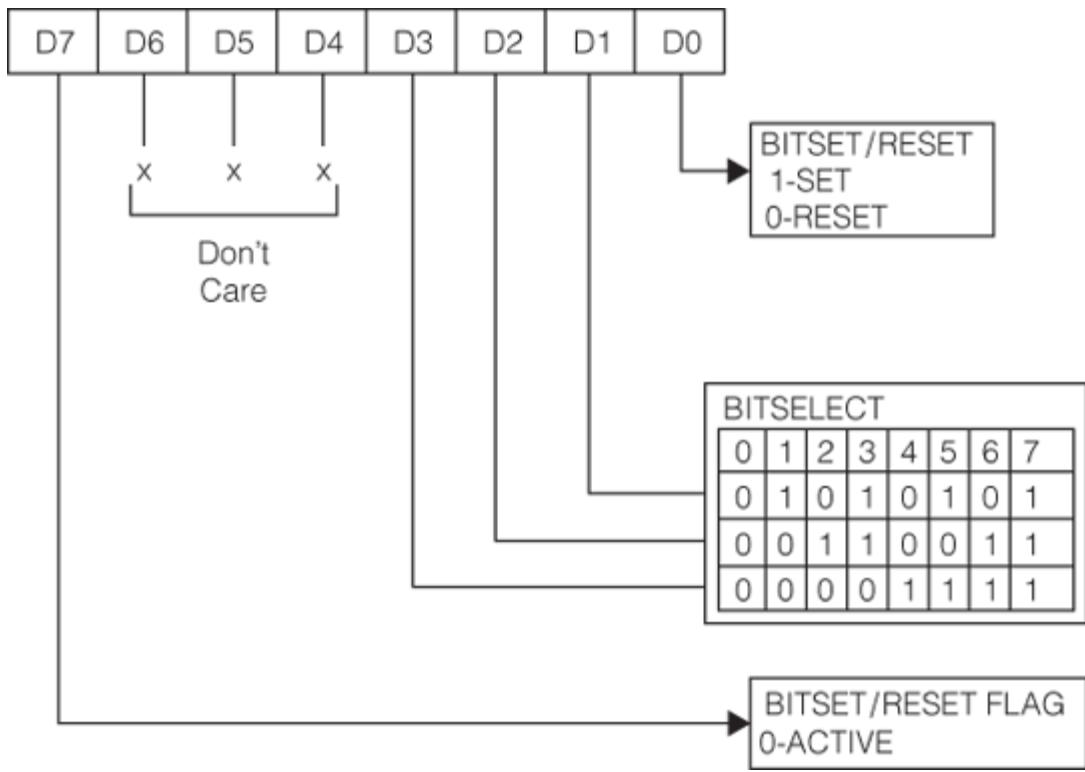


Figure 9.9 | Format of the BSR control word

Example 9.5

Write the BSR control words for the following cases:

1. PC_0 to be set
2. PC_7 to be reset
3. PC_1 to be set

Solution

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
i) PC_0 to be set	0	0	0	0	0	0	0	1
ii) PC_7 to be reset	0	0	0	0	1	1	1	0
iii) PC_1 to be set	0	0	0	0	0	0	1	1

Example 9.6

Write a program to do the following. Get data from input Port A. If the data is above 7FH, send a high value on PC_7 . Otherwise, make PC_7 to be zero. Use the port addresses in Example 9.1.

Solution

The first step in solving this problem is to configure Port A to be an input port and Port C_{upper} to be an output port (since a signal has to be generated on it) in mode 0.

The point is that though we might be using a BSR mode, we must first ‘configure’ the ports as input/output.

The control word format for this is 1 0 0 1 0 0 0 0 = 90H

The BSR control word for resetting PC₇ is 0 0 0 0 1 1 1 0 = 0EH

The BSR control word for setting PC₇ is 0 0 0 0 1 1 1 1 = 0FH

The program is

```
CR EQU 0C6H      ;address of control register
PA EQU 0C0H      ;address of Port A
PC EQU 0C4H      ;address of Port C

MOV AL, 90H      ;control word
OUT CR, AL       ;send it to the control register
IN AL, PA        ;take in data through Port A
CMP AL, 7FH      ;compare it with 7FH
JA LOW           ;if AL > 7FH, go to LOW
MOV AL, 0EH      ;AL < 7FH, write a word for PC7, = 0
OUT CR, AL       ;send it to the control register
JMP XIT          ;go to exit

LOW:  MOV AL, 0FH  ;AL > 7F, BSR word for PC7, = 1 in AL
      OUT CR, AL  ;send it to the control register

XIT:
      END          ;end the program
```

Thus, we see that with the BSR mode, we are able to do ‘bit addressing’ for port pins, but only for Port C, in the output mode alone.

9.5 | Mode 1

This is also called the handshaking mode. The features of this mode are that Ports A and B can be used for data input or output, and the bits of Port C are used as handshaking signals to control these data transfers. This mode is frequently referred to as strobed input/output mode also. The word ‘strobe’ is indicative of signaling – classically, by the use of light.

Recollect the grouping into groups A and B.

1. Each group contains one 8-bit port and one 4-bit port. The 4-bit port is used for 'handshaking'.
2. The 8-bit port can be either an input or an output.
3. Input and output are latched.
4. The 4-bit port is used for control and status of the 8-bit data port.
5. Interrupt logic is supported.

To understand it completely, we need to separate the cases of input and output.

9.5.1 | Strobed Input Mode

Figure 9.10 shows the signal definitions of Port C when Ports A and/or B are used as inputs. The definitions of the handshaking signals are as given:

1. **STB (Strobe):** The peripheral sends a data byte to the 8255 and indicates this by lowering the strobe line. This active low signal is given by an input device indicating that it has sent a byte of data to the 8255, which will be latched therein.
2. **IBF (Input Buffer Full):** This is the signal the 8255 sends as acknowledgement to indicate that the input latch has received the byte sent by the peripheral. This is reset when the 8086 reads the data.
3. **INTR (Interrupt Request):** A high on this is used to interrupt the CPU. This is set to high if **STB** is high, IBF is high and the INTE (interrupt enable) FF is set. It is reset by the falling edge of **RD** as shown in Fig. 9.11. For enabling the 'INTE' flip flop, use PC_4 (for $INTE_A$) and PC_2 ($INTE_B$). The setting /resetting of these FFs is done by the BSR mode.
Note PC_4 and PC_2 are used as strobe signals also, but remember that they are used in the BSR mode (for INTE). There is no effect of the BSR control word on the activities in Mode 1. Once the interrupt enable flip flops are set/reset, these pins are used as strobe signal pins.
4. In Mode 1 input mode, two pins PC_6 and PC_7 are not used for handshaking. They may be used as input or output lines as desired.

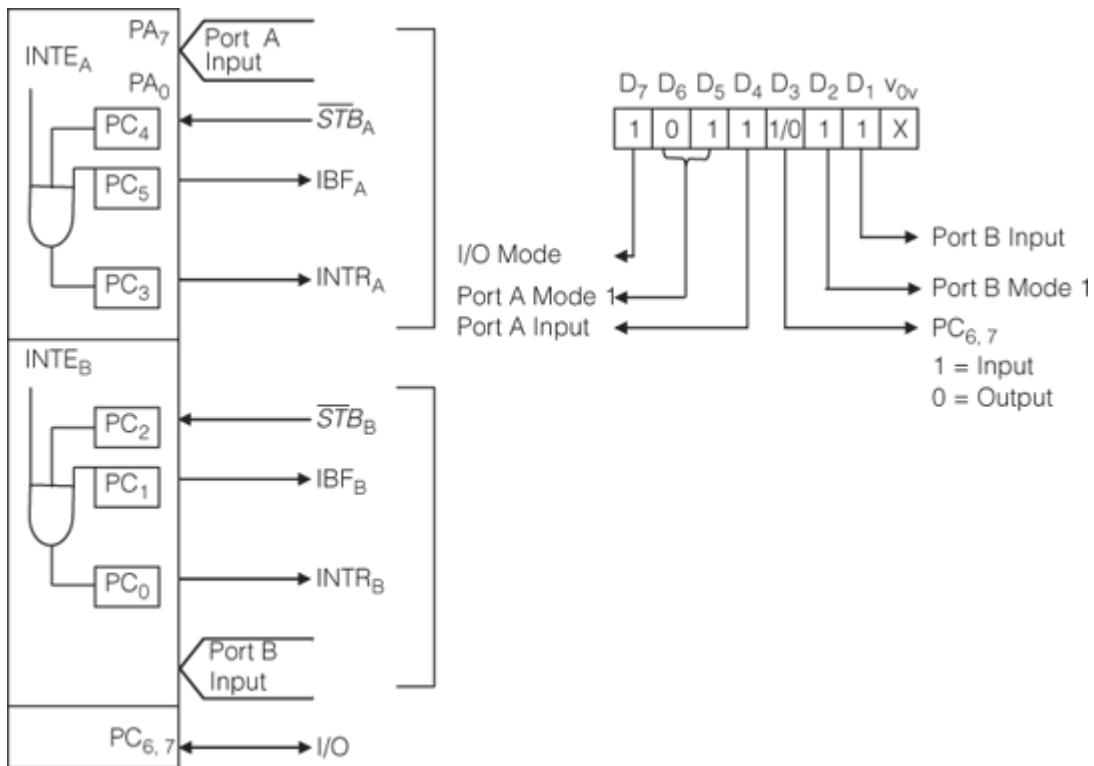


Figure 9.10 | Mode 1: Strobed input – Pin configuration and control word format

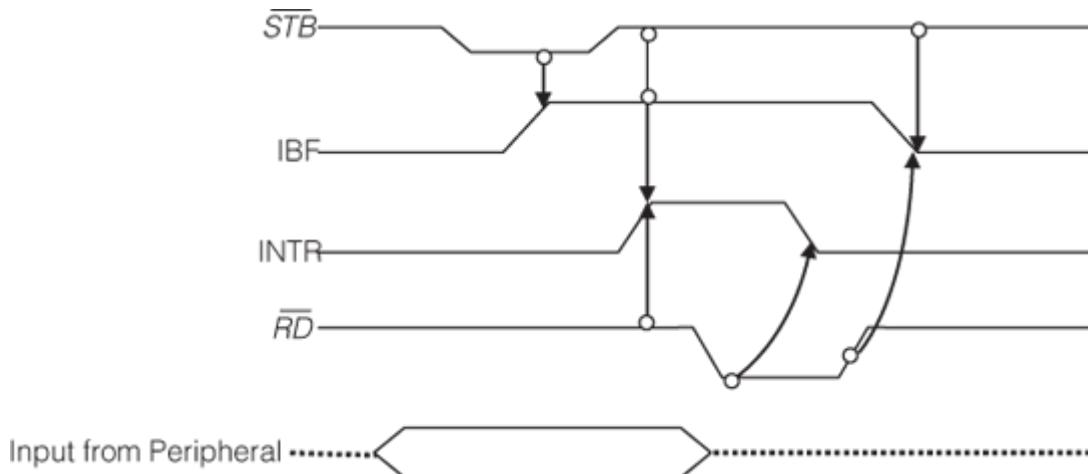


Figure 9.11 | Timing waveforms for strobed input mode

9.5.2 | Mode 1: Strobed Output

In a similar way, Mode 1 output uses the pins of Port C, as shown in Fig. 9.12. The timing diagram for data transfer is as in Fig. 9.13. Let us discuss the definitions of the handshaking signals here.

1. **OBF (Output Buffer Full):** When the processor writes a data byte to the output latch of the 8255, this signal goes low. It indicates that this data can be read into the peripheral. It goes high when the peripheral takes the data and acknowledges it.

2. **ACK**: This is the signal given by a peripheral when it accepts the data from 8255A. A ‘low’ on this input informs the 8255 that the data from the port has been accepted. In essence, a response from the peripheral device indicating that it has received the data output by the CPU.
3. **INTR (Interrupt Request)**: This signal goes high after the acknowledge signal is removed. This can be used to prompt the processor to deliver the next data byte into the output latch of the 8255. The setting of this signal requires that **OBF**, **ACK** and **INTE** (interrupt enable Flip flop) are all high.
4. **INTE (Interrupt Enable)**: This is an internal flip which can be set/reset by the BSR mode of port C on PC_6 and PC_2 respectively. This is an internal flip flop used to enable or disable the generation of the INTR signal. $INTE_A$ is controlled by the set/reset of PC_6 . $INTE_B$ is controlled by the set/reset of PC_2 .
5. In Mode 1 output mode, two pins PC_4 and PC_5 are not used as handshaking signals. They may be used as input or output lines as desired.

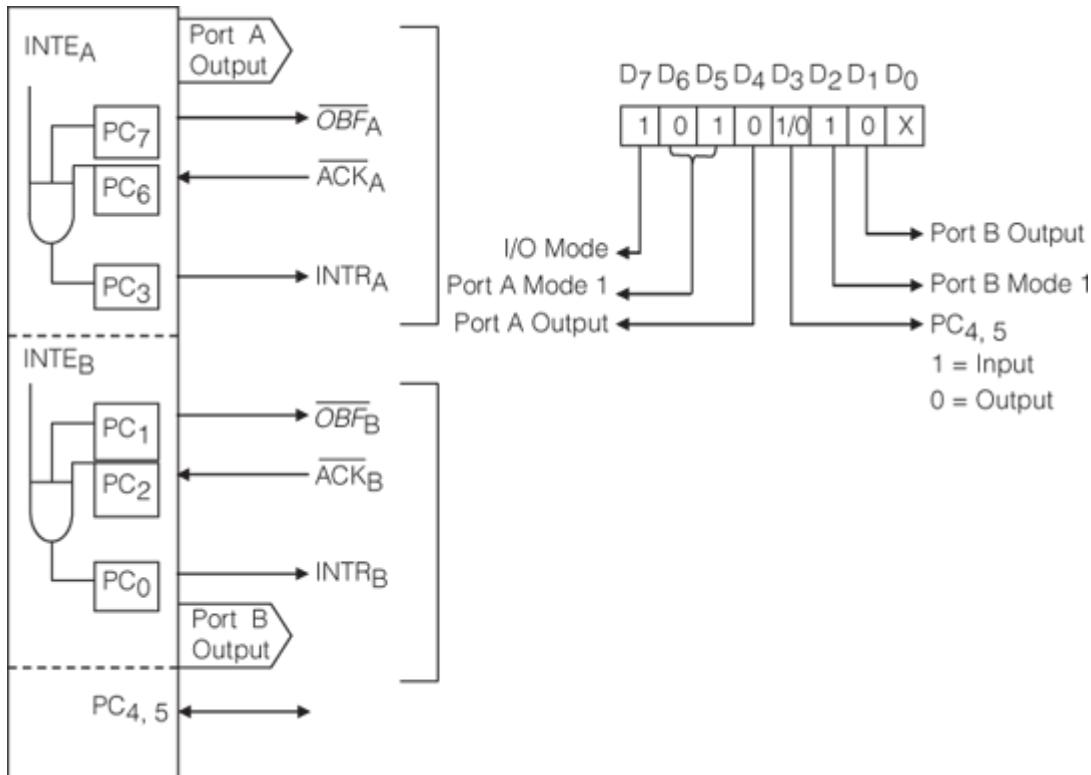


Figure 9.12 | Mode 1: Strobed output – Pin configuration and control word format

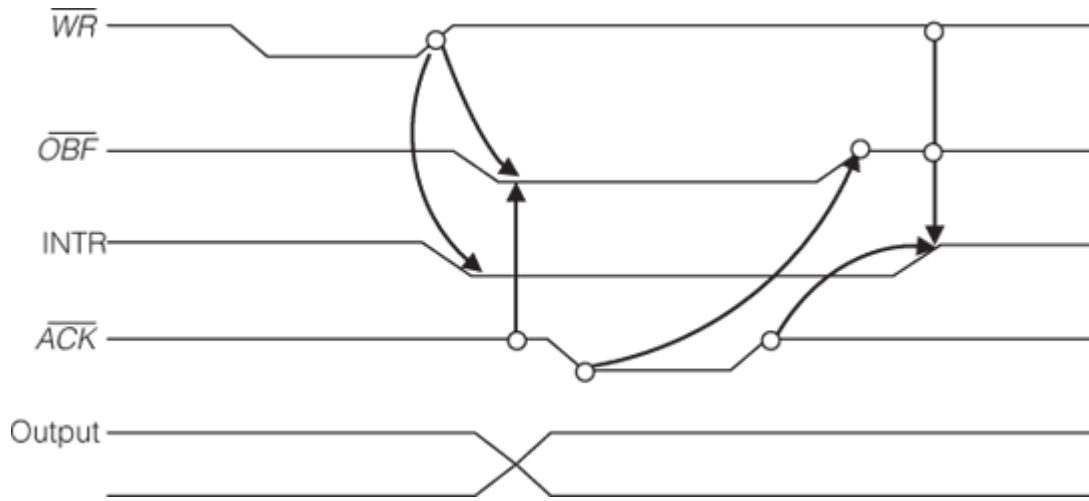


Figure 9.13 | Timing waveform for strobed output mode

Status Word

Sometimes, it may be necessary to know the current status of the handshaking signals and take decisions within the program accordingly. Since the handshaking signals are on Port C lines, it will be sufficient to read in the status of Port C, which will have the same signal definitions as the port pins, and the ‘status word’ is shown in Fig. 9.14. This status word is read by using the instruction IN AL, PORT C. Thus, we get the status of Port C pins in AL, which can be used to take decisions. It is also possible to use the PPI in such a way in Mode 1 that one port is in strobed output and the other is in strobed input mode. The two possibilities are shown here in Figs 9.15 and 9.16. Now we will examine a case in which Port A is in strobed input mode connected to a keyboard and Port B is connected to a printer in the strobed output mode (as in Fig. 9.15).

INPUT CONFIGURATION								
D7	D6	D5	D4	D3	D2	D1	D0	
I/O	I/O	IBF _A	INTE _A	INTR _A	INTE _B	IBF _B	INTR _B	
GROUP A					GROUP B			

Figure 9.14 | Status of Port C for input and output configurations of Mode 1

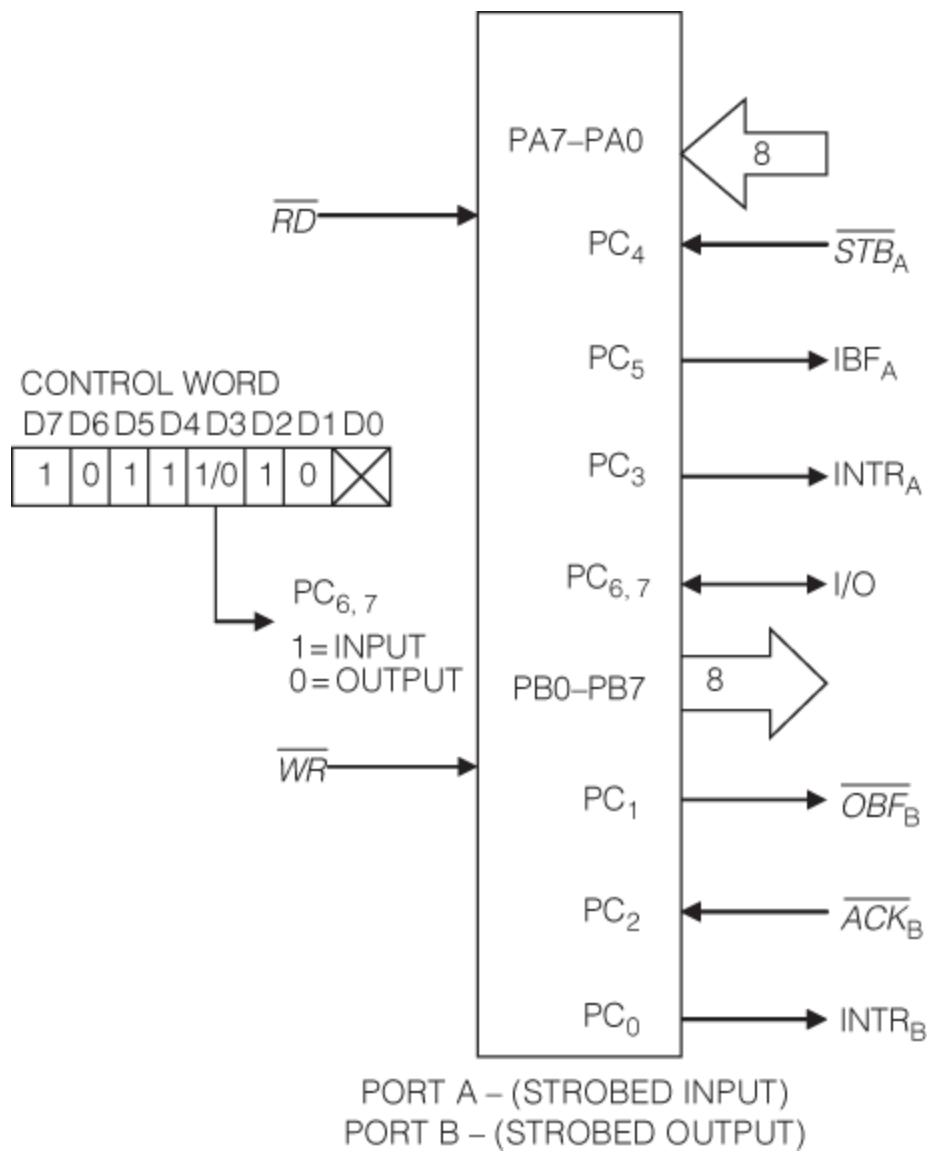


Figure 9.15 | Port A in strobed input and Port B in strobed output mode

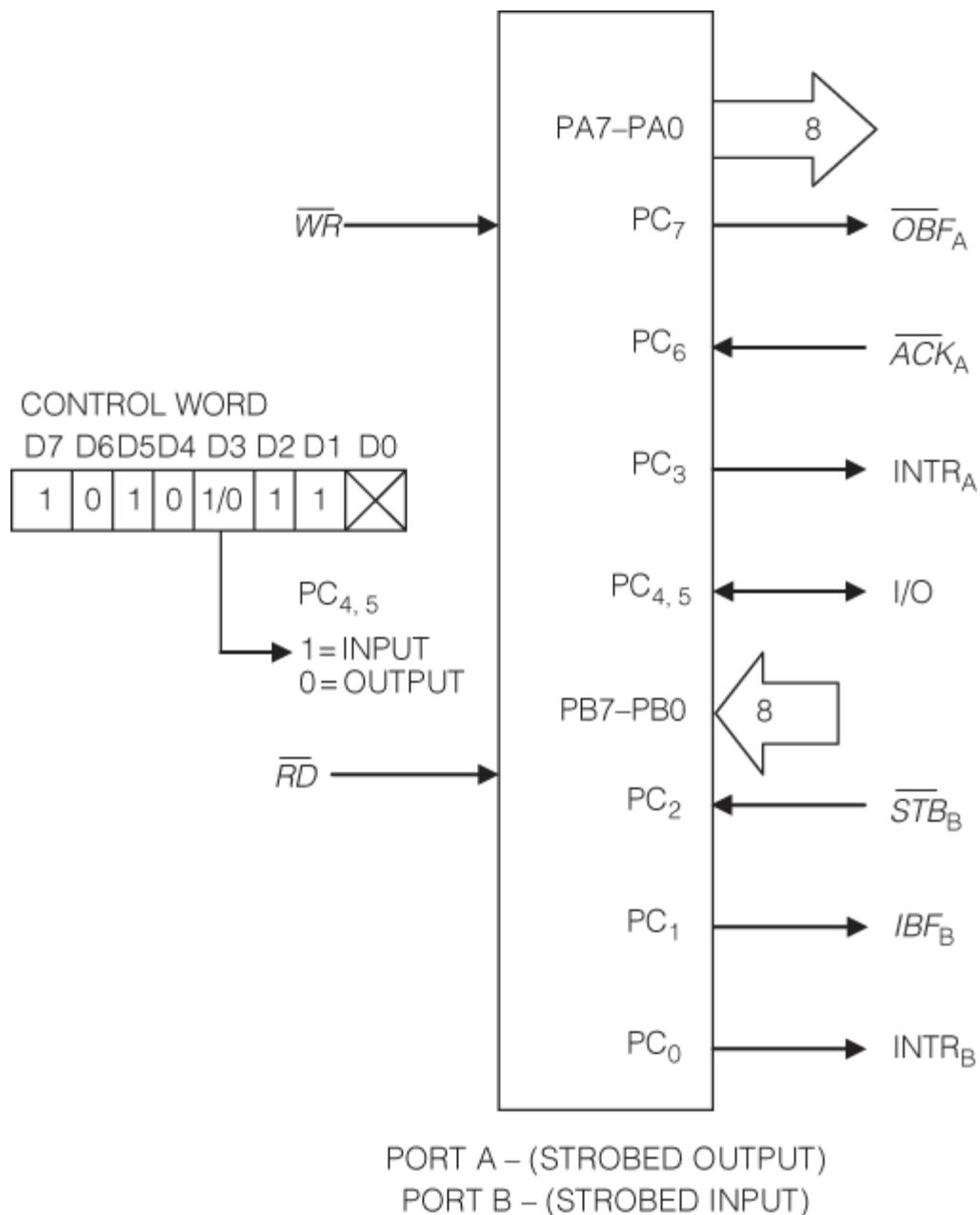


Figure 9.16 | Port A in strobed output and Port B in strobed input mode

Example 9.7

1. Write the control word for setting Port A as an input port and Port B as an output port.
2. Port C upper and Port C lower are used as control signals.
3. Write instructions to find the status of the signal IBF_A and OBF_B .

Solution

Refer to Fig. 9.4 or 9.15 for designing the control word.

The control word is thus

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	1	1	0	1	0	0

i.e., B4H

- In this application, Group A pins corresponds to an input configuration, and Group B pins corresponds to an output. Hence, the pins of Port C will have the status as designated (refer Fig. 9.14).

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
I/O	I/O	IBF _A	INTE _A	INTR _A	INTE _B	\overline{OBF}_B	INTR _B

Thus, the status of Port C will be as shown.

What is needed is to read in Port C and test the condition of D₅ for knowing the status of IBF_A. Test the condition of D₁ for the status of OBF_B.

```
IN AL, PORTC      ;read in the contents of Port C
TEST AL, 5        ;for testing the condition of D5, IBFA
TEST AL, 1        ;for testing the condition of D1, OBFB
```

Example 9.8

Consider a case in which the 8255 is used with Port A in the strobed input and Port B in the strobed output mode. The chip is connected to a keyboard as well as a printer as shown in Fig. 9.17. Write instructions to get data from the keyboard and send it to the printer to be printed.

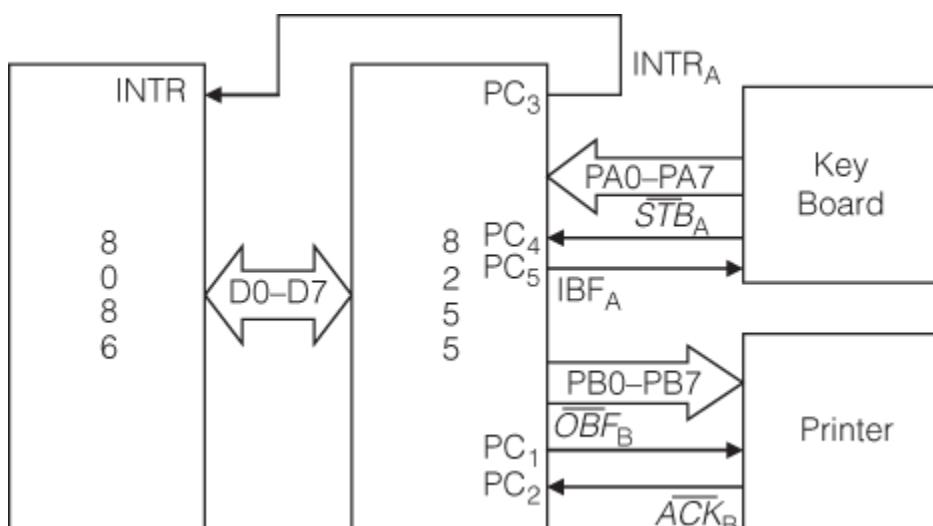


Figure 9.17 | Connecting a keyboard and a printer to the 8086 in Mode 1

Solution

The scenario is that when a key is pressed, the corresponding ASCII data is transferred to the 8086 using Mode 1 handshaking, and it generates an interrupt. This interrupt calls the print ISR, which is responsible for sending this data to the printer. The problem has two parts.

1. Port A is used as the input port to get data from the keyboard. The keyboard sends a strobe STB_A signal to the 8255 on its PC_4 pin, along with sending the ASCII code corresponding to the key pressed. This is read into the microprocessor only after it is confirmed that the signal IBF_A (Input buffer full) has been sent by the 8255 on Pin PC_5 . The status of all the pins of Port C can be read by ‘reading in’ Port C. Once PC_5 is confirmed to be high, the data that has been latched into Port A of the 8255 can be read into the processor. Now, see the timing diagram of Fig. 9.11. When STB_A signal goes high and IBF_A is high, $INTR_A$ (PC_3) is raised. To ensure this, the corresponding $INTE_A$ flip flop must be set. This is done by setting bit PC_4 using the BSR facility. In Fig. 9.17, the $INTR_A$ signal from pin PC_3 of 8255 is connected to the $INTR$ pin of the 8086. When the 8086 is interrupted, it runs the ISR ‘PRINT’, which sends the character that was read in from the keyboard.

2. In the second part, the PRINT ISR (Interrupt Service Routine) is executed.
It does the following:

Port C status is read to confirm that OBF_B is high. The processor considers the printer to be ready to accept data only when this signal is high. Only then will data be outputted to Port B. The status of this signal is obtained in D_1 bit of Port C. If it is high, the data which was brought from the keyboard and placed in memory is transferred to Port B using an OUT instruction. When this is done, the OBF_B signal will go low and the printer should respond to it with a ACK_B signal, which will cause OBF_B to go high again. (This will allow the system to send the next byte to be printed, when interrupted again.) The generation of the signals OBF_B will be done automatically by the 8255 in Mode 1 output once the WR goes low corresponding to a write operation by the processor.

Steps in the Program

1. Write the control word in Mode 1 with Port A as input and Port B as output.
2. Set the interrupt enable FF ($INTE_A$) by setting bit PC_4 .
3. Set the interrupt flag of the 8086 processor.
4. Read in the status of Port C.
5. Test the status of IBF_A .
6. When it is high, transfer data byte from Port A to AL of the processor.
7. Transfer the byte to a memory location.

- Since the interrupt enable flag of the 8255 has been set, the INTR_A will go high, and since the Interrupt flag of the 8086 is set, the 8086 gets interrupted and gets directed to the PRINT ISR.

Steps in the Print ISR

- Read in the contents of Port C.
- Test the OBF_B pin.
- If it is high, send the data to the printer.

```

CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A
PB EQU 0C2H      ;address of Port B
PC EQU 0C4H      ;address of Port C

LOC DB ?          ;space to store key data

MOV AL, 0B4H      ;mode 1, Port A input, Port B output
OUT CR, AL        ;send it to control register
MOV AL, 09         ;BSR word for setting PC4
OUT CR, AL        ;set bit PC4 for setting INTA
STI               ;set the interrupt flag of 8086
IN AL, PC         ;read in Port C
REPEA: TEST AL, 20H ;test bit D20H (IBFA)
JZ REPEA          ;if it is 0, keep testing
IN AL, PA         ;if IBFA is high, send Port A data to AL
MOV LOC, AL       ;move it to memory

PRINT ISR
IN AL, PC         ;read in Port C
REAP: TEST AL, 2   ;test bit D1 (OBFB)
JZ REAP           ;if low, test bit till high
OUT PB, AL
IRET

```

9.6 | Mode 2 (Strobed Bidirectional Bus I/O)

This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bidirectional bus I/O). ‘Handshaking’ signals are provided to maintain proper bus flow discipline in a similar manner to Mode 1. Interrupt generation and enable/disable functions are also available.

Mode 2 Basic Functional Definitions:

- Used in Group A only.
- One 8-bit, bi-directional bus port (Port A) and a 5-bit control port (Port C).

3. Both inputs and outputs are latched.
4. The 5-bit control port (Port C) is used for control and status for the 8-bit, bi-directional bus port (Port A).

Since this is used only for very special applications, detailed discussion of this mode will not be attempted here. For more details of this mode, the data sheet of the chip may be referred.

9.7 | Centronics Printer Interface

Now, let us discuss a very popular application of 8255 in Mode 1. The Centronics printer interface is one such application. The printer parallel port has been in use for many years but is getting to be more or less obsolete these days – this is because the USB port is now used in most applications replacing the parallel port as well as the conventional serial port. However, the parallel port is still maintained as a legacy port in many computers. Let us have a look at this application which has been in use for many, many years.

In the 1970s, a printer company named Centronics developed an electrical interface to its printer, which soon became very popular. By 1981, when IBM wanted to interface its PC to a printer, they decided to modify the connector on the PC side. The end result was that the printer interface consisted of a 25-pin connector on the PC side and a 36-pin Centronics connector on the printer side. This is also called the Epson FX-100 standard. It might seem quite weird to find 36 pins devoted to sending data at just one byte at a time, though with a few handshaking signals as well. However, in actuality, most of the control pins and all the data pins have individual ground return pins rather than a common ground. This has been done to reduce the effect of interfering electrical noise.

Table 9.3 shows the pin designations of the 36 pins of the Centronics interface as well as the corresponding 25 pins (DB-25) to which these 36 pins are connected to. Obviously, many of the 36 pins will be designated as ‘no connection’ (NC). Now, let us discuss how the PC and the printer communicate such that data bytes get printed one at a time. See the Timing diagram, Fig. 9.18, of the data transfer between the PC and the printer. The steps involved in sending one byte of data to the printer are:

1. The PC checks if the BUSY signal of the printer is asserted. If it is found low, it means the printer is not busy; i.e., it is willing to accept a data byte for printing.
2. The PC then sends out one-byte of data on the data bus.
3. The PC also asserts the *STROBE* signal (makes it low) for at least 0.5 μ secs.

4. The timing diagram also shows that the data should remain at the data pins at least 0.5 μ secs after the *STROBE* is made high.
5. Once the *STROBE* goes low, the *BUSY* signal goes high to indicate that it is busy in the process of printing this byte.
6. When the printing of this byte is done with, the printer sends an acknowledge signal (*ACK* to the PC indicating that it is ready to accept a new byte for printing). The PC can use either the *ACK* or *BUSY* signal to verify if the next byte can be sent for printing.
7. The ‘minimum’ timing requirements are to be met.

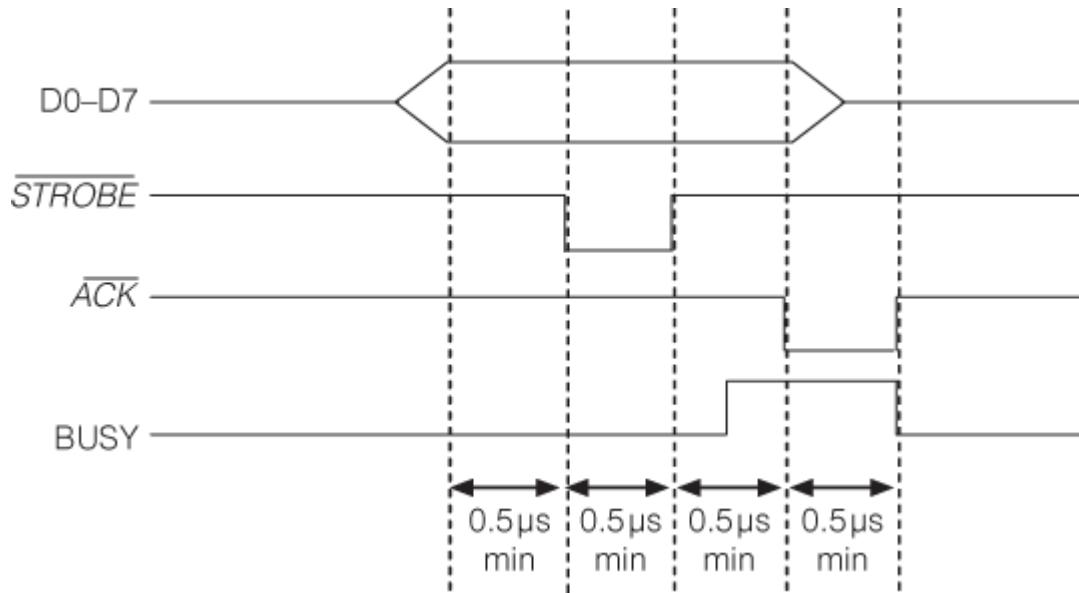


Figure 9.18 | Timing diagram of the Centronics interface

Table 9.3 | List of Centronics and DB-25 Pins *SLCTIN*

Centronics Pins	DB-25 Pin	Name	Direction	Description
1	1	<u>STROBE</u>	PC-to-Printer	Strobe
2	2	D0	PC-to-Printer	Data Bit 0
3	3	D1	PC-to-Printer	Data Bit 1
4	4	D2	PC-to-Printer	Data Bit 2
5	5	D3	PC-to-Printer	Data Bit 3
6	6	D4	PC-to-Printer	Data Bit 4
7	7	D5	PC-to-Printer	Data Bit 5
8	8	D6	PC-to-Printer	Data Bit 6
9	9	D7	PC-to-Printer	Data Bit 7
10	10	<u>ACK</u>	Printer-to-PC	Acknowledge
11	11	BUSY	Printer-to-PC	Busy
12	12	POUT	Printer-to-PC	Paper out
13	13	SEL	Printer-to-PC	Select
14	14	<u>AUTOFEED</u>	PC-to-Printer	Autofeed
15	N/A	n/c	N/A	Not used
16	N/C	0 V	N/A	Logic ground
17	N/C	CHASSIS GND	N/A	Shield ground
18	N/C	+5 V PULLUP	Printer-to-PC	+5 V DC (50 mA max)
19–30	18–25	GND	N/A	Ground reference for signal pins 1–12, in most cables as twisted pairs
31	16	<u>RESET</u>	PC-to-Printer	Reset
32	15	<u>FAULT</u>	Printer-to-PC	Fault (Low when offline)
33	N/C	0 V	N/A	Signal ground
34	N/C	N/C	N/A	Not used
35	N/C	+5 V	Printer-to-PC	+5 V DC
36	17	<u>SLCTIN</u>	PC-to-Printer	Select in (Taking low or high sets printer on line or off line respectively)

Various other status and control signals are seen for communications between a PC and a printer. Their functions are listed in Table 9.3. The printer interface will not be discussed in more detail because, as mentioned earlier, it is more or less obsolete. However, before we conclude, a few tit-bits from the history of parallel ports are presented here.

Note N/A stands for not available and N/C for no connection.

9.7.1 | History

The word LPT is sometimes used to designate the parallel port of the PC. The word stands for ‘Line Print Terminal’, which was designed for line printers meant for printing text data. Because there is no written standard, the timing relationships between the handshaking signals vary widely among printers from different manufacturers. As we have just

seen, the parallel port is unidirectional, because data was ‘sent’ for printing. However, printing is not the only function that a parallel port can do. A parallel port with unidirectional capabilities is called the ‘standard parallel port’. In this standard, there are a few registers involved in data transfer and specific timing too. The SPP has only a few differences from the original Centronics port.

The Bidirectional Port

The IBM PS / 2 (Personal System / 2) computer enhanced the standard PC parallel interface by adding bidirectional drivers to the eight data lines. The I/O connector and signal assignments remained the same. A parallel port with bidirectional drivers is often referred to as an Extended Parallel Port (EPP). EPP was developed to provide for high-speed, bidirectional data transfers that are compatible with the register map of the existing standard parallel port. The EPP specification assigns traditional microprocessor bus signaling functions to the standard parallel port lines (address strobe, data strobe) to access adapter hardware directly.

Extended Capabilities Port (ECP)

ECP is an extension to the bidirectional standard parallel port and was developed by Microsoft and Hewlett Packard in 1992. The specification defines automatic hardware handshaking, command and data cycles, and DMA transfers. The handshaking signals for data transfers have the same timing relationships as defined for standard parallel ports. In 1994, the IEEE 1284 standard was released. It included the two specifications for parallel port devices, EPP and ECP. In order for them to work, both the operating system and the device must support the required specification. This is seldom a problem today since most computers support SPP, ECP and EPP and will detect which mode needs to be used, depending on the attached device. If you need to manually select a mode, you can do so through the BIOS on most computers.

9.8 | Interfacing an Analog to Digital Converter to the 8086

Real world applications use sensors to get a signal voltage. For example, for measuring a temperature, a thermocouple may be the sensor used. This gives an analog voltage corresponding to the temperature value. Various other sensors measure quantities like humidity and speed and give the measured values as analog voltages. To use these values in a computer, these voltages have to be converted to digital numbers. This is the function of an analog to digital converter. There are various methods

employed in this conversion, which you might have learned in your basic digital circuitry course. Each method has its merits and demerits. There are various standard ICs available as ADCs. They vary in the number of digital output bits, the number of analog inputs (multiplexed) and speed of conversion. These parameters will have to be verified from the data sheet of the IC before choosing a particular ADC for an application.

Here, our interest will be to interface an ADC to the 8086 using the 8255 as an interface. Also, ADCs may be ‘parallel’ or ‘serial’. This is an important application for the parallel port IC 8255. When an analog voltage is given as an input to an ADC, it gets converted to a digital number which is transferred to the 8086. The digital value can be stored in the RAM of the system and may be displayed or used in further computations. See the block diagram of such a setup in Fig. 9.19 with the 8255 acting as an intermediary in this. What is the role of the 8255 chip in the conversion and data transfer? The answer is that the converted digital data from the ADC is transferred to the 8255 with one of its ports acting as an input. This data is sent to the processor through the data bus. Moreover, the ADC needs a few control signals for its operation. This is provided by selectively using the other port pins of the 8255 PPI.

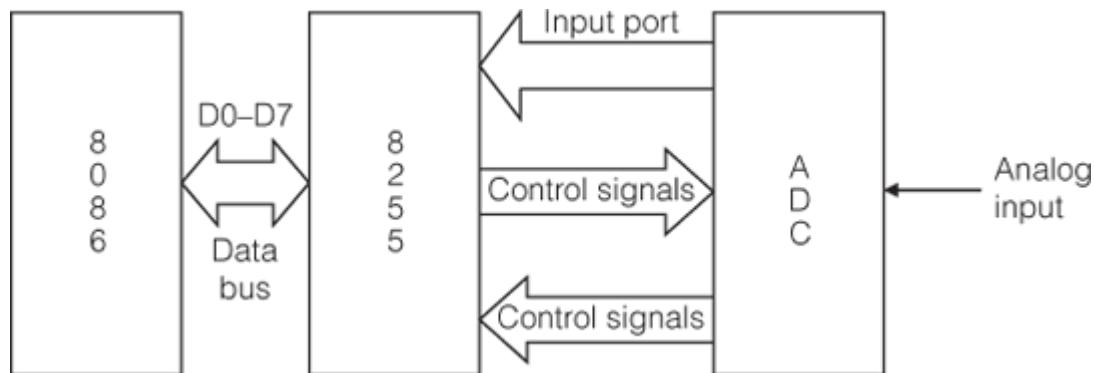


Figure 9.19 | General block diagram of the connection between an ADC, PPI and 8086

ADC 0808/0809

There are many different methods of analog to digital conversion and many different standard ADC chips. We will choose the ADC0808/ADC0809, which is 8-bit parallel ADC and microprocessor compatible. Its functional pin diagram is shown in Fig. 9.20. It is designated as an ‘8-Bit μ P Compatible A/D Converter with 8-Channel Multiplexer’. It uses the successive approximation technique for analog to digital conversion.

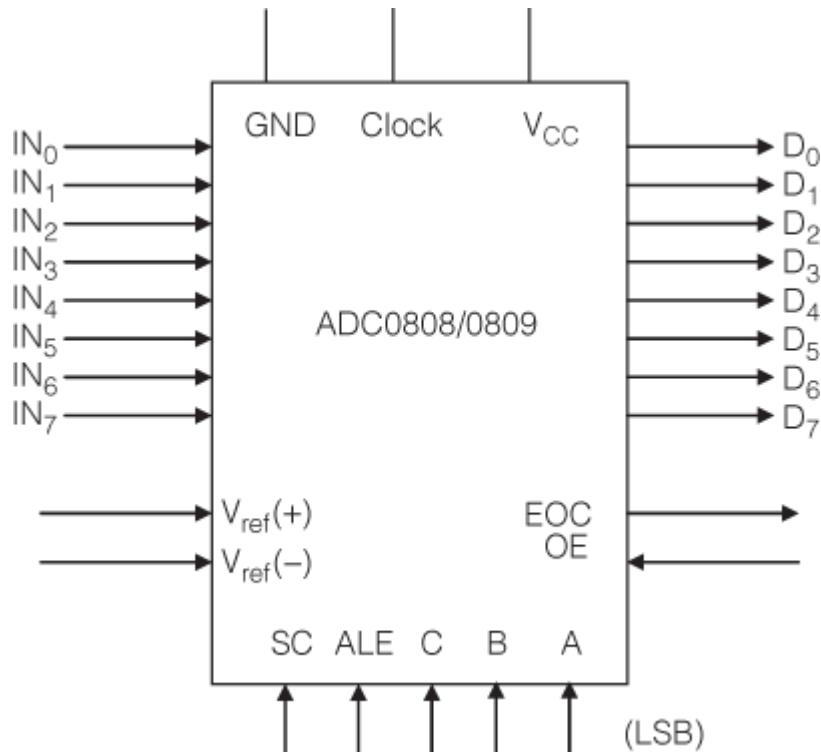


Figure 9.20 | Functional pin diagram of the ADC 0808/0809

Its key specifications are given as:

1. Resolution 8 Bits
2. Total Unadjusted Error $+/- \frac{1}{2}$ LSB and $+/- 1$ LSB
3. Single Supply 5 VDC
4. Low Power 15 mW
5. Conversion Time 100 μ secs

It is an 8-input multiplexed ADC – which means that it has 8 input analog signal lines, though only one of them can be operational at a time. This is selected by three address inputs A, B and C. Table 9.4 shows the address bit configuration for selecting specific input channels. For example, if IN0 is to act as the input, the address lines C, B and A all have to be low; for IN1, the values of CBA is to be 001 and so on. The first requirement in using the ADC is to select an input channel by giving the appropriate logic on the address pins. To latch this on to the chip, a signal called ALE (Address Latch Enable) is to be supplied on the ALE pin. ALE is to be a low to high transition (ref the timing diagram in Fig. 9.21). After the address is latched and the analog input is available at the selected input line, the ADC must be signaled to start conversion. This is a low to high pulse of minimum specified duration (as mentioned in the data sheet). The ADC requires a clock and the speed of conversion depends on the clock rate. The maximum clock frequency will be

specified in the data sheet. The clock of the processor can be divided to get the right frequency for the ADC.

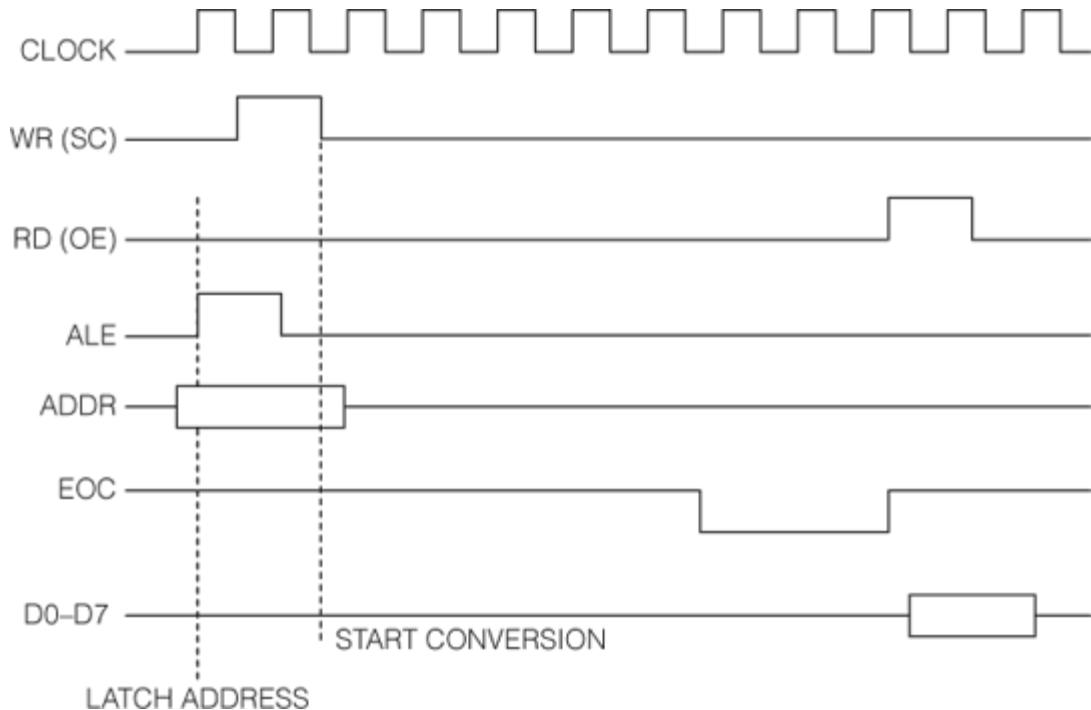


Figure 9.21 | Timing diagram for the ADC 0808/0809

Table 9.4 | Channel Selection Logic

Selected analog channel	C	B	A
IN0	0	0	0
IN1	0	0	1
IN2	0	1	0
IN3	0	1	1
IN4	1	0	0
IN5	1	0	1
IN6	1	1	0
IN7	1	1	1

The ADC takes a finite time to complete the conversion and then it notifies this fact by lowering the pin called EOC (End of Conversion). This should be brought to the notice of the processor. The EOC signal can be used to interrupt the microprocessor and allow the converted data to be transferred to the 8086 or this signal can be polled continuously. To receive the digital data, the output lines of the ADC are to be activated. This is done by making high the line OE (Output Enable). Once the output lines are activated, the converted digital data can be transferred to the 8086. The above is the sequence of actions necessary

to use the ADC chip 0809 to perform analog to digital conversion and then to transfer the digital data to the microprocessor.

Let us now use the pins of 8255 for the purposes specified above. Make the connections between the 8086, 8255 and the ADC, as shown in Fig. 9.22. The salient points regarding the connection are

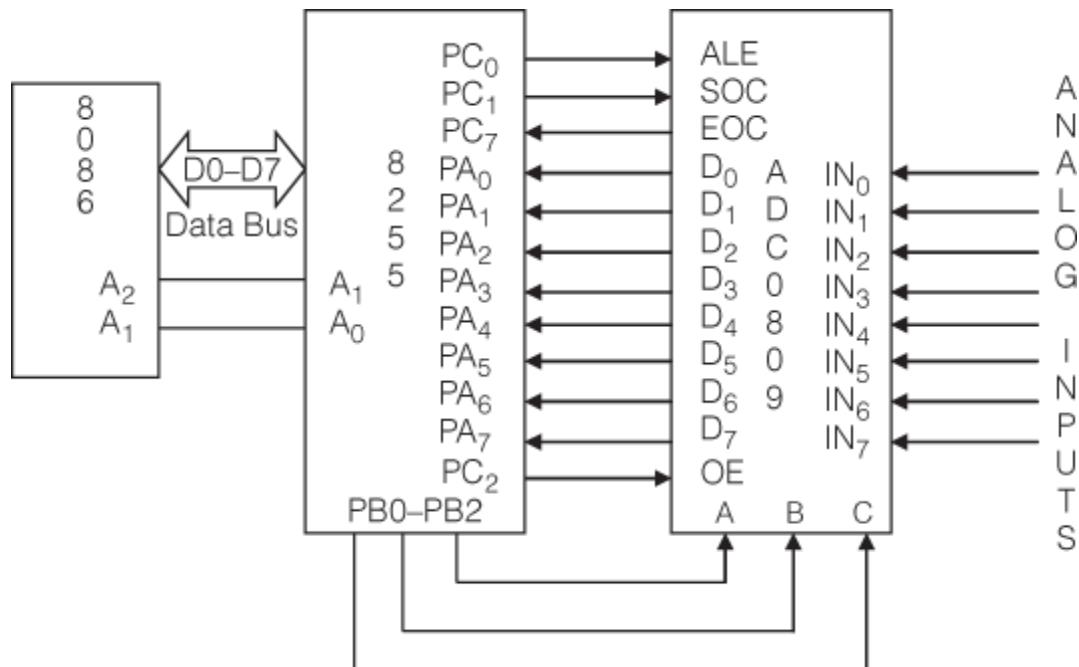


Figure 9.22 | ADC with its control and data pins connected to an 8086

1. Port A is used in the input mode to get the converted digital data from the ADC to 8255.
2. The port pins PB₀ to PB₂ are used in the output mode as the address selection pins A, B and C of the ADC.
3. The port pin PC₀ is used as ALE; it is to be an output pin.
4. The port pin PC₁ is used to give the start conversion (SC) pulse to the ADC. Hence, it is to be an output pin.
5. The port pin PC₇ is used in the input mode to receive the End of Conversion (EOC) signal from the ADC.
6. Port pin PC₂ is used as OE for the ADC. It is defined as an output pin.

Example 9.9

Write a program to interface the ADC 0809/0808 to the 8086 using 8255 as an interfacing chip.

Solution

Refer to Fig. 9.22 for the connection diagram. Also refer to the timing diagram (Fig. 9.21) to understand the following steps:

1. First, the control word in Mode 0 is to be finalized. Port A is to be input, Port B output, Port C_Loutput and Port C_U input. Hence, the control word is 1001 1000, i.e., 98H.
2. The ALE signal has to be pulsed; i.e., it should have a low to high transition and for that pin PC₀ can be used in the BSR mode. The BSR control word for clearing and setting PC₀ are 0000 0000, i.e., 00, and 0000 0001, i.e., 01.
3. To start conversion, the SC signal which uses pin PC₁ should be made high and then low. The corresponding BSR control words are 02 and 03.
4. After a delay, both ALE and SC are to be made low. The minimum delay necessary can be checked from the data sheet.
5. Then, the EOC signal is tested. This is done by rotating Port C left and getting PC₇ into the carry bit. When the carry bit is found to be low, the OE signal on pin PC₂ is raised. This enables the output pins of the ADC. The BSR control words for raising and lowering the OE pin are 07 and 06, respectively.

```

DAT DB ?      ;memory location for converted data
CR EQU 0C6H   ;address of the control register
PA EQU 0C0H   ;address of Port A
PB EQU 0C2H   ;address of Port B
PC EQU 0C4H   ;address of Port C
MOV AL, 98H    ;control word in AL

OUT CR, AL    ;send it to control register
MOV AL, 00    ;address corresponding to IN0
OUT PB, AL    ;send it to Port B
MOV AL, 00    ;BSR control word to clear ALE (PC0)
OUT CR, AL    ;send it to the control register
MOV AL, 01    ;BSR control word for setting ALE (PC0)
OUT CR, AL
MOV AL, 03    ;BSR control word to set SC(PC1)
OUT CR, AL
CALL DELAY    ;call a delay
MOV AL, 00    ;BSR control word for clearing ALE (PC0)
OUT CR, AL
MOV AL, 02    ;BSR control word for clearing SC(PC1)
OUT CR, AL

AGN: IN AL, PC    ;read in the contents of Port C
ROL AL, 1        ;rotate left to check if PC7 is high
JC AGN          ;if carry high, continue the checking
MOV AL, 07        ;if low, EOC got. Load word for OE
OUT CR, AL
IN AL, PB        ;read in converted data from PB
MOV DAT, AL      ;save it in memory
MOV AL, 06        ;BSR control word to lower OE(PC2)
OUT CR, AL      ;send it to control register

DELAY PROC NEAR  ;delay procedure
MOV CX, N        ;the value of N can be decided by

NXT:    LOOP NXT  ;referring the data sheet for
        RET        ;required delay
DELAY ENDP
END

```

9.9 | Interfacing to a Digital to Analog Converter

Converting a digital number is an important application in electronics – you must have learned the basic methods of this conversion. Here, we will use a DAC chip and connect it to the 8086 using an 8255 as the interface. A digital number sent from the processor is converted to an analog current / voltage by this arrangement. DACs of different resolutions are available – those with 8, 10, 12 or 16-bit inputs; the more the number of bits, the better the resolution. This is understandable because an 8-bit DAC gives 2^8 or 256 levels of voltage at the output – similarly, the others give 2^{10} , 2^{12} and 2^{16} discrete current levels respectively. Adding these current values give the resultant current, which is converted to a voltage by an I to V converter (using an operational amplifier). For an 8-bit DAC, the output current I_o is a function of the binary numbers at the data input pins D_7 to D_0 .

$$I_o = I_{ref} \times \frac{D_7}{2} + \frac{D_6}{4} + \frac{D_5}{8} + \frac{D_4}{16} + \frac{D_3}{32} + \frac{D_2}{64} + \frac{D_1}{128} + \frac{D_0}{256} \quad (9.1)$$

where I_{ref} is the reference input current corresponding to the V_{ref} (pin no:14) applied to the chip.

Figure 9.23 shows the pin diagram of the chip DAC0800/DAC0802, which is an 8-bit high speed digital to analog converter with a settling time of 100ns. We will now use the 8-bit data from the 8255 as the input to the DAC. The minimum connections are as shown in the Fig. 9.24. Additional components like Zener diodes may be used for protection purposes. The output of the DAC is a current which is converted to a voltage by the opamp at the output. Since the opamp is connected in a simple inverting configuration, only positive output voltages can be obtained here. However, if the opamp is used in a difference configuration, both positive and negative values may be obtained.

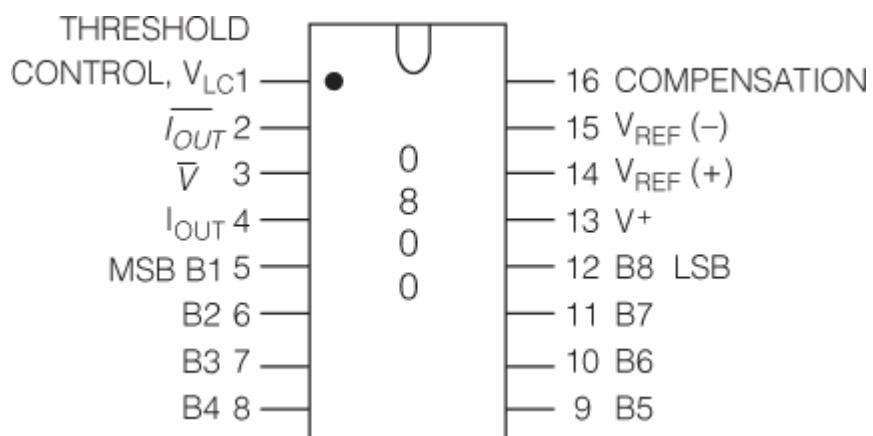


Figure 9.23 | Pin diagram of the DAC0800

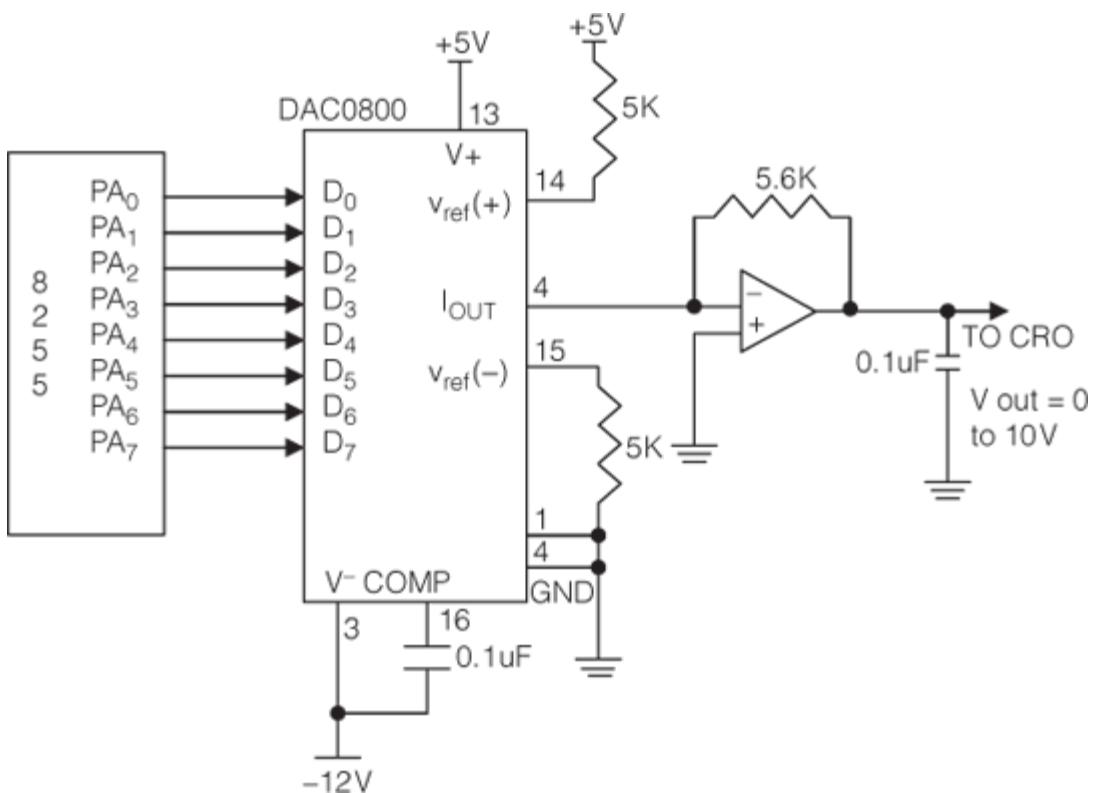


Figure 9.24 | Connections between the 8255 and DAC

Example 9.10

For the following binary numbers applied to a DAC, calculate the output analog voltage obtained given that $I_{ref} = 2 \text{ mA}$ and $R = 5.6 \text{ K}$:

1. 11000011
2. 00010111

Solution

Equation 9.1 can be used to find the value of the analog voltage for each case.

$$I_{ref} = 2 \text{ mA}$$

$$\begin{aligned} 1. \quad & \text{For } 11000011, I_o = I_{ref} (1/2 + 1/4 + 0 + 0 + 0 + 0 + 1/128 + 1/256) \\ & = 2 \times (195/256) \\ & = 1.52 \text{ mA} \end{aligned}$$

$$V_o = 5.6 \times 1.52 = 8.52 \text{ V}$$

$$\begin{aligned} 2. \quad & \text{For } 00010111, I_o = I_{ref} (0 + 0 + 0 + 1/16 + 0 + 1/64 + 1/128 + 1/256) \\ & = 2 (23/256) \\ & = 0.1796875 \text{ mA} \\ & V_o = 5.6 \times 0.1796 \end{aligned}$$

$$= \mathbf{1.006 \text{ V}}$$

Next, we will generate waveforms of various shapes using a DAC.

Example 9.11

Generate a triangular waveform using the DAC 0800.

Solution

The method is to increment the data outputted to the DAC from 0 to FFH. On conversion to analog form, it will give a gradually increasing output from 0 to the maximum output voltage. Then, decrement the number from FFH to 0. This will give the high to low triangular transition.

Note As the clock frequency of the 8086 is reasonably high, the time for incrementing the number 0 to FFH and decrementing from FFH to 0 will be quite small. Thus, the waveform generated will have a very short duration. To obtain a slower increase and decrease in analog voltage, a good idea will be to have a small delay after each increment/decrement of the digital number. The delay must be small, such that steps are not seen in the increasing/decreasing portions of the analog voltage. The amount of this small delay can also be used to control the periods of the generated waveforms. This has been done in this program.

```

CR EQU 0C6H           ;address of the control register
PA EQU 0C0H           ;address of Port A
MOV AL, 80H           ;control word for Port A as output
OUT CR, AL            ;send word to the control register
MOV AL, 0              ;AL = 0

REPEA:   OUT PA, AL      ;send AL to port A
        CALL DELAY       ;call a delay
        INC AL           ;increment AL
        CMP AL, OFFH     ;compare AL with FFH
        JNZ REPEA        ;if not equal, increment
AGN:    OUT PA, AL      ;otherwise decrement
        CALL DELAY       ;call a delay
        DEC AL           ;decrement AL
        CMP AL, 0         ;comapare AL with 0
        JNZ AGN          ;if not 0, continue decrementing
        JMP REPEA        ;if AL = 0, increment

DELAY PROC NEAR        ;delay procedure
    MOV CX, 03FH       ;value of N for a small delay
HOW:   LOOP HOW
    RET
DELAY ENDP

```

Example 9.12

Generate a staircase waveform with 5 steps.

Solution

A staircase waveform has steps in the increasing portion of the analog voltage. Once the voltage reaches the maximum value, it falls to zero in zero time. Here, we are to generate a staircase waveform with 5 steps. After each step, a delay is called. The method is as follows.

Divide the total range of 0 to 255 into 5 parts. Each increment can correspond to 51. The digital number range is 0, 51, 102, 153, 204 and 255. The program is as follows, and the output waveform is in Fig. 9.25.

```

CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A

MOV AL, 80H        ;control word for Port A as output
OUT CR, AL        ;send word to control register
STRT: MOV AL, 0    ;AL = 0
        OUT PA, AL   ;send AL to Port A
        CALL DELAY    ;call a delay
REPEA: ADD AL, 51  ;add 51 to AL
        OUT PA, AL   ;send AL to Port A
        CALL DELAY    ;call a delay
        CMP AL, 255   ;check whether AL = 255
        JNZ REPEA    ;if not, repeat the addition of 51
        JMP STRT    ;if AL = 255, go to START
        DELAY PROC NEAR ;delay procedure
        MOV CX, 200   ;value of N for 'Delay' in Fig. 9.25
AGN:  LOOP AGN    ;loop back to AGN
        RET
DELAY      ENDP      ;end of procedure
END

```

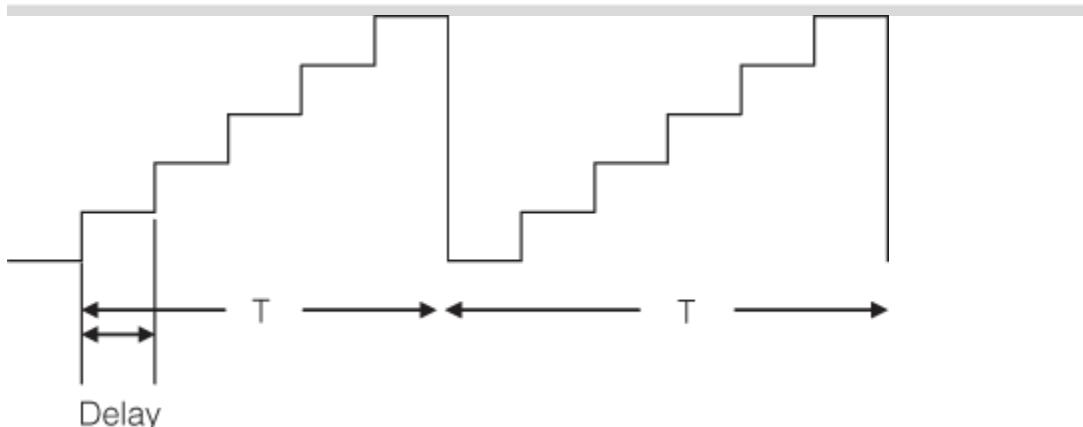


Figure 9.25 | Staircase waveform generated

Example 9.13

Generate a saw tooth waveform using a DAC.

Solution

For a saw-tooth waveform, the content of AL is incremented from 0 to FFH. Thus, the analog voltage varies slowly from 0 to the maximum value.

To enable a ‘slow’ variation, a delay is used every time an increment in AL is done. Once it reaches the maximum, it is made to go to 0, and then the whole procedure is repeated.

```

CR EQU 0C6H      ; address of the control register
PA EQU 0C0H      ; address of Port A

MOV AL, 80H      ; control word for Port A as output
OUT CR, AL       ; send word to control register
AGN:  MOV AL, 0    ; AL = 0
START: OUT PA, AL ; send AL to Port A
       CALL DELAY   ; call a delay
       INC AL       ; increment AL
       CMP AL, 0FFH ; compare AL with FFH
       JNZ START    ; if AL ≠ FFH, go to start
       JMP AGN     ; if AL = 0, go to AGN to start

DELAY PROC NEAR   ; delay procedure

        MOV CX, 03FH
HOW:   LOOP HOW
        RET
DELAY ENDP

```

9.10 | Interfacing Liquid Crystal Displays to the 8086

Liquid crystal displays called LCDs are very popular with their qualities of low power dissipation and ease of use. The only problem normally encountered is the problem of the viewing angle. The display is not equally clear at all viewing angles. Now, LCD modules of many different specifications (mostly differing in the number of lines, number of characters per line and so on) are available. An LCD module has registers writing into which the display can be easily programmed and controlled. Here, we will use a 16×2 character LCD, which looks as shown in Fig. 9.26.

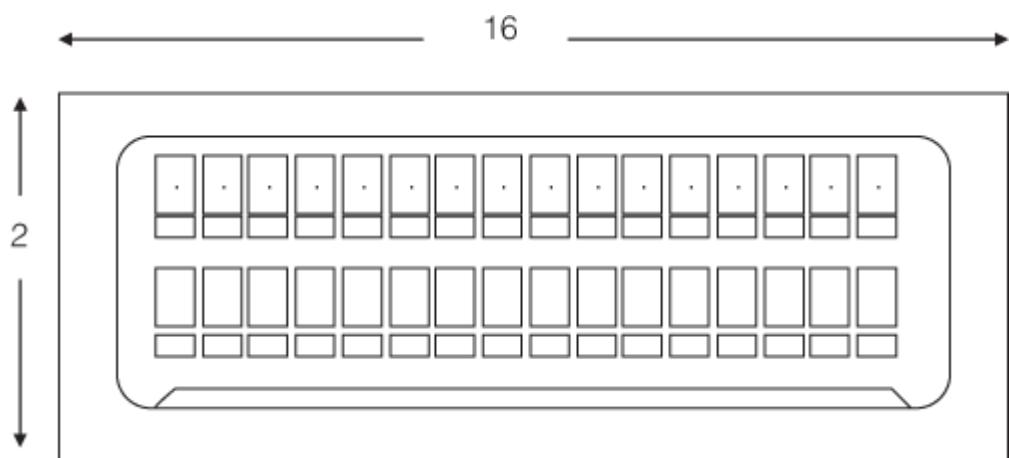


Figure 9.26 | A 16×2 LCD module

9.10.1 | Pins of the LCD

The LCD that we have selected has 16 pins as shown in Table 9.5. We see that DBo to DB7 correspond to the data pins. The others are the pins for control signals and the power supply. VEE is a pin used to adjust the contrast of the display. It is usually connected to a potentiometer so that contrast can be adjusted. Besides that, there are the VCC and ground pins. There are two pins for backlight adjustment, if necessary. Backlighting means extra lighting behind the LCD panel (usually LEDs) so that the display is visible in the dark also.

Table 9.5 | Pins of the 16×2 LCD Module

No	Symbol	Function
1	Vss	Power supply ground (0 V)
2	Vcc	Power supply (5 V)
3	VEE	Power supply for adjusting contrast
4	RS	Register select signal
5	R/W	Read write select signal
6	E	Enable signal
7	DB0	Data bus line
8	DB1	Data bus line
9	DB2	Data bus line
10	DB3	Data bus line
11	DB4	Data bus line
12	DB5	Data bus line
13	DB6	Data bus line
14	DB7	Data bus line
15	AV _{EE}	Positive voltage for back light
16	K	0 V for back light

RS Register Select

This pin selects between a command register and a data register. RS = 0 corresponds to the command register and RS = 1 to the data register. The data to be displayed is to be sent to the data register.

R/W: Read/Write

This pin allows the user to write to or read from the display. When there is no necessity for reading the display, this pin is grounded. If both reading and writing is required, this pin is made programmable.

E: Enable

The Enable pin has to be given a high to low pulse, which is maintained high for at least 450 ns (may be different for other LCD modules).

D_B0 to D_B7

These are the data pins of the LCD. Data to be written is to be sent through these pins, and data to be read will be received from the LCD through these pins. The data to be written for display are sent as ASCII characters. For writing into the command registers, there are predefined codes for the LCD. The codes for the LCD we are using are given in Table 9.6.

Table 9.6 | Command Codes of the LCD

Code (Hex)	Command
01	Clear display screen
02	Return home
04	Shift cursor left (decrement cursor)
05	Shift display right
06	Shift cursor right (increment cursor)
07	Shift display left
08	Display off, cursor off
0A	Display on, cursor on
0C	Display on, cursor off
0E	Display on, cursor blinking
0F	Display off, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80*	Force cursor to the beginning of the first line
C0*	Force cursor to the beginning of the second line
38	2 lines and 5 x 7 matrix

* For a 16 x 2 line display, the addresses of the cursor positions are 80 to 8F for the first line and C0 to CF for the second line

Busy Flag

It is seen that there is a minimum time required to latch one data on the LCD and get it displayed. Suppose we want to give a new data for display, the simplest way would be to introduce a small delay between sending the two display data (which can be given only one character at a time).

However, another method for sending consecutive characters is to check what is called the ‘Busy’ flag of the LCD. For testing the busy flag, make RS = 0 first. The ‘Busy’ flag is DB7 and can be read when R/W = 1 and RS = 0. If DB7 is found high, it means that the LCD is busy doing its operations and will not accept any new information. Keep checking this flag until it is low. Then, the next data can be written to it.

Note If the busy flag is to be read, the R/W pin has to be made programmable.

Now, let us do some display activities using a 16×2 LCD. Data and commands are sent from the 8086, using the 8255 as the interface.

Backlight: There is a lamp here instead of reflected light. If backlighting is provided by LEDs as in the case of many 16×2 LCDs, connect pin 16 to ground and pin 15 to Vcc through a 100Ω resistor.

The connections are to be done as follows:

- V_{ss} and R/W is connected to ground.
- V_{cc} is connected to 5 V supply.
- V_{ee} is connected through a 10 K pot to the supply for contrast adjustment.
- RS is connected to PC0 and E is connected to PC1.
- Pin 7-14 (DB0 to DB7) of the LCD module are connected to port A.
- Pins 15 and 16 of the LCD are used for backlight adjustment (not shown in Fig. 9.27).

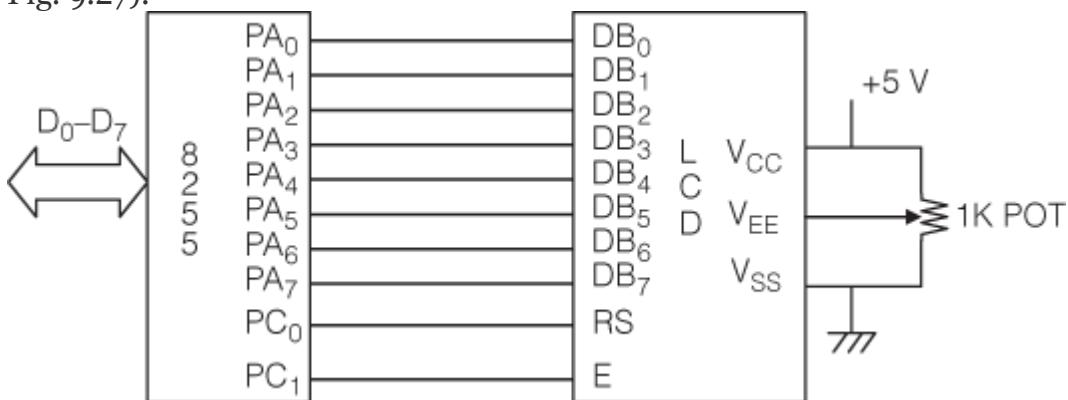


Figure 9.27 | The LCD connected to the 8255

Example 9.14

This program displays ‘YA’ on the LCD. The generation of control signals necessary for sending the commands to the command register are put in a procedure named ‘COMMAND’, and similarly, there is a procedure named ‘DAT’ which takes care of the control signals necessary for sending to the data register the characters which are to be displayed.

Note that first the LCD is initialized, then cleared, and then the cursor is positioned. This is done by sending command words to the LCD command register. For latching these words to the command register, RS has to be made low and E has to be given a high to low pulse. These are done in the COMMAND procedure.

Similarly, for displaying a character, the ASCII value of the data is to be sent to the data register and RS should be made '1'. Also, a high to low transition should occur at E. These issues are taken care of in the DAT procedure.

```

PA EQU 0C0H          ;address of Port A
CR EQU 0C6H          ;address of the control register
MOV AL, 80H           ;control Word for port A as output
MOV CR, AL            ;send it to the control register
MOV AL, 38H           ;initialize LCD, 2 lines, 5x7 matrix
CALL COMMAND          ;call the COMMAND procedure
CALL DELAY            ;call a delay
MOV AL, 0EH            ;command for display on, cursor on
CALL COMMAND          ;call the COMMAND procedure
CALL DELAY            ;call a delay
MOV AL, 01              ;command to clear LCD
CALL COMMAND          ;call the COMMAND procedure
CALL DELAY            ;call a delay
MOV AL, 80H           ;cursor at line 1, positon 1
CALL COMMAND          ;call the COMMAND procedure
CALL DELAY            ;call a delay
STRT: MOV AL, 'Y'      ;move to AL, the ASCII of 'Y'
CALL DAT              ;call the DAT procedure
CALL DELAY            ;call a delay
MOV AL, 'A'             ;move to AL, the ASCII of 'A'
CALL DAT              ;call the DAT procedure
CALL DELAY            ;call a delay
COMMAND PROC NEAR     ;procedure named COMMAND
OUT PA, AL             ;send data in AL to port A
MOV AL, 0               ;BSR word to make PC0(RS) low
OUT CR, AL             ;RS = 0 for command register
MOV AL, 03              ;BSR word to make PC1(E) high
OUT CR, AL             ;this is to make E = 1
CALL DELAY            ;call a delay
MOV AL, 02              ;BSR word to make E(PC1) low
OUT CR, AL             ;this is to make E = 0
RET

COMMAND ENDP          ;end the procedure

DAT    PROC NEAR        ;procedure named DAT
OUT PA, AL             ;send data in AL to port A
MOV AL, 01              ;BSR word to make PC0(RS) high
OUT CR, AL             ;RS = 1 is to select data register
MOV AL, 03              ;BSR word to make PC1(E) high
OUT CR, AL             ;this is to make E = 1
CALL DELAY            ;call a delay
MOV AL, 02              ;BSR word to make E(PC1) low
OUT CR, AL             ;this is to make E = 0
RET

DAT    ENDP

DELAY PROC NEAR         ;delay procedure
MOV CX, 0FFFH
AGN: LOOP AGN
RET
DELAY ENDP

```

Example 9.15

Write a program which rotates the word HELLO from right to left displayed on the LCD.

Solution

Refer Table 9.5. For rotating the display to the right, the command ‘18H’ is used.

For rotating the display in the opposite direction, use the code 1CH.

The program is the same as Example 9.14, except that the characters to be displayed are written as an array in memory and accessed one by one. Also, one more command code is added, that is, for rotation of the display. Since everything else is the same, only the additional lines of the program are shown here.

```
MESG DB 'H', 'E', 'L', 'L', 'O'  
-----  
-----  
-----  
-----  
-----  
  
REEP:    MOV AL, 18H      ;command to shift display left  
         CALL COMMAND  
         CALL DELAY  
  
         LEA BX, MESG    ;point to the characters  
         MOV CX, 5        ;CX = number of characters  
REPEA:   MOV AL, [BX]    ;move to AL the character  
         CALL DAT  
         CALL DELAY_ROLL  
         INC BX  
         LOOP REPEA       ;repeat this until CX = 0  
  
         JMP REEP         ;infinite loop for continuous  
                           rotation  
-----  
-----  
-----  
-----
```

Note The DELAY_ROLL procedure is used for deciding the rate at which rotation occurs. For slow rotation, this delay can be made high.

9.11 | Interfacing a Stepper Motor to the 8086

Introduction to Stepper Motors

A stepper motor is an electromechanical device which converts electrical pulses into discrete mechanical movements. When electrical pulses are

applied to it, the shaft of the motor rotates in steps and this type of movement gives the motor its name.

Principle of Operation

Stepper motors operate differently from normal DC motors. A DC motor rotates continuously when voltage is applied to its terminals. Stepper motors have multiple ‘toothed’ electromagnets arranged around a central gear-shaped piece of iron (see Fig. 9.28). The electromagnets are energized by an external control circuit which sends pulses to the motor. To turn the motor shaft, one of the electromagnets is given power first, which makes the gear’s teeth magnetically attracted to the electromagnet’s teeth. When one tooth of the gear is thus aligned to the energized electromagnet (Electromagnet ‘B’ and tooth ‘6’ are aligned in Fig. 9.28), others are slightly offset from the corresponding electromagnets. When the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next one, and from there the process is repeated. Each of those slight rotations is called a **step**, with an integral number of steps making a full rotation. In this way, the motor can be turned by a precise angle.

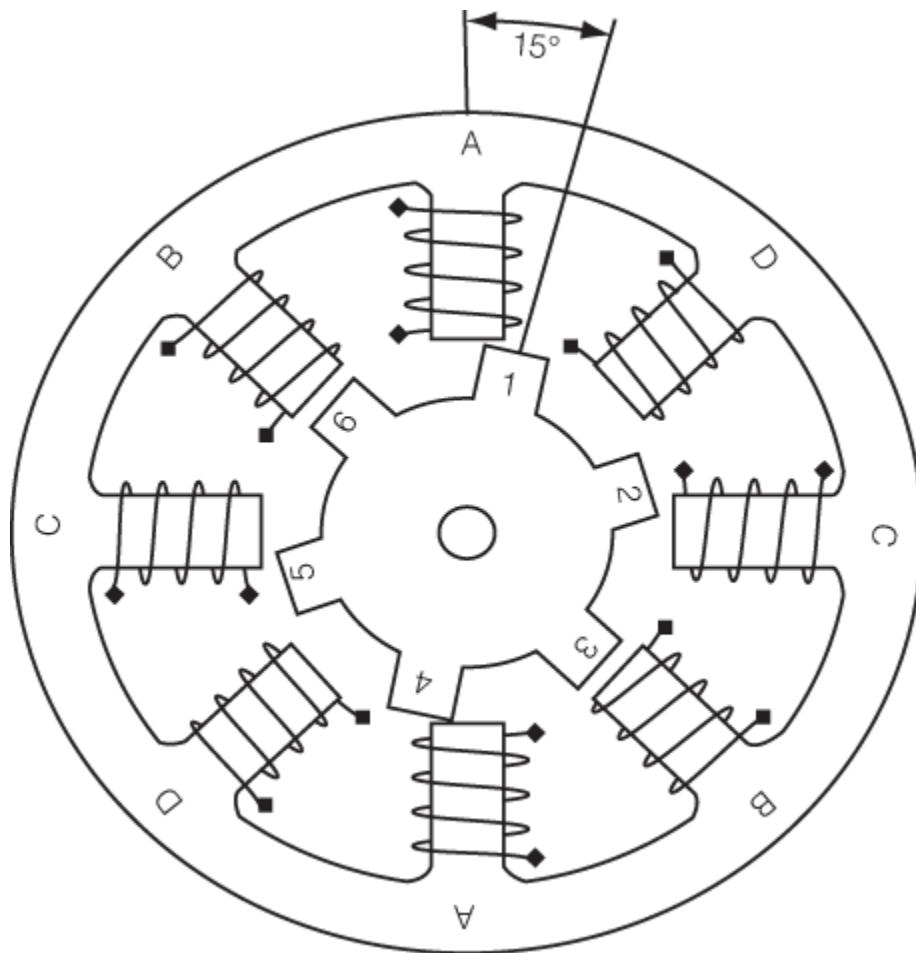


Figure 9.28 | Cross-section of a variable reluctance (VR) motor

The rotation of the motor is related to the sequence of the input pulses:

1. The order in which a particular sequence is applied decides the direction of rotation (clockwise or anti-clockwise).
2. The speed of stepping depends on the frequency of the pulses applied; i.e., higher the frequency, faster the stepping motion.

We can use stepper motors for movement which needs to be finely controlled. The fine control is obtained because these motors move in steps, and the steps can be quite small in size. For example, one step can be 2 degrees, and for one complete (360 degree) rotation, 180 steps are obviously needed. In order to obtain 90 degree rotation for such a motor, we must write a program to supply only 45 pulses to it. This can be used to advantage when it is needed to control aspects such as rotation angle, speed, position and synchronism. As such, they are used in applications such as printers, plotters, high end office equipment, hard disk drives, medical equipment, fax machines and automotive and industrial applications where precise and controlled rotation is required.

9.11.1 | Stepper Motor Types

There are three basic stepper motor types. They are

Variable reluctance

Permanent magnet

Hybrid

Variable Reluctance

This type of motor consists of a soft iron multi-toothed rotor and a wound stator. When the stator windings are energized with DC current, the poles become magnetized. Rotation occurs when the rotor teeth are attracted to the energized stator poles (Fig. 9.28).

Permanent Magnet

Such motors have permanent magnets added to the motor structure. The rotor has no teeth but is made with alternating north and south poles of permanent magnets situated in a straight line parallel to the rotor shaft (see Fig. 9.29). Applying current to each phase in sequence will cause the rotor to rotate by adjusting to the changing magnetic fields. Although it

operates at fairly low speed, the PM motor has a relatively high torque characteristic, as the magnetic flux intensity due to the permanent magnet is much higher.

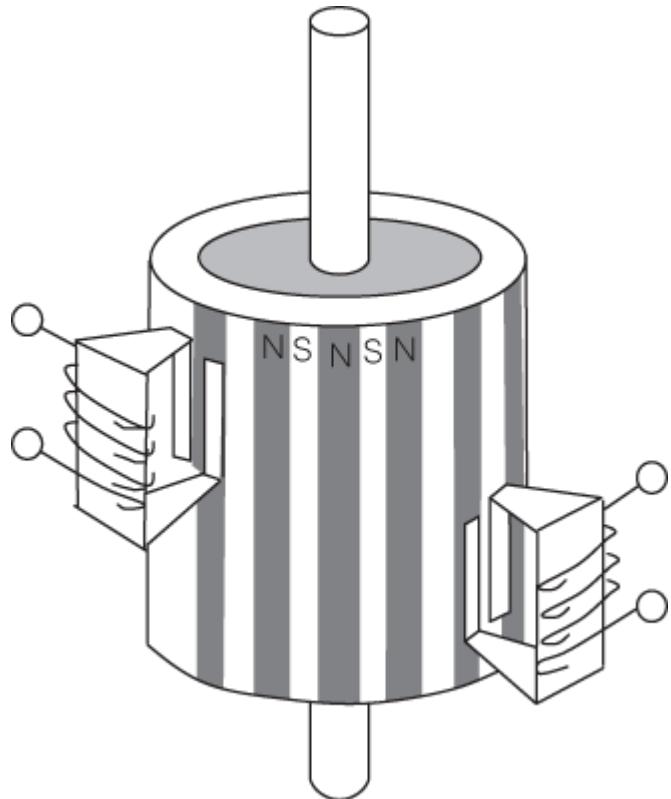


Figure 9.29 | A permanent magnet stepper motor

Hybrid

The hybrid stepper motor combines the best features of both the PM and VR type stepper motors. The rotor is multi-toothed like the VR motor and contains an axially magnetized concentric magnet around its shaft (see Fig. 9.30). The teeth on the rotor provide an even better path, which helps guide the magnetic flux to preferred locations in the air gap.

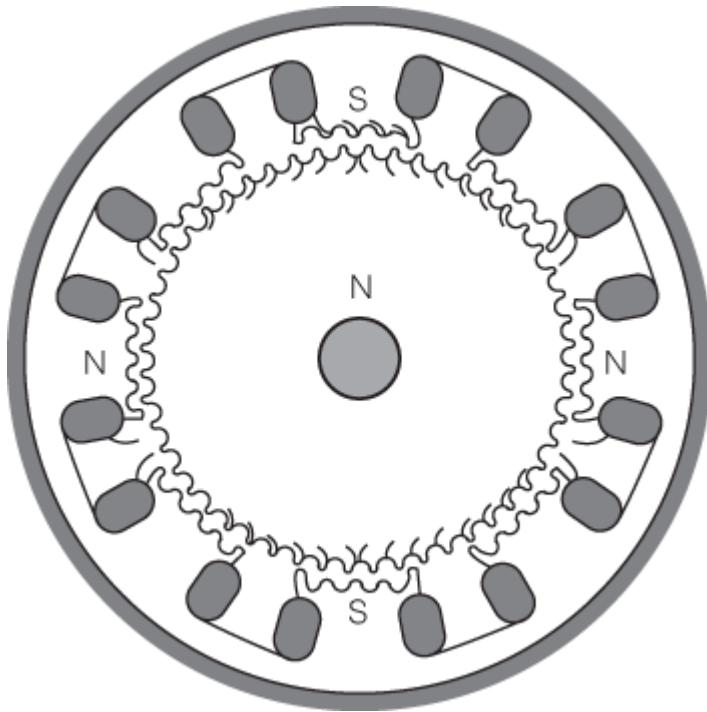


Figure 9.30 | Cross-section of a hybrid stepper motor

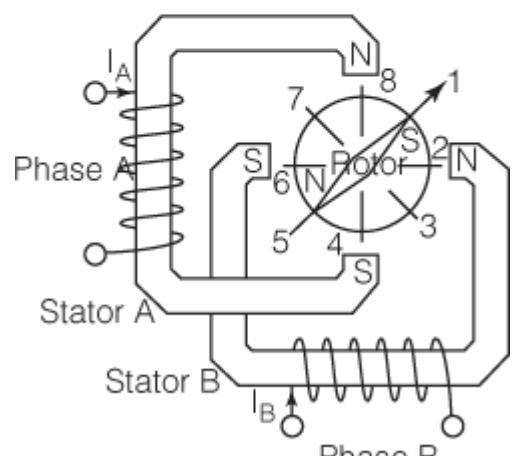
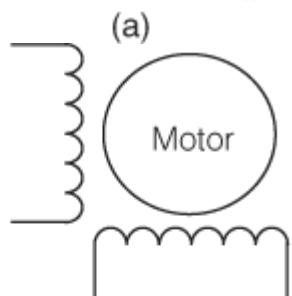
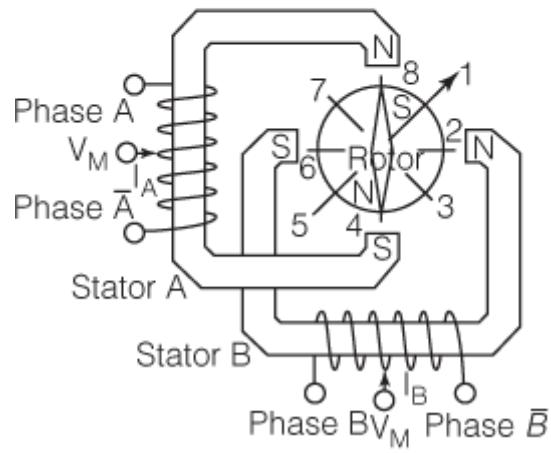
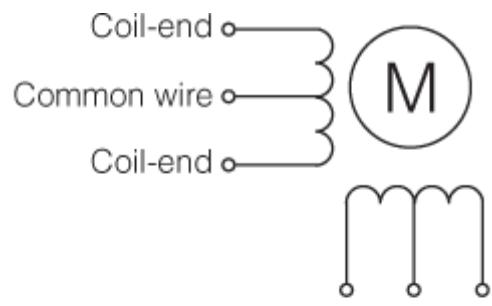
The permanent magnet and the hybrid types are the most popular types with the latter being more expensive but capable of being designed for greater resolutions, i.e., smaller step sizes.

9.11.2 | Two-phase Stepper Motors

There are two basic winding arrangements for the electromagnetic coils in a two-phase stepper motor: bipolar and unipolar.

Unipolar Motors

A unipolar stepper motor has logically two windings per phase, one for each direction of magnetic field. See Fig. 9.31a. In this arrangement, since a magnetic pole can be reversed without switching the direction of current, the commutation circuit can be made very simple for each winding. Typically, given a phase, one end of each winding is made common: giving three leads per phase and six leads for a typical two phase motor. Often, these two phase commons are internally joined, so the motor has only five leads.



(b)

Figure 9.31 | a Unipolar motor b Bipolar motor

Bipolar Motors

Bipolar motors have logically a single winding per phase, reversed in order to reverse a magnetic pole, so the driving circuit is more complicated. Figure 9.31b makes this clear. There are two leads per phase, none are common. A unipolar motor can be converted to a bipolar one (by discarding the common) but not vice versa.

Universal Motors

A universal stepper motor has 8 leads, while the unipolar has six and the bipolar has four. Usually, we use a universal motor, using only its four leads to which we feed in our electronic pulses. In our discussions, we will consider the application of a sequence of pulses to the four leads of such an arrangement. We will use a simple 90 degree PM motor with two phases. Applying current to each phase in sequence will cause the rotor to rotate by adjusting to the changing magnetic fields, and thus rotor alignment is done by pulsing.

9.11.3 | Driving a Stepper Motor

9.11.3.1 | Full Step Drive (two phases on)

This is the usual method for full step driving of the motor. Both phases are always on. The motor will have full rated torque. This is achieved by the sequence of ones and zeros as shown in Table 9.7, which is to be repeatedly applied. Reversing the order in which the sequence is applied gives anti-clockwise rotation. In short, for clock wise rotation, the sequence to be applied repeatedly is 09, 0CH, 06, 03 ... For anti-clock wise rotation, it is 03, 06, 0CH, 09 ...

Table 9.7 | Driving Sequence for a Stepper Motor-Full Step Drive

Step No.	Clockwise			
	A	B	\bar{A}	\bar{B}
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

9.11.3.2 | Wave drive

In this drive method, only a single phase is activated at a time. It has the same number of steps as the full step drive, but the motor will have significantly less than rated torque. This sequence is 8, 4, 2, 1 for

clockwise and 1, 2, 4, 8 for anti-clockwise rotation. Table 9.8 illustrates the driving sequence for this.

Table 9.8 | Driving Sequence for a Stepper Motor-Wave Drive

Step No.	Clockwise			
	A	B	\bar{A}	\bar{B}
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

9.11.3.3 | Half stepping

When half stepping, the drive alternates between two phases on and a single phase on. This increases the angular resolution, but the motor also has less torque at the half step position (where only a single phase is on). The advantage of half stepping is that the drive electronics need not change to support it. The step-angle is half that of the previous two cases – thus, the stepping resolution is increased. For anticlockwise rotation, the order of the above sequence should be reversed. Table 9.9 illustrates the driving sequence.

Table 9.9 | Driving Sequence for a Stepper Motor-Half Stepping

Step No.	Clockwise			
	A	B	\bar{A}	\bar{B}
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

9.11.4 | Using the 8255 to Interface a Stepper Motor

Since we are trying to run the stepper using a sequence generated by the 8086, we also use the 8255 to send the sequence through one of the ports. The motor, however, cannot be driven directly by the parallel port, because the motor requires a current much more than can be supplied by the 8255. (The exact current requirement depends on the specifications

of the particular motor being used.) As such, current drivers are needed between the 8255 port lines and the leads of the motor. Transistors with high current capability (e.g., Darlington pair or power transistors) can be used. Besides this, there are special motor driving ICs available. One such IC is the ULN 2003 driving IC whose pin diagram is shown in Fig. 9.32. This IC contains an array of seven Darlington pair transistors.

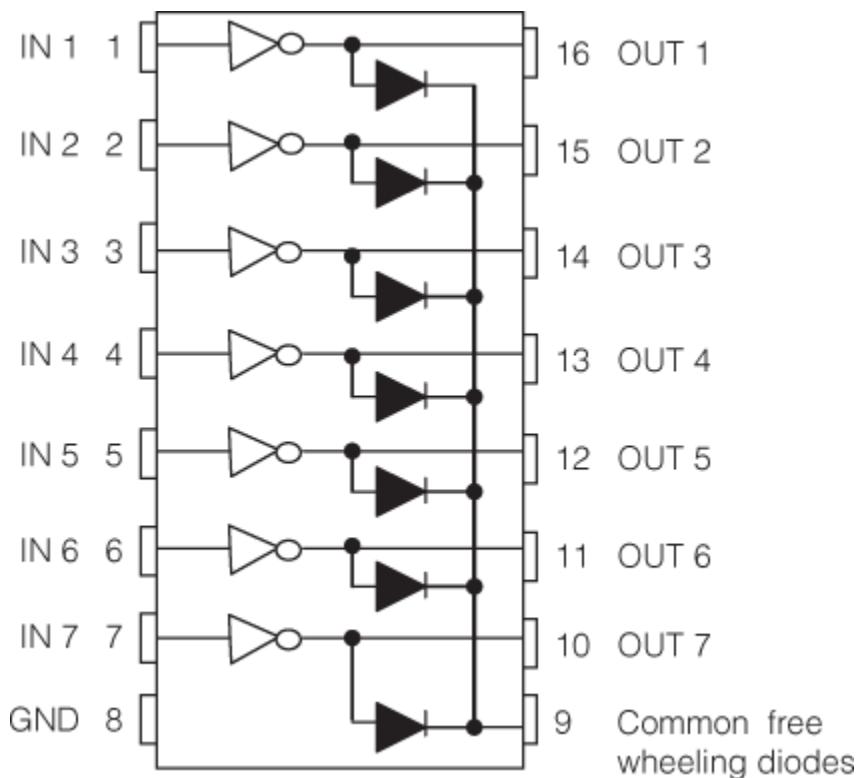


Figure 9.32 | Functional pin diagram of the driver IC ULN2003

Example 9.16

Write a program to rotate a stepper motor in the clockwise direction using the full step drive scheme.

Solution

Let us use the sequence shown in Table 9.7 for rotation in the clockwise direction. The connection is as shown in Fig. 9.33. Only the lower four pins of port B need to be used. The sequence is obtained by rotating right the data 0110 0110. On rotating this, the lower four bits are 06, 03, 09, and 0CH; this repeats to get a clockwise rotation of the motor.

```

CR EQU 0C6H ;address of the control register
PB EQU 0C2H ;address of Port B

MOV AL, 80H ;control word for Port B as output
OUT CR, AL ;send it to the control register
MOV AL, 66H ;load sequence in AL
BACK: OUT PB, AL ;send it to Port B
        ROR AL, 1 ;rotate right once
        CALL DELAY ;call a delay
        JMP BACK ;repeat the sequence

DELAY PROC NEAR ;delay procedure
    MOV CX, 0FFFH
AGN: LOOP AGN
    RET
DELAY ENDP

```

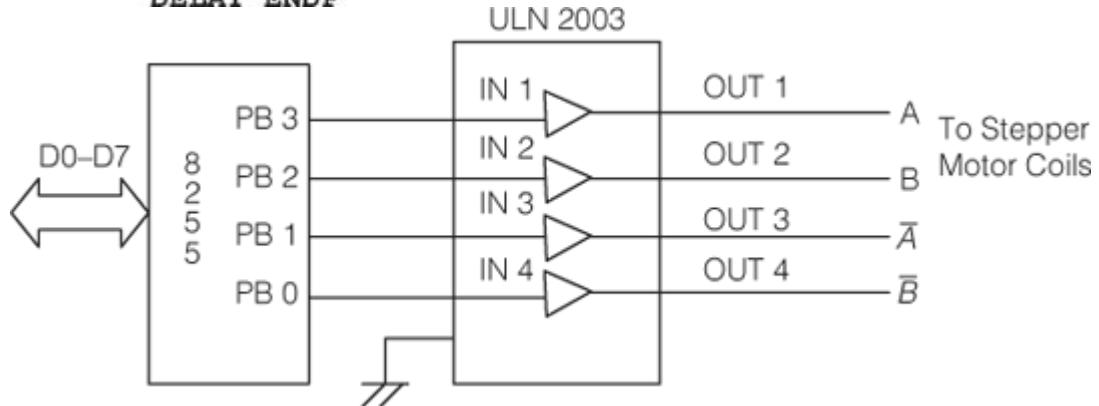


Figure 9.33 | Connection diagram of a stepper motor using the ULN 2003 IC

A delay is given between applying the sequence to the coils of the motor. The delay is needed for the coils to be energized to their new excitation sequence – also, **less the delay; more the speed of the motor. Thus, the speed of rotation is varied according to the frequency of the pulse stream.**

9.11.5 | Step Angle

Now, think of a motor with a step-angle of 1.8° . Suppose a rotation of just 90° is required. How many steps are required? Obviously,

No. of steps = $90 / 1.8 = 50$. This motor takes 200 steps to cover 360° .

Example 9.17

Write a program to rotate a motor by 90° in the clockwise direction and then rotate it 180° in the anticlockwise direction. The step-angle is 1.8° .

Solution

For rotating 90° in the clockwise direction, 50 steps are needed.

For rotating 180° in the anticlockwise direction, 100 steps are needed, and the sequence has to be given in the reverse order. So, the data 66H is shifted left in this case.

```
CR EQU 0C6H          ;address of control register
PB EQU 0C2H          ;address of Port B

MOV AL, 80H          ;control word for Port B as output
OUT CR, AL           ;send it to control register
MOV CX, 50            ;CX = 50
MOV AL, 66H          ;load sequence in AL
BACK:    OUT PB, AL      ;send it to Port B
         ROR AL, 1      ;rotate sequence once, to right
         CALL DELAY     ;call a delay
         LOOP BACK      ;repeat until CX = 0

         MOV CX, 100     ;CX = 100
         MOV AL, 66H     ;load sequence in AL
BACK1:   OUT PB, AL      ;send it to AL
         ROL AL, 1      ;rotate it once, to the left
         CALL DELAY     ;call a delay
         LOOP BACK1     ;repeat until CX = 0

DELAY PROC NEAR        ;delay procedure
    MOV BX, 0FFFH
AGN:    DEC BX
        JNZ AGN
        RET
DELAY ENDP
```

Example 9.18

Write a program to rotate a stepper motor in the wave drive 4-step sequence in the clockwise direction.

Solution

```

CR EQU 0C6H           ;address of control register
PB EQU 0C2H           ;address of Port B

MOV AL, 80H            ;control word for Port B as output
OUT CR, AL             ;send it to control register
MOV CX, 50              ;CX = 50
MOV AL, 66H             ;load sequence in AL
BACK: OUT PB, AL        ;send it to Port B
      ROR AL, 1           ;rotate sequence once, to right
      CALL DELAY          ;call a delay
      LOOP BACK           ;repeat until CX = 0

MOV CX, 100             ;CX = 100
MOV AL, 66H             ;load sequence in AL
BACK1: OUT PB, AL        ;send it to AL
      ROL AL, 1           ;rotate it once, to the left
      CALL DELAY          ;call a delay
      LOOP BACK1          ;repeat until CX = 0

DELAY PROC NEAR         ;delay procedure
  MOV BX, 0FFFH
AGN: DEC BX
  JNZ AGN
  RET
DELAY ENDP

```

9.11.6 | Other Issues Regarding Stepper Motors

An important issue to take care of when using stepper motors is that there is a chance of back emf being produced during the energization of the coils. This can damage the circuits producing the sequence and hence diodes are connected which block these spikes. Figure 9.34 shows the diodes connected in a circuit which uses Darlington pair transistors for producing the high current required to drive the motors. Note that the diodes are not shown in Fig. 9.33 where the motor driving IC ULN 2003 is used. This is because such a diode is in-built within the IC. See Fig. 9.34 where the drive for the stepper motor is given by high current Darlington pair transistors TIP 120. Diodes IN4001 are used to suppress the back emf spikes.

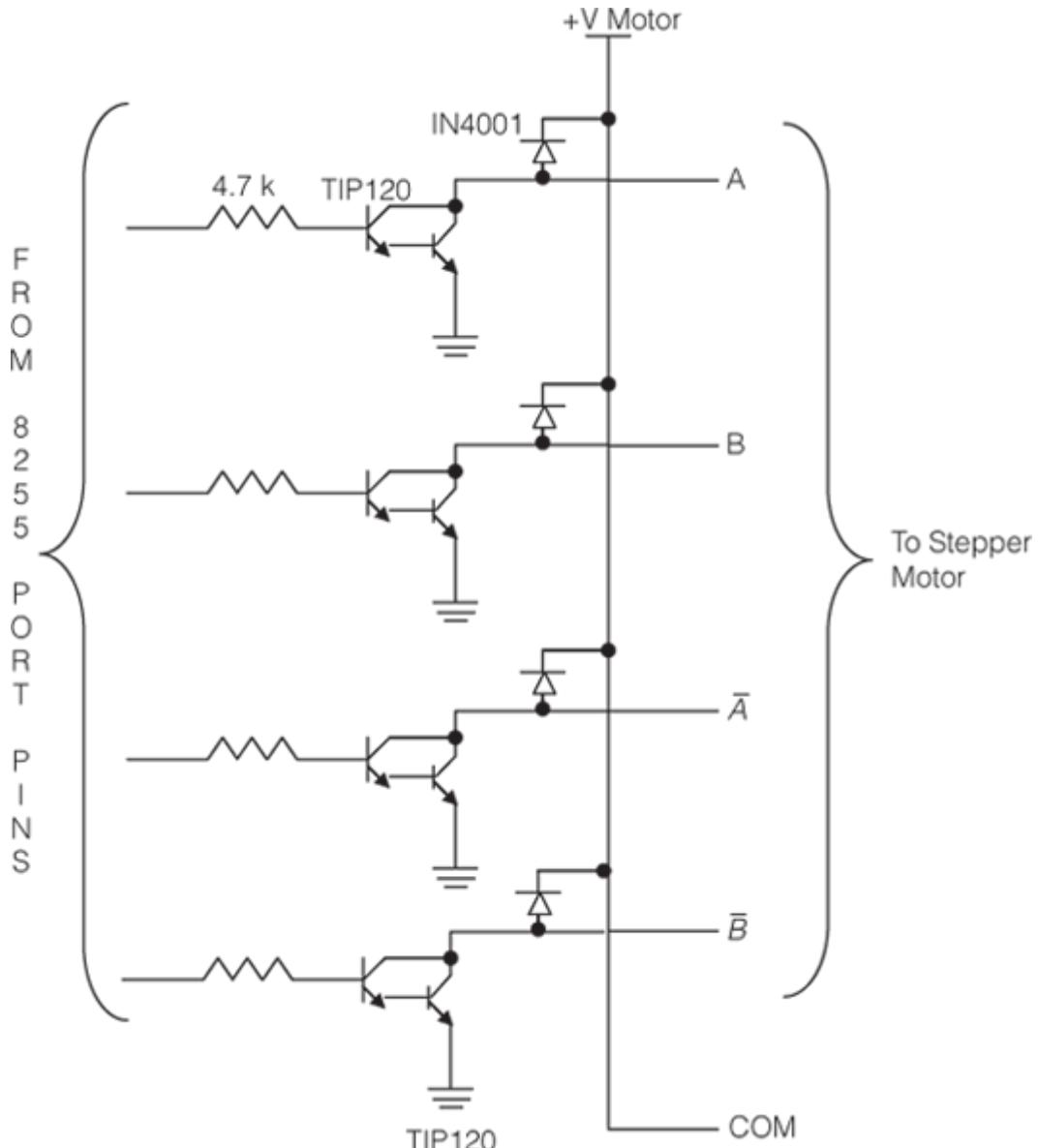


Figure 9.34 | Current driver Darlington transistors and back EMF suppressing diodes in a stepper circuit

9.11.7 | Optocouplers

An optocoupler is another device helpful in damping the back EMF produced in a motor circuit. An optocoupler or optical isolator is a device that uses a short optical transmission path to transfer a signal between elements of a circuit while keeping them electrically isolated. Typically, they come in a small 6-pin or 8-pin IC package but are essentially a combination of two distinct devices: an optical transmitter, typically a gallium arsenide LED (light-emitting diode) and an optical receiver such as a phototransistor or light-triggered diac. The two are separated by a transparent barrier which blocks any electrical current flow between the two but does allow the passage of light. The basic idea is shown in Fig. 9.35. Usually, the electrical connections to the LED section are brought

out to the pins on one side of the package and those for the phototransistor or diac to the other side to physically separate them as much as possible. Figure 9.36 shows the pin diagram of a quad optocoupler ILQ74. Figure 9.37 shows this optocoupler used in a stepper motor interfacing circuit, along with a driving IC ULN 2003.

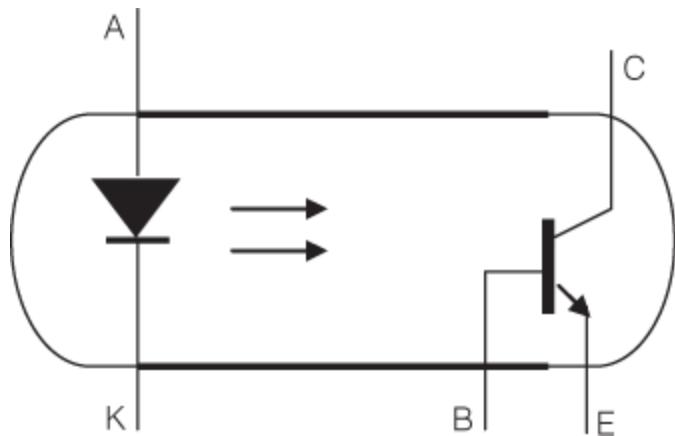


Figure 9.35 | Principle of operation of an optocoupler

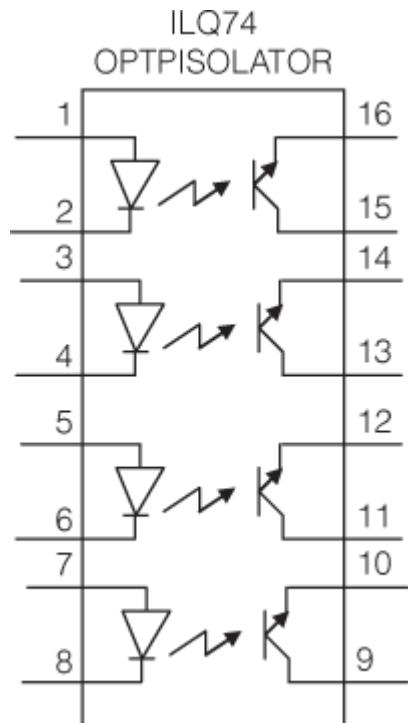


Figure 9.36 | An optocoupler IC

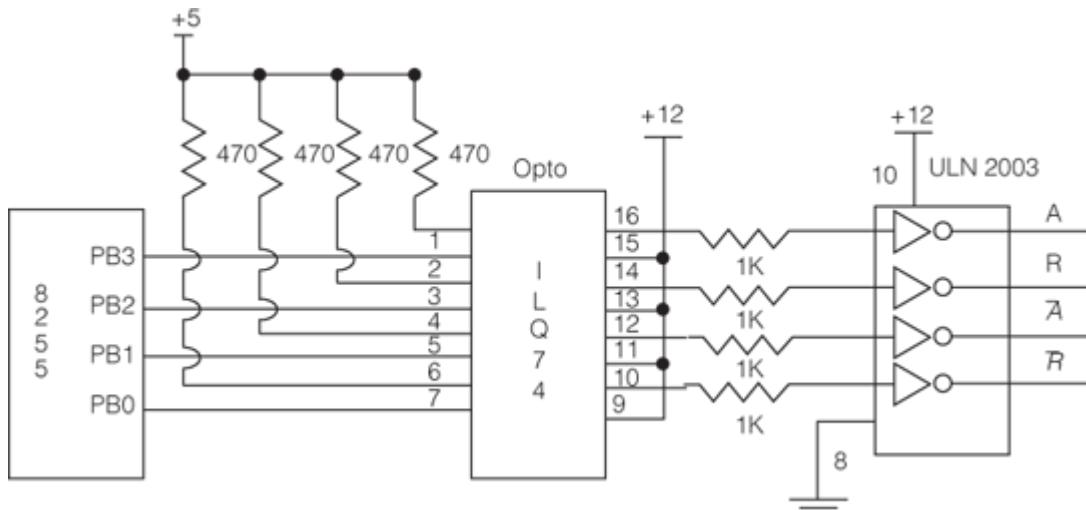


Figure 9.37 | Optocoupler used in a stepper motor driving circuit

9.12 | Hex Keyboard Interfacing

In Chapter 8, we discussed the ideas of interfacing a keyboard to the PC. It was seen that specialized microcontrollers are used to handle all the issues associated with keyboard interfacing. As such, the main processor of the PC is free of the task of identifying the key pressed. However, still it is important to understand the mechanism of how a keypress is detected and how the pressed key is identified. For this purpose, we use a hexadecimal keyboard, which has 16 keys with the characters o to F. This keyboard is interfaced to the 8086 through the port lines of the 8255. The program identifies the key that has been pressed and displays it.

9.12.1 | Hex Keyboard

See the diagram of the hex keyboard. This is a 4×4 matrix keyboard connected to the 8255. For our purpose, we will use two ports of the 8255, Port A as an output port connected to the row lines and Port B as an input port connected to the column lines. Only the lower 4 bits of the two ports are needed. See Fig. 9.38 for the connection, with the key positioned at the interconnection of a row and a column. Only when a key at a junction is pressed, a path is established between the corresponding row and column.

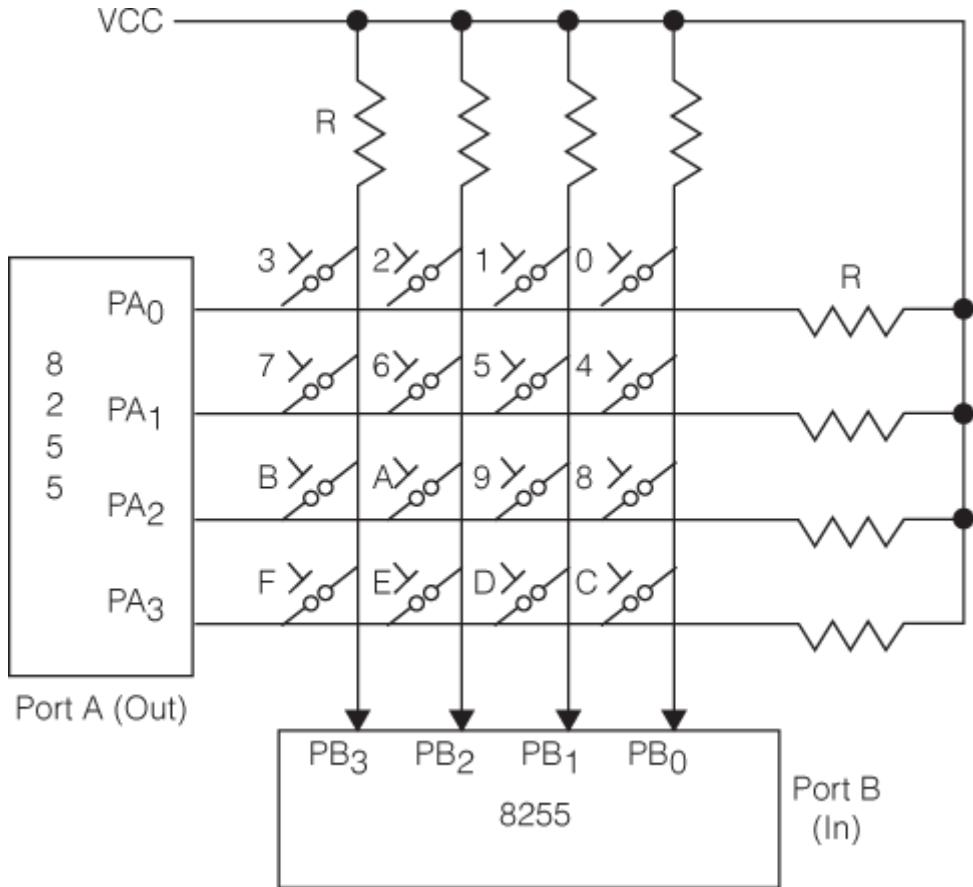


Figure 9.38 | Hex keyboard connected to an 8255

9.12.2 | Detecting a Keypress

Remember that the rows are connected to an output port and the columns to an input port. Now, send zeros from the four row outputs, and read in the contents of the columns. If no key has been pressed, there is no connection between a row and column and hence only ‘1’ s are received from all the columns. We get 1111 from PB₃ to P_{B0}. This should be clear from Fig. 9.38. If a key is pressed, the ‘0’ from the row will pass through the closed switch to the column and will be read in from Port B. Say, the key ‘9’ is pressed. Then, a ‘0’ is received from that column; i.e., the PB₃ to P_{B0} will read as 1101. An information we get from this input port reading is that a key has been pressed. So, we say that a ‘keypress has been detected’.

9.12.3 | Identifying the Key

The next task is that of identifying the key that has been pressed. For the case of key ‘9’ being pressed, looking at the number received from Port B makes it clear that a key in column 1 is the pressed key. However, we have no clue to the identity of this key, since we do not know the row in

which the key resides. Thus, identifying the row of this key is the solution to this problem. How do we go about this?

The technique is to ground the rows one by one and read in the columns, starting with Row 0. If the data read from the columns is 1111, it is obvious that this particular row does not contain the key that was pressed – if a number less than 1111 is received, the row gets identified, since we know which row is grounded now. Using this information along with the column information identifies the key. If the key ‘9’ was pressed, we get an input reading from Port B of 1101 only when the row grounded is Row 2. Thus, we identify that the pressed key is the one at the junction of the second row and first column, which is ‘9’. If a display mechanism is available, we can arrange it to display ‘9’.

Note The rows and columns are numbered from 0 to 3.

9.12.4 | Key Debounce

The key that we are using has mechanical contacts and is susceptible to what is called ‘key jitter or bounce’. That is, when a key is pressed, the key moves back and forth between the contacts for a certain time before it settles. To ensure that the key settles before we read the key, we allow a certain time for it – and this is called the key debounce time. It may range from 10 ms to 20 ms, depending on the type of keyboard we have. After this time has elapsed, if the key is still found to be pressed, it is a valid keypress. Otherwise it could have been due to a noise spike.

Now, the steps are listed out in order.

1. Keep the rows grounded and read in the columns. If any column gives a ‘0’, we know that a key might be pressed. Wait for a time corresponding to the key debounce time. After this, if the column reading is still 0, it means that there is a valid keypress.
2. Ground one row at a time and read in the columns. When a grounded row produces a column reading, which is not ‘1111’, the row is identified.
3. The next job is to identify the column of the key that was pressed. For this, the data read in through Port B is shifted right. If a column had the key press, the carry bit will be 0. Otherwise, it is shifted right again. Within 4 right shifts, the ‘0’ bit will be detected. This will give the column position.
4. The number corresponding to each key is in a look up table, stored in the order of the keys of the keyboard. Once identification is done, this is brought and saved in a memory location, after which it can be displayed if necessary.

5. The program in Example 9.19 finally saves the pressed key in a location named 'KEY'.

Example 9.19

This program tests for a key press in a hex keyboard and identifies the key.

```

ROW_0 DB 0, 1, 2, 3
ROW_1 DB 4, 5, 6, 7
ROW_2 DB 8, 9, 0AH, 0BH
ROW_3 DB 0CH, 0DH, 0EH, 0FH
KEY DB 0

CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A
PB EQU 0C2H      ;address of Port B

MOV AL, 82H      ;8255 control word
OUT CR, AL       ;send to control register

MOV AL, 0          ;AL = 0
OUT PA, AL        ;send zeros to all the rows
KEYP:   IN AL, PB    ;read in the columns
        AND AL, 0FH   ;mask the upper nibble
        CMP AL, 0FH   ;compare with 0FH
        JZ KEYP      ;if equal, no key, keep checking
        CALL DELAY   ;if key press, wait for debounce

        IN AL, PB    ;read in the columns
        AND AL, 0FH   ;mask the upper nibble
        CMP AL, 0FH   ;compare with 0FH
        JZ KEYP      ;if equal, no key, check again
        ;Next, ground one row at a time

```

```

ROW0: MOV AL, OFEH ;send '0' to Row0 alone
      OUT PA, AL
      IN AL, PB ;read in the columns
      AND AL, OFH ;mask the upper nibble
      CMP AL, OFH ;compare with OFH
      JZ ROW1 ;no keypress in Row0, check Row1
      LEA SI, ROW_0 ;not equal, SI to point to Row0 data
      JMP COL_ID ;go to finding the column
ROW1: MOV AL, OFDH ;send '0' to Row1 alone
      OUT PA, AL ;repeat steps
      IN AL, PB
      AND AL, OFH
      CMP AL, OFH
      JZ ROW2
      LEA SI, ROW_1
      JMP COL_ID
ROW2: MOV AL, OFBH ;send '0' to Row2 alone and
      OUT PA, AL ;repeat steps
      IN AL, PB
      AND AL, OFH
      CMP AL, OFH
      JZ ROW3
      LEA SI, ROW_2
      JMP COL_ID
ROW3: MOV AL, OF7H ;send '0' to Row3 alone and
      OUT PA, AL ;repeat steps
      IN AL, PB
      AND AL, OFH
      CMP AL, OFH
      JZ KEYP
      LEA SI, ROW_3
COL_ID: SHR AL, 1 ;to identify column, shift right
      JNC FOUND ;if no carry, column is found
      INC SI ;otherwise point SI to next row
      JMP COL_ID ;go back to finding the column
FOUND: MOV AL, [SI] ;get data pointed by SI into AL
      MOV KEY, AL ;this is the pressed key. Store in KEY
      DELAY PROC NEAR
      MOV CX, 6000 ;delay for key debounce
AGN:  LOOP AGN
      RET
DELAY  ENDP
END

```

9.13 | Interfacing Led Displays

We know that LEDs are very popular display devices. They are easy to use and give very bright and pleasing display, which can be viewed equally well from any viewing angle, unlike LCDs. The only drawback with LED displays is the high amount of current they need, unlike LCDs, which need very low power. Figure 9.39 shows that a single LED can be connected to a positive power supply as shown. The value of the current limiting resistor depends on the current rating of the LED. Suppose we need a number of LEDs for displays, we usually use only one power source for all of them. In that case, they are connected together either as ‘common anode’ or ‘common cathode’ LED displays. In Fig. 9.40b, the cathodes of the three LEDs are connected together. Suppose we want to light up only the first and third LED, i.e., apply a ‘1’ only at A1 and A3. In

Fig. 9.40a, which is ‘common anode’, K₁ and K₃ alone should be grounded for the same result.



Figure 9.39 | A single LED

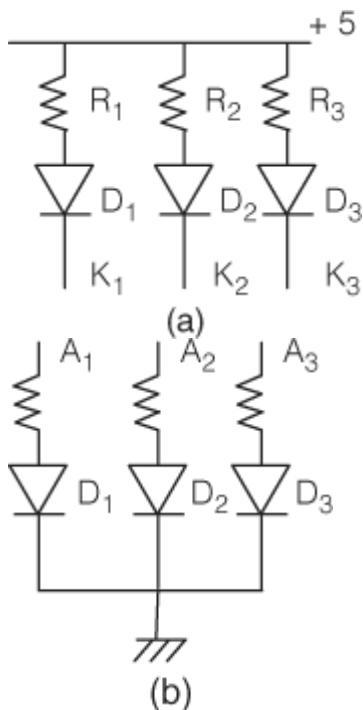


Figure 9.40 | **a** Common anode LEDs **b** Common cathode LEDs

9.13.1 | Seven Segment Led

However, more important applications of LEDs are as alphanumeric displays. In that case, seven segment LEDs are used, in which seven LEDs are arranged as the segments of a display arranged in a particular shape, which when selectively lighted up give the display of alphanumeric characters (see Fig. 9.41). By lighting up all the segments, we get ‘8’ displayed. We can have one more segment in this display and it is for the decimal point. In Fig. 9.41, there are eight segments, including the segment for the decimal point. In spite of this, we still designate such displays as ‘seven’ segment displays. Such LED modules also can be used in the common anode or common cathode configuration. To light up a seven segment display LED of the common cathode type, ensure that the

cathode is grounded and give a ‘1’ to the segments which are to be lighted. The first information (grounding the cathode) corresponds to the ‘digit’ data and the second is the segment data. By selectively lighting up segments, we can have all alphanumeric characters displayed.

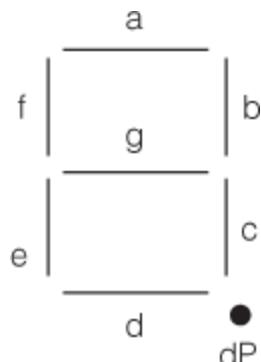


Figure 9.41 | A seven segment LED display unit

9.13.2 | Static Seven Segment Displays

Now, suppose we want to use such a module to display a single character. Let us think of a scenario wherein the number to be displayed is sent from an 8086 through an 8255 to the display module. We just send the code (called seven segment code) corresponding to the segments of the LED. This code gives the information as to which of the segments are to be lighted up for the display of a specific character.

Suppose we use only a one digit display module. If it is a common cathode type, we ground the common cathode and then we send the seven segment code directly to activate the required segments. This causes some of the segments to be ON and some to be OFF. Either way, as long as the display is ON, the module draws its required current from the power source. This may be in the range of 5 to 30 mA for a single segment to be lighted up. The display is ON all the time, and hence it is called a ‘static’ display. Suppose we need an eight digit display. If we use the same kind of static display, the current drawn is multiplied by 8, and this becomes quite a large amount. Multiply $7 \times 25 \times 8$ mA. This gives a value of 1.4 A, which is much too large for an electronic circuit. For this reason, static displays are not preferred for multiple digit displays.

9.13.3 | Dynamic Display

When there is an array of digit display units, say 8 seven segment LEDs arranged in digit form in a row, a continuous display can be obtained by lighting up just one digit at a time. The next instant this digit is switched off and the next one is lighted up. This is done continuously and cyclically from digits 1 to 8 and repeated at a rapid rate. Because of the

property of persistence of vision of the eyes, an illusion of continuous display is obtained. This is also called a multiplexed display.

The important points to note here are:

1. The common anode/cathode of a digit is to be activated for a digit to be active.
2. At a time, only the segments of one digit are 'ON'.
3. After a specified delay, this digit is switched off and the segments of the next digit are ON. The information displayed here is different from that of the previous case.
4. Thus, for display multiplexing, consecutive digits should be switched on in a cyclic fashion, and for each digit, the segment information should be supplied.

Now, let us use this concept in a system in which an 8086 handles a dynamic display using the 8255 as an interfacing chip (see Fig. 9.42). This is an 8-digit display of the **common cathode** type. The ports of 8255 are used in such a way that Port A supplies the digit information and Port B supplies the segment information. Digit information is to select which digit is being activated at a particular time. For segment information, the seven segment code of each digit should be sent as a byte. To identify which segment corresponds to which bit in a byte, see Fig. 9.43. This is just the way that the port pins are connected to the leads of the seven segment module in this example, meaning that it is not a standard connection.

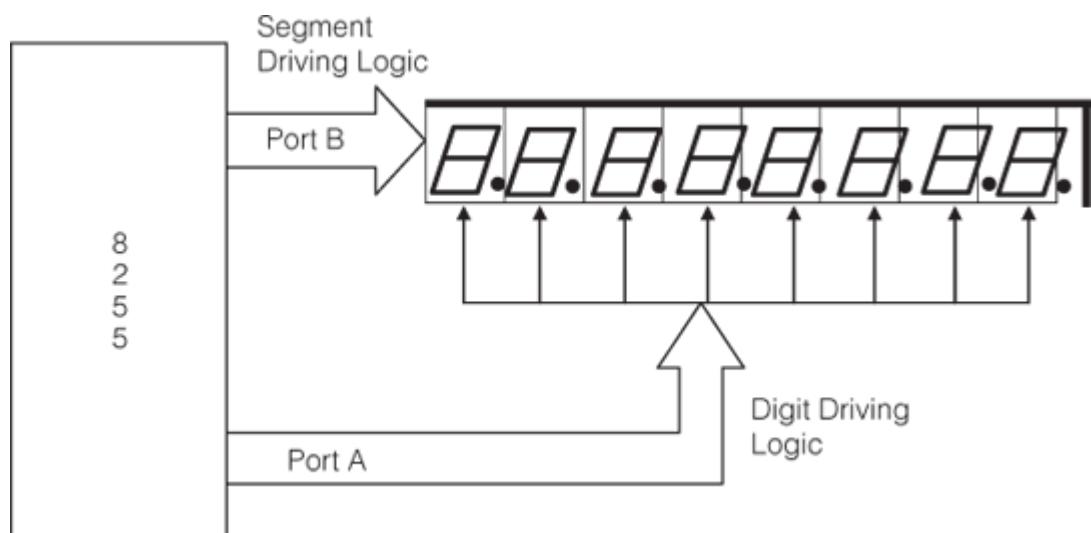


Figure 9.42 | Eight digit dynamic display connected to two ports of an 8255

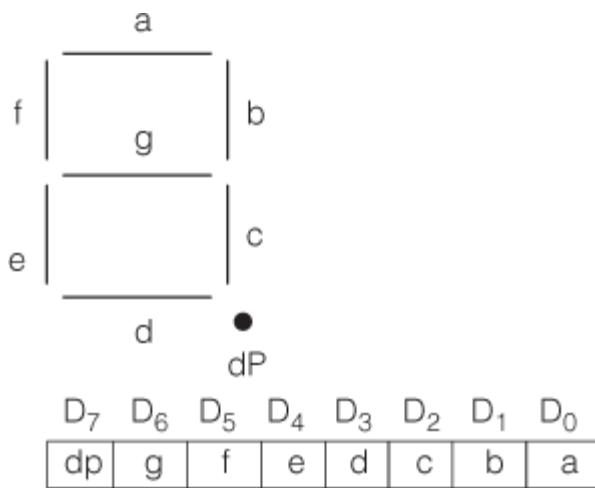


Figure 9.43 | Segments of the display and the data byte

Example 9.20

Assuming a common cathode type of display, find the seven segment codes to be sent to Port B for displaying

- i) 8, ii) A and iii) b.

Solution

Since it is a common cathode type, the common cathode of the LEDs of a digit are to be grounded. Then, supply the segment information to the anodes.

For displaying 8, the segments to be lighted up are a, b, c, d, e, f and g.

Since we are using a common cathode type of display, the data to light up a segment is '1'. Hence, the bits from D₀ to D₆ is '1'.

Thus, the seven segment code for the display of 8 is 0111 1111, i.e., 7 FH.

1. Similarly, for 'A' it is 0111 0111, i.e., 77 H.
2. For 'b' it is 0111 1100, i.e., 7 CH.

9.13.4 | An Eight-Digit Dynamic, Multiplexed Display

In Fig. 9.44, the connection between the LED, the port lines and other extra components are shown. Port A is the digit driving port. It gives the information about which digit is to be ON at a particular time. This is achieved by switching on one of the digit driving transistors Q1 to Q8. These are PNP transistors and are turned on if a '0' is applied at their bases. This '0' goes to the emitters of the transistors, which are connected to the cathodes of all the segment LEDs. At a time, Port A

gives a ‘o’ only on one of its port lines. To understand this clearly, observe Fig. 9.44. The most significant digit (or the left most digit of the display) is activated by a ‘o’ on D₇ of Port A. Thus, when Port A outputs the binary sequence 0111 1111, the left most digit is scheduled to be on, but the segment information also is to be obtained from Port B. If Port B gives the data 77 H, the first digit displays ‘A’. This is the method for displaying one digit. This is to be repeated for all digits as long as we want this data to be displayed. The steps are listed as follows.

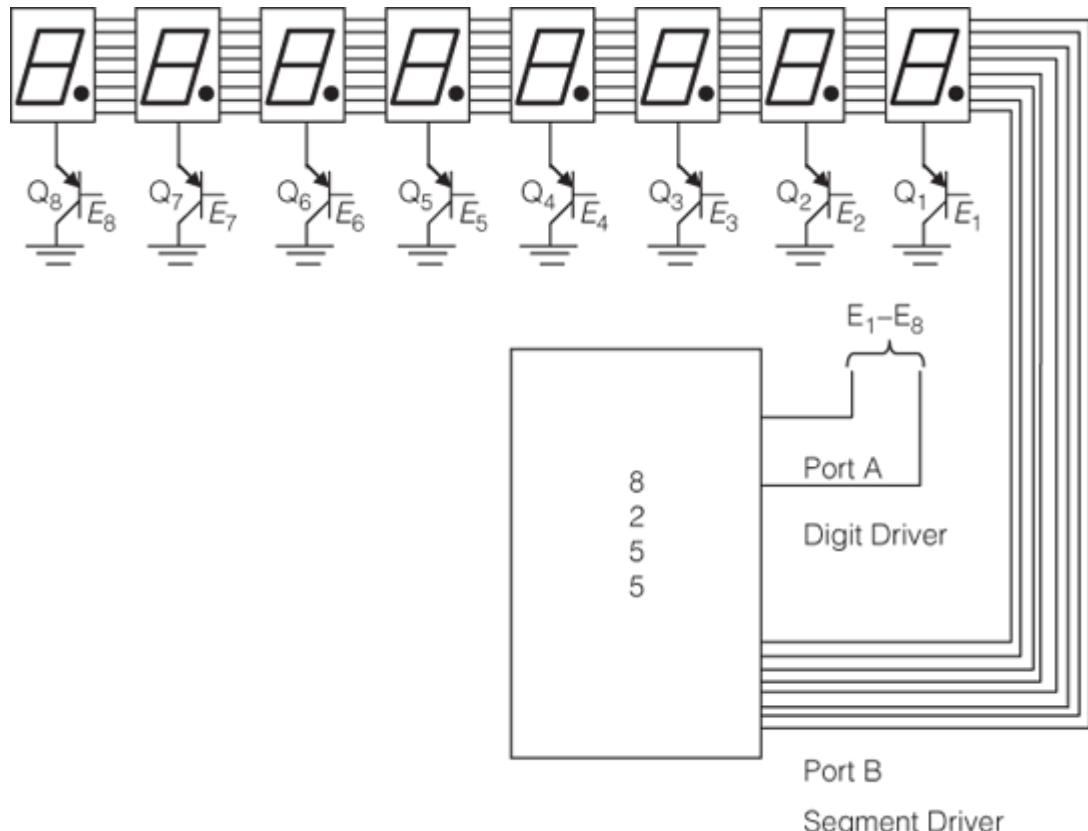


Figure 9.44 | Eight digit dynamic display with segment and digit drivers

1. Select the first digit to be displayed and send a suitable word to Port A to activate this digit.
2. Send the segment code through Port B.
3. Call a delay of, say, 1.5 msec.
4. Switch off this digit and repeat steps 1 to 3.
5. Repeat this sequence for all eight digits.
6. Then, start again from the first step.

With 8 digits and 1.5 msec delay, we can get back to the first digit every 12 msec. This corresponds to a refresh rate of around 83 times a second, which is sufficient to fool the eye into believing that all the digits are ON at the same time.

Now, let us write a program to use the above circuit for displaying data.

Example 9.21

Display the word ‘attitude’ on the 8 LEDs.

Solution

Here, for activating the leftmost digit, the byte 0111 1111 is sent. For the second digit, this sequence is rotated right once. The segment code for each character is stored in the location named TABLE.

```
CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A
PB EQU 0C2H      ;address of Port B

TABLE DB 5FH,78H,78H,04,78H,3EH,5EH,7BH

MOV AL, 80H      ;CW with ports A and B as outputs
OUT CR, AL       ;send it to the control register

STRT: LEA SI, TABLE ;let SI point to the display data
      MOV CX, 8    ;CX to contain the number of digits
      MOV BL, 01111111 ;bit sequence for leftmost digit

AGN:  MOV AL, BL   ;copy it to AL
      OUT PA, AL   ;send it to Port A
      MOV AL, [SI]  ;copy display data to AL
      OUT PB, AL   ;send it to Port B
      CALL DELAY   ;call a delay
      INC SI       ;increment pointer
      ROR BL, 1    ;activate next digit

      LOOP AGN    ;repeat until CX = 0
      JMP START   ;repeat the sequence of display

DELAY PROC NEAR
      MOV DX, 300   ;delay loop
      ;calculate delay value

BACK: DEC DX
      JNZ BACK
      RET
DELAY ENDP

END
```

Example 9.22

Write a program for moving the character ‘U’ from right to left continuously.

Solution

The data corresponding to the character to be displayed is stored in the location DAT. When the first digit is switched on, it displays ‘U’. Next, after a delay, this same character is displayed only on the second position. The important thing here is that the delay between successive activation of the digits should be high enough so that the persistence of vision does not apply. Thus, the letter ‘U’ is seen to move from the left to right continuously on the display. The delay should be very high (compared to the case in Example 9.21) to create such an effect.

```

CR EQU 0C6H      ;address of the control register
PA EQU 0C0H      ;address of Port A
PB EQU 0C2H      ;address of Port B

DAT DB 3EH       ;data to be displayed - 'U'

MOV AL, 80H       ;CW for ports A and B as outputs
OUT CR, AL        ;send it to the control register
STRT: MOV CX, 8    ;CX = 8, the number of digits
           MOV BL, 01111111 ;sequence for one digit to be 'on'
AGN:   MOV AL, BL    ;copy it to AL
           OUT PA, AL    ;send it to Port A
           MOV AL, DAT    ;data of 'U' is loaded in AL
           OUT PB, AL    ;send it to Port B as segment data
           CALL DELAY    ;call delay
           ROR BL, 1     ;rotate BL to turn on next digit
           LOOP AGN     ;repeat for 8 digits, until CX = 0
OK:    JMP STRT    ;repeat the display sequence

DELAY PROC NEAR   ;procedure for a long delay
      MOV DX, 30000
BACK:  DEC DX
      JNZ BACK
      RET
DELAY ENDP

```

This program rolls one character along the display. A very interesting program to try will be to have a message roll from left to right. Suppose we want to rotate the sequence ABCDEFGH, the display will be as shown in the Table 9.10. D₁, D₂ ... shows the digit positions and T₁, T₂ ... corresponds to the time instants at which the corresponding characters are seen on the display. The delay must be large enough to create a feeling of movement. The program for this problem is left as a challenge to interested readers.

Table 9.10 | Display Output for Rolling Display of ABCDEFG

	D1	D2	D3	D4	D5	D6	D7	D8
T1	A	B	C	D	E	F	G	H
T2	H	A	B	C	D	E	F	G
T3	G	H	A	B	C	D	E	F
T4	F	G	H	A	B	C	D	E
T5	E	F	G	H	A	B	C	D
T6	D	E	F	G	H	A	B	C
T7	C	D	E	F	G	H	A	B
T8	B	C	D	E	F	G	H	A

KEY POINTS OF THIS CHAPTER

- A ‘trainer kit’ with an 8086 and associated circuitry helps us to understand the ideas of hardware interfacing to the 8086.
- For interfacing the processor to various peripherals, specialized peripheral controller chips are used.
- One chip which acts as an intermediary in parallel data transfer is the programmable peripheral interface (PPI) chip 8255.
- The PPI has three 8-bit ports A, B and C, which are programmable as input/output ports.
- The PPI has three modes of operation.
- Mode 0 is the simple I/O mode, while the modes 1 and 2 are handshaking modes.
- The parallel port of the PC uses mode 1 of the 8255 for all parallel data transfer, but the parallel port is getting to be more or less obsolete.
- An analog to digital converter can be interfaced to the 8086 using an 8255 as the intermediary.
- A DAC can be used to get analog waveforms generated by the 8086.
- LCDs are very popular display devices. Data and command words can be sent to its internal registers to get displays in different formats.
- Stepper motors are used to get precise and controlled rotation. They can be fed pulse sequences from the 8086 channeled through the PPI.
- The stepper motor needs more current than can be supplied by the 8255. Hence, current driver transistors /ICs are used in such circuits.
- For understanding how keyboards operate, a hex keyboard can be used.
- LEDs are popular displays, but they draw a lot of current. Hence, seven segment digit LEDs are used in a multiplexed fashion.

QUESTIONS

1. What is the special feature of Port C when used in the following modes?
 1. Mode 0

2. Mode 1

2. What does the term handshaking mean to you?
3. What is the importance of the BSR mode of the 8255?
4. Distinguish between the SPP and EPP protocols.
5. How can we operate an ADC in the interrupt mode?
6. How is the current output of a DAC converted to a voltage?
7. How can a sine wave be generated by a DAC?
8. What is meant by a backlit LCD?
9. What is the function of the BUSY pin of an LCD module?
10. How do we make the LCD display move from left to right?
11. Name a few applications of the stepper motor. Why cannot dc motors be used here?
12. Distinguish between the full step drive and half step drive sequences of the stepper motor driving sequences.
13. Why cannot we directly connect the stepper motor from the port pins of the 8255?
14. Why are optocouplers popular in stepper motor circuits?
15. What is meant by the step angle of a stepper motor?
16. Why are dynamic displays preferred in multi-digit displays?
17. Distinguish between common anode and common cathode displays.

EXERCISES

1. Design the control word of the 8255 with:
 1. all ports as input ports,
 2. Port A and B as input and port C as output.
2. Write the BSR control word to set PC3 and PC2 and clear PC5 and PC4.
3. With a 5 MHz clock, write a program to get a symmetric square wave from port B of the 8255.
4. Write a program to get an asymmetric square wave of 20% duty cycle from port pin PC2.
5. Write the control word for mode 1 operation with Port A in strobed output and Port B in strobed input.
6. Write instructions to test OBF_B and IBF_B for the previous problem.
7. Find the name of any standard 12-bit ADC and list its control signals. Draw the figure showing how it can be connected to an 8086 through a parallel port.
8. Interface a DAC to the 8086 with a clock frequency of 5 MHz and write programs to generate
- o. a triangular waveform of T = 1 msec,
 1. a ramp of T = 3 msec,
 2. a staircase waveform of 6 steps each of time period 0.2 msec.
9. Use a 16 × 2 LCD module and write programs to do the following:

o. Display HELLO and WORLD on different lines and move the entire display back and forth.

1. Write four strings on different lines of the LCD which move.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

10

Peripheral Interfacing-II

IN THIS CHAPTER, YOU WILL LEARN

- The architecture and programming features of the timer chip 8253/8254.
- The method of generating waveforms of different frequencies using this chip.
- The use of the different modes of operation of the timer chip.
- The use of a dedicated chip 8279 for keyboard and display interfacing.
- The connections between the 8279 and the display circuits.
- How to use this chip in a circuit with a keyboard and a display.
- How interrupts are managed by the programmable interrupt controller chip 8259.
- The use of initialization and operational control words of 8259.
- The method of changing the priority settings using this chip.
- The use of 8259 in PCs.

Introduction

In Chapter 9, we used the parallel port interfacing IC 8255 for many applications. This was done to illustrate the fact that data is available for transfer in the parallel form so it can be sent to various external peripherals, each of which has specific functions. In all these - applications, the 8255 PPI mostly was used as a data transfer

intermediary, along with the capability of generating signals for controlling the individual peripherals.

In this chapter and the next, we will familiarize ourselves with a few more peripheral chips with specialized applications. The first is the 8253/8254 timer chip.

10.1 | The Programmable Interval Timer 8253/8254

So far, whenever we wanted to create delays, we used the idea of software delay, wherein the processor was kept in a loop repeatedly executing a set of instructions, thus creating a delay loop. This idea was extended to create square waves of required frequencies.

However, this is not a very good solution – first, because the processor is held up in this loop unable to do anything else, and second, delays so created are not very accurate.

The use of dedicated hardware for generating delays and generating waveforms of required frequencies and pulse widths has solved the above problems.

The chip 8253 is a programmable timer chip used for timing applications. Another chip 8254 is also available, which is the same as 8253 except that it is capable of being operated at higher frequencies. In fact, 8254 is a superset of 8253, and all programs written for 8253 can be used for 8254. Both these timer chips provide three independent 16-bit timers, but 8253 can handle clock inputs of only up to 2.6 MHz, while 8254 can be used up to 10 MHz. Both chips are available as 24-pin DIP, and the pin diagram is as shown of 8253/8254 in Fig. 10.1. In the IBM PC, the 8253 was used initially, but now in the PC-AT, the timer chip used is the 8254 which is pin compatible with 8253.

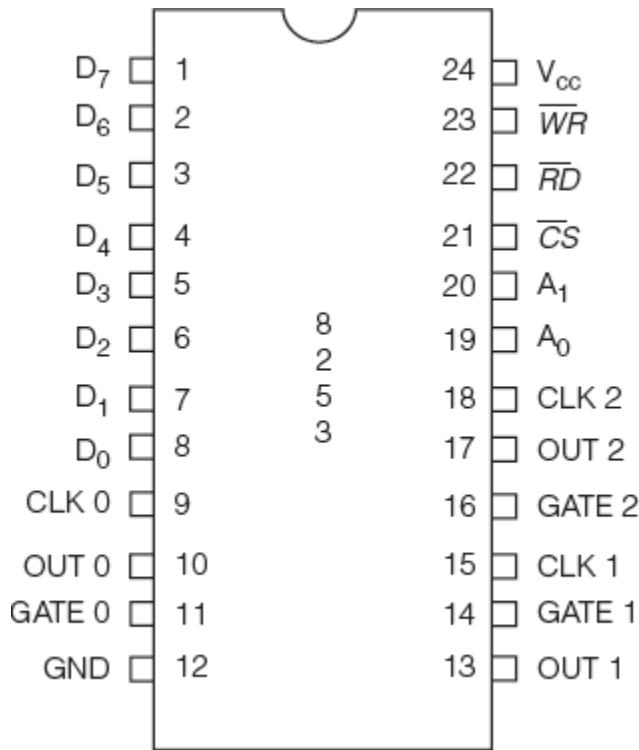


Figure 10.1 | Pin diagram of 8253/8254

Now, let us have a good look at the internal details of the chip. As the internal block diagram shows, there are three independent counters and the operation of these is managed by a control register. For selecting these four entities, two address pins A_1 and A_0 and the chip select line CS are used as shown in Table 10.1.

Table 10.1 | Address of the Control Register and Counters

CS	A_1	A_0	Selected entity
0	0	0	Counter 0
0	0	1	Counter 1
0	1	0	Counter 2
0	1	1	Control register

10.1.1 | Counters

The block diagram (Fig. 10.2) shows that each counter has three pins associated with its operations – CLK, GATE and OUT.

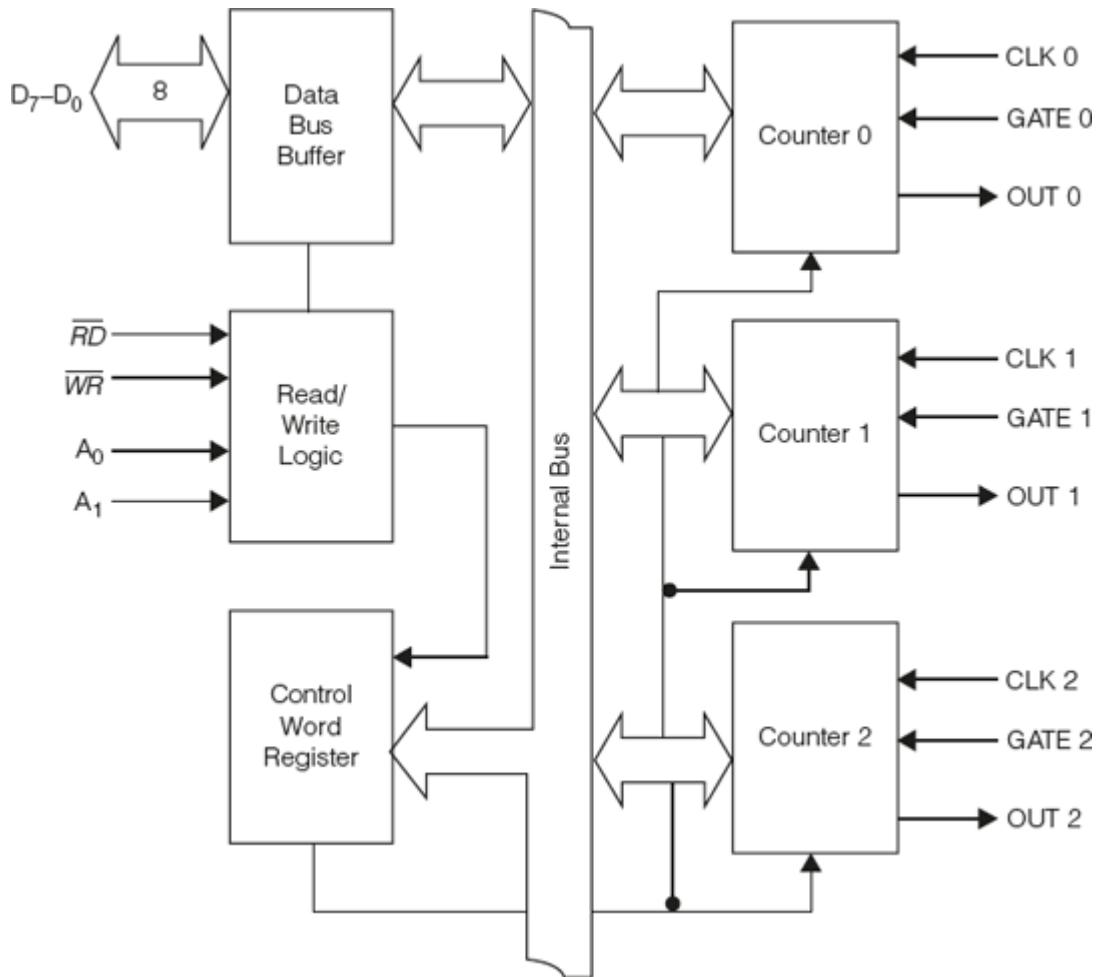


Figure 10.2 | Functional block diagram of 8253/8254

Clk

This pin corresponds to the pin on which an input frequency is received, and all delays are calculated based on this frequency f . $T = 1/f$ is the time period of this input frequency and we will use T in all our calculations. This frequency should not be higher than 2.6 MHz for 8253 but can be as high as 10 MHz for the 8254.

Gate

This functions as a control pin for each of the timers. This has to be high (Gate = 1) in most of the modes of operation but is used as a trigger pin in some modes and then it should be a 0 to 1 pulse.

Out

Each counter has an output pin and the waveforms generated by programming the specific counter can be observed at this pin.

Data and Control Pins

The other pins of the chip are the data pins D_0 to D_7 and the RD and WR pins. The data pins are connected to the lower order data pins of the processor. The read and write control pins are activated by the I/O read and I/O write cycles, respectively, of the processor.

10.1.2 | Programming the Chip

All operations are decided by the control word loaded into the control register. For each of the counters, there is a count-register, which is 16 bits in size. A number is written into this register and stored in the counter block. The maximum size of this number is FFFFH, i.e., 16-bit, but since the counter's internal register is only 8 bits long, this number must be written as two 8-bit chunks. The operation of the counter occurs by creating a delay by decrementing this number down to 0 – the rate at which this occurs depends on the input clock frequency. Hence, this number decides the factor by which the input frequency gets divided to give an output frequency.

Suppose the number loaded is 3. Then, during each input time period ' T ', the number decrements to 2, 1 and then 0. Thus, it creates a delay to divide the input frequency by 3. This is the way with any number ' N ' loaded in the counter register of a particular counter. The maximum delay is obtained by loading '0' as the number – it will decrement to 65,535 and then back to 0. Figure 10.3 gives an idea of what constitutes the hardware associated with a **single counter**. The control register is common to all the counters, but the other blocks are replicated for each counter. CR stands for count-register and the subscripts H and L are for high byte (MSB) and low byte (LSB), respectively. CE is the counting element and OL stands for output latch. This block diagram will come in handy for understanding the procedure for reading the content of a - count-register. This is discussed in Section 10.1.13.

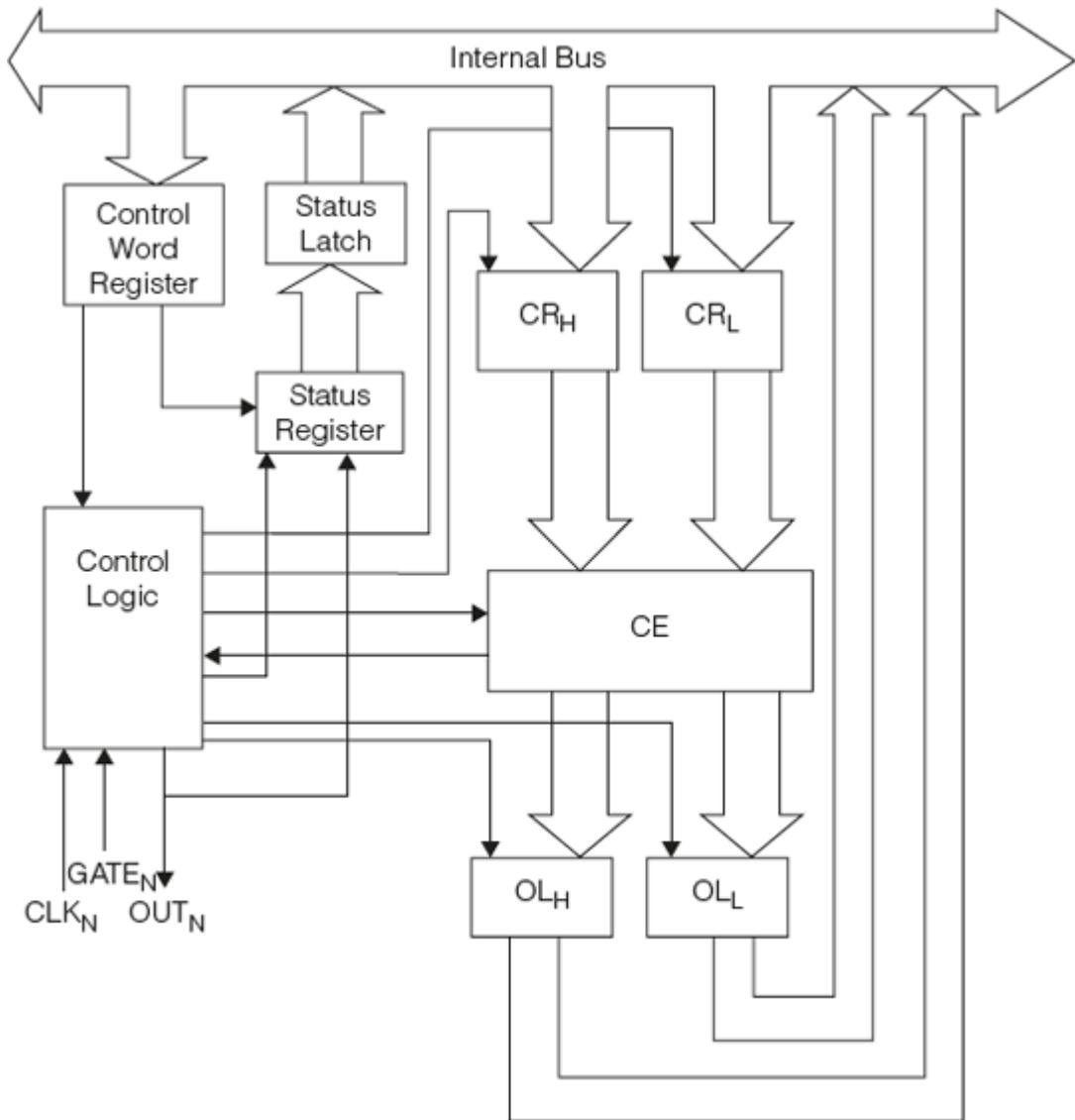


Figure 10.3 | Internal architecture of a counter

10.1.3 | Control Word

Figure 10.4 shows the control word and the definition of each bit of the word. D₇ and D₆ (SC1 and SCo) are used to select the counter intended to be used. D₅ and D₄ (RLo and RL1) point to 4 options. One option is 'latch'. This will be needed only when we want to 'read' the contents of the count-register. The other three options are for writing a number 'N' into the count-register. This number can be 8 or 16 bits long. There might be confusion about why there are three ways of loading the count in the counter. One possibility is that the count will be two bytes in size. In that case, use RL1, RLo = 11 in the control word and then load the count. So, load the LSB first, followed by the MSB. Later, if the LSB byte alone needs to be changed, then the control word should be changed, and reloaded with RL1 and RLo as 01 or as 10 if the MSB alone need to be changed. After configuring the control word in this way, the count can

be loaded in the count-register. D₃ to D₀ (M2 to M1) select the mode in which the counter is to be used. The chip has 6 modes, which will be explained in detail soon. D₀ is the bit which informs the counter of the format of the ‘count’. If it is BCD in form, this bit is set to be 1; if binary, it is ‘0’. In binary, the count can be from 0 to FFFFH – in BCD, the number can vary only from 0 to 9999.

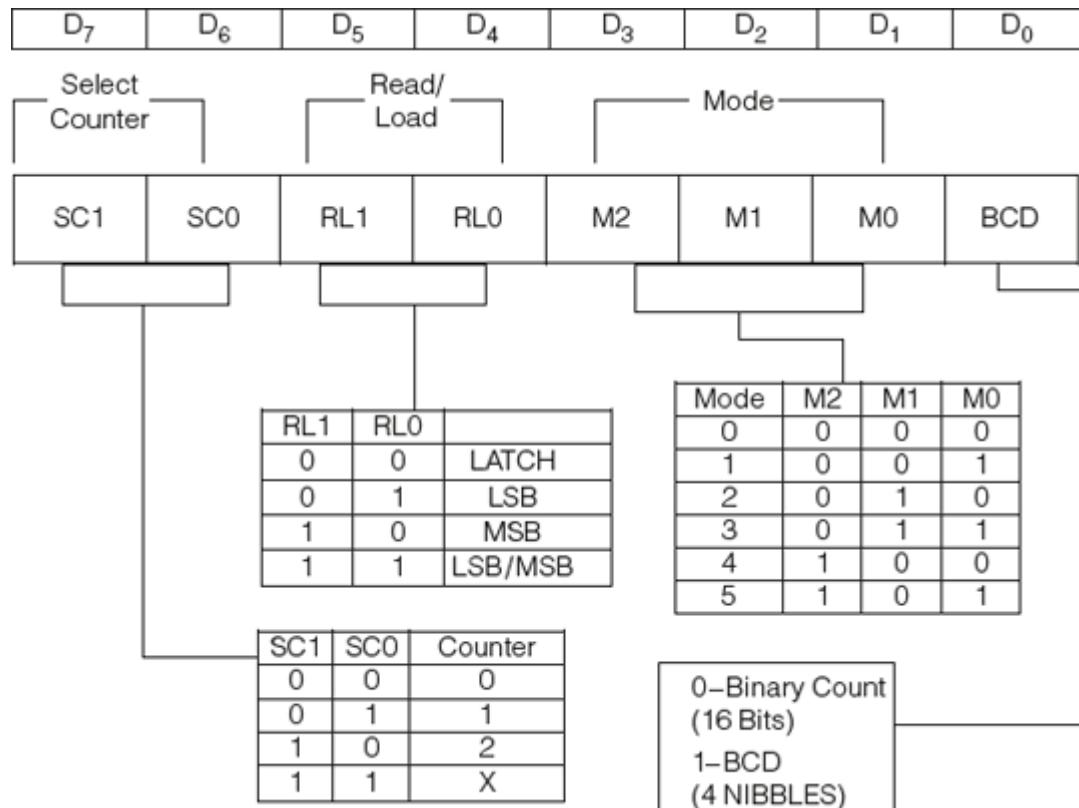


Figure 10.4 | Control word format of the 8253 / 8254

Recollect that the maximum BCD count can only be 9999, which is to be written as 9999H (refer Section 0.6.4), while the maximum binary count can be FFFFH.

Example 10.1

Configure the control word for the following cases:

1. Counter 0 in mode 3, count in BCD, specify that a 16-bit count is written with LSB first.
2. Counter 2 in mode 0, count in binary format, count loaded as in the previous case.

Solution

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	1	1	0	1	1	1

The control word is written referring to Fig. 10.4 and is 37H.

Solution

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	1	1	0	0	0	0

The control word in this case is BoH.

Example 10.2

Find the addresses of the control registers and the three counters for the decoding logic shown (Fig. 10.5).

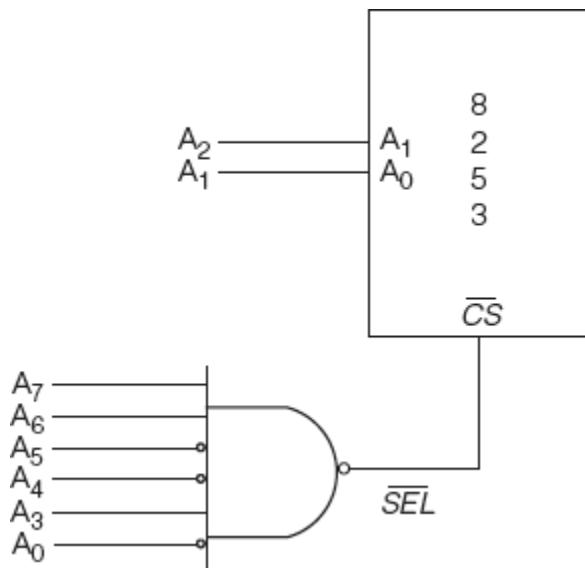


Figure 10.5 | Decoding logic for the chip

Solution

In the circuit, we see that the address lines A₂ and A₁ of the 8086 are connected to A₁ and A₀ of the 8253. The other address lines A₇ to A₃ and A₁ are used for address decoding as shown. The address of various entities within the chip are as follows:

Entities	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	Address (Hex)
Control Register	1	1	0	0	1	1	1	0	CE
Counter 0	1	1	0	0	1	0	0	0	C8
Counter 1	1	1	0	0	1	0	1	0	CA
Counter 2	1	1	0	0	1	1	0	0	CC

Now, let us use the timer chip in its various modes. The different modes are listed as:

Mode No.	Mode bits			Operation
	M2	M1	M0	
0	0	0	0	Interrupt on terminal count
1	0	0	1	Programmable one shot
2	X	1	0	Rate generator
3	X	1	1	Square wave generator
4	1	0	0	Software triggered strobe
5	1	0	1	Hardware triggered strobe

To get a feel about the use of the timer chip, we will first use it in Mode 3, which is the simplest and direct of the modes. Mode 3 is used to generate square waves of a desired frequency.

Example 10.3

The input frequency to the timer chip is 1.5 MHz. Generate a square wave of 1.5 KHz using Timer 0.

Solution

For an input frequency of 1.5 MHz, $T = 1 / (1.5 \times 10^{-6}) = 0.66 \mu \text{secs}$

To get an output frequency of 1.5 KHz, the time period = 0.66 msec

N is the number of clock cycles of the input frequency to get 0.66 msecs time period for the output signal.

$$N = 0.66 \text{ msecs} / 0.66 \mu \text{secs} = 1000 \text{ cycles}$$

Thus, the number to be loaded in the count-register is 1000; i.e., 1000 is the number that divides 1.5 MHz to get 1.5 KHz. What this means is that when the counter starts running, this number decrements down to zero, and 1000 cycles of the input signal occur during this period. Before

writing the program, the control word is to be designed (refer Fig. 10.4). The control word is 0 0 1 1 0 1 1 0, i.e., 36H.

This word selects counter 0 and uses mode 3. It also loads the count-register with LSB first and then the MSB. The count is loaded as a binary number. Hence, D₀ is '0'.

Program

```

CR      EQU  0CEH
CNT0   EQU  0C8H

MOV  AL, 36H    ;control word
OUT  CR, AL    ;send control word to control register
MOV  AX, 3E8H    ;load count = 1000 in AX
OUT  CNT0, AL    ;send lower byte of count to counter 0
MOV  AL, AH    ;move higher byte of count to AL
OUT  CNT0, AL    ;send higher byte of count to counter 0
END

```

The output signal from the terminal OUT 0 divides the input frequency by 1000. It is seen that the input frequency is divided by N in this mode. For this mode, if the number N is even, the high and low portions are of the same period. If N is odd, the high pulse is $(N + 1) / 2 \times T$ and the low pulse is $(N - 1) / 2 \times T$. Note that T is the period of the input frequency. It is mandatory to keep GATE = 1 for the counter to run. We load a number into the count-register which decrements down to 0. After that, this number gets re-loaded into the count-register to generate a continuous square wave, as shown in Fig. 10.6.

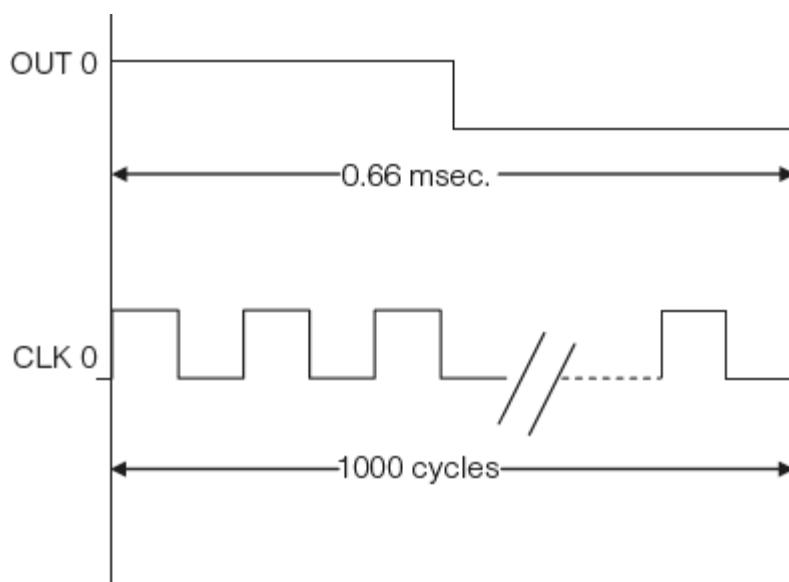


Figure 10.6 | Output waveform corresponding to Example 10.3

10.1.4 | Using all the Counters

Note There are three channels in this chip, i.e., three different counters with individual clocks, gates and output pins. Hence, three different signals can be obtained from the chip simultaneously. If, say, we need three square waves of different frequencies, we can program all the three counters in mode 3. Try the following program.

Example 10.4

Generate three different frequencies from the 8253 timer. Use a clock frequency of 1.5 MHz for all the counters.

Solution

```
CR    EQU  0CEH ;address of control register
CNT0 EQU  0C8H ;address of counter 0
CNT1 EQU  0CAH ;address of counter 1
CNT2 EQU  0CCH ;address of counter 2

MOV  AL, 37H   ;control word to use counter 0
OUT  CR, AL   ;send control word to control register
MOV  AX, 10H   ;load count = 10 in AX
OUT  CNT0, AL  ;send low byte of count to counter 0
MOV  AL, AH    ;move high byte of count to AL
OUT  CNT0, AL  ;send high byte of count to counter 0

MOV  AL, 77H   ;control word to use counter 1
OUT  CR, AL   ;send control word to control register
MOV  AX, 100H  ;load count = 100 in AX
OUT  CNT1, AL  ;send low byte of count to counter 1
MOV  AL, AH    ;move high byte of count to AL
OUT  CNT1, AL  ;send high byte of count to counter 1
MOV  AL, 0B7H   ;control word to use counter 2
OUT  CR, AL   ;send control word to control register
MOV  AX, 1000H  ;load count = 1000 in AX
OUT  CNT2, AL  ;send low byte of count to counter 2
MOV  AL, AH    ;move high byte of count to AL
OUT  CNT2, AL  ;send high byte of count to counter 2

END
```

With this program, counter 0 divides the input frequency by 10, counter 1 by 100 and counter 2 by 1000. **We have used the BCD of bit D₀ (= 1) of the control word and so used the numbers 10H, 100H and 1000H as the count for the three cases.** It should be clear that to write BCD in hex, 1000 should be written as 1000H, 100 as 100H and so on (refer Section 0.6.4). To use the binary option, we

should write the counts as 0AH, 64H and 3E8H after ensuring that the D₀ bit in the control word is made ‘0’ for each of the cases. The control words of the three counters would then be 36H, 76H and B6H.

Now, let us examine all the modes of the timer chip. Keep in mind that associated with each of the three counters, there are three pins – the clock, the gate and the out. The gate is a control input. In some modes, it is mandatory to keep GATE = 1, but in certain other modes, the GATE should have a low to high transition for the operation in that mode to take effect.

10.1.5 | Mode 0: Interrupt on Terminal Count

This can be used to generate an event after a programmed time period. GATE = 1 should be maintained for this mode to function. What we need to do to use in this mode is to load a number N in the count-register. The output is normally low and will go high after the terminal count is reached, i.e., when N rolls down to 0. The width of the low pulse is $N \times T$, where T is the duration of the input clock pulse. This mode can be used to interrupt the microprocessor after a delay period.

Note If, in between, the GATE pin goes low, counting will be stopped and the low duration will be extended by the amount of time that the gate is kept low.

Example 10.5

Use counter 0 in mode 1 to generate a delay of 10 msec. After this delay, the OUT 0 pin must go high. The clock frequency used for the counter chip is 1.5 MHz.

Solution

Calculation to generate a delay of 10 msec:

$$T = 10 \text{ msec} / (0.66 \mu\text{s}) = 15151 = 382 \text{ FH}$$

Referring to Fig. 10.4, write the control word for mode 0.

CW is 00110000.

```

CR      EQU 0CEH ;address of control register
CNT0    EQU 0C8H ;address of counter 0

MOV AL, 30H          ;control word for mode 0, counter 0
OUT CR, AL           ;send it to control register
MOV AX, 382FH         ;load count in AX
OUT CNT0, AL          ;send LSB of count to counter 0
MOV AL, AH             ;move AH to AL
OUT CNT0, AL          ;send MSB of count s to counter 0
END

```

The output will become high after 10 msec.

This mode can be used as an event counter and to signal a processor when the number of events has reached a predefined number. Suppose it is needed to have a signal when 100 bottles have passed through a conveyor belt. An optical sensor can sense this and generate a pulse for each bottle, and the output of the sensor is fed to the clock input of counter 0 in mode 0. A count of 64H is loaded in the count-register. When the counter counts down to 0, the OUT pin of the counter goes high, and this signals the event of 100 bottles passing the conveyor belt. If the OUT pin is connected to the INTR pin of the 8086, appropriate action can be programmed in the interrupt service routine of the processor. Figure 10.7 illustrates this.

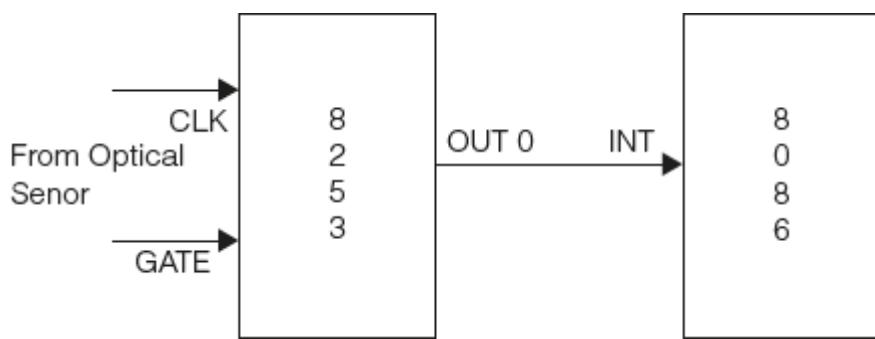


Figure 10.7 | An example of a practical application of mode 0

10.1.6 | Mode 1: Programmable One Shot

This mode is also called the **hardware re-triggerable** one-shot. After the count-registers are loaded, the GATE must be given an L to H transition, which constitutes the trigger for the one-shot. Once the trigger is obtained, the OUT pin goes low and remains low until $N \times T$, where N is the count loaded in the count-register and T is the period of the input clock. Meanwhile, if the GATE is given another L to H transition, the count is re-loaded and counting starts again. **This re-**

triggering capability is what gives it the name of a ‘re-triggerable one shot’. Refer Figs 10.8a and 10.8b.

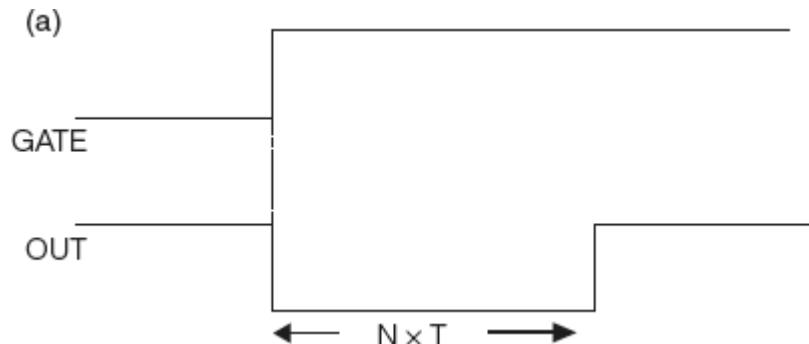


Figure 10.8a | Output waveform corresponding to mode 1

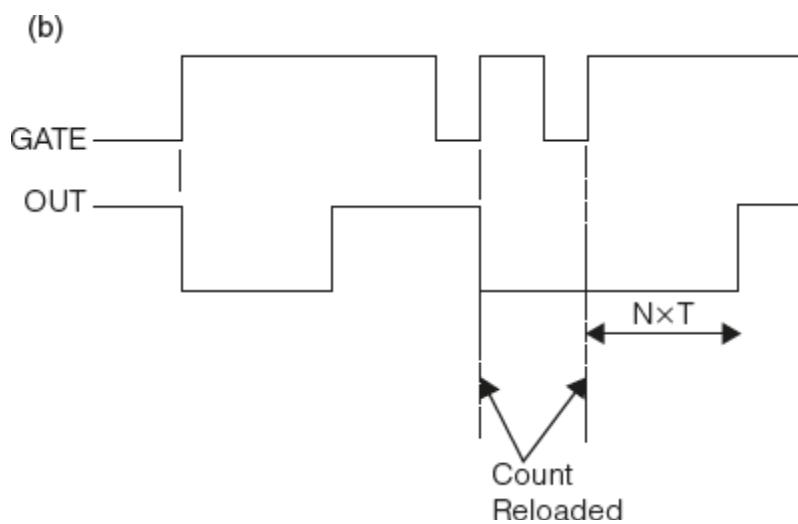


Figure 10.8b | Output waveform when a new L to H transition is given to the gate before $N \times T$ is reached

Example 10.6

Consider a gate input waveform of 1 KHz and clock input of 1.5 MHz ($T = 0.66$ msec).

The following program gives a low output pulse of 0.2 msec from the output of counter 0 only when the gate is given a low to high transition. To calculate the count value, $N \times 0.66 \mu\text{secs} = 0.2 \text{ msec}$. $N = 303 = 12 \text{ FH}$.

Apply a 1 KHz signal at the gate input and draw the output waveform at OUT o. The low pulse is not produced when the count is loaded into the count-register – it starts only when a low to high transition occurs on the gate.

Solution

Control word is 0011 0010 - 32H

```
CR EQU 0CEH ;address of control register  
CNT0 EQU 0C8H ;address of counter 0  
  
MOV AL, 32H      ;control word for mode 1, counter 0  
OUT CR, AL      ;send it to control register  
MOV AX, 12FH      ;move count to AX  
OUT CNT0, AL      ;send LSB to counter 0  
MOV AL, AH      ;move MSB to AL  
OUT CNT0, AL      ;send it to counter 0  
END
```

Figure 10.9 shows the output wave form.

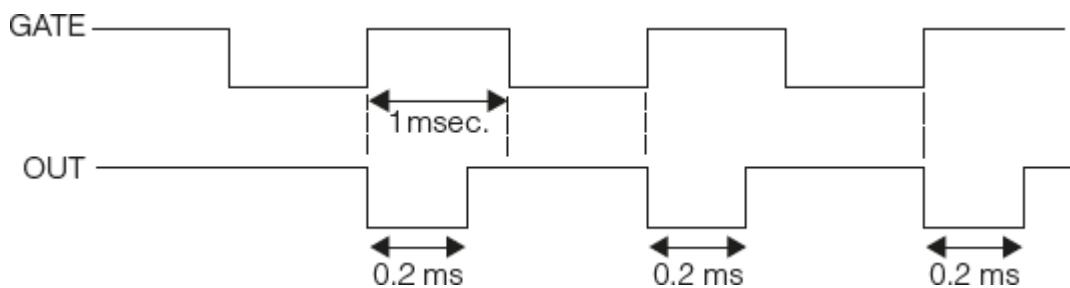


Figure 10.9 | Output waveform of Example 10.6

10.1.7 | Mode 2: Rate Generator

Here, if GATE = 1 and a count is loaded into the count-register, the output will be high for the duration $N \times T$ and will then go low for a period of one clock cycle. Thus, the period of the output waveform can be considered to be $(N + 1) T$, where it is high for the duration $N \times T$ and low for one cycle. The count is reloaded automatically and the counter continues to produce the output waveform. This mode is also called a divide by N counter. In Fig. 10.10, WR corresponds to the time in which the count is written into the counter.

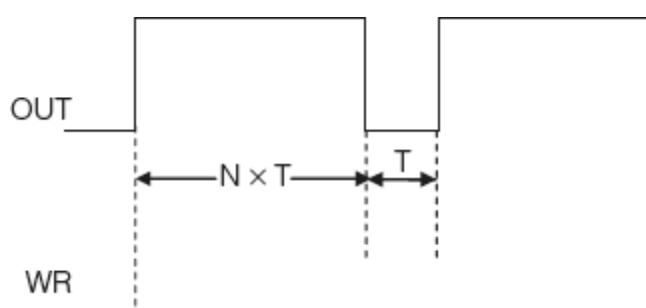


Figure 10.10 | Output waveform corresponding to mode 2 operation

Example 10.7

```

CR      EQU  0CEH ;address of control register
CNT0    EQU  0C8H ;address of counter 0

MOV AL, 35H          ;control word for counter 0, mode 2
OUT CR, AL           ;send it to control register
MOV AX, 0005          ;load count N = 5 in AX
OUT CNT0, AL          ;send LSB to counter 0
MOV AL, AH            ;move MSB to AL
OUT CNT0, AL          ;send MSB to counter 0
END

```

This program generates a low pulse of one T at OUT o after a period of 5 T cycles.

Example 10.8

Generate a square wave of 10 KHz from counter 0 and use counter 2 to get an asymmetric output waveform.

Solution

Apply a clock of 1.5 MHz to CLKo. Connect OUT o to CLK2 pin, and observe the output at OUT2 pin.

```

CR      EQU  0CEH ;address of control register
CNT0    EQU  0C8H ;address of counter 0
CNT2    EQU  0CCH ;address of counter 2
MOV AL, 37H          ;control word to use counter 0, mode 3
OUT CR, AL           ;send control word to control register
MOV AX, 0150H          ;load count in AX
OUT CNT0, AL          ;send low byte of count to counter 0
MOV AL, AH            ;move high byte of count to AL
OUT CNT0, AL          ;send high byte of count to Counter 0
MOV AL, 0B4H          ;control word to use counter 2 in mode 2
OUT CR, AL           ;send control word to control register
MOV AX, 0003          ;load count in AX
OUT CNT2, AL          ;send low byte of count to counter 2
MOV AL, AH            ;move high byte of count to AL
OUT CNT2, AL          ;send high byte of count to counter 2
END

```

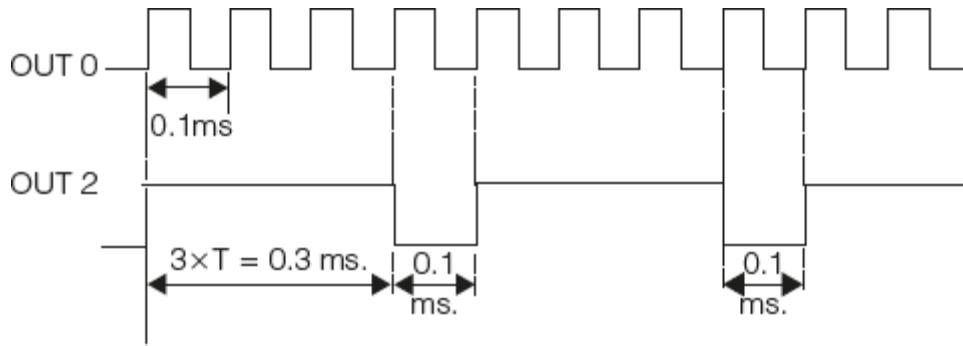


Figure 10.11 | OUT 0 and OUT 2 outputs

The signal from OUT 0 is a symmetric square wave obtained by dividing CLK0 (1.5 MHz) by 150. Its frequency is 10 KHz and its T is 0.1 msecs. Counter 2 is programmed to run in mode 2 with a count of $N = 3$. Its input frequency is 10 KHz. It will generate an output signal which is high for $3T$ (0.3 msecs) and low for one period of its input clock, i.e., 0.1 msec. Hence, an un-symmetric output waveform is obtained, as shown in Fig. 10.11.

10.1.8 / Mode 3: Square Wave Generator

This mode has been discussed earlier in Example 10.3 and Section 10.1.4.

10.1.9 / Mode 4: Software Triggered Mode

In this mode, if GATE = 1, the output will be high for the duration $N \times T$ and will go low for one clock period. It then goes high again. Repetition of the sequence is possible only after reloading the count-register. This mode is similar to Mode 2, except that, the counter is not re-loadable here.

10.1.10 / Mode 5: Hardware Triggered Mode

In the previous mode, counting started when the count-register was loaded. However, here the counting starts only when an L to H transition is given to the GATE. In Fig. 10.12, WR is the time at which the count is written into the count-register.

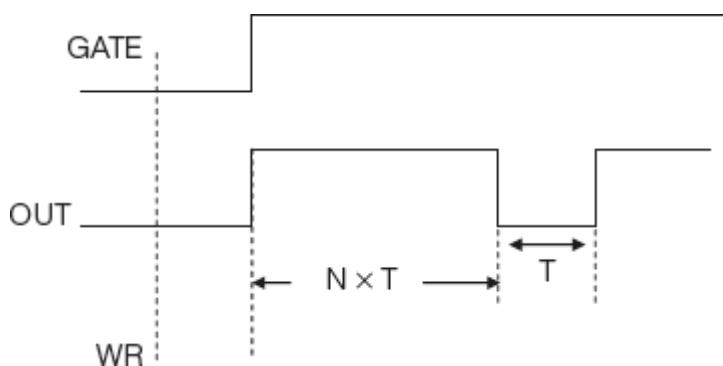


Figure 10.12 | Output waveform for mode 5, starting from the L to H transition of the GATE

10.1.11 / Reading a Count

In many applications, it may be important to read the contents of the count-register. This can be done by stopping the counting. To stop the counting, the method is to make the GATE pin equal to 0 or to inhibit the clock. However, the better method is to do the reading ‘on the fly’, which does not cause any interference on the circuit operation. Referring to Fig. 10.3 shows that there is a latch associated with the counting element. To read a stable value from a counter, the method is to latch the count and then read the content of the latch. Refer to Fig. 10.4 which shows the control register format – RL1 and RL0 being 0 gets the count to be latched. After that, the contents can be read. The following are the steps for reading the count of counter 0. This program segment can be used anywhere in the program once counting starts, which means that it performs reading ‘on the fly’.

```
MOV AL, 00000000 ;control word for latching counter 0
OUT CR, AL        ;send it to control register
IN AL, CNT0       ;input the LSB of counter 0
MOV AH, AL        ;move it to AH
IN AL, CNT0       ;input the MSB of counter 0
XCHG AH, AL       ;exchange the contents of AH and AL
MOV COUNT, AX      ;save the count in memory
```

Now, the current count is available in AX, which is to be moved into COUNT, a memory location.

10.2 | The Programmable Keyboard Display Interface – 8279

In Chapter 9, you had an exposure to the techniques of interfacing keyboards as well as displays to the 8086. What should have been noticed is that when a key is expected to be pressed, the processor is fully involved in verifying this and then identifying the key that has been pressed. Similarly, for displaying data in multiplexed LED displays, the processor is fully involved in this work of refreshing individual digits, sending segment information and so on.

We would like to have a peripheral chip which will take care of these activities, such that the processor has only to give commands for the actions. One chip which takes up the responsibility of managing these functions is the 8279 keyboard display interface. This chip has two

portions – one for providing scanned display interface for LED, incandescent and other popular display technologies in alphanumeric form and the other for interfacing with keyboards or an array of sensors.

We will discuss the interfacing of keyboards and displays separately and then use them jointly. Our approach will be to discuss the features of the chip with the help of an example in which the 8279 is interfaced to a keyboard as well as a display. Figure 10.13 shows the functional block diagram of the chip. BD is used to blank the display during digit switching or by a display blanking command. We will discuss the important pins when we use them in practical circuits, as we go ahead.

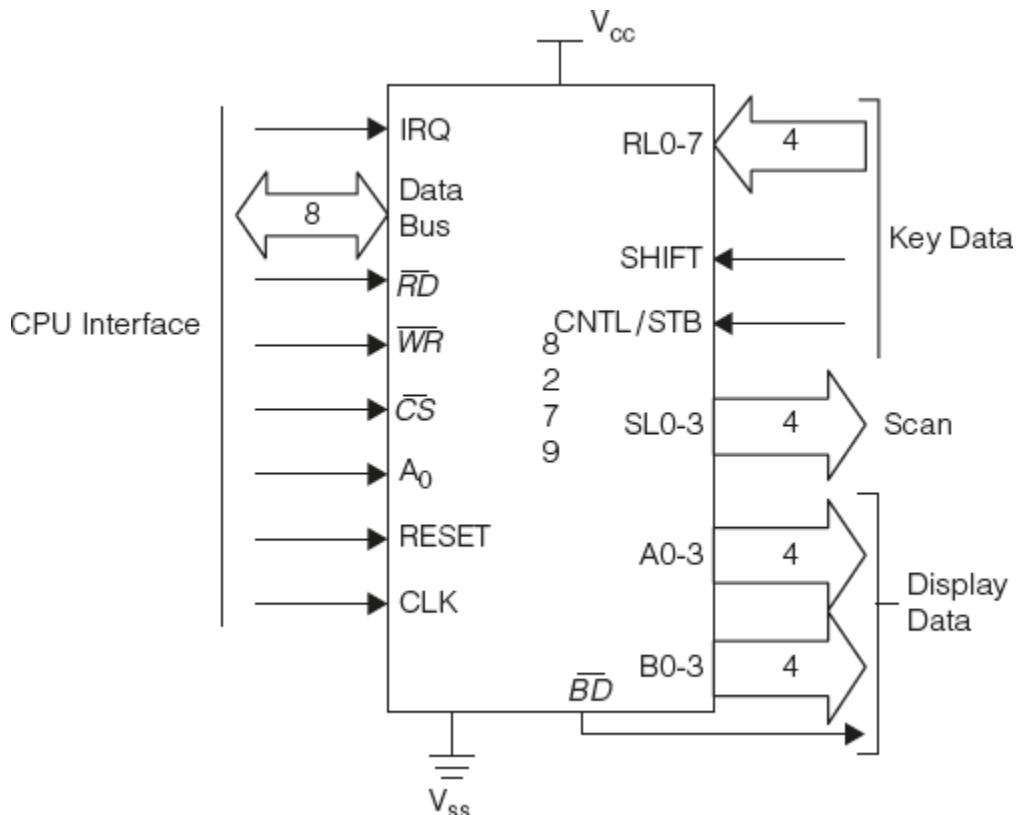


Figure 10.13 | Functional pin diagram of the 8279

10.2.1 / Display Interface

Let us consider using the chip for interfacing to a multiplexed seven segment LED of 8 digits. This will help to understand the use and capabilities of the chip. Refer to Fig. 10.14. However, do not be intimidated by the seemingly complicated diagram. It is actually very simple and easy to understand.

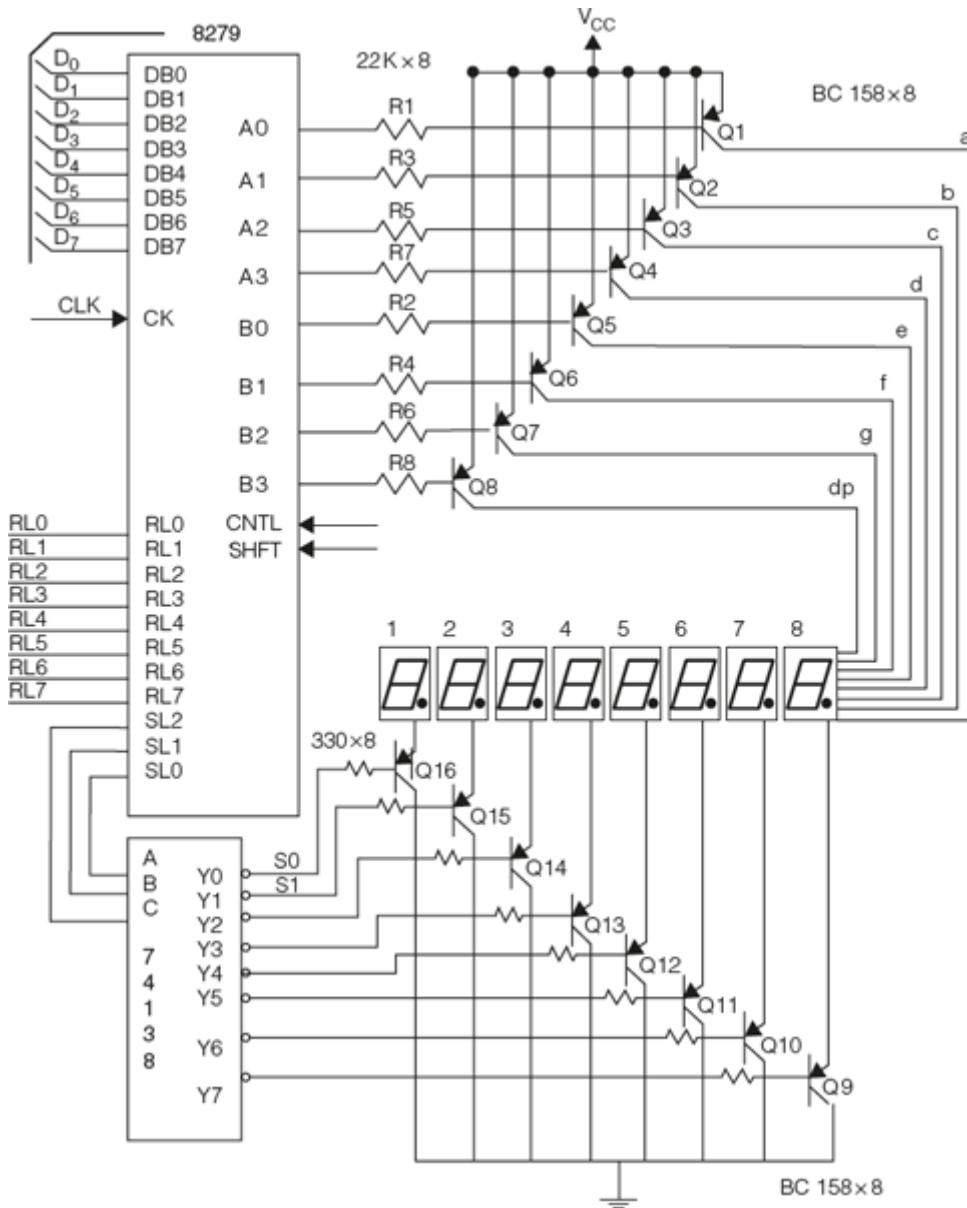


Figure 10.14 | Display interface part of the 8279

The circuit is for using the 8279 for interfacing an 8-digit, multiplexed dynamic display. As discussed in Section 9.13.2, for a multiplexed display, two signal sequences are required.

1. Only one digit is to be activated at a time, and so for a sequence which switches 'ON', only one of the digit transistors is needed. The scan lines SL0 to SL3 of the 8279 generate a continuous sequence of numbers from 0000 to 1111. In the case that we are discussing, since only eight digits are to be displayed, SL0 to SL2 are connected to a three to eight decoder 74LS138. This gives a low signal only at one of its eight outputs, and thus, each digit transistor is switched on in sequence. Only one of the eight transistors Q9 to Q16 will be ON at a time. The LEDs are connected as common cathode, and when, say, transistor Q9 is ON, it grounds the cathode of the

LEDs of the corresponding digit. So, when segment information is given at the anodes, this digit can display the character corresponding to the segment information.

2. When a particular digit is activated, the segment information corresponding to that digit will get its output from the chip. This is done at the output lines A0 to A3 and B0 to B3 of the chip. For increasing the current driving capability (to give a bright display), transistors Q1 to Q7 are used between the output lines and the segments of the LED. Thus, a segment gets lighted up only when a 'high' appears on the collectors of the driving transistors (note that the transistors are PNP type). **For that to happen, the output lines A0 to A3 and B0 to B3 must be low – which means that for a segment to be lighted up, the bit from the 8279 must be a '0'.** However, note that this is not a feature of the 8279; rather, it is due to the connections used between the chip and the display.
3. The chip has a 16-byte display RAM into which the seven segment codes of the display characters are to be written. The chip automatically cycles through a process of sending out the code for one digit, turning on the digit for a short time and then repeating the procedure for the next. In between the displaying of one digit and the next, the chip also sends a 'blanking code' which turns off all the segments so as to prevent 'ghosting' of information from one digit to the next.

10.2.2 / Segment Definition

The segment definitions are as shown in fig. 10.15. The table below shows the correspondence between the data bus and the output port bits of the 8279. This definition is only because of the way the lines A0 to A3, and B0 to B3 have been connected in this case.

Data bus	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
8279 outputs	A3	A2	A1	A0	B3	B2	B1	B0
Segments	d	c	b	a	dp	g	f	e

The diagram illustrates the connection of the seven segments (a, b, c, d, e, f, g) and the decimal point (dp) to the data bus lines. The segments are arranged as follows: segment 'a' is the top horizontal bar, 'b' is the right vertical bar, 'c' is the left vertical bar, 'd' is the bottom horizontal bar, 'e' is the bottom-left vertical bar, 'f' is the top-left vertical bar, and 'g' is the middle vertical bar. The decimal point (dp) is indicated by a dot. The connections are summarized in the table above, where D₇ corresponds to segment d, D₆ to segment c, D₅ to segment b, D₄ to segment a, D₃ to segment dp, D₂ to segment g, D₁ to segment f, and D₀ to segment e.

Figure 10.15 | correspondance between the segments and the data bus lines

Example 10.9

What data should be outputted from the 8279 to display the following characters with reference to the connection in Fig. 10.14?

1. A
2. 5
3. blank

Solution

For the connections in Fig. 10.14, for any segment to be lighted, the corresponding bit is to be '0'.

1. For A, the segments to be activated by a '0' are a, b, c, e, f and g. From Fig. 10.15, the corresponding byte is 1000 1000, i.e., 88H.
2. For 5, the segments to be lighted up are a, c, d, f and g. The byte is 0010 1001, i.e., 29H.
3. For no segment to be lighted up, the byte is 1111 1111, i.e., FFH.

10.2.3 / Keyboard Section

The chip has the capability to be used with keyboards as well as with sensor matrices. However, here we will discuss only the former case. Let's consider a matrix keyboard and recollect how a key press is identified (Section 9.12). '0's are sent on one dimension of the matrix and the other dimension is read in – the lines that are read in are called 'return lines (RL0 to RL7 here). However, here we have to send out a binary sequence. That is done by the scan lines SL0 to SL3. There are two options for using these scan lines. There are two kinds of keyboard scans – encoded scan and decoded scan.

Encoded Scan

For the keyboard also, the scan lines SL0 to SL3 of the chip are used. In the encoded scan mode, these lines generate a binary sequence from 0000 to 1111. A decoder is then used to generate a low on only one of its output lines. This is shown as is used in our example.

Decoded Scan

In this case, a decoder is not needed. The pins SL0 to SL3 generate a sequence of 'low' pulses directly on the lines. This can be used if there are only 4 digits to refresh.

To detect a keypress and identify the key, the lines RL0 to RL7 are the ‘return’ lines. There are return buffers inside the chip which buffer and latch the return lines. In the keyboard mode, these return lines are scanned looking for key closures in that row. If a key press is sensed, a debounce circuit (inside the chip) waits for 20 msecs to confirm the keypress. If key closure is confirmed, the address of the key (in terms of its row and column position and status of SHIFT and CONTROL pins) is stored in a FIFO RAM.

Let us understand this by using the keyboard shown in Fig. 10.16. Here, we use a hex-keyboard with 16 keys, besides two keys for CNTL (control) and SHFT (shift). The 16 keys are arranged in two rows and eight columns. The 3 to 8 decoder (used for the display interface as well) gives two signals S0 and S1 (in response to the sequence produced in SL0 to SL2 – ref Fig. 10.14 also), and these two lines are connected to the rows of the keyboard, while the columns are connected to the pins RL0 to RL7 of the chip. The RLs are ‘return’ input lines from the columns to the chip. The idea is that the scan lines cause ‘lows’ to be outputted on the rows and the status of the columns are returned to the chip. Remember that this is the same as the principle of keyboard interfacing discussed in Section 9.12, but here, the 8279 does this automatically in response to the control words we use.

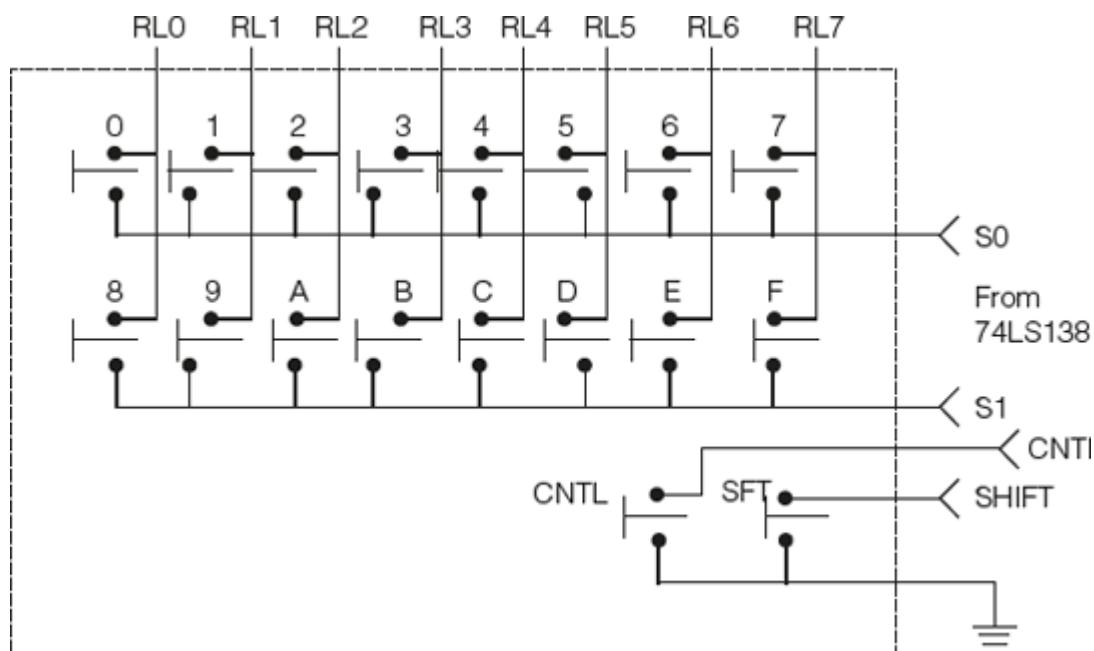


Figure 10.16 | Keyboard section

The return key code which is produced has a format as shown in Fig. 10.17. This code is stored in an eight-byte FIFO RAM, where FIFO stands for ‘First-In First-Out’. This RAM thus stores the key codes in the order in which the keys have been pressed.

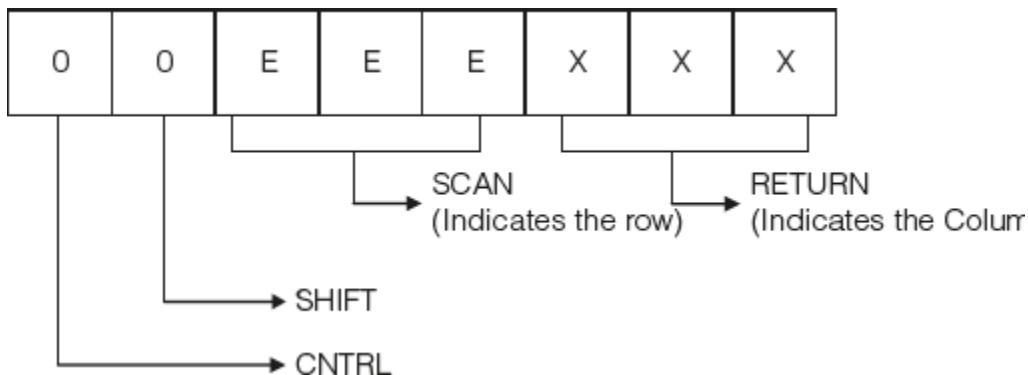


Figure 10.17 | Return key code

A valid keypress also causes an interrupt to be generated on the IRQ line of the chip. This can be connected to the interrupt line of the processor so as to operate the keyboard in an interrupt driven mode. Another thing that a valid keypress causes is the incrementing of a FIFO count in a status register. This information can be used to verify if the FIFO contains a valid key code.

10.2.4 / System with Keyboard and Display Interfaced

Figure 10.18 shows the block diagram of the system that we are using, i.e., 8-digit multiplexed LED display and a 2×8 hex keyboard with SHFT and CNTRL keys. With this minimum information, let us get to the task of programming the chip for our application.

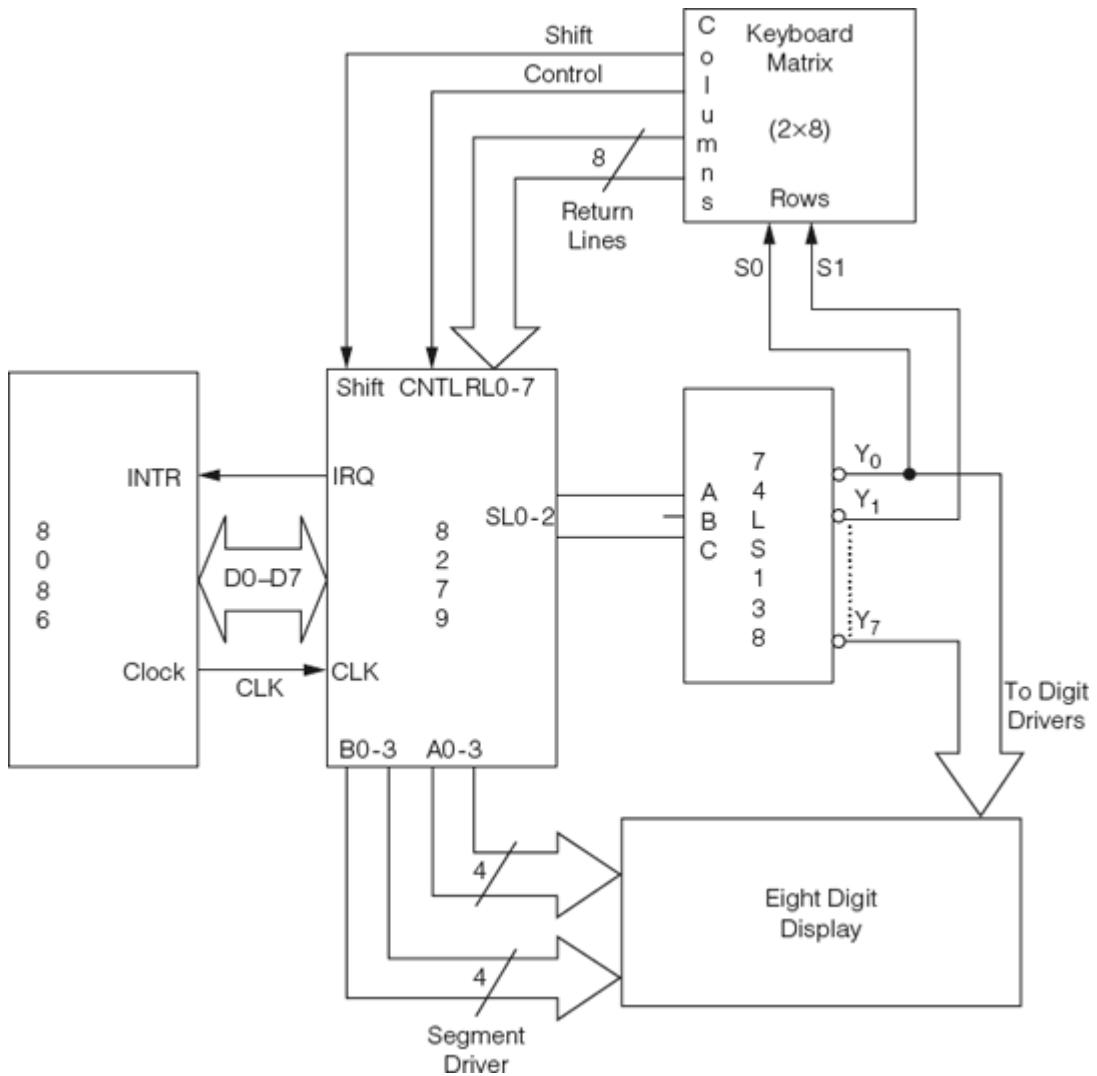


Figure 10.18 | Connections between the 8279, an 8 digit display and a 2×8 keyboard

CLK

This is connected from the system clock to generate timing.

A0

The pin A0 is used to select between data and command. If A0 is low, it indicates that the data bus contains data; otherwise, it is to be considered as command.

10.2.5 / Control Words

It was mentioned earlier that when A0 is high, the byte sent from the processor is a command. Thus, it is obvious that the control register has one address only (and the data register has another). However, because of the many functions that this chip performs, a number of control words are

to be written. **The upper three bits of the control register specify the type of the control word.** We will discuss the control words in the order in which they are to be used for a particular application. Only the control words necessary for the application will be discussed. For more information, the datasheet of the chip may be referred.

i) Keyboard / Display Mode Setup

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	0	D	D	K	K	K

In this word, DD stands for display and KKK for keyboard.

DD has the following options:

- 0 0 Eight 8-bit character display – Left entry
- 0 1 Sixteen 8-bit character display – Left entry
- 1 0 Eight 8-bit character display – Right entry
- 1 1 Sixteen 8-bit character display – Right entry

The display can be initialized to 8 or 16 character display. In our example, since we have only an 8-digit display, DD can be set for 8 character display. This will cause the refresh lines SL0 to SL2 to change from 000 to 111.

Left Entry

This is the simplest display format and is like a typewriter. The first character is displayed in the left most position, the second one will be displayed right of the first one and so on – just like the typewriter types on paper.

Right Entry

This is the way most electronic calculators display data. The first entry is placed on the right most position. For the second entry, the earlier one is shifted left and the new character is placed again at the right most position. This is repeated for all the characters.

Since the characters come from the display RAM, see Fig. 10.19 for the way the contents of the display RAM are displayed for each case. T₁ to T₄ are the time instants at which the characters stored in the display RAM appear on the display. The seven segment code of the characters A, B, C, D is in the RAM at address 0, 1, 2, 3.

KKK has the following variations:

- 0 0 0 Encoded Scan Keyboard – 2 Key Lockout
- 0 0 1 Decoded Scan Keyboard – 2 Key Lockout
- 0 1 0 Encoded Scan keyboard – N Key Rollover
- 0 1 1 Decoded Scan Keyboard – N Key Rollover
- 1 0 0 Encoded Scan sensor matrix
- 1 0 1 Decoded Scan sensor matrix
- 1 1 0 Strobed input, Encoded Display Scan
- 1 1 1 Strobed input, Decoded Display Scan

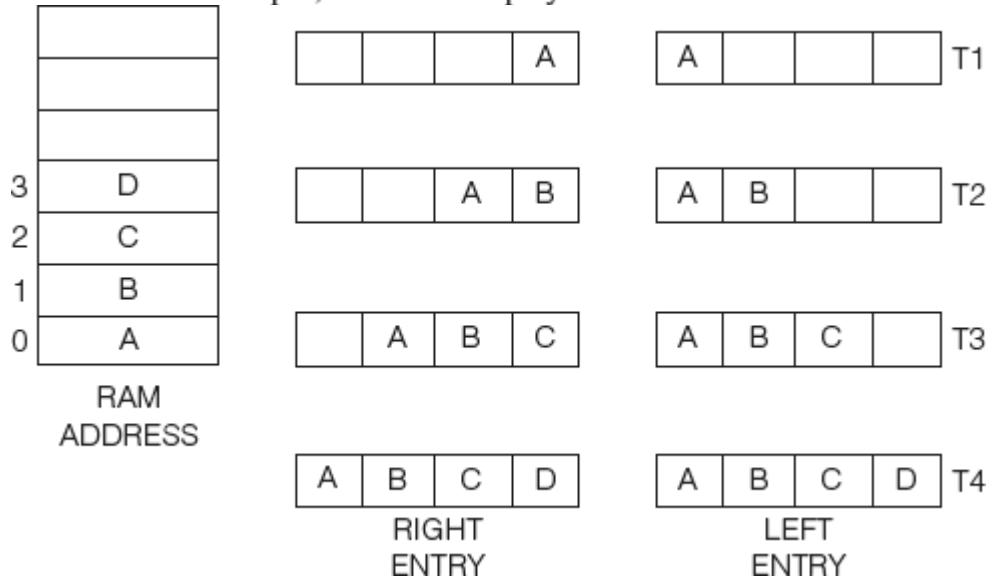


Figure 10.19 | Contents of the display RAM displayed for right entry and left entry

Two key lockout happens when the keys are such that only if the first key is released will the second key be considered at all. In N key rollover, if, say, two keys are pressed almost simultaneously, both of them are debounced and their codes will be stored in FIFO RAM in the order in which they were pressed.

A sensor matrix is a matrix of switch type sensors, the condition of which can be stored in the FIFO RAM ($8 \times 8 = 64$ switch states).

Example 10.10

Write the keyboard/display control word for the case of an 8-character display with right entry and encoded scan keyboard – 2 key lockout.

Solution

Refer the keyboard/display mode setup control word. The control word is

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	0	1	0	0	0	0

i.e., 10H

ii) Program Clock

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	1	P	P	P	P	P

All timing and multiplexing signals for the 8279 are generated by an internal pre-scaler. This pre-scaler divides the external clock by a programmable integer. Bits PPPPP determine the value of this integer, which ranges from 2 to 31. Choosing a divisor that yields 100 KHz will give the specified scan and debounce times. For instance, if the 8279 is clocked by a 2 MHz signal, PPPPP should be set to 10100 to divide the clock by 20 to yield the proper 100 KHz operating frequency. For dividing by 30, the control word is 0011 1110.

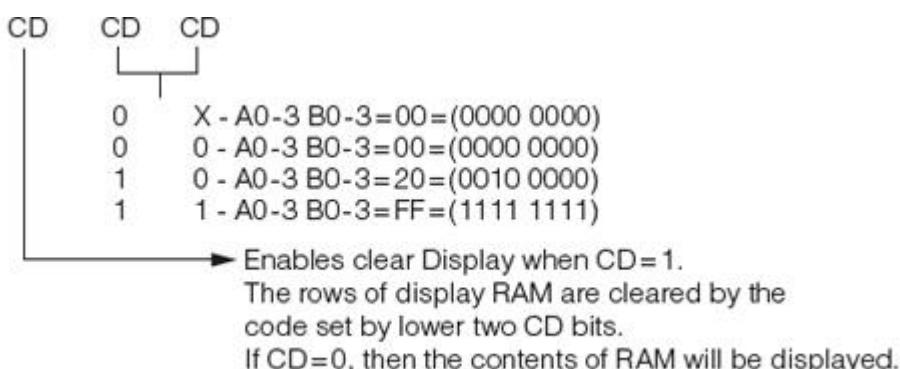
iii) Clear Display

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	1	0	C _D	C _D	C _D	C _F	C _A

C_D C_D C_D – The lower two C_D bits specify the blanking code to be sent to the segments to turn them off while the display is switching from one digit to the next.

C_F – If C_F = 1, FIFO status is cleared, interrupt and output lines are reset, and sensor RAM pointer is set to Row 0.

C_A – Clear all bit has the combined effect of C_D and C_F. It uses the C_D clearing code on the display and clears the FIFO status. It also synchronizes the internal timing chain.



Example 10.11

Find the clear display control word for the circuit of Fig. 10.14.

Solution

We have discussed that for this circuit the bit required to blank a segment is '1'. Hence, D_3 and D_2 are 11. To enable clear display, $D_4 = 1$. C_F is not relevant and may be 0. We can have $C_A = 1$.

Hence, the required control word is 1101 1101, i.e., CCH.

iv) Write Display Ram

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	AI	A	A	A	A

To display data, it must be entered into the display RAM. After this word is written, all subsequent writes will be to the display RAM. Remember that commands are written using an address which ensures that pin $A_0 = 1$. Data is written with $A_0 = 0$.

AI – Auto Increment. If AI = 1, the row address selected in the display RAM will be incremented after each write.

AAAA – Selects one of the 16 rows in the display RAM – thus it varies from 0000 to 1111.

Example 10.12

Write the display RAM control word

1. with auto-increment
2. without auto-increment

Solution

Here, let us select the first location in the display RAM.

1. With AI = 1, the control word is 1001 0000, i.e., 90H.
2. With AI = 0 (no auto-increment), the control word is 1000 0000, i.e., 80H.

v) Read Fifo / Sensor Ram

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	AI	X	A	A	A

This control word is for setting up the FIFO / SENSOR RAM.

AI – Auto Increment. Irrelevant for scanned keyboard mode but used in sensor RAM mode.

AAA – Irrelevant for scanned keyboard mode but used in sensor RAM mode.

Example 10.13

Write the control word for setting up the FIFO so as to read in keys that are pressed.

Solution

The control word is 0100 0000, i.e., 40H.

Example 10.14

Write a program to display ‘5’ in the first LED position of the LED display used, as per the block diagram of Fig. 10.14. The display consists of 8 LEDs to be used in the left entry mode. The keyboard is of type ‘encoded scan keyboard – 2 key lockout’. The address of the data register is 0C0H and of the control register is 0C2H.

Solution

The control words for setting up the display, clearing the display, and writing into the display RAM have been designed in Examples 10.10 to 10.12 and may be used here.

```
DATR EQU 0C0H
CNTR EQU 0C2H

MOV AL, 00      ;control word for 8 digit, left entry display
OUT CNTR, AL   ;send it to the control register
```

```

MOV AL, 3EH ;word to divide clock frequency by 30
OUT CNTR, AL ;send it to control register
MOV AL, 0CCH ;control word for clearing display
OUT CNTR, AL ;Send it to control register
MOV AL, 90H ;word to write display RAM with auto-increment
OUT CNTR, AL ;Send it to control register
MOV AL, 29H ;seven segment code to display '5'
OUT DATR, AL ;Send it to data register
MOV AL, OFFH ;seven segment code for no display
MOV CX, 07 ;7 display LEDs to be blanked
NEXT: OUT DATR, AL ;send the data to the data register
LOOP NEXT ;repeat for seven digits

```

This program displays '5' in the first digit location. The rest of the digits are not lighted up because the data given to these digits is FFH, which corresponds to no segment being lighted.

Example 10.15

Write a program to display a rolling message 'H' continuously moving left.

Solution

Here, the seven segment code for 'H' is in the location named TABLE. Then, the codes of 'no display', i.e., FFH, is written 3 times and this is repeated once more. A counter of 8 causes this sequence to be displayed once. The display seen is 'H' moving left followed by three blanks. In effect, it shows 'H' moving from right to left continuously.

```

DATR EQU 0C0H
CNTR EQU 0C2H
TABLE DB 98H, OFFH, OFFH, OFFH, 98H, OFFH, OFFH, OFFH

STRT: LEA SI, TABLE ;let SI point to the address of table
      MOV CX, 07 ;CX to act as a counter
      MOV AL, 10H ;word for 8 digit, right entry display
      OUT CNTR, AL ;send it to control register
      MOV AL, 3EH ;word to divide clock frequency by 30
      OUT CNTR, AL ;send it to control register
      MOV AL, 0CCH ;word to clear display
      OUT CNTR, AL ;send it to control register
      MOV AL, 90H ;write to display RAM with auto-increment
      OUT CNTR, AL ;send it to control register
NEXT:  MOV AL, [SI] ;get data in table to AL
      OUT DATR, AL ;send it to the data register
      CALL DELAY ;call a delay

```

```

INC SI           ;increment pointer
LOOP NEXT       ;repeat until CX = 0
JMP STRT        ;start all over again

DELAY PROC NEAR  ;delay procedure
    MOV DX, 0AFFFH
AGN:   DEC DX
    JNZ AGN
    RET
DELAY ENDP

```

10.2.6 | Detecting a Keypress

Whenever a key is pressed, the code of the key enters the FIFO RAM. This action also increments a count in a **status** register which gives the information regarding the number of characters in the FIFO and gives, besides other things, the indication as to whether the FIFO is full. This status register may be read to get the required information. Figure 10.20 shows the bit configuration of the status register, which has the same address as the control register. The difference is that one ‘writes’ into a control register but ‘reads’ from a status register. Observe that the lowest three bits indicate the number of characters in the FIFO RAM. If these bits are zero, it is understood that no valid key presses have occurred. There is bit D₃ to indicate that the FIFO is full. The two error conditions it indicates are the overrun and underrun errors. The former will be set when writing into a full FIFO is attempted. What it means is that more than 8 characters have been written in the FIFO RAM, and hence the oldest data has been lost. Under run error implies an attempt to read an empty FIFO.

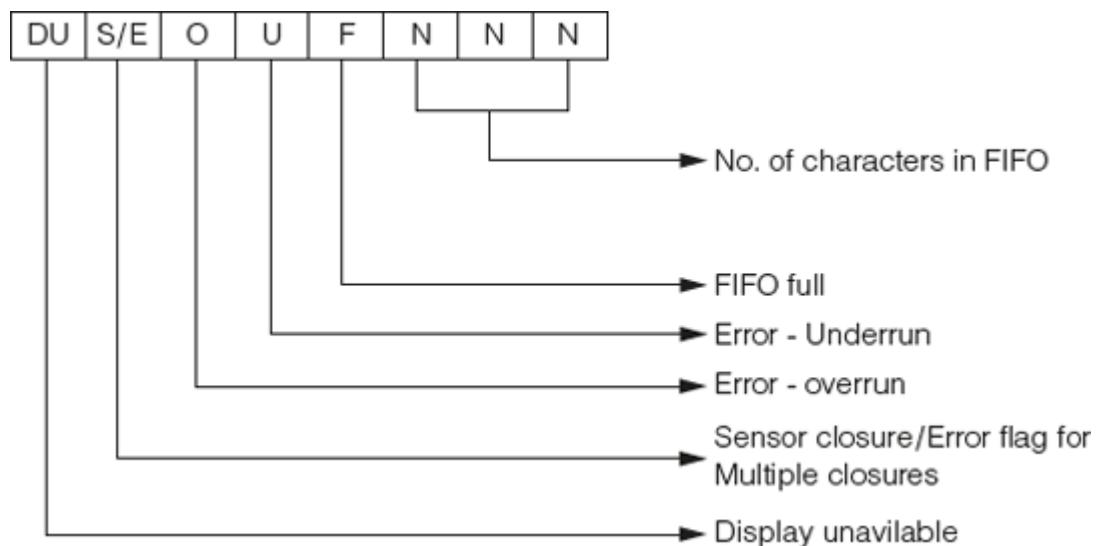


Figure 10.20 | Format of the status word

Example 10.16

Write a program to read a key that is pressed and store the key code in memory.

Solution

Refer to Fig. 10.17, which shows the way a key code is specified. This code will be entered into the FIFO RAM when the key is pressed. To read the key, the steps are (after pressing the key):

1. Read the status word, and check if the lowest three bits are zero. If they are, it means that the FIFO RAM is empty, meaning that no valid key press has been detected. The status register has the same address as the control register, the difference being that a status word is ‘read’ while a control word is ‘written’.
2. If a valid key press is identified, read the code from the FIFO RAM after writing the ‘read FIFO RAM’ control word into the control register.

```
DATR EQU 0C0H
CNTR EQU 0C2H
DAT DB ? ;the code of the key should be here

LEA SI, DAT ;let SI point to a location in memory
NEXT: IN CNTR, AL ;read in the status register
      TEST AL, 01 ;check whether D0 = 1
      JZ NEXT ;if not, continue checking
      MOV AL, 40H ;otherwise, write the word to read FIFO
      OUT CNTR, AL ;send it to the control register
      IN AL, DATR ;read the FIFO RAM
      MOV [SI], AL ;send the content to the location DAT
```

In the above program, while reading the status register, only the D₀ bit is tested. This is because it is assumed that the FIFO was empty earlier, and so, a key press makes the count equal to 1.

In the location DAT, we get a key code C6 if ‘6’ is pressed, CE if ‘E’ is pressed and so on. The key code is obtained as shown in Fig. 10.17 and it depends on the positioning of the keys in the key matrix. This code corresponds to the keyboard used here.

Example 10.17

Write a program which displays the key that has been pressed.

Solution

Here, the seven segment code corresponding to the keys is stored in memory, and the program accesses these codes and displays it, depending on the key pressed. The seven segment codes are stored in the order 0, 1, 2 ... F for the 16 keys. No provision has been made to account for pressing the shift and control keys.

When a key is pressed, and a key code is obtained, the upper nibble of this byte is masked off. Thus, for the key 6, the key code is C6, which is 06 after the masking operation. The seven segment code of this key is available in the location TABLE + 6.

```

DATR EQU 0C0H
CNTR EQU 0C2H
TABLE DB 0CH, 9FH, 4AH, 0BH, 99H, 29H, 28H, 8FH,
      08, 09, 88H, 38H, 6CH, 1AH, 68H, 0E8H

LEA SI, TABLE ;pointer to the seven segment codes
MOV CX, 08 ;counter
MOV AL, 0 ;word for display RAM, right entry
OUT CNTR, AL ;send it to control register
MOV AL, 3EH ;word to divide clock frequency by 30
OUT CNTR, AL ;send it to control register

MOV AL, 3EH ;word to divide clock frequency by 30
OUT CNTR, AL ;send it to control register
MOV AL, 0CCH ;word to clear display
OUT CNTR, AL ;send it to control register
MOV AL, 90H ;word to write display RAM
OUT CNTR, AL ;send it to control register
MOV AL, 0FFH ;data to blank out the display
AGN: OUT DATR, AL ;send it to the data register
      LOOP AGN ;repeat for 8 digits
NEXT: IN AL, CNTR ;read in the status register
      TEST AL, 07 ;test if D0 = 1
      JZ NEXT ;if not, continue testing
      MOV AL, 40H ;otherwise write the word to read FIFO
      OUT CNTR, AL ;send it to control register
      IN AL, DATR ;read in the FIFO
      AND AL, 0FH ;mask upper nibble of key code
      MOV BL, AL ;copy it to BL
      MOV BH, 0 ;BH = 0
      ADD SI, BX ;add BX to SI
      MOV AL, [SI] ;take corresponding display code
      OUT DATR, AL ;send it to data register
      JMP NEXT ;continue for more keys

```

This program displays every key that is pressed. After 8 entries, the FIFO drops the earliest entries and continues the displaying of the keys pressed.

10.3 | The Programmable Interrupt Controller (PIC) 8259

10.3.1 / Introduction

Before trying to understand the description and working of the programmable interrupt controller (PIC), it will be worthwhile to read Section 8.1 to get a precise and clear understanding of the mechanism of interrupts in general and also the specific details of the interrupt structure of the 8086. This will go a long way in the understanding of the use and functionality of this chip, which is dedicated to the management of the hardware interrupts of the system.

Recollect that there are only two hardware interrupt lines for the 8086. An external device can place its interrupt request on either of these pins. One pin called NMI is a ‘vectored’ interrupt, in the sense that its interrupt handler is pointed to by a particular type number and that is ‘2’. Thus, NMI is a Type 2 interrupt, and since it is non-maskable it is used for important and high priority services. As such, this interrupt pin is not available for general interrupt requests from various peripherals. That leaves just one pin for accepting hardware interrupt requests, and if many peripherals want to use this pin, obviously a number of issues have to be resolved. It is in this context that a dedicated interrupt controller is necessary.

10.3.2 / 8259 – Features

The first interrupt controller 8259 was designed for the 8080 microprocessor only. Later, the 8259A was made available that gives support to the x86 processor as well. However, we will use the word 8259 itself for 8259A, because the original 8259 is obsolete now. This is the standard chip which acts as an interrupt manager for a system – depending on the size and requirements of the system, there can be one or more interrupt controllers. See the basic connection between the 8086 and the 8259 in Fig. 10.21.

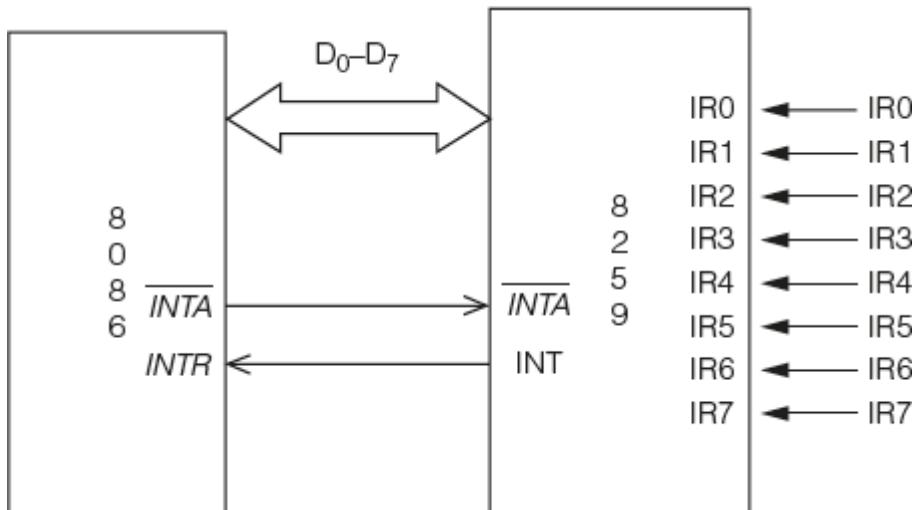


Figure 10.21 | Basic connections between a PIC 8259 and the 8086

The PIC has eight interrupt request lines IR0 to IR7 on which peripherals can place their interrupt requests. If one interrupt request alone is received on the chip, that request is channeled to the INTR line of the processor. As is the usual sequence, the processor sends back an acknowledge request in the form of the INTA signal. Then, the PIC (on behalf of the peripheral which has made the request) sends to the 8086 the interrupt type number corresponding to that specific interrupt. This implies that the PIC is to have been programmed to send a specific type number for a particular interrupt request.

Another activity of the chip is to resolve priorities. Suppose interrupt requests arrive at more than one interrupt input of the chip. Which one of these requests should be channeled to the processor? This decision is made by the PIC and it has to be programmed for this. There should also be the provision to change the priorities.

Suppose there are more than 8 possible interrupt sources, more PICs can be used, which means that the chip has pins for cascading more of such chips and a particular chip can be a master or a slave. With these ideas in mind, let us see the finer details of the chip.

10.3.3 / Block Diagram

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It can cascade up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single 5 V supply. Circuitry is static, requiring no clock input. The 8259 A is designed to minimize the software and real time overhead in handling multi-level priority interrupts.

It has several modes, permitting optimization for a variety of system requirements. Figure 10.22 depicts a functional block diagram of the PIC.

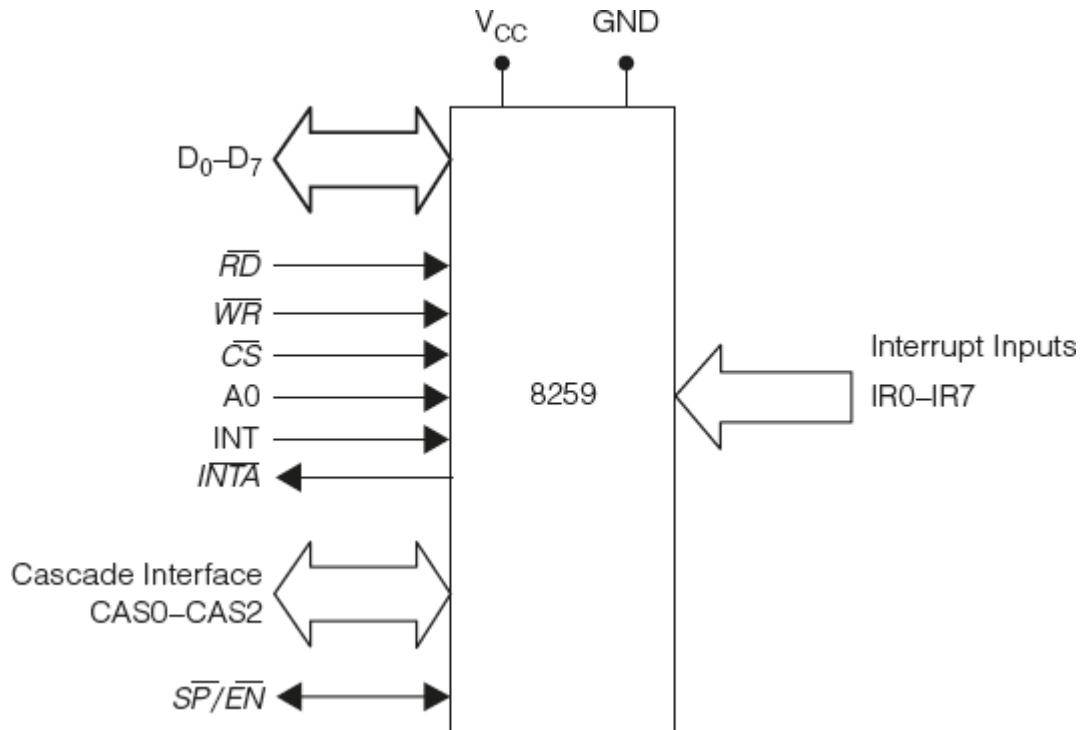


Figure 10.22 | Functional block diagram of the PIC

The block diagram in Fig. 10.23 shows the different functional parts of the chip. It has 8 interrupt request lines IR0 to IR7, which are asynchronous inputs lines. An interrupt request is executed by raising an IR input (low to high) and holding it high until it is acknowledged (Edge Triggered Mode) or just by a high level on an IR input (Level Triggered Mode).

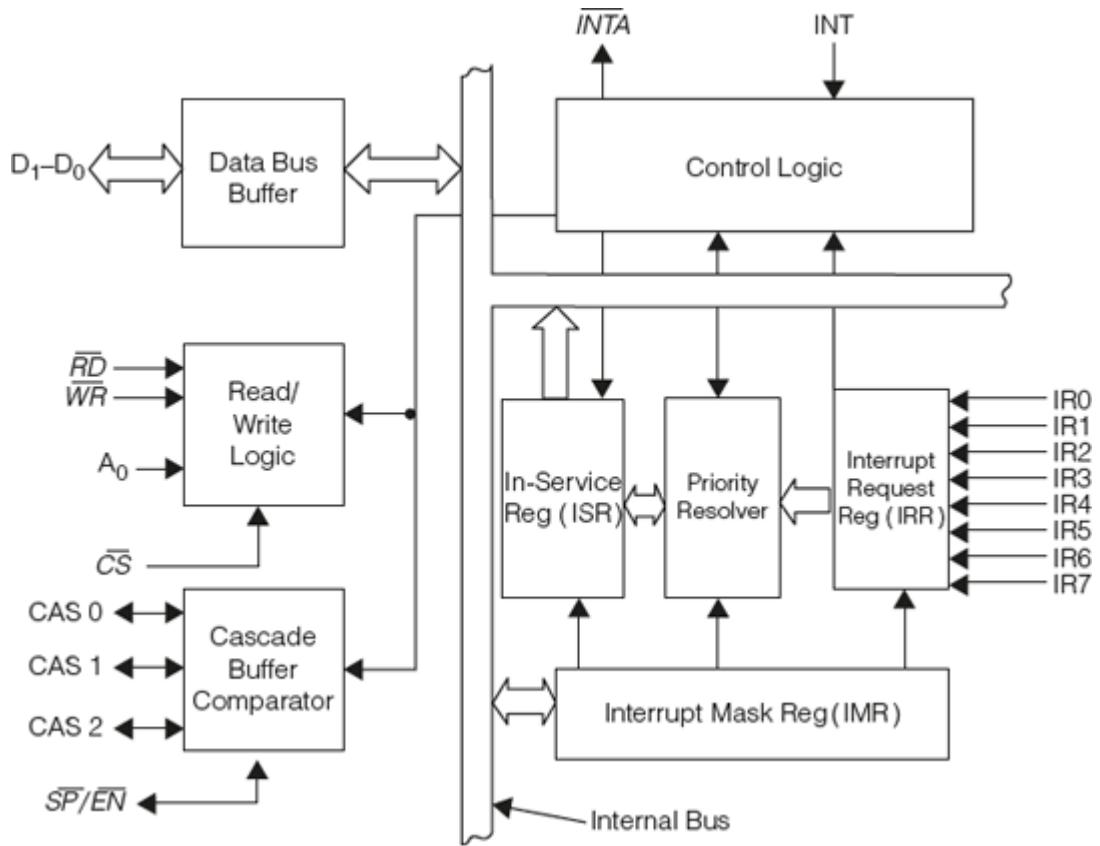


Figure 10.23 | Internal block diagram of a programmable interrupt controller (PIC)

Int (Interrupt)

This output goes directly to processor interrupt input and causes the processor to send back the *INTA* as an acknowledgement to the 8259.

Interrupt Request Register (Irr) and In-Service Register (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all the interrupt levels which are requesting service, and the ISR is used to store the interrupt levels which are being serviced.

Priority Resolver

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR (in service register) during the INTA pulse.

Interrupt Mask Register (IMR)

The IMR stores the bits which mask the interrupt lines to be masked. The IMR operates on the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

CAS0–CAS2

These three bi-directional lines can be used to cascade several such chips to expand the number of interrupts up to 64 in a master or slave mode. The associated three I/O pins (CAS0-2) are outputs when the 8259A is used as a master and are inputs when the 8259A is used as a slave.

SP / EN

This stands for Slave Program/Enable Buffer. This is also a bidirectional pin. In the buffered mode, it can be used to control buffer transceivers (*EN*). In other modes, it indicates whether the 8259 is a master (*SP* = 1) or whether it is a slave (*SP* = 0).

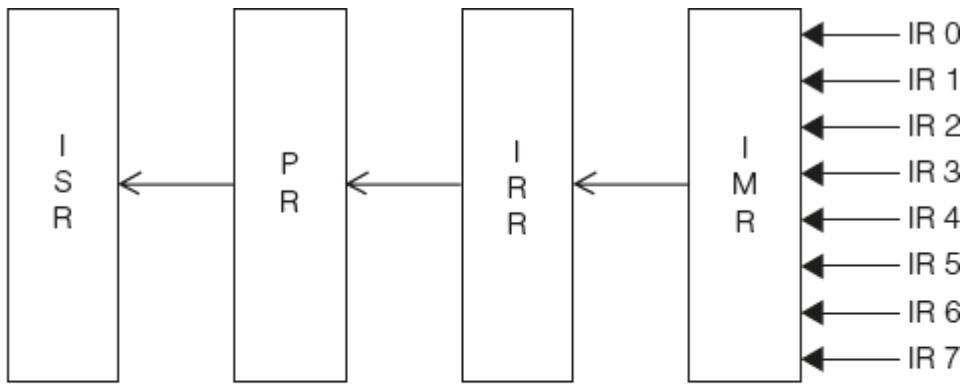
R / W LOGIC

Just as in the case of any other peripheral chip, the decoding logic on the *CS* and *A0* determines the addresses of the registers within the chip, and in conjunction with the *RD* and *WR* signals, the registers can be written into and read from (to get status information).

10.3.4 / Interrupt Sequence for an 8086 Based System

The 8259 has been designed to interface the processors 8080 and 8085 as well as the x86 family of microprocessors. The interrupt processing sequence of the 8080 and 8085 is different from that of the x86 processors. Here, we will discuss only the sequence for the x86 processors.

Let us list out the sequence of operations that occur when a peripheral places an interrupt request in an x86–8259 system. (Refer Fig. 10.24.)



IMR: Interrupt Mask Register

IRR: Interrupt Request Register

PR : Priority Resolver

ISR : In-Service Register

Figure 10.24 | Interrupt Sequence and the registers involved

1. One or more of the interrupt request lines (IR0 to IR7) are raised high. The 8 individual interrupt request lines are first passed through the Interrupt Mask Register (IMR) to see if they have been masked or not. If they are masked, then the request is not processed any further. However, if they are not masked, they will register their request with the Interrupt Request Register (IRR) setting the corresponding IRR bit(s).
2. The Interrupt Request Register will hold all the requested IRs until they have been dealt with appropriately. The Priority Resolver simply selects the IR of highest priority.
3. The 8259 then sends an INTR signal to the CPU.
4. The CPU acknowledges the INTR signal and responds with an INTA pulse.
5. Upon receiving the processor's interrupt acknowledge, the IR (Interrupt Request) which the PIC is processing at the time is stored in the In Service Register (ISR), which, as the name suggests, shows which IR is currently in service. This IR bit is also reset in the Interrupt Request Register (IRR), as it is no longer requesting service but is actually getting service. The 8259 does not drive the data bus during this cycle.
6. The 8086 will initiate a second INTA pulse. During this pulse, the 8259 releases an 8-bit pointer (type number) onto the data bus where it is read by the CPU, and the interrupt service routine of the selected device thus gets executed.
7. This completes the interrupt cycle.
8. Once the ISR has done everything it needs, it sends an End of Interrupt (EOI) to the PIC, which resets the In-Service Register.

10.3.5 / Using the 8259

The chip must be programmed for our use, and the programming sequence starts with writing the necessary control words into the control register. There are two types of control words:

1. Initialization Command Words (ICWs) are needed for starting the operation. These are a sequence of 4 bytes, some of which are optional.
2. Operational Control Words (OCWs) are the words which command the 8259 to operate in various interrupt modes.

10.3.6 | Initialization Command Words (ICWs)

Note that there is only one address line, i.e., A_0 , to communicate with the chip (see Table 10.2). Thus, the initialization control words are selected using this pin, and it is obvious that ICW1 uses a port address different from the other three initialization control words. See Fig. 10.25 for the initialization flow chart.

Table 10.2 | Value of A_0 for the Different ICWs

CS	A_0	ICW
0	0	ICW1
0	1	ICW2
		ICW3
		ICW4

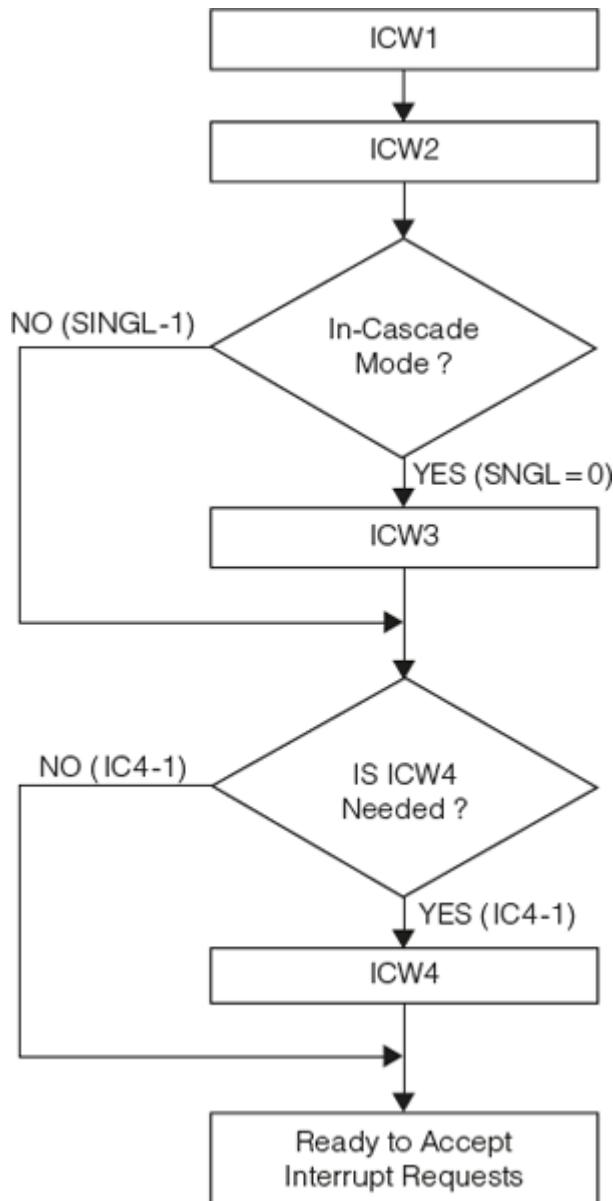


Figure 10.25 | Flow chart showing the initialization sequence for the PIC

10.3.7 | Initialization Sequence

10.3.7.1 / ICW1 (Initialization Command Word1)

For initialization, always ICW1 should be sent first. Whenever a command is issued with **A0 = 0 and D4 = 1, this is interpreted as Initialization Command Word 1 (ICW1)**. ICW1 starts the initialization sequence.

This control word carries the information required to distinguish between ICW2, ICW3 and ICW4. The important thing is that all these words may not be necessary.

D₀ indicates whether ICW4 is necessary.

D_1 indicates whether the system is operating in a slave or master mode. If $D_1 = 1$, it is a slave mode and hence ICW3 is not needed.

$D_2 = 0$ always for the x86 processor.

D_3 selects the options of edge or level triggering for the interrupt lines IR0 to IR7.

D_4 is high always in the ICW1 and D_5 to D_7 are low for x86 processors.

10.3.7.2 / ICW2 (Initialization Command Word2)

This word allows us to decide the type of numbers to be associated with the interrupt input lines of IR0 to IR7 (See Fig. 10.25).

This word is sent with $A_0 = 1$. The bits D_2 to D_0 vary from 000 to 111 and, together with the upper bits D_7 to D_3 , decide the type numbers of the interrupts (IR0 to IR7). Thus, we have to write the bits D_7 to D_0 . Figure 10.27 shows how this is done.

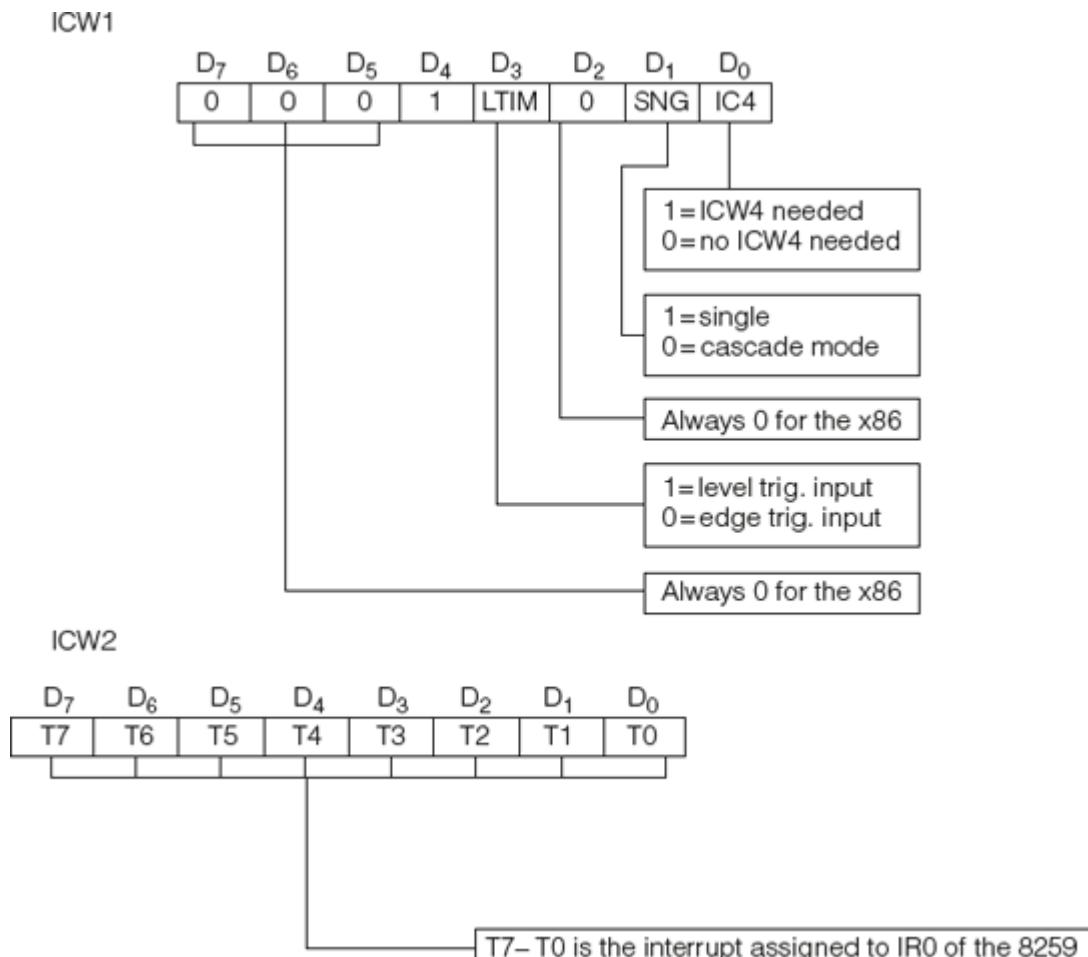


Figure 10.26 | Format of the initialization command words 1 and 2

Content of Interrupt Vector Byte
for 8086 System Mode

	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
IR7	T7	T6	T5	T4	T3	1	1	1
IR6	T7	T6	T5	T4	T3	1	1	0
IR5	T7	T6	T5	T4	T3	1	0	1
IR4	T7	T6	T5	T4	T3	1	0	0
IR3	T7	T6	T5	T4	T3	0	1	1
IR2	T7	T6	T5	T4	T3	0	1	0
IR1	T7	T6	T5	T4	T3	0	0	1
IR0	T7	T6	T5	T4	T3	0	0	0

Figure 10.27 | The method of deciding interrupt type numbers for each IR

Example 10.18

Design the initialization control word 2 (ICW2) for a system to associate type numbers 78H onwards for the interrupt inputs IR0 to IR7.

Solution

The method is that ICW2 should be assigned the value corresponding to the type number of IR0. Here, for this to be done, ICW2 should be designed with D₇ to D₃ to be 01111. Thus, ICW2 can be

0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

i.e., 78H

This will assign the type number 78H to IR0, 79H to IR1 and so on. The idea is that the type number assigned to IR0 should have D₂ to D₀ to be 000. It cannot be assigned a type, say, 76H because D₂ to D₀ are 110.

10.3.7.3 / ICW3 (Initialization Control Word3)

This control word is necessary only when the two or more 8259s are cascaded. A single 8259 can be connected to 8 slave chips, thus giving the possibility of up to 64 hardware interrupts. In cascade modes, there are separate ICW3 words for the master and the slave. For the master, this word specifies which IR input has a slave connected and another ICW3 informs the slave which IR input of the master is connected to it. See Fig. 10.27.

10.3.7.4 / Initialization Command Word 4 (ICW4)

D_0 indicates whether the processor is 8080/85 ($D_0 = 0$) or x86 ($D_0 = 1$).

D_1 specifies whether an AEOI (automatic end of interrupt) will be used ($D_1 = 1$) or whether EOI commands should be deliberately inserted at the end of an interrupt service routine ($D_1 = 0$).

D_2 and D_3 specify the options possible when the data bus has bi-directional transceivers (Section 6.1.4.2) and when it does not.

D_4 indicates the use of the special fully nested mode ($D_4 = 1$). This mode is to be used when the chip acts as a master.

D_5 to D_7 are always '0' in this control word.

Example 10.19

Assume that address decoding is done such that when ICW1 is to be used, the port address is C0H, and for the other ICWs, the address is C2H.

Write the initialization instruction for setting up the system with the following specifications:

1. The system has a single 8259, with edge triggered interrupt inputs and interrupt type numbers from 90H, 91H and 92H for IR0 to IR2.
2. It is also to operate in slave buffered mode with normal EOI.

Solution

First, the ICWs are to be designed. It is designed with $D_0 = 1$; i.e., ICW4 is needed so as to be able to use the second set of specifications (i.e., slave buffered mode with normal EOI).

ICW1 is as follows (refer Fig. 10.26):

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	1	0	0	1	1

i.e., 13H

ICW2 is designed with D_7 to D_3 to be 10010 for specifying the interrupt type numbers.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	1	0	0	0	0

i.e., 90H

Since the system has only a single 8259, the control word ICW3 is not needed. ICW4 is

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	0	0	1	0	0	1

i.e., 09

The port address to send ICW1 is C0H and for the other ICWs is C2H.

Program

```

MOV AL, 13H           ;value of ICW1
OUT 0C0H, AL          ;send it to port C0H
MOV AL, 90H           ;value of ICW2
OUT 0C2H, AL          ;send it to port C2H
MOV AL, 09             ;value of ICW4
OUT 0C2H, AL          ;send it to port C2H

```

10.3.8 / Operational Command Words

Now that we know how to initialize the chip, the next thing will be to understand its different operating modes. For this, it is necessary to use the control words called ‘Operational Command Words’. We will discuss the modes and features of the chip step by step, along with the operational command words that are used for these modes. The OCWs are sent with A₀ = 1 for OCW1 and A₀ = 0 for OCW2 and OCW3. See Table 10.3.

Table 10.3 | Value of A₀ for the Different OCWs

CS	A₀	OCW
0	0	OCW2
0	1	OCW3
		OCW1

10.3.8.1 / Ocw 1 (Operational Command Word 1)

This word is used to mask any of the interrupt requests IR0–IR1. For masking, the specific bit is ‘1’. See Fig. 10.28. Note that this word has the

interrupt mask structure. To know the current mask status, OCW1 can be read. For example, to mask the interrupt requests on IR0, IR3 and IR7, OCW1 has to have the value

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	0	1	0	0	1

i.e., 89H

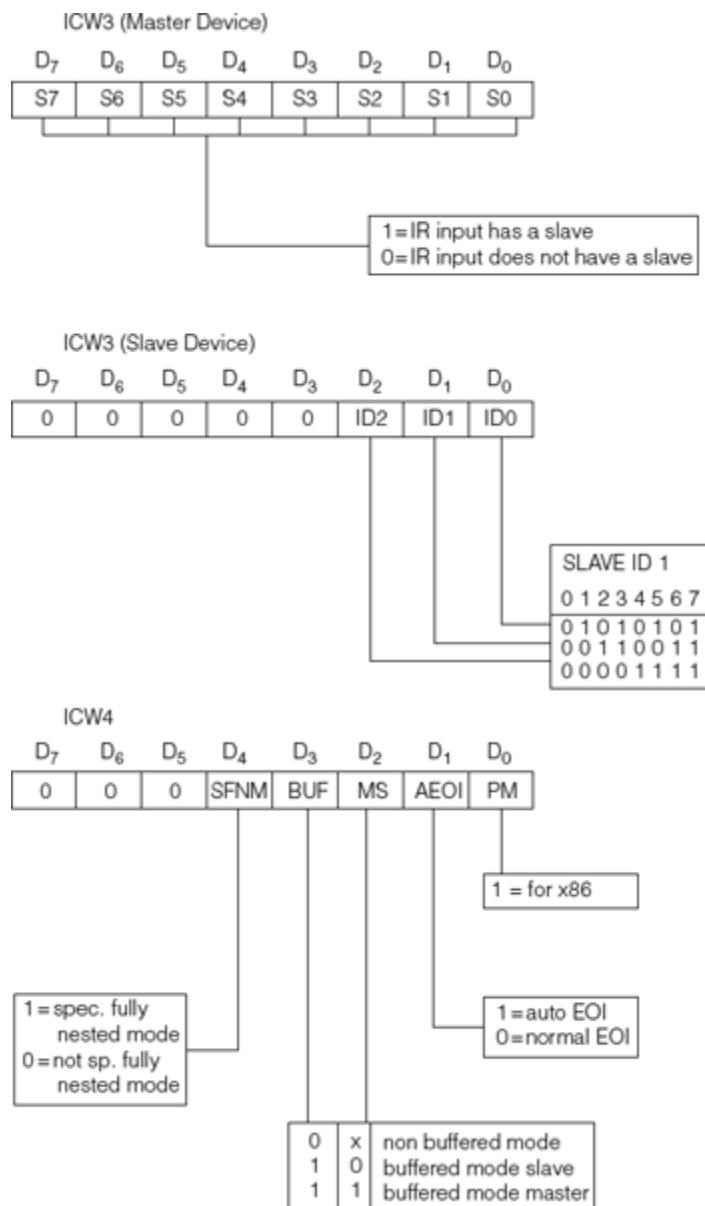


Figure 10.28 | ICW4 and ICW3

```

IN AL, 0C0H          ;read IRR
MOV BL, AL           ;save IRR in BL
MOV AL, 69H          ;write OCW3 for reading ISR
IN AL, 0C0H          ;read ISR
MOV CL, AL           ;save it in CL

```

Example 10.20

Write instructions to read the current status of the IMR. After that, change the masking structure such that only IR0 is masked.

Solution

The status of the IMR is to be read. Read the status of OCW1. The instruction is

```
IN AL, 0C2H ;C2H is the address we have used in  
               Example 10.19 and it is the same as that of  
               ICW2 (with A0 = 1)  
MOV AL, 01    ;mask bit for IR0 is set  
MOV BL, AL  
OUT 0C2H, AL ;send to OCW1
```

10.3.8.2 / OCW2 (Operational Control Word2)

This word is used to assign priorities to the interrupt requests. There is a default priority assigned on initialization of the chip. This can be changed dynamically by the bit patterns of OCW2. Refer Fig. 10.29 for the bit patterns for the various priority assignment schemes. Also, let us have a look at the various priority options available.

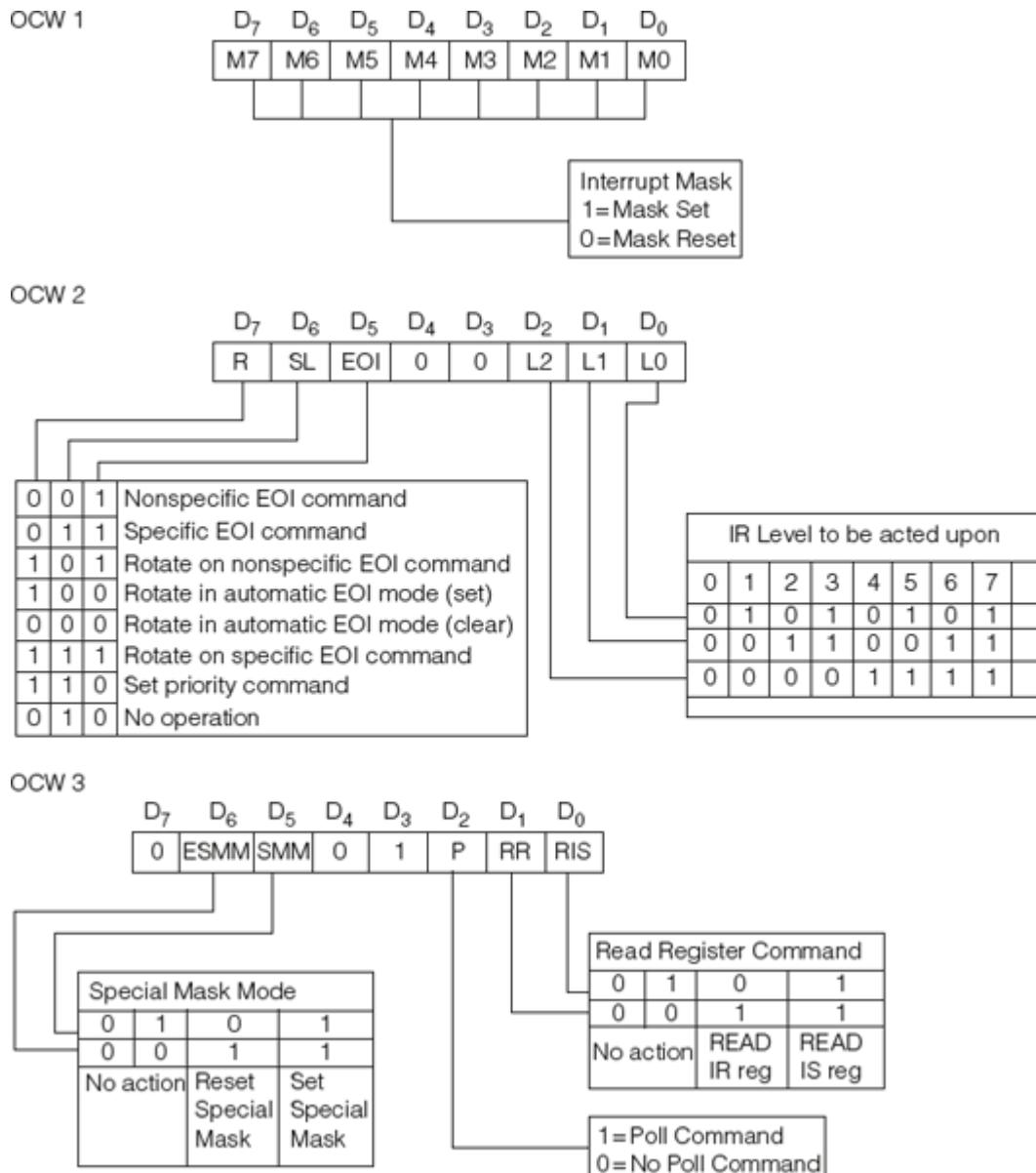


Figure 10.29 | Operational Control Words 1 to 3

Fully Nested Mode

This mode is entered after initialization until another mode is programmed. The interrupt requests are ordered in priority from 0 through 7 (0 highest). When an interrupt is acknowledged, the highest priority request is determined and its vector placed on the bus. Additionally, a bit of the In-Service register is set. This bit remains set until the microprocessor issues an End of Interrupt (EOI). When an interrupt is being serviced, all lower priority interrupts are inhibited, but higher priority interrupts can be acknowledged, provided the IF is ensured to be re-enabled using software.

After the initialization sequence, IR0 has the highest priority and IR7 the lowest. Thus, if IR3 and IR4 requests occur simultaneously, IR3 will be

serviced first. However, IR3 can be interrupted by a request on IR2, provided the interrupt flag has been enabled by software (IF = 1).

Automatic Rotation mode

This mode is useful when a just and equitable distribution of service is meant to be allotted for devices which may be considered to be of equal importance and thus of equal priority. The idea is that once an interrupt for a device is serviced, it goes to the status of lowest priority until all other requests are serviced.

Specific Rotation Mode

This mode has similarities to the previous mode in that the one that has been serviced will be made to go to the lowest priority status. The difference is that the priorities can be made to follow a specific sequence. The programmer can change priorities by programming the bottom priority and thus fixing all other priorities; i.e., if IR3 is programmed as the bottom priority device, then IR4 will have the highest priority device connected to it. Priorities are now IR4, 5, 6, 7, 0, 1, 2, 3 ... in that order of decreasing levels.

In Fig. 10.29, D₂ to D₀ are used to assign new priority schemes – if the highest priority is to be IR4, D₂–D₀ should be 100. D₄ to D₃ must always be 0 for OCW2. D₇ to D₅ can have values according to the options needed.

Example 10.21

Find the value of OCW2 for the following options:

1. Bottom priority level IR5.
2. Specific EOI command with rotating priority.

Solution

The word is

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	1	1	0	0	1	0	1

i.e., E4H

D₇ to D₅ = 111, because of using the mode ‘specific rotation’.

D_2 to $D_0 = 101$, because IR5 is the bottom priority.

Eoi Command

This is the ‘end of interrupt’ command. At the end of any ISR, this command should be present so that the bit corresponding to that interrupt request level in the In-Service Register (ISR) is reset. To clarify this point, remember that when an interrupt on line IR2, say, is taken up for servicing, a bit in the in-service register is set to indicate the ‘interrupt servicing’ status of that request. Once that is over and done with, the bit is to be reset and this is done only on receiving an EOI command. If this is not done, new interrupt requests cannot be placed on this line.

10.3.8.3 / OCW3 (Operational Control Word 3)

This word can be read to get the status of the In-Service register and the Interrupt Request Register. Also, it is used to enable some high level features of the chip.

Example 10.22

Write a program to read the contents of the following registers of the 8259:

1. IMR
2. IRR
3. ISR.

Assume that initialization has been done earlier.

Solution

```
IN AL, 0C2H          ;to read IMR, READ OCW1
MOV AL, 6AH          ;write OCW3 for reading IRR
IN AL, 0C0H          ;read IRR
MOV BL, AL           ;save IRR in BL
MOV AL, 69H          ;write OCW3 for reading ISR
IN AL, 0C0H          ;read ISR
MOV CL, AL           ;save it in CL
```

Note The 8259 is a complex chip, and for finer details, the data sheet of the chip is to be referred to.

10.4 | Cascade Mode

The 8259 can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels. The master controls the slaves through the 3 line cascade bus. The cascade bus acts like chip selects to the slaves during the INTA sequence.

In a cascade configuration, the slave interrupt outputs are connected to the master interrupt request inputs. When a slave request line is activated and afterwards acknowledged, the master will enable the corresponding slave to release the type number corresponding to the device interrupt handler.

The cascade bus lines are normally low and will contain the slave address code from the trailing edge of the first *INTA* pulse to the trailing edge of the third pulse. Each 8259 in the system must follow a separate initialization sequence and can be programmed to work in a different mode. An EOI command must be issued twice: once for the master and once for the - corresponding slave. An address decoder is required to activate the Chip Select *CS* input of each 8259. The cascade lines of the Master 8259 are activated only for slave inputs; non-slave inputs leave the cascade line inactive (low). Figure 10.30 shows three 8259s cascaded together.

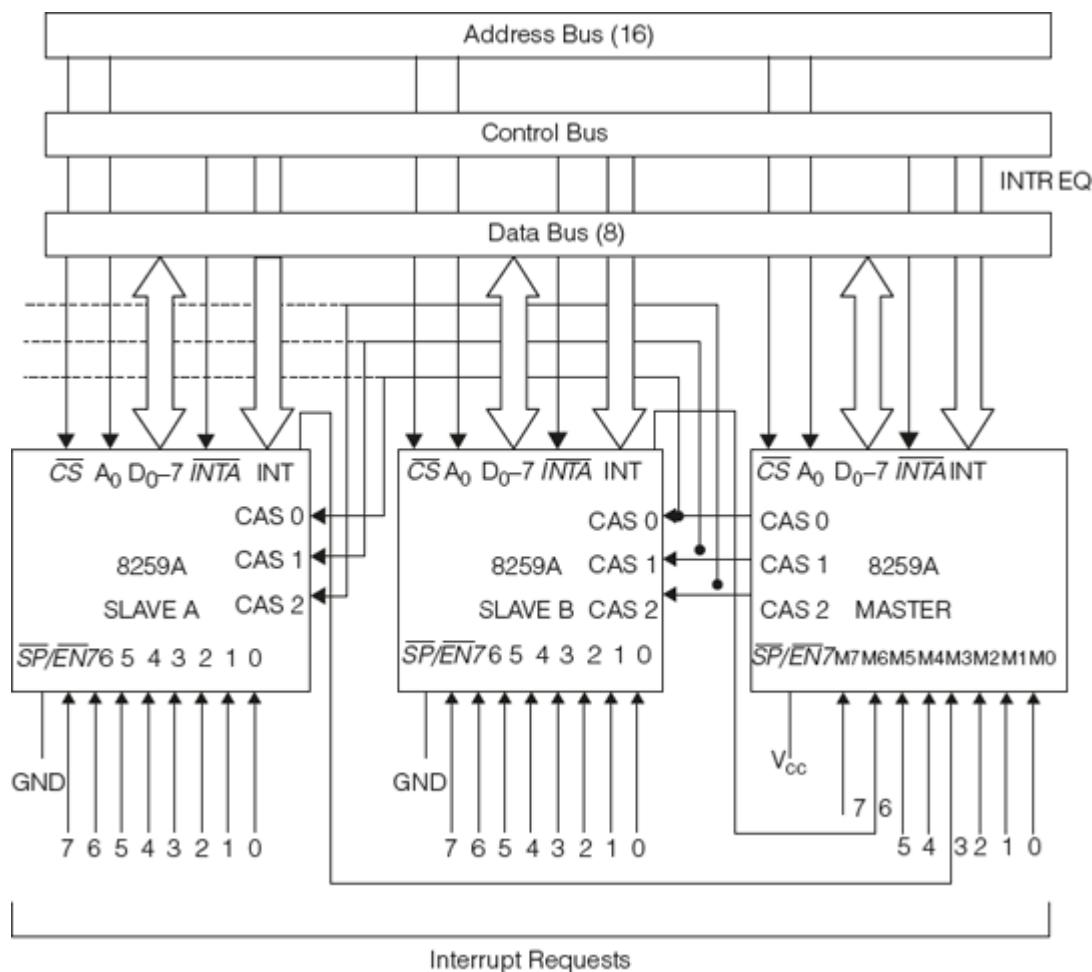


Figure 10.30 | Cascaded connection of three PICs

10.4.1 / Programmable Interrupt Controllers in the PC

In Table 8.2, a list of some of the standard PC interrupts was seen. Some of them are software interrupts and some are hardware interrupts. The hardware interrupts are routed through an 8259 and each one is denoted as an IRQ (Interrupt Request). In the IBM PC and the original PC-XT, there was only one 8259 chip and thus only eight IRQ lines. However, users soon needed more devices to be interrupt driven, and thus, an additional 7 IRQ's were added to the PC. This involved attaching another 8259 to the existing one. Compatibility always causes problems as the new configuration still had to be compatible with old hardware and software. The new configuration is shown in Fig. 10.31.

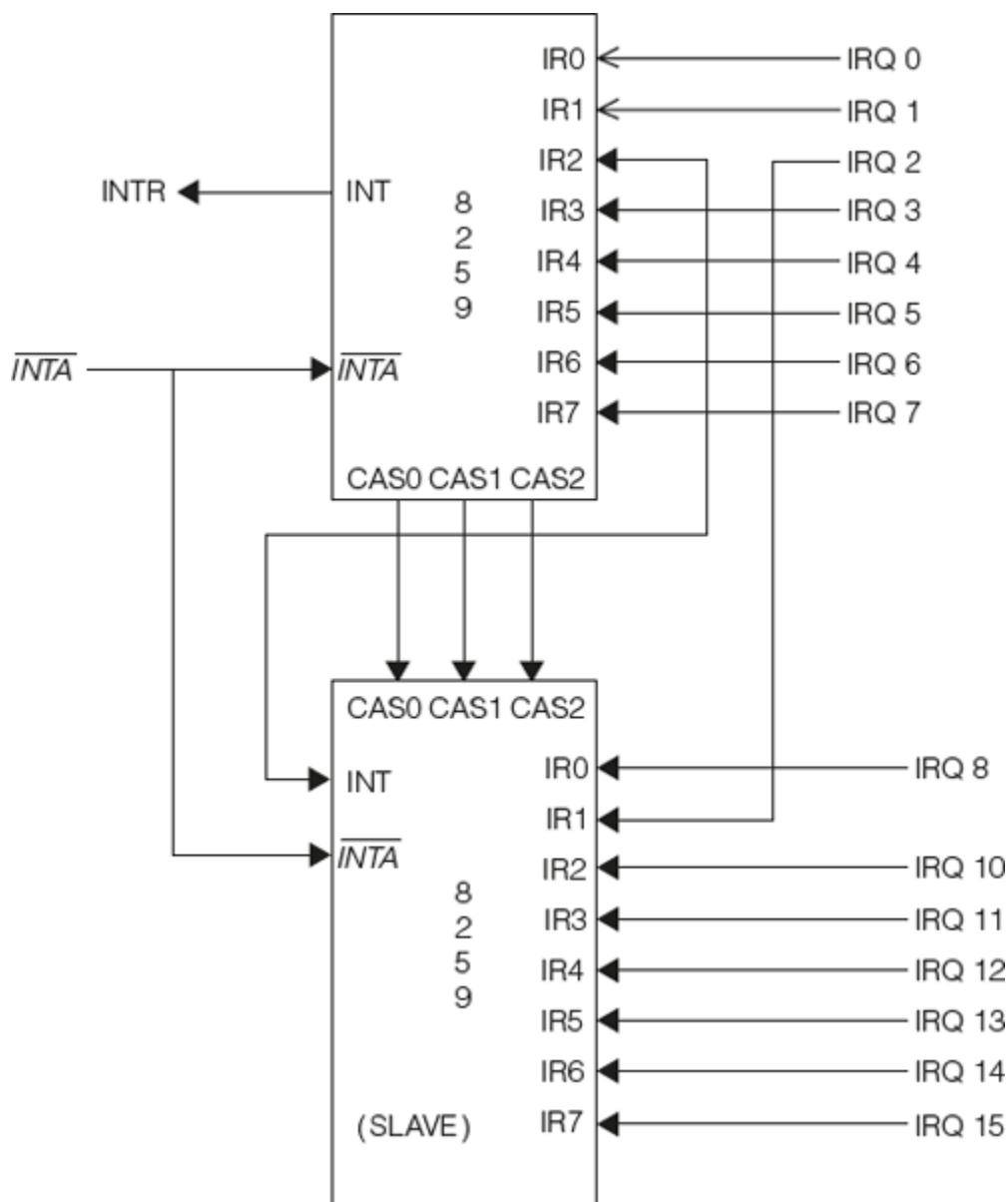


Figure 10.31 | PICs used in the PC-AT

The CPU only has one interrupt line; thus, the second controller had to be connected to the first controller in a master/slave configuration. IRQ2 was selected for this. By using IRQ2 for the second controller, no other devices could use IRQ2. What happened to all these devices using IRQ2? Nothing; the interrupt request line found on the bus was simply diverted into the IRQ 9 input. As no devices yet used the second PIC or IRQ9, this could be done.

As shown in Fig. 10.31, IRQ0 through IRQ7 are the master 8259's interrupt lines, while IRQ8 through IRQ15 are the slave 8259's interrupt lines. The actual names on the pins on an 8259 are IR0 through IR7. IRQ0 through IRQ15 are the names of the ISA bus lines to which the 8259's are attached. The devices connected to the IRQ lines are shown below. This is standard for the PC-AT.

Master 8259

- IRQ0 – Intel 8253 or Intel 8254 Programmable Interval Timer, i.e., the system timer
- IRQ1 – Intel 8042 keyboard controller
- IRQ2 – not assigned in PC/XT; cascaded to slave 8259 INT line in PC/AT
- IRQ3 – 8250 UART serial port COM2 and COM4
- IRQ4 – 8250 UART serial port COM1 and COM3
- IRQ5 – hard disk controller in PC/XT; Intel 8255 parallel port LPT2 in PC / AT
- IRQ6 – Intel 82072A floppy disk controller
- IRQ7 – Intel 8255 parallel port LPT1 / spurious interrupt

Slave 8259 (PC/AT and later only)

- IRQ8 – real-time clock (RTC)
- IRQ9 – no common assignment
- IRQ10 – no common assignment
- IRQ11 – no common assignment
- IRQ12 – Intel 8042 PS / 2 mouse controller
- IRQ13 – math coprocessor
- IRQ14 – hard disk controller 1
- IRQ15 – hard disk controller 2

Initially, IRQ7 was a common choice for the use of a **sound card**, but later, IRQ5 was used when it was found that IRQ7 would interfere with the printer port (LPT1). The **serial ports** are frequently disabled to free an IRQ line for another device.

KEY POINTS OF THIS CHAPTER

- The programmable interval timer chips 8253 and 8254 are similar in most ways except that the latter can be used at higher operating frequencies.
- The timer chip has three internal counters each of which has a clock input, a gate and an output pin.
- The count in any count-register can be read 'on the fly' by latching it into a latch.
- There are six operating modes for the timer, and writing suitable words into the control registers will enable any of these modes.
- The 8279 is a keyboard display interface chip which does automatic keyboard scanning as well as refreshing digits in a dynamic display.
- The 8279 has a display RAM into which a seven segment code can be written for displaying a character.
- The 8279 also has a FIFO RAM which stores the key codes corresponding to the keys pressed.
- A number of control words are needed to operate this chip.
- The third interfacing chip discussed in this chapter is the programmable interrupt controller 8259.
- This chip is designed to handle all the issues related to hardware interrupts arriving at the INTR pin of the 8086.
- It has a number of internal registers to handle interrupt requests, interrupt masks and so on.
- A number of initialization command words and a number of operational command words are needed to use this chip.
- In the current PC, two PICs are used to handle standard hardware interrupts.

QUESTIONS

1. In what way is the 8253 different from the 8254?
2. In which mode can symmetric square waves be obtained from the 8253?
3. What is meant by reading a count 'on the fly' and how is it done?
4. Explain the use of mode 1 of the timer chip.
5. What is the maximum BCD number that can be used as a count?
6. What are the actions performed by the 8279 to interface it to a dynamic display?
7. What are the actions performed by 8279 to let it be interfaced to a keyboard?
8. Differentiate between left and right entry display.
9. What information does a 'key code' hold?
10. What is the role of the decoder 74LS138 in Fig. 10.17?
11. Briefly explain how the 8259 channels an interrupt request from a peripheral to the 8086.
12. Discuss the roles of the following registers in handling multiple interrupt requests:

1. IRR
 2. IMR
 3. ISR
13. How is a type vector installed for a particular interrupt request?
 14. How can interrupt priorities be changed?
 15. Name two features which can be set using operational command words.

EXERCISES

1. Using the 8253 chip, generate square waves of frequencies 1 KHz, 12 KHz and 500 Hz, assuming that the input frequency is 2 MHz.
2. Using any two modes of the chip, generate asymmetric square waves.
3. Write a program with a count of 8888H loaded into count-register 2, and draw the output waveform for the waveform in Mode 1. What additional condition is required to facilitate the functioning of this mode?
4. Use an 8254 timer chip with a clock of 10 MHz and program it to operate in Mode 2 with three different values of N. Draw corresponding output waveforms.
5. Referring to Fig. 10.14, write a program to display ABCD on the display.
6. Referring to Fig. 10.14, write a program to get a rolling display.
7. Interface a keyboard to the 8279 as in Fig. 10.17, write a program that allows a key press and displays the corresponding key. Now, modify it such that when 1 is pressed, 6 is displayed, when 2 is pressed, 7 is displayed and similarly for all the keys.
8. For an 8259 PIC, design an ICW to fix up interrupt type numbers of your choice for the interrupt requests.
9. Draw a decoding circuit for an 8259, and write the corresponding ICW to make all the interrupt requests to be level triggered.
10. Write instructions to read the contents of the ISR and IRR.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

Peripheral Interfacing—III

IN THIS CHAPTER, YOU WILL LEARN

- The principles involved in serial communications.
- The terminology and standards of serial communications.
- RS-232 pins and connections.
- The use of the USART 8251 as an interface to computers.
- The modes and programming of 8251.
- The concept of direct memory access.
- The necessity and functions of a DMA controller in a computer system.
- Different types and modes of DMA.
- How to use the DMA controller IC 8237.

11.1 | Serial Communication Principles

Transferring data is what we mean when we use the word ‘communication’. Data is to be sent from the source to the destination, and it is necessary for the source and destination formats to be similar for compatibility between them to be ensured.

We have seen such data transfer in many contexts, and in all these cases, a number of bits are sent together, i.e., 8 bits, 16 bits or 32 bits. This is ‘parallel communication’. All the bits are sent and received together. Data transfer between registers in a processor is done this way. This is fine as long as the source and the destination are in close proximity, but when they are placed far apart physically, like, say, two computers in two different buildings, a lot of problems crop up. If we are sending 8 bits, eight long wires would be required, and the problem aggravates as the bit size of the data increases. To circumvent this and other related problems, ‘serial communication’ is chosen in preference to the parallel communication just discussed. Serial communication is when we send digital data one bit at a time, one after the other. Thus, 8 bits need 8 times the time required, compared to the previous case (assuming the same sending rate), but the direct advantage is that only one physical wire is required for transmission. This is a very great advantage when dealing with larger data word sizes, but along with this, a new set of issues arrive, which need to be dealt with effectively.

First, let us discuss the various terms likely to be encountered while learning serial communication.

11.2 | Simplex, Half Duplex and Full Duplex Communication

A Simplex Connection

is a connection in which the data flows in only one direction, from the transmitter to the receiver. This type of connection is useful if data does not need to flow in both directions (for example, from the computer to the printer or from the mouse to the computer.)

A Half-duplex Connection

(sometimes called an alternating or semi-duplex connection) is a connection in which data flows in one direction or the other but not both at the same time. At a time, transmission is done only in one direction and thus the full bandwidth of the line can be used for the transmission (like when talking on a radio).

A Full-duplex Connection

is a connection in which data flows in both directions simultaneously. Each end of the line can thus transmit and receive at the same time, which means that the bandwidth is divided by two for each direction of data transmission if the same transmission medium is used for both directions. Talking over a telephone line is a typical case of full duplex transmission. The serial port on the PC is a full-duplex device meaning that it can send and receive data at the same time. In order to be able to do this, it uses separate lines for transmitting and receiving data.

11.2.1 | Synchronous vs Asynchronous Communications

Now, let us assume we have a block of bytes to be transmitted and that the transmitter and receiver are connected by a serial line. Serial data transfer depends on accurate timing in order to differentiate the bits in the data stream. This timing can be handled in one of two ways: asynchronously or synchronously. **In asynchronous communication, the scope of the timing is a single byte at the maximum. In synchronous communications, the timing scope comprises one or more blocks of bytes.** In both cases, the transmitter and the receiver should know the state of each other. In synchronous transmission, synchronizing information is sent between the transmission of each block of data. The size of this 'block' depends on

the system. Even when real data is not being sent, a constant flow of bits allows each device (transmitter or receiver) to know the state of the other at any given time. That is, each character that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required, but only between the large blocks.

11.2.2 | Asynchronous Communications

In asynchronous transmission, the beginning and end of each byte of data must be identified by start and stop bits. The start bits indicate when the data byte is about to begin and the stop bits signal when it ends. The requirement to send these additional bits causes asynchronous communication to be slightly slower than the synchronous type.

An asynchronous line that is idle is identified with a value of 1 (also called a mark state). By using this value to indicate that no data is currently being sent, the devices are able to distinguish between an idle state and a disconnected line. When a character is about to be transmitted, a start bit is sent. A start bit has a value of 0 (also called a space state). Thus, when the line switches from a value of 1 to a value of 0, the receiver is alerted that a data character is about to be sent. Asynchronous communication is the prevailing standard in the personal computer industry, both because it is easier to implement and because it has the unique advantage that bytes can be sent whenever they are ready, as opposed to waiting for blocks of data to accumulate.

We will discuss this mode of communication in more detail. When the transmitter intends to send a byte, it sends a start bit to the idle line which is at the ‘mark’ or ‘1’ state. The start bit is a ‘space’ character, i.e., a ‘0’ bit. This indicates that what follows is the actual data byte. **Note that the LSB is sent first.** After all the eight bits are sent, one or two stop bits are sent. The type and number of stop bits must be defined for the system. The total character frame includes the data and the start and stop bits and parity bits for error checking and correction. A bit time can be defined as the time given for the transmission of one-bit, and the receiver samples the received voltage level at periodic intervals, known as the bit time, to determine whether a 0-bit or a 1-bit is present on the line. Observe Fig. 11.1. Note that one character can be 8 bits or less. A parity bit may be used for error control. The parity bit in each character can be set to none (N), odd (O) or even (E). However, the practice now is to have no parity bit but instead to have some other communication error control protocol to be used. The data sent using this method is said

to be framed. That is, the data is framed between the Start and Stop bits. **In transmission, the start bit is sent first, then the data bits with LSB first, followed by the parity and stop bits.**

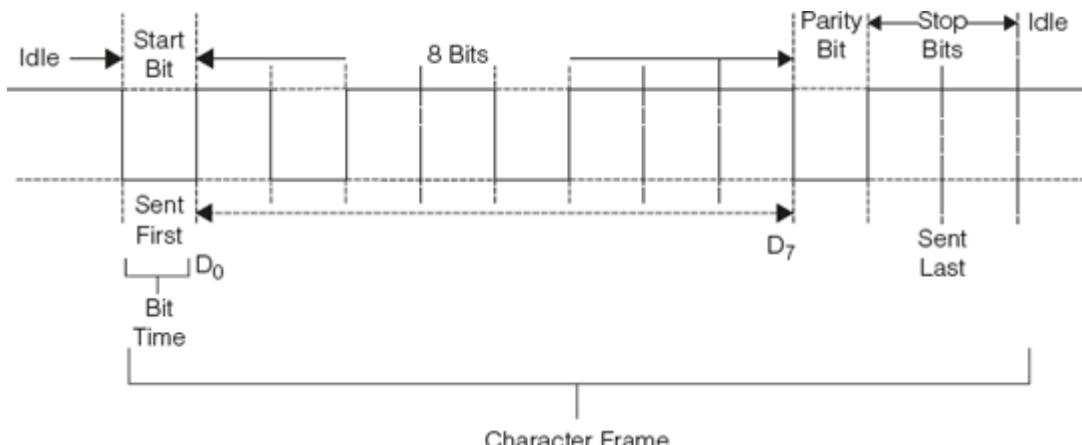


Figure 11.1 | A typical asynchronous character frame

11.2.3 | Transmission Rate

There are two terms in common use relating to transmission. One is bits per second. This is what we have seen in our discussion above. If the bps is specified to be 1200, it means 1200 bits are sent per second.

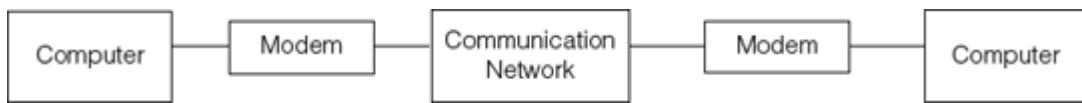


Figure 11.2 | A computer to computer communication system using a modem

There is another term that is commonly used in serial communications, and that is baud rate. Many people use these words interchangeably, which is not correct, but to understand the difference, we need to discuss the equipment called ‘modem’. For the case of direct serial data transmission from one computer to another in simple binary form, the baud rate and bps are the same.

11.2.4 | Modems

The need to communicate between distant computers led to the use of the existing phone network for data transmission. Most phone lines were designed to transmit analogue information – voice, while computers and their devices work in digital form – using pulses. So, in order to use an analog medium, a converter between the two systems is needed. This converter is the **modem**, which performs Modulation and Demodulation of data. It accepts serial binary pulses from a device and modulates some property (amplitude, frequency, or phase) of an analog

signal in order to send the signal in an analog medium. It also performs the opposite process, enabling the analog information to arrive as digital pulses at the computer or device on the other side of connection.

Wireless modems convert digital data into radio signals and back. Early modems used frequency shift keying wherein one frequency was used for digital ‘1’ and another frequency for ‘0’. In order to create faster modems, modem designers had to use techniques far more sophisticated than frequency-shift keying. First, they moved to phase-shift keying (PSK) and then to quadrature amplitude modulation (QAM).

11.2.5 | Baud Rate vs Bps

The difference between the two is complicated and intertwining. They are dependent and inter-related. However, the simplest explanation is that a bit rate is the number of data bits transmitted per second. Baud rate is the measurement of the number of times per second a signal in a communications channel changes per second.

Bit rates measure the number of data bits (i.e., 0's and 1's) transmitted in one second in a communication channel. A figure of 2400 bits per second means 2400 zeros or ones can be transmitted in one second, hence the abbreviation **bps**.

A baud rate, by definition, means the number of times a signal in a communications channel changes state or varies. For example, a baud rate of 2400 means that the channel can change states up to 2400 times per second. The term ‘change state’ means that it can change from 0 to 1 or from 1 to 0 up to X (in this case, 2400) times per second. It also refers to the actual state of the connection, such as voltage, frequency or phase level.

The main difference between the two is that one change of state of the communication signal can transmit one bit – or more or less than one bit and that depends on the modulation technique used. So, the bit rate (bps) and baud rate (baud per second) have this connection:

$$\text{bps} = \text{bauds per second} \times \text{the number of bits per baud}$$

The number of bits per baud is determined by the modulation technique. See these examples. When FSK ('Frequency Shift Keying', a transmission technique) is used, each baud transmits one bit; only one change in state is required to send a bit. Thus, the modem's bps rate is equal to the baud rate.

When we use a baud rate of 2400, and use a modulation technique called quadrature phase shift keying (QPSK) that transmits four bits per baud, we get the result that $2400 \text{ baud/second} \times 4 \text{ bits per baud} = 9600 \text{ bps}$. Such modems are capable of 9600 bps operation.

11.2.6 | RS-232 Standards

We have been talking about sending data as bits as either ‘1’ or ‘0’. As per TTL standards, these levels correspond to 5 V and 0 volts. However, during transmission over short distances (without a modem), say from one room to another, the TTL level signals will get easily corrupted by noise.

However, this problem does not occur normally, because we don’t send or receive at TTL levels; instead, we use a standard called RS-232C, which defines a different set of voltages/currents. RS-232 stands for Recommend Standard number 232 and C is the latest revision of the - standard. By this standard, before being transmitted, the TTL voltage levels are changed to a level between -3 to -25 V for a ‘1’ and $+3$ to $+25$ V for a ‘0’ (the voltages between -3 V and $+3$ V is undefined). This means that before sending, the bits should be changed to this level and reconverted to TTL levels on receiving. There are some standard chips available for doing this.

11.2.7 | RS-232 Level Converters

Almost all digital devices that we use require either TTL or CMOS logic levels. Two common RS-232 Level Converters are the 1488 RS-232 Driver and the 1489 RS-232 Receiver. Each package contains 4 inverters of one type, either Drivers or Receivers. See Fig. 11.3 which shows how and where these converters are used to connect two computers in close proximity using an RS-232 cable. A better IC for the same purpose is the MAX232, which can be used for conversion in both directions (see Fig. 11.4).

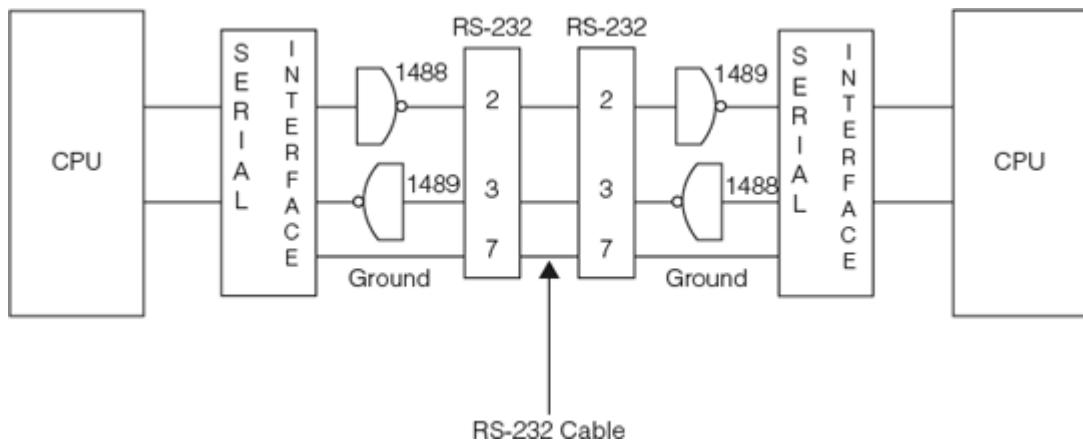


Figure 11.3 | Serial communication using RS 232 cable and line drivers/receivers

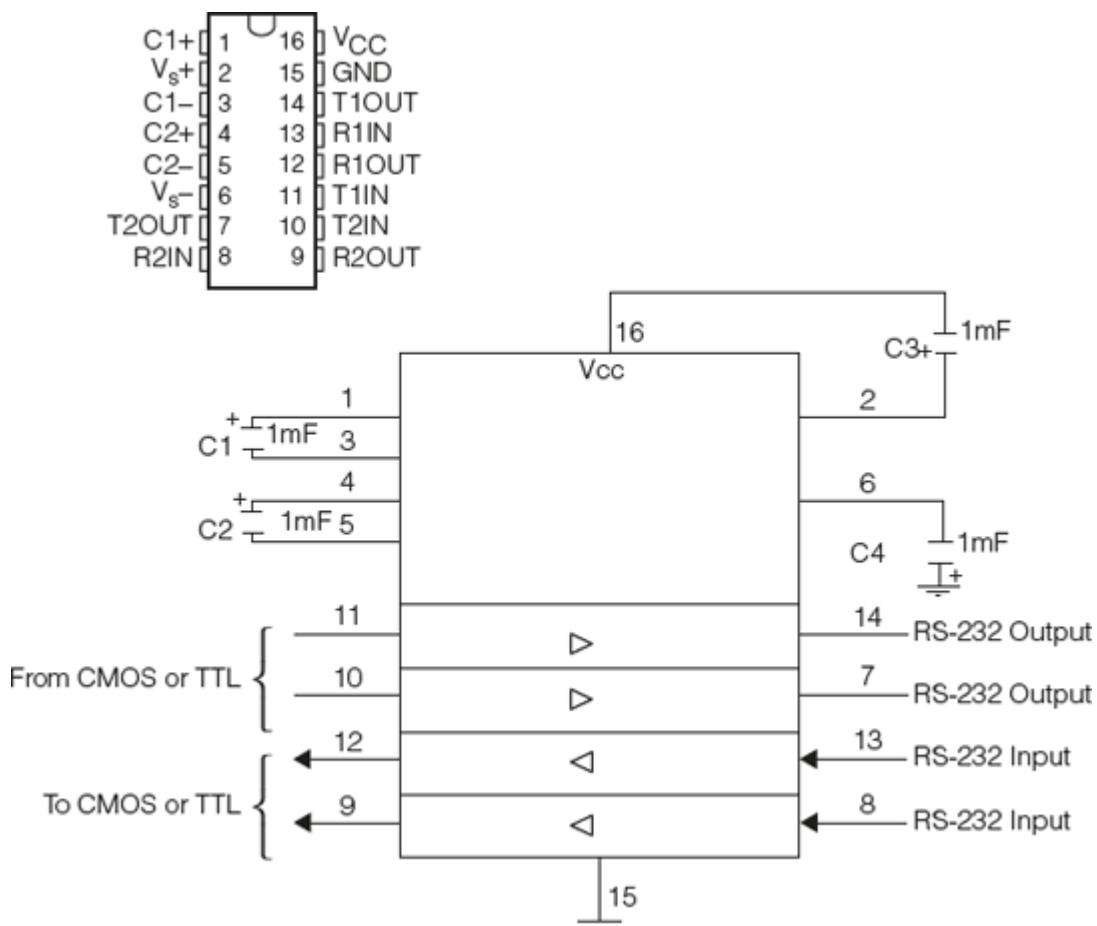


Figure 11.4 | MAX 232 chip pin diagram and a typical connection

11.2.8 | RS-232 Connectors

The serial ports on most computers use a subset of the RS-232 standard. The full RS-232 standard specifies a 25-pin D connector of which 22 pins are used. Most of these pins are not needed for normal PC communications, and indeed, most new PCs are equipped with male D type connectors having only 9 pins. Figure 11.5 shows the DB-25 and DB-

9 connectors and Tables 11.1 and 11.2 describe the pin functions of these connectors.

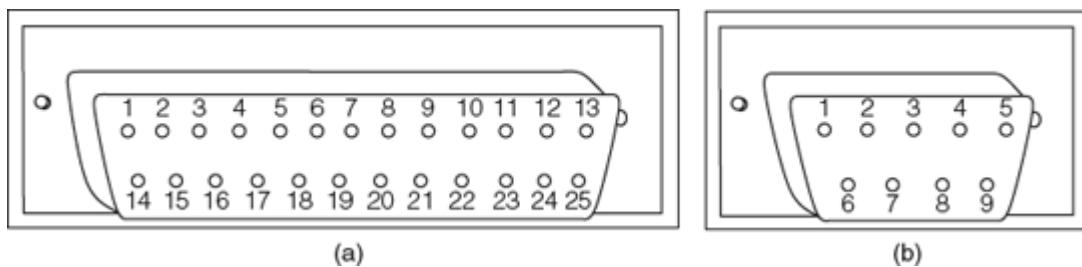


Figure 11.5 | RS-232 connectors **a** DB-25 male connector **b** DB-9 male connector

Table 11.1 | Pin Functions of the DB-25 Connector

Pin No.	Description	Pin No.	Description
1	Protective ground	13	Secondary clear to send
2	Transmitted data ($T \times D$)	14	Secondary transmitted data
3	Received data ($R \times D$)	15	Transmit signal element timing
4	Request to send (\overline{RTS})	16	Secondary received data
5	Clear to send (\overline{CTS})	17	Receive signal element timing
6	Data set ready (\overline{DSR})	18	Unassigned
7	Signal ground	19	Secondary request to send
8	Data carrier detect (\overline{DCD})	20	Data terminal ready (\overline{DTR})
9	Data test pin	21	Signal quality detector
10	Data test pin	22	Ring indicator
11	Unassigned	23	Data signal rate select
12	Secondary data carrier detect	24	Transmit signal element timing
		25	Unassigned

Table 11.2 | Pin Functions of the DB-9 Connector

Pin No.	Description
1	Data carrier detect (\overline{DCD})
2	Received data ($R \times D$)
3	Transmitted data ($T \times D$)
4	Data terminal ready (\overline{DTR})
5	Signal ground (GND)
6	Data set ready (\overline{DSR})
7	Request to send (\overline{RTS})
8	Clear to send (\overline{CTS})
9	Ring indicator (RI)

11.2.9 | DCE and DTE Devices

The two terms that are commonly seen in serial communication literature are DTE and DCE. DTE stands for Data Terminal Equipment, and DCE stands for Data Communications Equipment. These terms are used to indicate the pin-out for the connectors on a device and the direction of the signals on the pins. The computer and terminals that send and receive data are DTE devices, while other devices like the modem that transfer data are usually DCE devices. To avoid confusion, remember that a typical DTE is a PC and a typical DCE is a modem. The RS-232 standard states that DTE devices use a 25-pin male connector, and DCE devices use a 25-pin female connector. It is possible therefore to connect a DTE device to a DCE using a straight pin-for-pin connection as in Fig. 11.6.

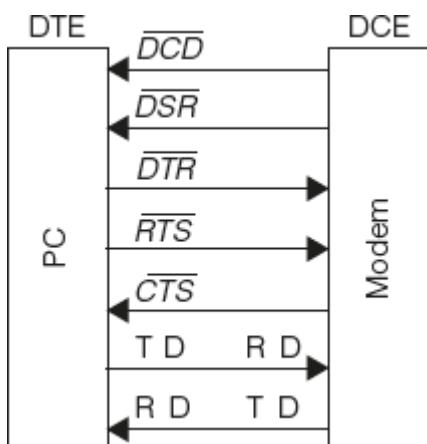


Figure 11.6 | Typical DTE–DCE connection

However, to connect two like devices, say a DTE with another DTE device, like two PCs, a null modem cable is to be used. What is a null modem? See Fig. 11.7. Null modem cables cross the transmit and receive lines in the cable, so that the transmit pin of one is connected to the receive pin of the other and so on. In addition to transmit and receive, the signals *DTR* and *DSR* as well as *RTS* and *CTS* are also crossed in a null modem connection.

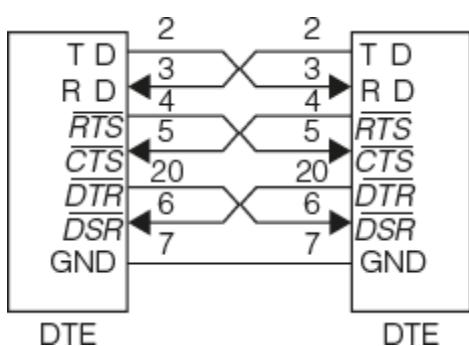


Figure 11.7 | DTE–DTE connection using a null modem connection

We can say that serial communication using RS-232 standards is possible using 25 pins, 9 pins and also just three pins (See Fig. 11.8). In the last case, transmission and reception is done without additional handshaking. T × D is the line for transmitted data and R × D for received data.

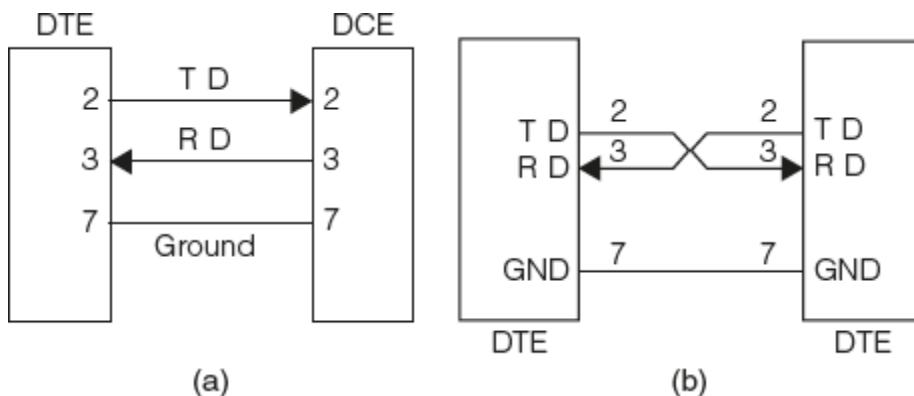


Figure 11.8 | **a** DTE–DCE connection using 3 pins **b** DTE–DTE connection with a null modem

11.2.10 | Handshaking in the RS232C Protocol

A number of signals seem to be involved in the RS232C protocol, some of which are control signals. They are used for handshaking and let us see the sequence in which these signals occur. Assume that a DTE equipment (for example, a COM port of a PC) and a DCE equipment (serial port of a modem) are to perform data transfer. The handshaking signals involved in this data transfer are as follows. Refer to Fig. 11.6 to understand the direction of the signals.

DTR (Data Terminal Ready)

When the DTE terminal is turned on, after its self test, it sends out the DTR indicating the robustness of the COM port and its readiness for communications.

DSR (*Data Set Ready*)

When the DCE terminal is turned on, it sends out the DSR signal after its self-test. In this case, the modem sends this signal to the PC's Com port, which also indicates that it has made a connection to the network (telephone or wireless network). Together DTR and DSR are used to confirm that both devices are connected and turned on.

RTS (Request to Send)

Now that both the equipments are ready, data transmission can occur. If the DTE has a byte to send, it asserts the RTS signal, which is received by the DCE equipment (modem in this case).

CTS (Clear to Send)

The receiving equipment (the modem in this case) responds to the RTS signal by this signal. CTS makes it known to the transmitting equipment that buffer space is available for the incoming data. Together RTS and CTS are used for hardware flow control; it regulates the flow of data. If no buffer space is available in the receiver side, the transmission must wait until the buffer is cleared.

DCD (Data Carrier Detect)

This is sent by the modem to indicate that a valid carrier has been detected.

RI (Ring Indicator)

This is an output from the modem that the telephone is ringing. The modem toggles (goes on and off) the state of this line when an incoming call rings the phone. The Data Carrier Detect (DCD) and the Ring Indicator (RI) lines are only available in connections to a modem. Because most modems transmit status information to a PC when either a carrier signal is detected (i.e., when a connection is made to another modem) or when the line is ringing, these two lines are rarely used. The RI signal is not shown in Fig. 11.6.

11.3 | The Programmable Serial Communication Interface

Since we have had an overview of serial communications, let us dig a bit deeper. We know that within the processor and between the processor and memory of most I/O devices, data is transferred in parallel. So, if we need to take it out from such a system in serial form, it is obvious that a parallel to serial conversion is necessary. Similarly, if serial data is received from a serial line, it needs to be converted to parallel form before the processor can process it. Another point is that all the serial communication rules and protocols must be ensured before we try to send and receive serial data. All these matters will have to be taken care of by a dedicated hardware and there are several specialized ICs available for this purpose. Listing out a few, we have:

8250 – Universal Asynchronous Receiver Transmitter (UART)

8251 – Universal Synchronous Asynchronous Receiver Transmitter (USART)

16550 – Universal Asynchronous Receiver Transmitter (UART)

Now, let us try to understand the features of the USART 8251.

11.3.1 | Universal Asynchronous/Synchronous Receiver/Transmitter – USART 8251

The 8251 USART is designed to be directly compatible with the x86 processors as well with the other Intel family processors. It caters to synchronous as well as asynchronous transmission. It can accept data from the processor in parallel form and convert it into a continuous serial stream for transmission. Simultaneously, it can also receive serial data streams and convert them to parallel format for use by the processor. It signals the processor whenever it is ready to accept a new character for transmission and also whenever it has received a character for the CPU. It inserts and deletes the framing bits as necessary in the communication mode it is set. In effect, the intricacies of the communication protocols being used are not felt by the programmer beyond the difficulty of writing into command registers. Additionally, it allows the CPU to read its complete status at any time.

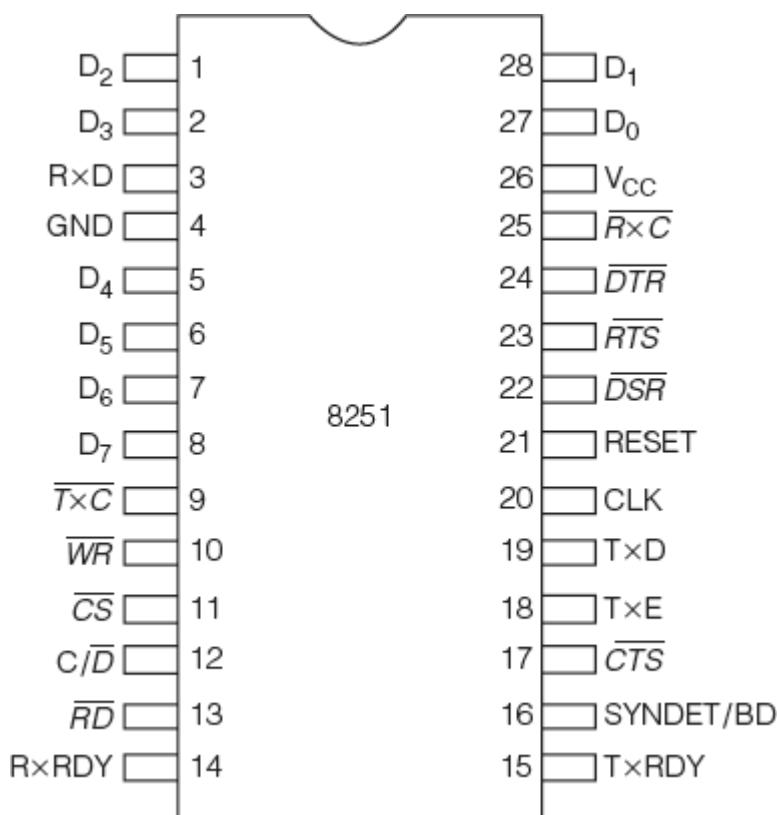


Figure 11.9 | Pin diagram of the USART

11.3.2 | Functional Block Diagram

Let us examine the functional block diagram of the chip.

11.3.2.1 | Data Bus Buffer

It is the block to which the data bus (three state bi-directional and 8-bit) of the processor is connected. It is through the data bus that control words and status words and data are sent from the processor to the chip and vice versa.

11.3.2.2 | Transmitter Buffer

This block accepts parallel data from the data bus buffer, converts it to a serial bit stream, inserts the appropriate extra bits (based on the communication technique) and outputs a composite serial stream of data on the $T \times D$ (Transmitter Data) output pin on the falling edge of $T \times C$. On reset, the $T \times D$ line will be held in the mark state (which corresponds to an idle state). It will begin transmission upon being enabled if $CTS = 0$. When the transmitter is empty, or if CTS or $T \times$ Enable is OFF, the line will once again go to the idle state of ‘mark’.

11.3.2.3 | Transmitter Control

This block manages all the activities associated with transmission. It accepts and issues signals required for this. The signals are:

$T \times RDY$ (Transmitter Ready)

This is an output signal saying that the transmitter is ready to accept a data character to be sent. This can be used to interrupt the processor, or the processor can check its state by reading the status register, in the polled mode of operation.

$T \times Empty$ (Transmitter Empty)

When the 8251 has no characters to send, this output pin will go high. It is cleared on receiving a character from the processor. It can be used to indicate the end of a transmission, so that next, the CPU can turn around for reception (in the case of half duplex mode, say).

$T \times C$ (Transmitter Clock)

This is the transmitter clock input pin, which controls the rate of transmission. In the synchronous transmission mode, the baud rate is

equal to the $T \times C$ _frequency. In asynchronous transmission mode, the baud rate is a fraction of this frequency. This is selected by the mode control word. The falling edge of the $T \times C$ clock shifts the serial data out of the 8251.

11.3.2.4 | Receive Buffer

The receiver accepts serial data, converts this serial input to parallel format, checks for data that are unique to the communication technique and sends an ‘assembled’ character to the processor. Serial data is taken as input on the $R \times D$ pin and is clocked on the rising edge of the $R \times C$.

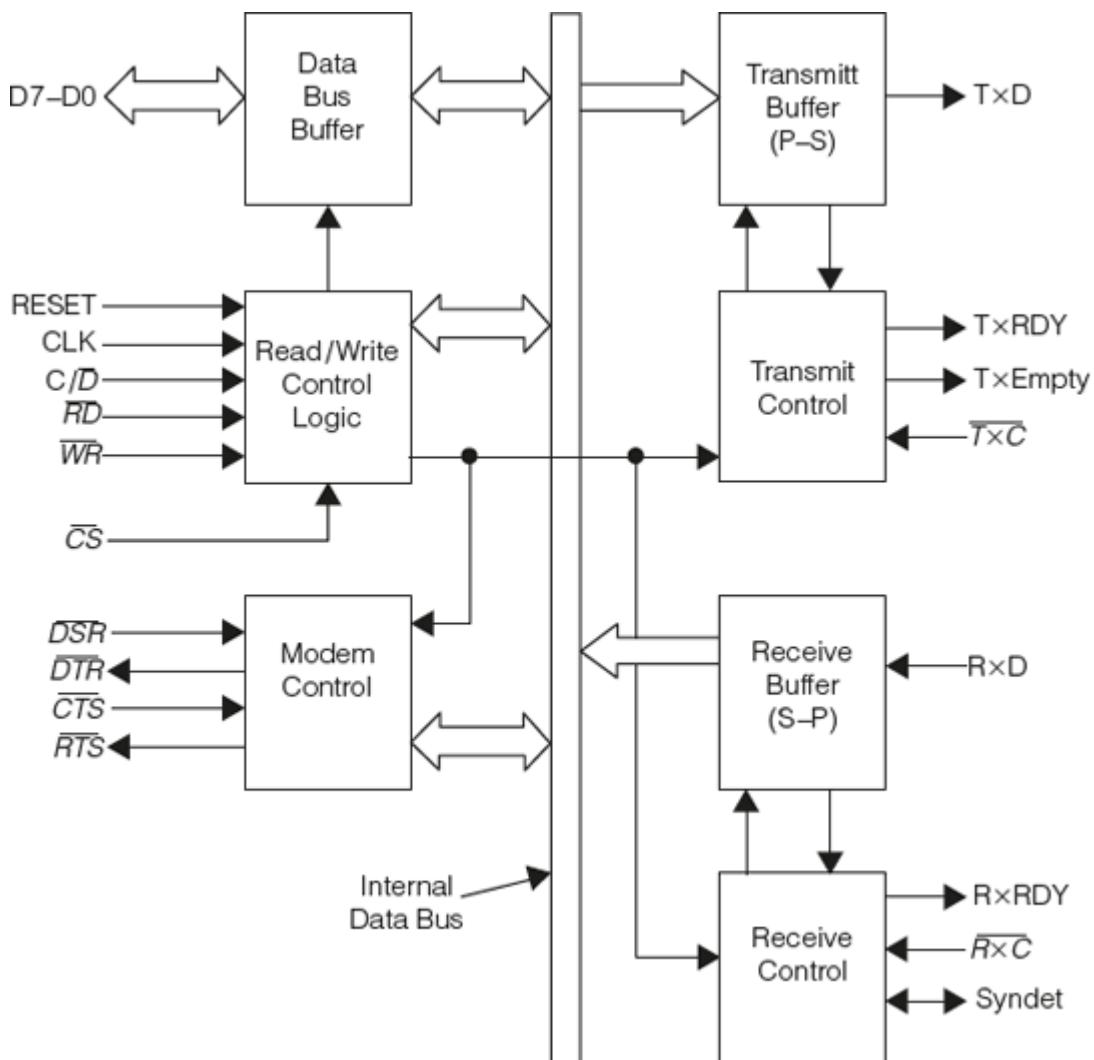


Figure 11.10 | Internal block diagram of the 8251

11.3.2.5 | Receiver Control

This functional block manages all receiver related activities. Before starting to receive serial characters on the $R \times D$ line, a valid ‘1’ must first be detected after a chip master reset. Once this is obtained, a search for a

valid low (Start bit) is enabled. This is so only in the asynchronous mode, and this is only done once for each master reset. The signals associated with receiver control are:

$R \times RDY$ (*Receiver Ready*)

This output signal indicates that the 8251 contains a character that is ready to be input to the CPU. $R \times RDY$ can be connected to the interrupt structure of the CPU. If polling is being used, the CPU can check the status of this pin using a status read operation.

$R \times C$ (*Receiver Clock*)

This input pin controls the rate at which receiving is done. In the synchronous mode, the baud rate is equal to the actual frequency of $R \times C$. In the asynchronous mode, the baud rate is a fraction of this frequency. Data is sampled in at the rising edge of the $R \times C$ signal.

Note In most cases, the 8251 will be handling the transmission and reception of the same link. Hence, the baud rates will be the same and so the $R \times C$ and $T \times C$ can be tied together to a single frequency source.

Syndet / Brkdet (Sync detect or Break Detect)

This is a bidirectional pin and is used in the synchronous mode as SYNCH DETECT. It is used either as input or output, programmable using the control word. In the asynchronous mode, it is used as an output pin as 'BREAK DETECT'. It will go high to indicate a 'break' or 'disconnection'. It goes high whenever the receiver remains low through two consecutive stop bit sequences (including start bits, data bits and parity bits). Break detect may also be read as a Status bit.

11.3.2.6 | R/W Control Block

In this block, there are the read *RD*, write *WR*, chip select *CS*, reset and *C/D*. Only the last pin needs an explanation.

C/D (Control/Data)

This is an input signal which, in conjunction with the *CS*, can be used for selection of the data register or control/status register.

\overline{CS}	C/\overline{D}	Register selected
0	0	Data Register
0	1	Mode, command and status register

11.3.2.7 | Modem Control Block

The pins in this block perform the function of interfacing to a modem and they are the control / status signals used therein. The 8251 has a set of control input and outputs that can be used to simplify the interface to any modem. However, these signals are general-purpose in nature and can be used for functions other than modem control. Refer to Section 11.2.10 to know how these signals are important in serial communication, in general.

DSR (Data Set Ready)

The DSR input signal is a general-purpose, one-bit input port. Its condition can be tested by the CPU using a status read operation. The DSR input is usually used to test modem conditions such as Data Set Ready.

DTR (Data Terminal Ready)

The DTR output signal is a general-purpose, one-bit output port. It can be set low by programming the appropriate bit in the command instruction word. The DTR output signal is normally used for modem control.

RTS (Request to Send)

This output signal is a general-purpose, one-bit output port. It can be set low by programming the appropriate bit in the Command instruction. The RTS output signal is normally used for modem control.

CTS (Clear to Send)

A low on this input enables the 8251 to transmit serial data if the T × Enable bit in the command byte is set to a ‘one’. If either a T × Enable OFF or CTS OFF occurs while the transmitter is in operation, the transmitter will transmit all the data in the USART written prior to the Transmitter Disable command before shutting down.

11.3.3 | How This Chip Operates

Like any other interfacing chip, the chip functions according to the way we program it. A set of control words must be sent to it to support the communication mode in which it is to operate. Once the chip is programmed for use, it is ready for transmission or reception. For transmission, the $T \times RDY$ output is raised high to tell the CPU that it is ready to transmit the character the CPU sends. On receiving this affirmation, the CPU writes a character into the 8251 and then the $T \times RDY$ pin is reset.

In the case of reception, the 8251 receives a character from a modem or serial I/O device and then it raises the output pin $R \times RDY$ to communicate this to the CPU. The CPU then reads into it the received data and the $R \times RDY$ pin is reset. $T \times C$ and $R \times C$ are connected to the same clock source. $T \times RDY$ and $R \times RDY$ pins can be used in the interrupt mode or status checking mode (polling) for transmission and reception to occur as outlined above. $T \times D$ is the pin on which serial data is transmitted and $R \times D$ is the pin on which serial data is received.

11.3.4 | Programming the 8251

Prior to starting data transmission or reception, the 8251 must be sent a set of control words. This must be done after an external or internal reset. The control words are split into two formats.

1. Mode control word
2. Command word

Just after reset, the mode word must be sent to the chip. Any word that comes after the mode word is considered to be the command word, and command words can be written any time during the operation of the 8251. To change the mode, we can write a new mode word but this can be done only after an internal or external reset. An internal reset can be initiated by setting the ‘reset’ bit in the command word (bit D_6).

The USART, as the name implies, can be used for synchronous as well as asynchronous communications. The communication format is different for the two cases and they should be considered separately. We are interested only in the asynchronous mode, and only that will be discussed here. The data sheet of the chip may be referred for details of the synchronous mode.

11.3.5 | Asynchronous Mode

Let us start by writing the control words. Figure 11.11 shows the format of the mode word.

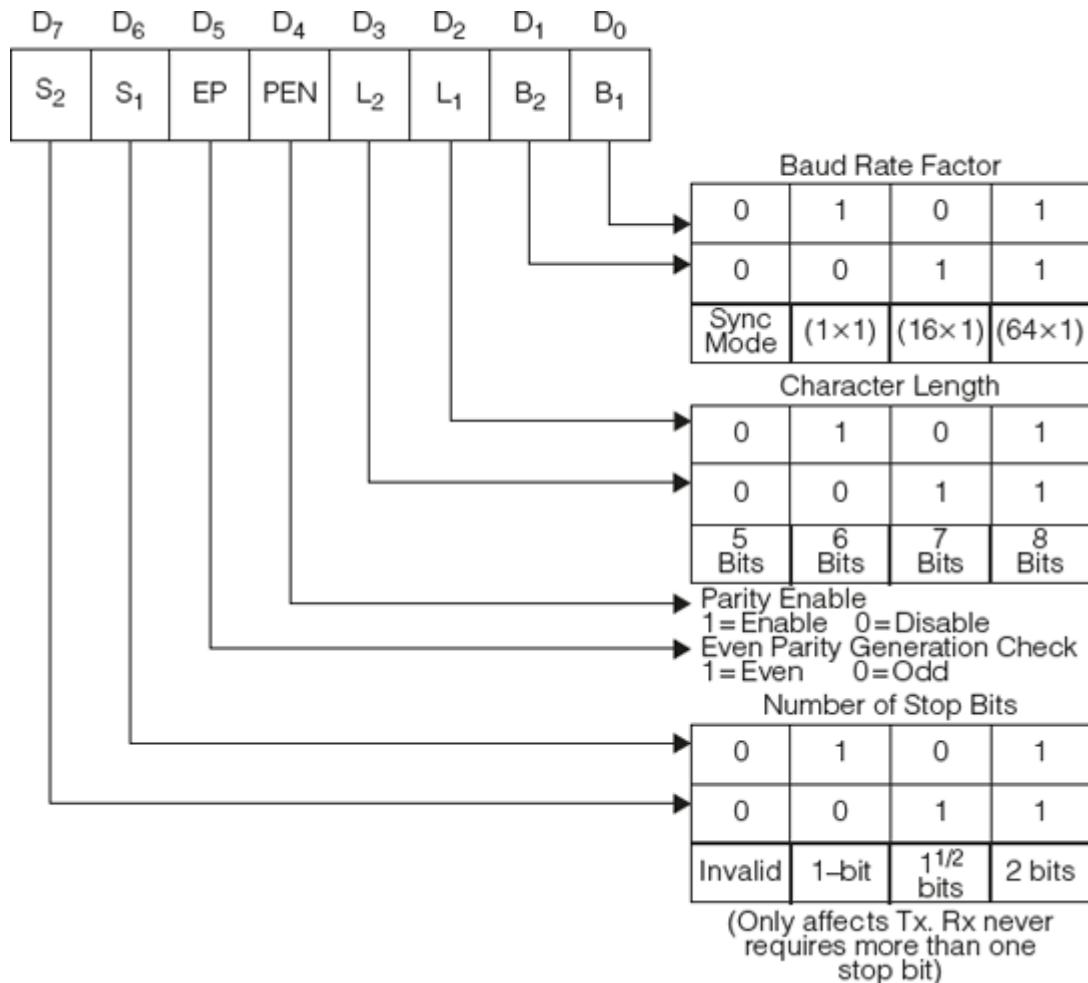


Figure 11.11 | Mode control word

11.3.5.1 | Mode Instruction Format

The mode word, as shown, fixes up the baud rate, number of characters, and stop bits for transmission. Remember that when the line is idle, it is in the mark ('1') state. A start bit is always low. This should be followed by the data bits and then the stop bits. The insertion of start and stop bits along with a character is called 'framing'. The baud rate is a fraction of the clock frequency ($T \times C$ and $R \times C$ are tied together for a link) and the factor can be 1, 16 or 64. If $T \times C = 9600$ Hz, the baud rate is 9600 bps for a factor of 1 and $9600/16 = 600$ for a factor of 16. If $D_1 D_0 = 00$, it means 'synchronous operation'. For using the chip in asynchronous mode, these bits should have values other than 00, which fix up the baud rate factor as well.

The character length can be 5, 6, 7 or 8 bits and it does not include the framing bits or parity bits. If the number of characters is less than 8, the

least significant data bus bits will hold the data; unused bits are don't cares when writing data into the 8251 and will be zeros when reading. When parity is enabled, it is not considered as one of the data bits, and the parity bit cannot be read on the data bus. The upper most two bits specify the number of stop bits. Figures 11.12 and 11.13 show clearly the transmission and reception formats. It should be clear that a character is framed and sent. On reception, the framing bits are removed and only the 'character' is considered as the received byte.

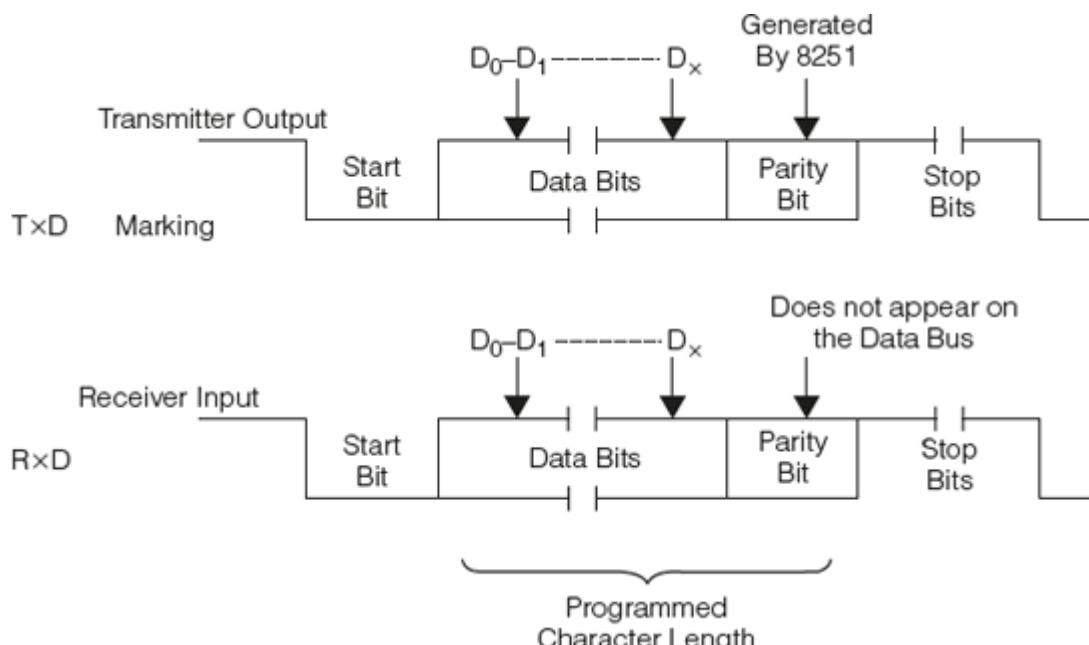


Figure 11.12 | The framed character at T × D and R × D pins

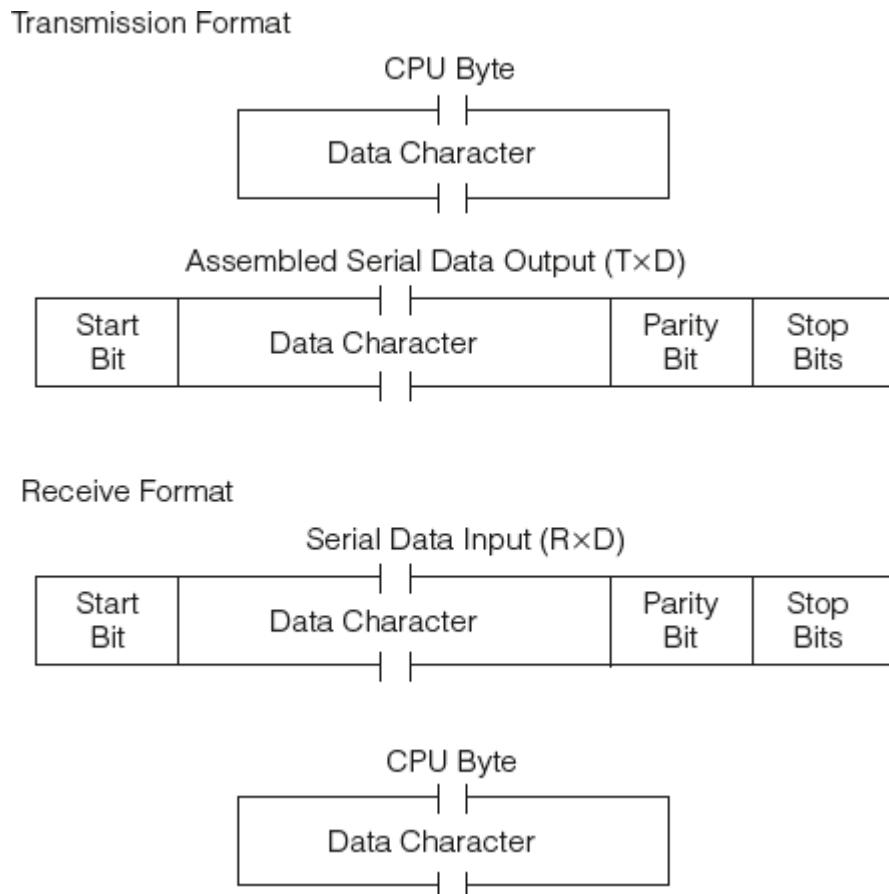


Figure 11.13 | Transmission and reception formats

Example 11.1

Write the mode control word for a case of asynchronous transmission, with an 8-bit data format, one stop bit, odd parity and 10000 baud rate approximately. The $T \times C$ clock is 150 KHz.

Solution

The mode control word is defined referring to Fig. 11.11.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	1	1	1	0

i.e., 5EH

$D_7, D_6 = 01$ for 1 stop bit

$D_5, D_4 = 01$, odd parity enabled

$D_3, D_2 = 11$, 8-bit character

$D_1 D_0 = 10$ baud rate factor of 16x

The clock source gives a clock of 150 KHz; the baud rate factor is 16x, i.e., 1/16.

Hence, the baud rate is $(150 \times 10^3) / 16 = 10,000 = 10\text{ K}$ approximately.

11.3.5.2 | Command Instruction Format

Figure 11.14 shows the command instruction format.

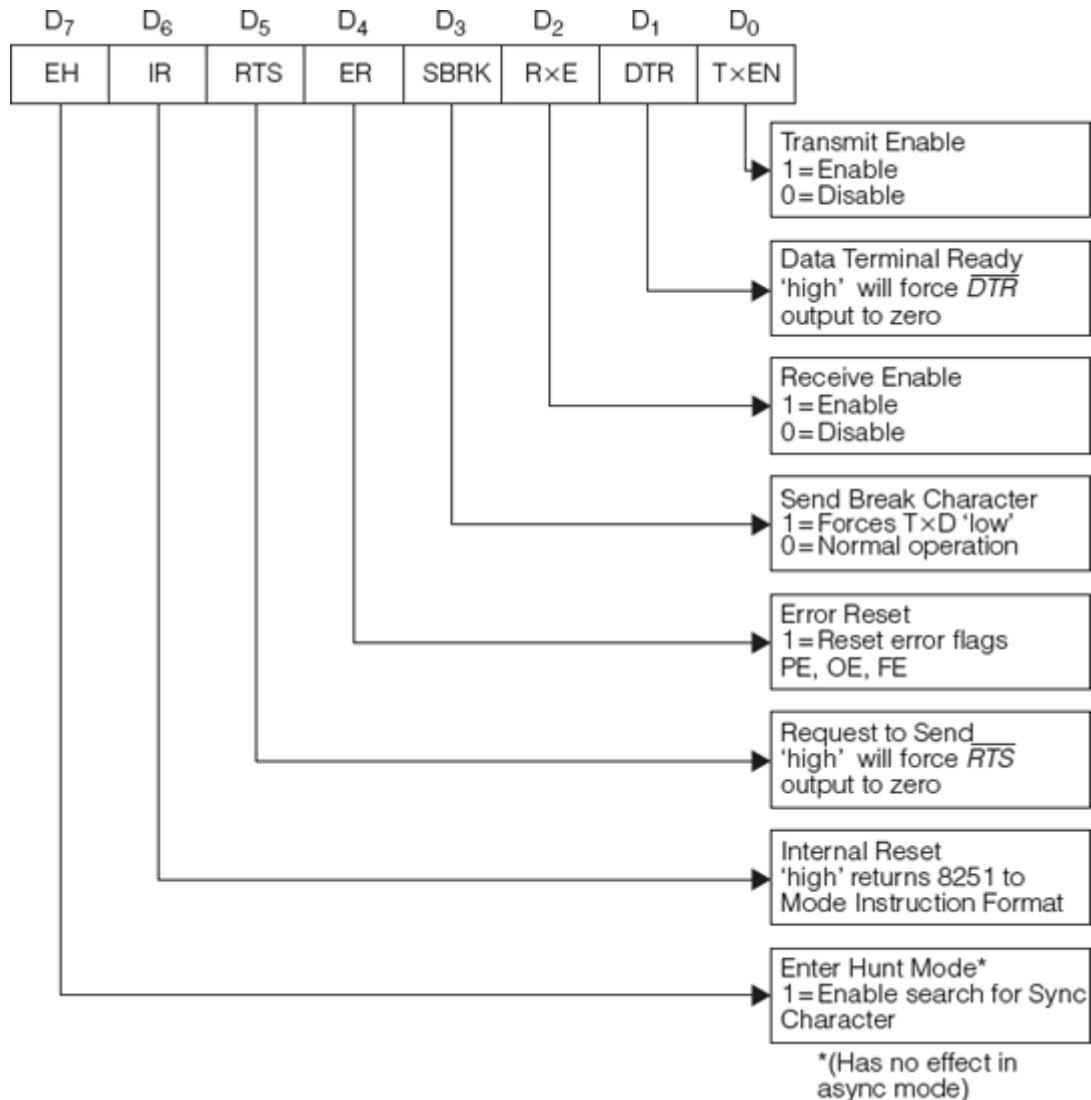


Figure 11.14 | Command instruction format

Example 11.2

For a simple operation, we have to enable transmit and receive, make DTR to be 0 and RTS to be 0 and enable error reset. Design the command word.

Solution

The command word is

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	1	1	0	1	1	1

i.e., 37H

11.3.5.3 | Status Word

The 8251 has a status word which enables us to read the status of the device during its operation. This can be read as and when necessary and used to perform transmission and reception in the polled mode.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
DSR	SYNDET	FE	OE	PE	T × EMPTY	R × RDY	T × RDY

Let us examine the important status bits:

T × RDY (Transmitter Ready)

This indicates that the USART is ready to accept data or command from the processor to which it is connected.

R × RDY (Receiver Ready)

This indicates that the chip has received a character and is ready to transfer it.

T × Empty (Transmitter Empty)

This means that the transmitter buffer is empty. There are three error flags in the status word but their setting does not inhibit the operation of the 8251. They are reset by the command word.

PE (Parity Error)

This bit is set when there is a parity error.

OE (Overrun Error)

This flag is set when the processor does not read a character before the next one becomes available. In the case that this bit gets set, the previous overrun character is lost.

FE (Framing Error)

This is applicable for asynchronous communication only. This flag is set when a valid stop bit is not detected at the end of every character.

Syndet–Synch Detect

This pin is used in synchronous mode for sync detect and is used in asynchronous mode for break detect.

DSR (Data Set Ready)

This bit indicates that DSR is at zero level.

11.3. 6 | Programming the 8251

Now that we have got an idea of the important aspects of the chip, we can program the chip for transmission and reception.

Loop Back Mode

We will first do transmission and reception in the same chip – send a character from the transmitter section of the USART and receive it in the receiver section therein. For that, the following connections are to be done (see Fig. 11.15). The data register has an address of CoH and the control/status an address C2H. The transmitter and receiver clocks are from the same clock source. T × D and R × D are to be interconnected, so also the *CTS* and *RTS* pins. Then, initialize the 8251 with the mode and command words, transmit a character, receive it, and save it in memory.

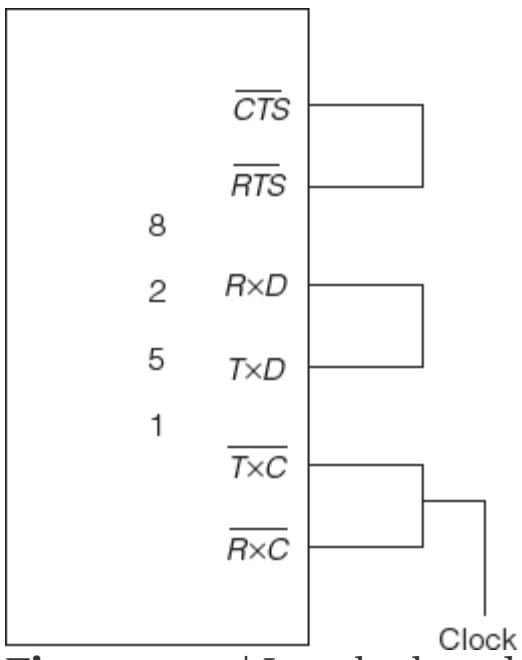


Figure 11.15 | Loop back mode

Example 11.3

Write the program for transmission and reception in the loopback mode.

Solution

The mode control word used is shown here. Parity is disabled.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	1	0	0	1	1	1	0

i.e., 4EH

D₇ D₆ = 01 for 1 stop bit

D₅ D₄ = 00 parity disabled

D₃ D₂ = 11, 8-bit character

D₁ D₀ = 10 baud rate factor of 1/16

```

TRANS DB 'S'           ;the character to be transmitted
RECD DB?              ;space for received character

DATR    EQU 0C0H
CSR     EQU 0C2H

MOV AL, 4EH            ;mode word
OUT CSR, AL           ;send it to control/status register
MOV AL, 37H            ;command word
OUT CSR, AL           ;send it to control/status register
MOV AL, TRANS          ;move character to AL
OUT DATR, AL           ;send it to data register
IN AL, DATR            ;move received data to AL
MOV RECD, AL           ;save it in memory
END

```

Example 11.4

Next, we will connect two microprocessor kits serially and perform transmission and reception. The transmission through one kit will be through its RS-232 cable and reception by the other cable will also be similar. Both the kits are clocked from the same source.

Solution

The sequence should be such that the receiver program is run first. The receiver then is in a loop waiting for a start bit to be received. Then, run the transmitter program. When the receiver senses the start bit, it starts reception and saves the received character string in memory.

TRANSMITTER PROGRAM

```

DATR    EQU 0C0H
CSR     EQU 0C2H

TRANS DB 'INDIAN'    ;location for character string

```

```

LEA BX, TRANS      ;point BX to character string
MOV CX, 06         ;CX = the number of characters
MOV AL, 4EH        ;mode word
OUT CSR, AL        ;send it to control register
MOV AL, 33H        ;command word to enable Tx
OUT CSR, AL        ;send it to control register
REPEA: IN AL, CSR ;read in status register
        AND AL, 01 ;test if D0 is 1. i.e., is Tx ready?
        JE REPEA ;if not '1', keep testing
NXT:   MOV AL, [BX] ;otherwise get character in AL
        OUT DATR, AL ;send it to the data register
        INC BX       ;increment pointer
        LOOP NXT    ;continue until CX=0
        END

```

RECEIVER PROGRAM

```

DATR   EQU 0C0H
CSR    EQU 0C2H

RECD DB 6 DUP(0)  ;location for received data
LEA SI, RECD     ;point SI to character string
MOV CX, 06        ;CX = the number of characters
MOV AL, 4EH        ;mode word
OUT CSR, AL        ;send it to control register
MOV AL, 36H        ;command word to enable receiver
OUT CSR, AL        ;send it to control register
REPEA: IN AL, CSR ;read in status register
        AND AL, 02 ;test if D1 is 1.
                      ;i.e., is Rx ready?
        JE REPEA ;if not '1' keep testing
NXT:   IN AL, DATR ;get received character into AL
        MOV [SI], AL ;move it to memory
        INC SI       ;increment pointer
        LOOP NXT    ;continue until CX=0
        END

```

The salient points of the programs are:

1. The mode control word for both the transmitter and receiver are the same. CSR is the control /status register and DATR is the data register.
2. The command words enable transmission or reception and causes RTS pin to be low. It is sufficient to enable only the transmitter in the transmitter program and the receiver in the receiver program. Thus, the command word for the transmitter is 33H, and for the receiver, it is 36H.
3. Each of the transmitted bytes is framed and sent. In the receiver, the framed character is received only if the start bit is sensed. This is done automatically by the 8251.
4. After the mode word and command word are sent to the control/status register, this register is 'read' to confirm if T × RDY is high (for transmission) and R × RDY is high for reception. Only after this is ensured,

- the data for transmission is sent to the data register (for transmission), and the received byte is read (in the receiver program).
5. The characters to be sent are stored in a memory location named TRANS at the transmitter. On reception, they are saved in a location named RECD at the receiver side.

11.4 | Internal Reset on Power Up

When power is first turned on, the 8251 is expected to come to the reset state but sometimes it does not do so. Instead, it may come up in the mode, sync character or command format. For normal operation, we need it to be ‘reset’; then we send the mode word and then the command word. To ensure this to be done, we can get it into the ‘command instruction format’ and from there cause an ‘internal reset’. This is done by executing the worst-case initialization sequence (sync mode with two sync characters). Loading three OOS into the control register configures sync operation and writes two dummy OO sync characters. An internal reset command may then be issued to bring the chip to the idle state. The following sequence of instructions must precede the program lines of Examples 11.3 and 11.4

```

MOV AL, 00          ;AL = 00
OUT CSR, AL         ;send AL to CSR
OUT CSR, AL         ;send AL to CSR
OUT CSR, AL         ;send AL to CSR
MOV AL, 01000000B   ;command word with D6 = 1
OUT CSR, AL         ;send to CSR for internal reset

```

The Serial Port of the PC

As already mentioned, the PC supports only asynchronous serial communication. It uses the UART chip 8250 for interfacing. This chip is, incidentally, a more powerful chip than the 8251. The 8250 was improved along the years that the processors used in PCs became more and more powerful – now 16450, which is a faster version of 8250, is being used. Recently, a better version, NS16550, has appeared in the market and is starting to be used widely.

Is the Serial Port on the PC Obsolete?

The serial port used to be found on almost all PCs but not anymore. Thus, in 2014, it is obsolete and not found in PCs or even embedded boards. It is now called a ‘legacy’ port. Laptops and Macs stopped being sold with serial ports several years before desktops did. However, if one

needs a serial port, it is possible to buy one and install it. However, as is seen now, the serial port as well as the parallel port are obsolete, because of being superseded by the USB port.

11.5 | Direct Memory Access

Data transfer between peripherals and memory is a frequent activity in any computer system. There are three methods by which this can be done; we have seen two such methods – polling and interrupt based data transfer. Polling is unacceptable, except for very small systems, because it keeps the processor in a waiting loop. Interrupt based data transfer is fine, because only when data transfer is needed, is the processor called into action for this.

Now, there is a third method of data transfer and that is ‘direct memory access’. A brief introduction to this was given in Section 6.3.2. The word ‘direct’ means that data can be transferred between a peripheral and memory without the intervention of the processor – a direct connection is established between these two (one a source and the other, a destination). Refer Fig. 6.19. Since the processor is effectively isolated from this data transfer function, it is left free to do other processing activities, but only such activities that do not require the use of the system bus (which is now being used for DMA). Also, since the processor is left without access to the system bus, this sort of data transfer is justified only for transferring large chunks of data. Thus, DMA is used to transfer big blocks of data from/to memory and peripherals. Figure 6.19 is re-drawn here to explain the operation of DMA.

HOLD and HLDA are two pins of the processor. Whenever a device wants a DMA operation to be initiated, it places a request on the HOLD pin of the processor. As the timing diagram indicates, HOLD is sampled at the middle of a clock edge. Then, the processor completes the current bus cycle (not the current instruction cycle) and enters the hold state after sending a HLDA signal (Hold Acknowledge) to the requesting device. HOLD has a higher priority than either the INTR or NMI interrupts. Also, note that interrupts are acknowledged only after the end of the current instruction cycle, but DMA operations start after completing the current bus cycle.

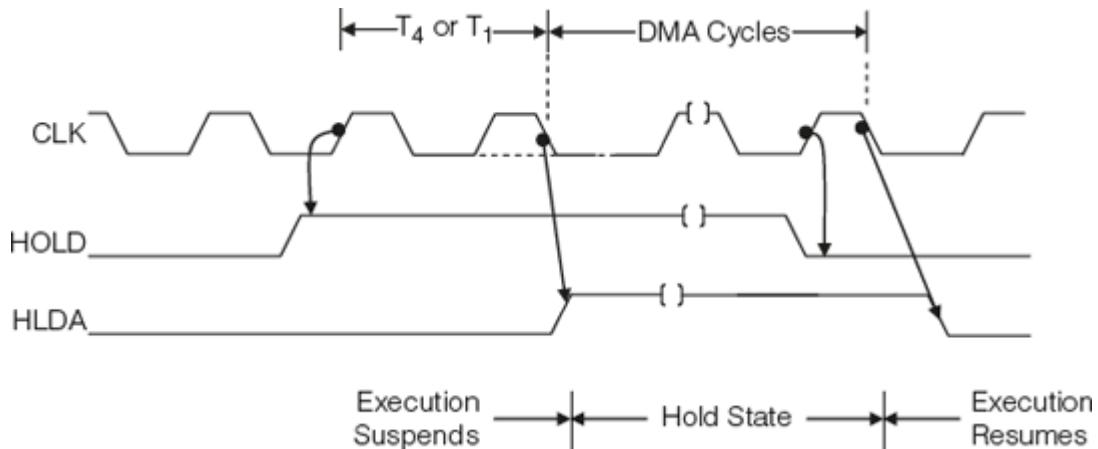


Figure 11.16 | DMA cycle timings

11.5.1 | DMA Controller

Let us review the concept of DMA. The idea is one of transferring data in one of the following cases:

1. from memory to a peripheral
2. from a peripheral to memory
3. from memory to memory (a special case)

It is obvious that there is the need for a controller which gives information to the CPU about the source and destination addresses and also the number of bytes to be transferred. Thus, DMA operations require what is called a DMA controller.

DMA controllers vary as to the type of DMA transfers and the number of DMA channels they support. There are two types of DMA transfers, namely flyby DMA transfers and fetch-and-deposit DMA transfers. There are also three common transfer modes – single, block, and demand transfer modes. These DMA transfer types and modes are described in the following paragraphs.

11.5.2 | Flyby DMA

The fastest DMA transfer type is referred to as a single-cycle, single-address, or flyby transfer. In a flyby DMA transfer, a single bus operation is used to accomplish the transfer, with data being read from the source and written to the destination simultaneously.

In flyby operation, the device requesting service asserts a DMA request on the appropriate channel request line of the DMA controller. The DMA controller responds by gaining control of the system bus from the CPU and then issuing the pre-programmed memory address. Simultaneously,

the DMA controller sends a DMA acknowledge signal to the requesting device. This signal alerts the requesting device to drive the data onto the system data bus or to latch the data from the system bus, depending on the direction of the transfer. In other words, a flyby DMA transfer looks like a memory read or write cycle, with the DMA controller supplying the memory address and the I/O device reading or writing the data.

DMA terminology dictates that a DMA write occurs when data is moved from an I/O device to memory by activating *MEMWR* and *IORD* simultaneously. A DMA read is just the inverse operation when data is read from memory and written to I/O. Note that only the memory address can be placed on the address bus, while the I/O address should be implicit. Because flyby DMA transfers involve a single memory cycle per data transfer, these transfers are very efficient; however, memory-to-memory transfers are not possible in this mode. Figure 11.17 shows the flyby DMA transfer signal protocol.

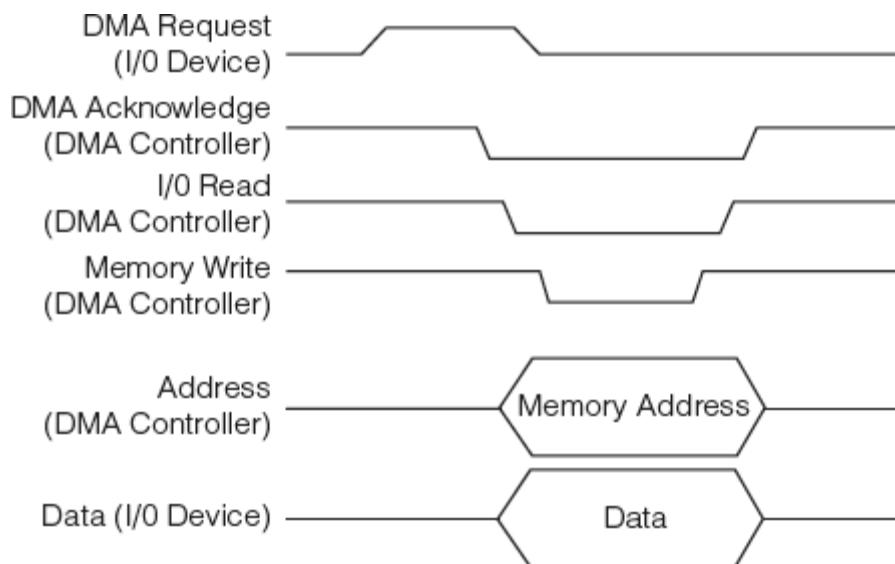


Figure 11.17 | Timing associated with flyby DMA operation (DMA write)

11.5.3 | Fetch and Deposit DMA

The second type of DMA transfer is referred to as a dual-cycle, dual-address, flow-through, or fetch-and-deposit DMA transfer. As these names imply, this type of transfer involves two memory or I/O cycles. The data being transferred is first read from the I/O device or memory into a temporary data register internal to the DMA controller. The data is then written to the memory or I/O device in the next cycle. Although inefficient because the DMA controller performs two cycles and thus retains the system bus longer, this type of transfer is useful for interfacing devices with different data bus sizes. For example, a DMA

controller can perform two 16-bit read operations from one location followed by a 32-bit write operation to another location. A DMA controller supporting this type of transfer has two address registers per channel (source address and destination address) and bus-size registers, in addition to the usual transfer count and control registers. Unlike the flyby operation, this type of DMA transfer is suitable for both memory-to-memory and I/O transfers. Figure 11.18 shows the timing associated with this.



Figure 11.18 | Timing for fetch and deposit DMA

11.5.4 | DMA Transfer Modes

In addition to DMA transfer types, DMA controllers have one or more DMA transfer modes. Single, block, and demand are the most common transfer modes.

Single Transfer Mode

In this mode, one data value is transferred for each DMA request assertion. This mode is the slowest method of transfer because it requires the DMA controller to arbitrate for the system bus with each transfer.

Block and Demand Transfer Modes

These modes increase system throughput by allowing the DMA controller to perform multiple DMA transfers once the DMA controller has gained control of the bus. For block mode transfers, the DMA controller performs the entire DMA sequence as specified by the transfer count register at the fastest possible rate in response to a single DMA request from the I/O device. For demand mode transfers, the DMA controller performs DMA transfers at the fastest possible rate as long as

the I/O device asserts its DMA request. When the I/O device de-asserts the DMA request, transfers are signaled off.

DMA Operation

There should be a DMA controller to co-ordinate all these activities. The controller is a programmable peripheral chip which can be programmed to act as directed, as we will see in the following discussion. It is not a ‘processor’, which has execution capability. However, it can be made to act to control and co-ordinate the DMA requirements of the system. See Fig. 11.19. Initially the system works as a normal computer with all the switches in the upper position. In this state, the processor controls the address, data and control buses. Now, say, a peripheral device wants DMA service. It sends a request to the DMA controller on the pin DREQ. The DMA chip consults its various internal registers and issues a HRQ (Hold Request) to the CPU. It gets a HLDA (hold acknowledge) signal from the processor. This signals the start of DMA operations, with the processor being disconnected from the system. Now, the switches in Fig. 11.19 are toggled to the lower position, giving the DMA chip the control of the buses. The DMA chip relays this information as a DACK (acknowledge) signal to the peripheral.

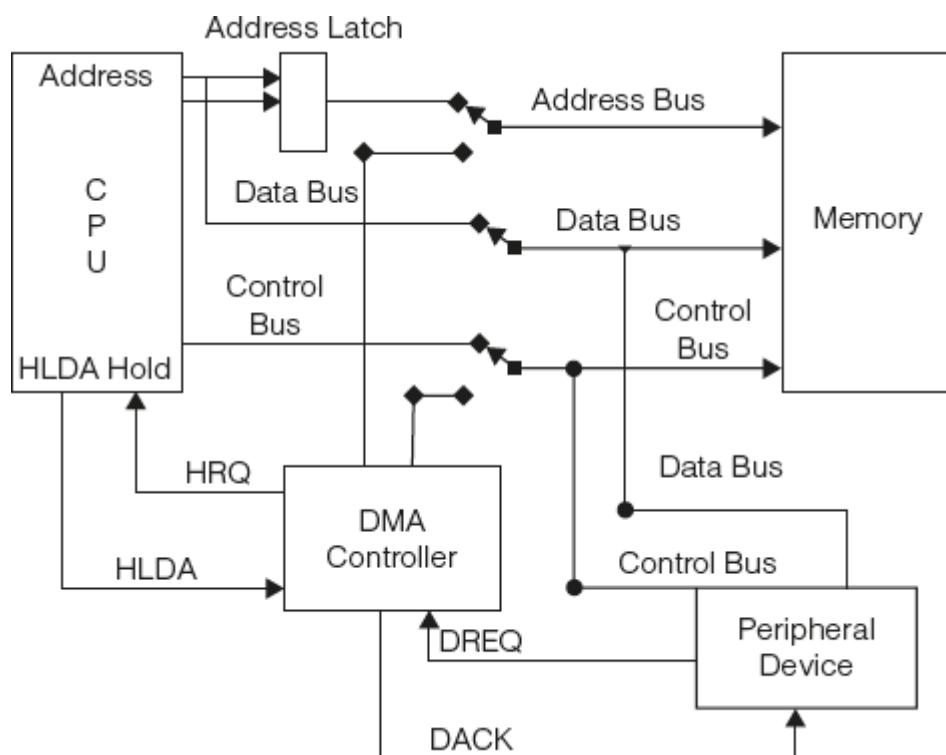


Figure 11.19 | DMA concept

After this initial signaling schedule, actual data transfer occurs. After the complete transfer is over, the DMA controller deactivates its hold

request (HRQ) and control returns to the processor, which means that the switches are back in position 1.

11.5.5 | DMA Controller Features

In a system, there can be a number of I/O devices needing DMA action. Each of them will have to be supported by the DMA controller, which should have a number of ‘channels’ each one catering to one DMA requesting device. For each channel in the DMA controller, certain components are replicated in the controller, while certain others are common to all the channels. Keep in mind that only one DMA operation is possible in the system at a particular time.

Now, see Fig. 11.20. There may be n channels in the DMA controller of which let the currently active channel be designated as channel ‘N’. For DMA transfers, the starting address in memory and the number of bytes to be transferred have to be specified. For each channel, the DMA controller saves the programmed address and count in the base registers and maintains copies of the information in the current address and current count registers, as shown in the figure.

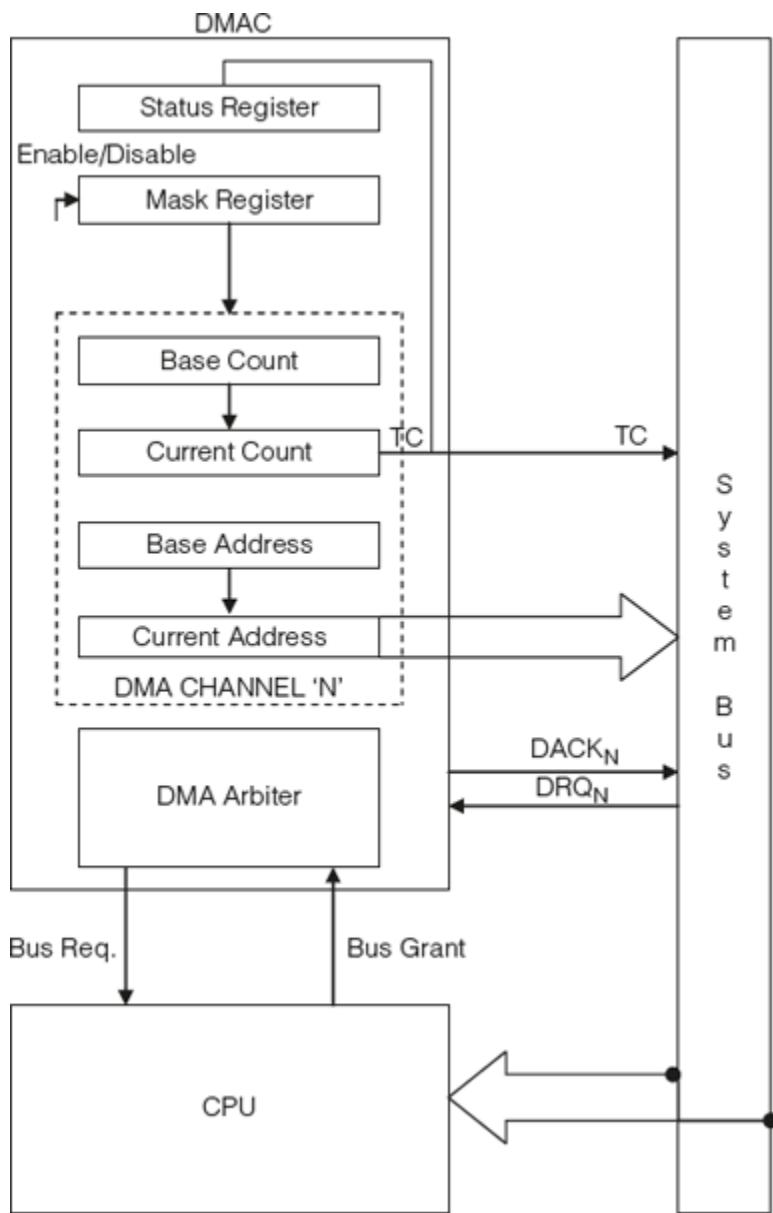


Figure 11.20 | A general block diagram of a DMA controller

Each DMA channel is enabled and disabled via a DMA mask register. When DMA is started by writing to the base registers and enabling the DMA channel, the current registers are loaded from the base registers. With each DMA transfer, the value in the current address register is driven onto the address bus, and the current address register is automatically incremented. The current count register determines the number of transfers remaining and is automatically decremented after each transfer. When the value in the current count register goes to 0, a terminal count (TC) signal is generated, which signifies the completion of the DMA transfer sequence.

Now that we have discussed DMA operations in general, let us see the features of Intel's 8237A multimode DMA controller chip, which has all the features mentioned above.

This is a 40-pin chip with four channels for transferring data, each of which caters to one device. For each channel, there are two signals, DREQ (DMA Request) and DACK (DMA Acknowledge). The former is a request from a peripheral device asking for DMA facility, and the latter is the reply from the controller informing the device that its request has been accepted and acknowledged. From the DMA controller to the processor, there is only one HOLD and HLDA signals, meaning that at least four devices can request DMA services, but only one of them will get service and that is decided by the DMA controller, according to the way priorities are programmed in the chip. However, each channel must be initialized separately for the base address and the count of bytes to be transferred. After this is done, a control word will enable and control the channels. See the functional block diagram of the 8237 chip in Fig. 11.21.

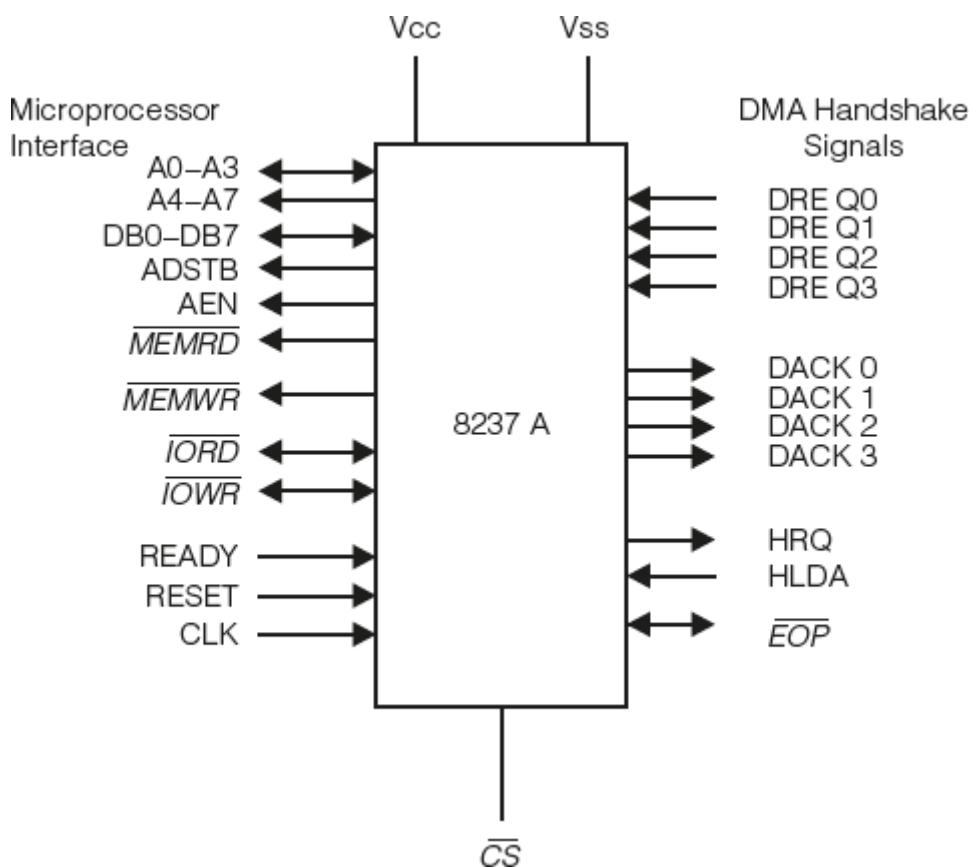


Figure 11.21 | Functional block diagram of the 8237 chip

11.6.1 | Programming the 8237 DMA Controller

Just like any other interfacing chip, this chip also is programmed by writing suitable words into its internal registers. Let us see how this is

done. Keep in mind that there are four channels, which need certain information to be supplied to them. There are two registers associated with each channel:

1. Base address register.
2. Base word count register.

Before DMA operation starts, these 16-bit registers should contain the starting address of memory, and the count of the number of bytes to be transferred. Thus, these registers are to be written into, and the content of these registers do not change during the DMA cycle. However, the content of these registers are also written (automatically) into two other registers called ‘current address register’ and ‘current count register’. The content of the former is incremented after each transfer, while the content of the latter is decremented. Thus, the current status of the DMA operation can be known by reading these registers. The DMA chip has four pins A₃ to A₀ to select these eight registers.

Example 11.5

Find the addresses of the base address and base count registers for the four channels when the decoding logic of Fig. 11.22 is used.

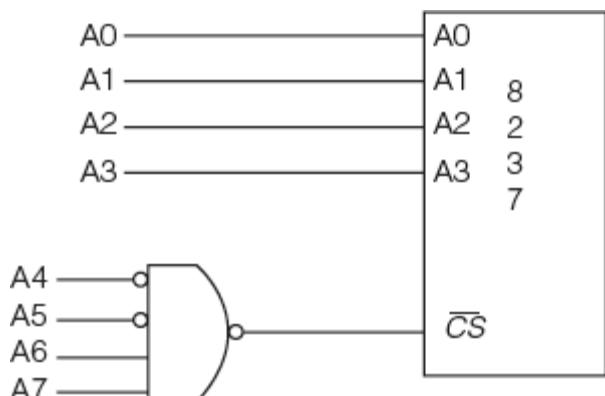


Figure 11.22 | Address decoding

Solution

The decoding logic needs A₇ to A₄ to be 1100, i.e., 0CH, for selecting the chip. The other bits for selecting each register is given in column 4 of Table 11.3. Hence, the address of the base address register and base count registers for each of the channels is given in column 5 of the table.

Table 11.3 | Addresses of the Registers of the Four Channels

Channel	Register name	Operation	A3	A2	A1	A0	Address (Hex)
0	Base Address	Write	0	0	0	0	C0
	Current Address	Read	0	0	0	0	C0
	Base Count	Write	0	0	0	1	C1
	Current Count	Read	0	0	0	1	C1
1	Base Address	Write	0	0	1	0	C2
	Current Address	Read	0	0	1	0	C2
	Base Count	Write	0	0	1	1	C3
	Current Count	Read	0	0	1	1	C3
2	Base Address	Write	0	1	0	0	C4
	Current Address	Read	0	1	0	0	C4
	Base Count	Write	0	1	0	1	C5
	Current Count	Read	0	1	0	1	C5
3	Base Address	Write	0	1	1	0	C6
	Current Address	Read	0	1	1	0	C6
	Base Count	Write	0	1	1	1	C7
	Current Count	Read	0	1	1	1	C7

Note We ‘write’ into the base address register and base count register. Later, at any time, the current values of address and count are ‘read’ from the current address register and current count register. That should explain Column 3 of Table 11.3.

11.6.2 | Functional Block Diagram

Now, observe the functional block diagram in Fig. 11.21. Most of the pins seem familiar, but the functions of a few (AEN, ADSTB, EOP) need more explanation. We will come to that soon. However, some features and some anomalies need to be noted.

1. The base count register is only 16 bits long, implying that only 64 KB of data can be transferred by one DMA operation.
2. The base address register is only 16 bits in size – but for a processor like 8086, the address is 20 bits long, so how can the upper 4 bits be supplied?
3. The DMA chip has only an 8-bit address bus, so how does it place a 20-bit address on the address bus to address memory?

The first point is a feature of the chip, but the second and third are obviously inadequacies of the chip which have to be addressed. Let us get to that.

Example 11.6

Write into the base register and count register of channel 1, given that 4KB of data are to be transferred from a memory location 13230H to a peripheral.

Solution

The address of the base address register and the base count register for channel 1 are C2H and C3H, respectively. The number of bytes to be transferred is 4K, i.e., 4096 bytes.

MOV AL, 30H	;move the LSB of the base address to AL
OUT 0C2H, AL	;send it to the base address register
MOV AL, 32H	;move the MSB of the base address to AL
OUT 0C2H, AL	;send it to the base address register
MOV AX, 4096	;move the byte count to AX
OUT 0C3H, AL	;send LSB of count to the count register
MOV AL, AH	;move MSB of count to AL
OUT 0C3H, AL	;send MSB of count to the count register

Note that there is no provision to send the upper hex digit of the 20-bit address 13230H to the base address register, since it can store only 16 bits. Let us see how this problem is taken care of.

11.6.3 | Memory Addresses above 64 KB

In the program of Example 11.6, only the lower 16 bits of the 20-bit address can be sent to the base address register, which means that during a DMA cycle, when the DMA controller controls the system bus, only the 16-bit part of the starting address of the memory block can be placed on the address bus. So, who sends the upper 4 bits to the address bus? The answer is that an extra four bit latch should be included in the system hardware. The latch's input should be hardwired to carry the upper 4 bits of address, or it should be sent to the latch through program instructions. Thus, during a DMA cycle, the upper 4 bits of the address are obtained at the output of this latch.

Now, look at Fig. 11.21 once again. Note that the DMA chip has only 8 address lines and they are numbered as A₇ to A₀. That means the chip has no lines to put the upper eight bits (A₁₅ to A₈) of the address during a DMA cycle. This necessitates the use of another latch which is 8-bit in size.

Now, let us review the address bus operation totally.

1. Before DMA operation starts, the processor controls the address bus. It uses the address bus to write words into the DMA controller's control and mode registers for initialization and mode setting of the chip. Then, the AEN (Address Enable) signal of the DMA chip goes high, and after that, the processor is isolated from the system bus and the DMA controller is in charge of the bus.
2. When the DMA operation starts, the 20-bit address of the memory block starting address should be put on the system address bus. Remember that 16 bits of this address are available in the DMA chip's internal register (base address register). The chip puts lower 8 bits of this address on its pins A7 to A0. It also puts on its data lines (DB7 to DB0), the upper 8 bits of the address available with it. These data lines are fed to the input of an 8-bit latch (see Fig. 11.23) which is clocked by the ADSTB (Address Strobe) signal. Thus, at the output of this 8-bit latch, the upper 8 bits of the 16-bit address is available and it can then be removed from the data lines of the 8237 chip.

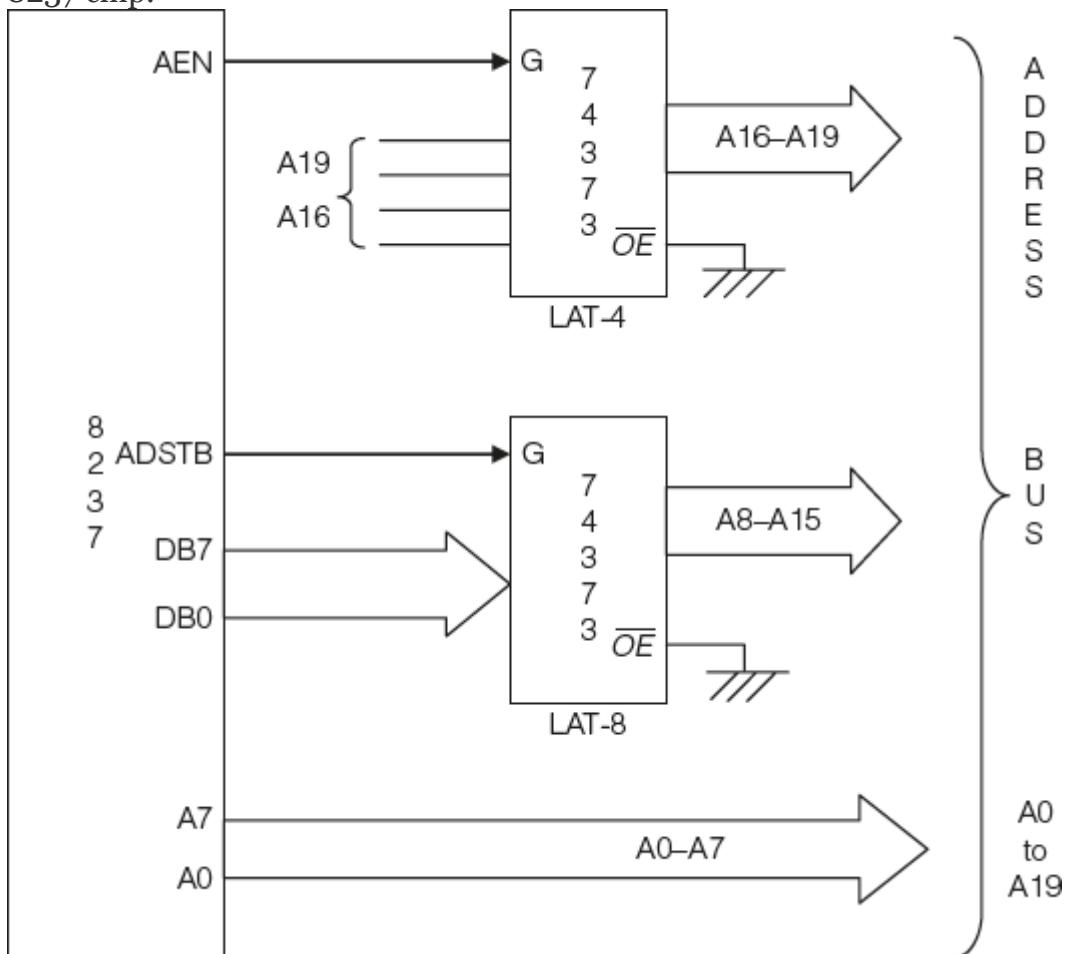


Figure 11.23 | Address generation for DMA operation

3. During DMA operation, the address on the address bus goes on incrementing. The lowest 8 bits increments from 00 to FFH and then has to overflow to the upper 8 bits, which have to be incremented by one now. For

example, for a 16-bit starting address of 1200H, the lower 8 bits roll from 00 to FFH, and then the address must change to 1300H. At this point, the number 13H must be sent from the DMA chip through its data pins (DB7 to DB) and this must be latched on to the latch IC. At this instant, ADSTB goes high again acting as a clock to this latch. This is an additional operation during a DMA cycle, but since it occurs only once in 255 DMA cycles, it is not a big overhead.

4. What about the uppermost 4 bits of the address A_{19} to A_{16} ? See the 4-bit latch in Fig. 11.23. When AEN goes high, these addresses are latched to the latch, and the output of this latch constitutes the uppermost 4 bits of the address bus. How does the 4-bit address reach the input of the 4-bit latch? For a particular application, if this part of the address is fixed, it can be hardwired. Or it can be sent from the processor from an output port and latched by the use of AEN during DMA operation.

The EOP signal

When the programmed number of bytes have been transferred, i.e., when the terminal count has been reached, the DMA controller lowers the EOP (End of Process) signal, de-asserts its HRQ pin, lowers the AEN signal and releases the system bus, so that the process can resume its normal operation.

11.6.4 | Initialization and Programming the 8237

Continuing with the idea of programming the 8237 chip, let us see the registers it has for command, status and modes. Since we have designed the logic of CS as in Fig. 11.22, the addresses of these registers are as in Table 11.4.

Table 11.4 | Addresses of Internal Registers

Register name	A3	A2	A1	A0	Address (Hex)
Status/Command Register	1	0	0	0	C8
Request Register	1	0	0	1	C9
Single Mask Bit Register	1	0	1	0	CA
Mode Register	1	0	1	1	CB
Clear Byte Pointer	1	1	0	0	CC
Master Clear/Temperory Register	1	1	0	1	CD
Clear Mask Register	1	1	1	0	CE
All Mask Register	1	1	1	1	CF

Command/Status Register

The command register is the register used for controlling the operation of the DMA chip. For that, the register must be written to. The same register is used as a status register when its contents are read from. First, let us see the format of the command register.

Example 11.7

Write the command word for using the 8237 DMA chip with the following requirements:

1. memory to peripheral transfer,
2. compressed timing,
3. rotating priority,
4. late write,
5. Both DREQ and DACK active high.

Solution

Refer to the command word format in Fig. 11.24.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	1	1	0	0	0

The command word is 98H.

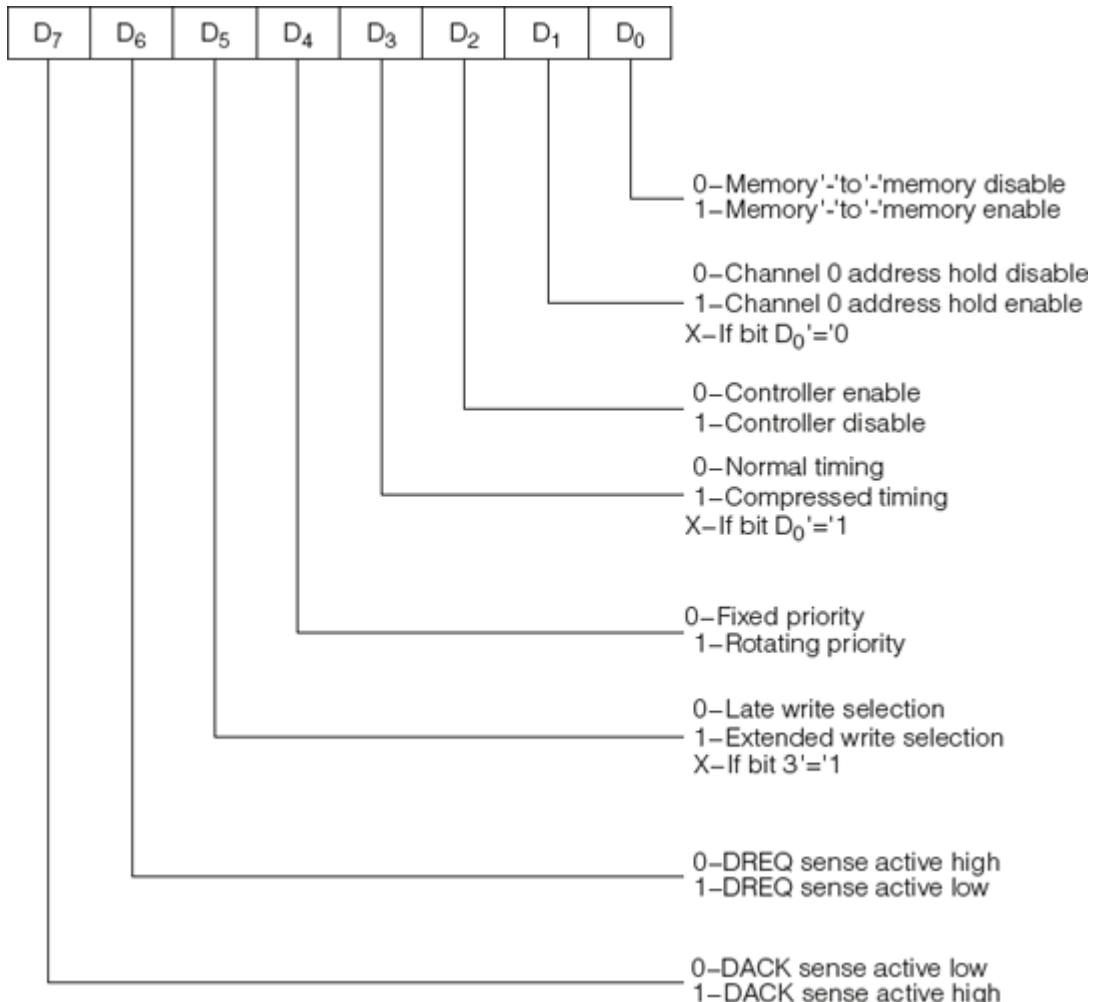


Figure 11.24 | Status of the command register

The Status Register

This register is available to be read out and it contains information about the status of the channels of the chip. See Fig. 11.25. This information includes which channels have reached a terminal count and which channels have pending DMA requests. Bits 0 to 3 are set every time a terminal count is reached by that channel or an external EOP is applied. These bits are cleared upon reset and on each status read. Bits 4–7 are set whenever their corresponding channel is requesting service.

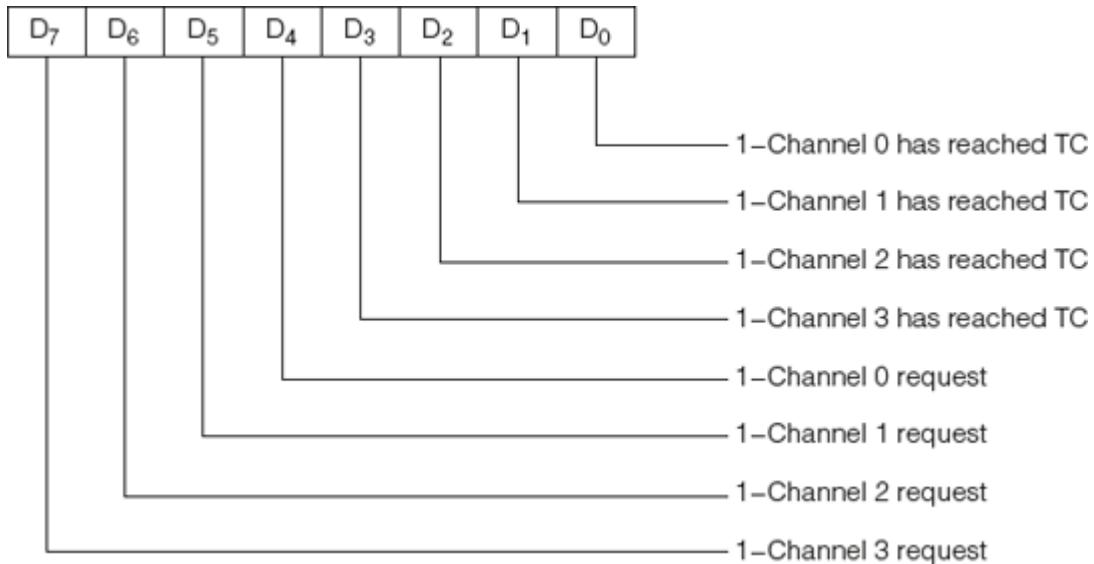


Figure 11.25 | Status register format

Mode Register

Each channel has an 8-bit mode register associated with it, the format of which is shown in Fig. 11.26. When the register is being written to, in the program condition, bits 0 and 1 select which channel's mode register is to be written to. The rest of the bits select various mode options. There are a few more registers for the DMA chip. For more details of programming the chip, the data sheets of the chip may be referred to.

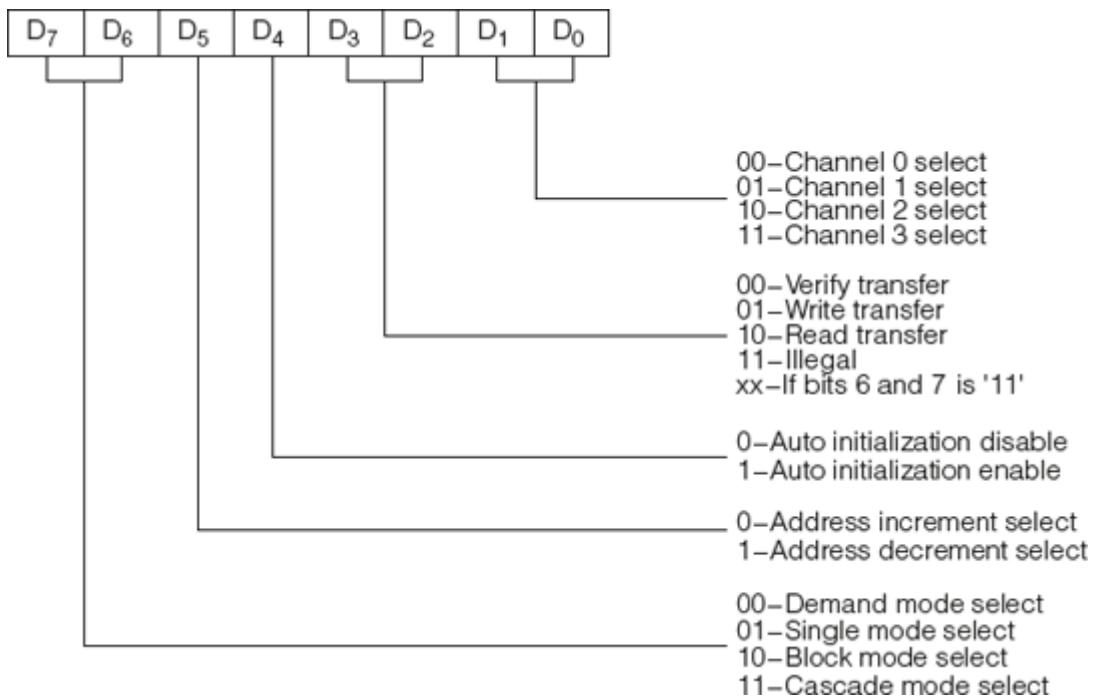


Figure 11.26 | Mode register format

In the original IBM-PC, only one 8237 chip was used, and its four channels were allocated to the following devices:

1. Channel 0 for refreshing DRAM.
2. Channel 1 unused but allowed for any I/O application.
3. Channel 2 for floppy disk controller.
4. Channel 3 for hard disk controller.

Remember that the 8237 has only a 16-bit address bus capability; hence, for the addresses beyond 64KB, the extra bits are provided by a page register, which is a latch for holding the extra address bits.

11.7.1 | DMA and PC-AT

With the advent of the ISA bus for PC-AT type computers, two DMA controllers came to be used. Two Intel 8237-DMA controllers were cascaded together, and each chip had four channels. Channel allocation was as follows:

- 0 DRAM Refresh (obsolete),
- 1 User hardware,
- 2 Floppy disk controller,
- 3 Hard disk (obsolete),
- 4 Cascade from XT DMA controller,
- 5 Hard Disk (PS/2 only), user hardware for all others,
- 6 User hardware,
- 7 User hardware.

11.8 | PCI Based Computers

The latest development in PCs is the PCI bus. This bus has a different philosophy as compared to the ISA bus. As far as DMA is concerned, the scenario is as follows.

A PCI architecture has no central DMA controller, unlike ISA. Instead, any PCI component can request control of the bus (become the bus master) and request to read and write from the system memory. More precisely, a PCI component requests bus ownership from the PCI bus controller (usually the *southbridge* in a modern PC design), which will arbitrate if several devices request bus ownership simultaneously, since there can only be one bus master at one time. When the component is granted ownership, it will issue normal read and write commands on the PCI bus, which will be claimed by the bus controller and forwarded to the memory controller using a scheme which is specific to each chipset.

KEY POINTS OF THIS CHAPTER

- Serial communication is the preferred method of data transmission over long distances.
- In synchronous transmission format, data can be 8 bits or less, with/without parity bits, one start bit and one or more stop bits.
- Transmission rate in serial communications can be expressed in bauds or bits per second, but both are not the same in all cases.
- Modems are used to connect computers to an analog transmission medium.
- RS-232 is a serial communication standard in which voltage levels are higher than TTL levels.
- Converters are needed for TTL to RS232 level conversion and vice versa.
- RS-232 connectors have 25 pins, but 9 pins are sufficient and so most computers are equipped only with DB-9 connectors.
- Two terms commonly used in serial communications are DTE and DCE.
- There are a number of handshaking signals in the RS-232 protocol.
- There are many chips used as serial communication interfaces, but the chip discussed here is the USART 8251.
- This chip caters to both synchronous and asynchronous communications.
- The method of programming this chip is to write into mode and command words.
- Direct memory access is a high priority activity in any computer system.
- Flyby DMA is much more efficient than fetch and deposit DMA.
- A DMA controller with a number of channels is necessary to manage DMA requests from various peripherals.
- The DMA controller chip 8237 has four channels and a number of registers to handle DMA operations.
- The chip has only the capability to handle 16-bit addresses, so extra hardware is required for processors with greater address widths.
- DMA chips are used in PCs, but the new PCI bus uses a different concept in bus mastering.

QUESTIONS

1. Distinguish between simplex and duplex communications.
2. Compare the merits and demerits of asynchronous communications with synchronous serial communications.
3. What is RS-232C ?
4. Explain how baud rate and bps may be different.
5. Where is the chip MAX 232 used and for what purpose is it used?
6. What is the use of a null modem connection?
7. How many lines are needed to have serial communications between two computers?
8. Explain the functions of the pins T × RDY and R × RDY of the 8251.

9. What happens to the start and stop bits at the receiving point?
10. How is baud rate decided when using the 8251?
11. Why is DMA said to be a high priority activity?
12. What are the capabilities expected of a DMA controller?
13. What is meant by the word 'Terminal Count'?
14. When is memory-to-memory DMA required? Mention a specific case.
15. Why is it said that flyby DMA is very efficient?

EXERCISES

1. Write a program to send a string of characters between a transmission and reception point using the 8251 in
 1. loop back mode,
 2. two different kits.
 2. Write mode words for transmission with different baud rates, different parity settings and different character lengths.
 3. Write instructions to read the following conditions:
- o. T × RDY**
1. T × EMPTY
 2. DSR
 3. PE
4. For the DMA controller, design a decoding scheme and get a different set of addresses for the count and address registers.
 5. For the 8237, design a command word for the following conditions:
- o. peripheral to memory transfer,**
1. normal timing,
 2. DREQ active low,
 3. DACK active high,
 4. fixed priorities.

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

14

Advanced Processors—80286 to Pentium

IN THIS CHAPTER, YOU WILL LEARN

- The impact that 80286 made to the realm of PCs.
- The major changes that the 80386 architecture offered.
- The programming enhancements of 80386.
- The features of PVAM and the concept of virtual memory.
- The mechanism of address translation in the protected mode.
- The importance and implementation of protection mechanisms in the 80386.
- The issues and methods of task switching.
- The use of the interrupt descriptor table and exceptions of 80386.
- The features of 80486 which are an enhanced set of the '386 features.
- The idea of burst mode data transfers for both 80486 and Pentium.
- Why data alignment is important for multibyte processors.
- The features of Pentium.
- The cache structure of Pentium.

Introduction

In this chapter, we will discuss the 80286, 80386 and 80486 processors, as well as pentium. The first 32-bit processor of Intel, i.e., the 80386 is discussed with the aim of cracking the intricacies of a complex processor. Once the '386 is understood well, the '486 and Pentium have nothing more to it than a few 'enhancements'. This means that the internal architecture of the '486 and Pentium is more or less the same as that of the '386. They have the same memory management unit, protection mechanisms and multitasking units. Thus, a discussion on these features is redundant and we will concentrate on the additional features of the 80486 and Pentium, in addition to that of 80386.

14.1 | The 80286 Processor

You know that the first Intel processor used in a personal computer by IBM was the 8088. The PC based on 8088 and manufactured by IBM

came to be known as IBM PC. It had only a floppy disk drive internally but could have an external hard disk. All PCs with similar hardware and standards came to be called IBM PC clones. Intel later released the PC-XT (extended technology), which included a standard hard disk internally.

The next important processor of Intel was 80186. However, we have seen in the previous section that the 80186 has never been used as a processor in a PC but instead found applications in the field of embedded systems.

Soon after that, in 1982, Intel released the 80286, which came to be used in PCs that came to be called PC-AT where AT stands for ‘advanced technology’. This turned out to be a big leap for Intel and for PCs as well. The PC-AT used 80286 as the processor and its technology could run all the software written for its predecessor. It could be operated at frequencies of 6 MHz, 8 MHz, 10 MHz and 12.5 MHz. The 80286 turned out to be a big improvement over the 8086/8088. It could boast of a performance double that of the 8086. It had a number of features that were entirely new to the realm of processors.

The major enhancements can be listed as:

1. The data bus remained the same at 16 bits (internally and externally), but there was no need for the multiplexed address/data bus as separate pins were made available for them. The pin count increased so that the packaging could no longer be a Dual in line package like the 8086. Instead, the packaging was a pin grid array (PGA).
2. The address bus was increased to 24 bits, which could address 2^{24} bytes or 16 Mbytes of physical memory.
3. The memory cycle time was reduced from 4 cycles to 2 cycles, which made a direct improvement to the speed factor. Calculation of the more complex addressing modes (such as base + index) had less clock penalty because it was performed by a special circuit. The 8086, its predecessor, had to perform effective address calculation in the general ALU, taking many cycles. Also, complex mathematical operations (such as MUL / DIV) took fewer clock cycles compared to the 8086.
4. The biggest change and enhancement, however, was in the form of the introduction of what was called the ‘Protected Virtual Addressing Mode (PVAM)’. Let us try to understand what exactly this is. For the 8086, there is no ‘protection’ provided between users or between system software and application software. MS-DOS allowed access to anyone to any programs or any hardware through the mechanism of interrupts. Also, there was no facility to use the processor in a multi-tasking environment. 80286 reversed

all this – it provided new mechanisms and instructions which provided protection and multi-tasking. Also, it could use the concept of ‘virtual memory’ whereby the memory as seen by the user becomes much larger than the actual physical memory present. This is the essence of PVAM.

When the processor is powered on, it is in the ‘real mode’ where it behaves as an accelerated 8086 – it then has only 20 active address lines and 1 MB of physical memory. From this mode, a special instruction switches it into the protected mode where the active address lines become 24 in number, the physical memory becomes 16 MB and all the protection and virtual memory mechanisms become activated. With all these enhancements and changes, this processor was popular for a short while, but one complaint against it was that it could not be switched back to real mode without resetting it.

But it was this processor that made the real change in the way a computer could be designed and used. The concepts in the protected mode allowed the first GUI (graphic user interface) based operating system of Microsoft, i.e., windows 3.1 to be designed comfortably for a PC. The design of the 80286 was followed up in Intel leading to a better and bigger (in terms of data and address size) processor named as the 80386 (1985). All the features of the 80286 were used here, and more features were added as well – we can say that the features of 80286 are a subset of the features of that of the 80386. Thus, it will be to our advantage to go into the details of the architecture of the 80386, which will give us all the wisdom necessary to understand the ’486 and Pentium which followed very soon.

14.2 | The 80386

In 1985, Intel released the 80386 processor which carried in it all the new features it had introduced in its previous processor and also made one great leap – that was the doubling of the data bus width to 32 bits and the increase in the address bus width to 32 bits. Thus, the data band width doubled, and the addressable physical memory became 2^{32} , i.e., 4 GB. Thus, 80386 was Intel’s first 32-bit processor. Intel also manufactured a model called 80386 SX, which was a 32-bit processor internally but had an external data bus width of 16 bits. However, our discussion will be on the 80386 DX, which is a 32-bit processor internally and externally.

When powered on, the ’386 is in the real mode with 20 address lines and capability to access only 1 MB of physical memory. From this state, it

can be taken to the protected mode using a special instruction, the details of which we will soon come to.

14.2.1 | 80386 Enhancements

The 80386 has all the features of 80286 – additional instructions, protected virtual addressing mode – plus a lot more. It has more instructions, higher memory addressing capability, double the data bus width and address bus and also includes the use of memory as ‘pages’ in addition to segmentation. Let us list out the new features.

- The external data bus is now 32 bits in width.
- The registers and, hence, the internal data bus width is 32 bits in width.
- The address bus is 32 bits in size, and hence, the memory addressing capability is 2^{32} or 4 GB.
- All addresses are 32 bits long, and hence, the address registers are also 32 bits long.
- A new addressing mode called the scaled addressing mode is available.
- A set of bit manipulation instructions is available.
- The ’386 design overcame the biggest complaint against its predecessor – this processor can switch from protected mode to real mode without a hardware reset.
- The processor has all the features of PVAM that its predecessor had – as well as an additional ‘paging’ mode.

Let us go into the details of this processor – first the internal architecture and then the hardware features.

14.3 | Internal Architecture

14.3.1 | Registers

Let us first view the working registers of the processor – the ones used for computation, address calculation and the like (see Fig. 14.1). The processor has 16 registers, which consist of scratchpad (general-purpose) registers, segment registers, flag register and instruction pointer.

Comparing it with 8086 shows that the 16-bit registers AX to DX have been re-designated as EAX to EDX, making them 32-bit registers. These registers can be used as 8 bits or 16 bits as well, but only the lower portions of the registers can be used, i.e., the register EAX can be used as EAX (32-bit), AX (16-bit), AH (8-bit) and AL (8-bit). Similar is the case for the registers EBX, ECX and EDX.

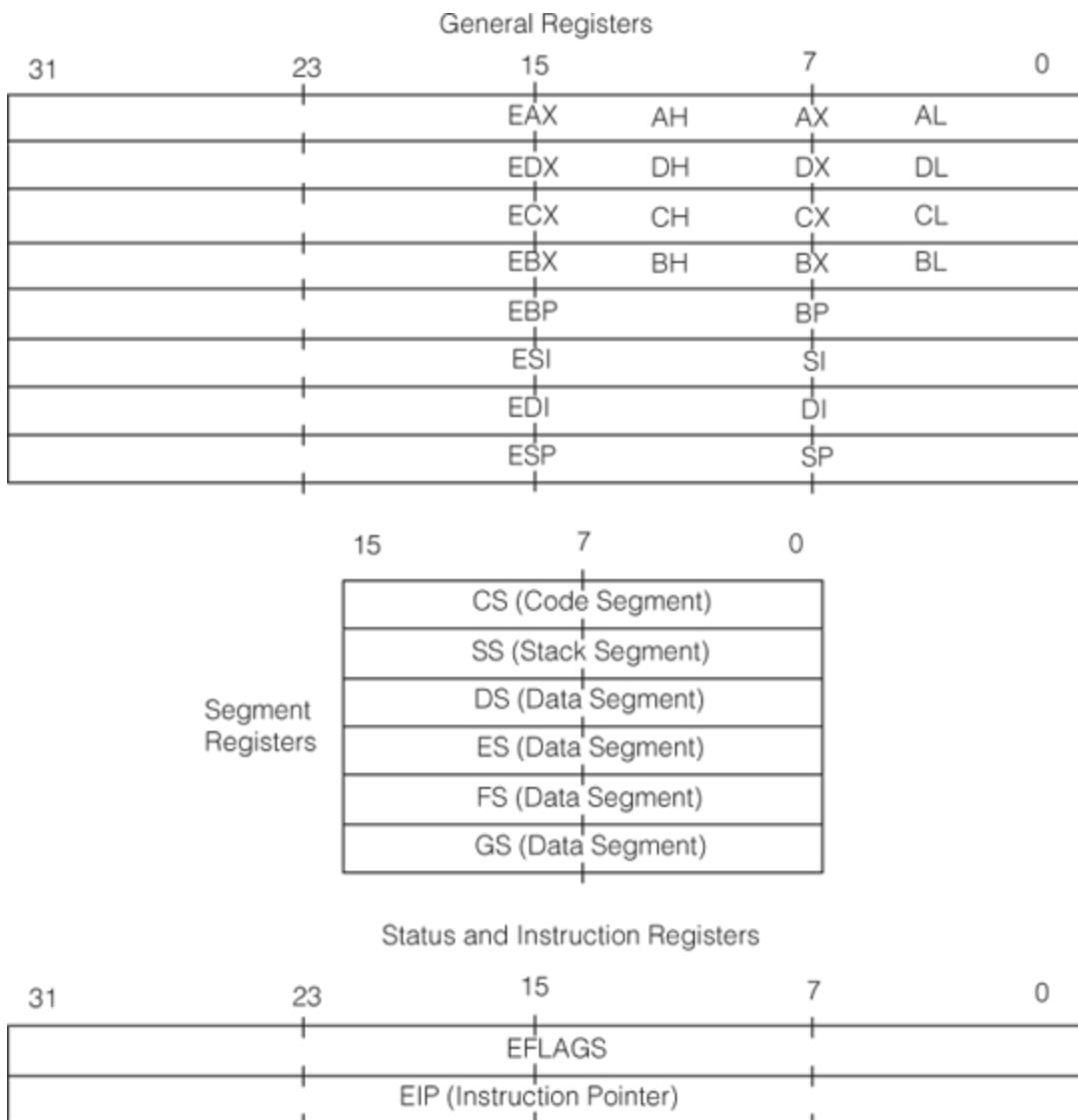


Figure 14.1 | General registers and address registers used in the real mode

The registers EBP, ESI, EDI and ESP are 32-bit address registers as all addresses are 32 bits in length. The instruction pointer EIP is 32 bits long as each memory address is 32 bits in size. The same is the case of the stack pointer, which is now designated as ESP.

14.3.1.1 | Segment Registers

The segment registers are 16-bit, but we will see later that they have what is called a ‘hidden part’, which will then make them totally 48 bits long (this becomes effective only in the protected mode). The visible part of the segment registers are still 16 bits and you can see that there are two new segment registers FS and GS. They are used to store the base addresses of the GS and FS segments, which are data segments itself.

14.3.1.2 | Control and Status Registers

The flag register, i.e., now EFLAG, contains the values of the status flags (i.e., C and Z), the control flags (i.e., DF, IF and TF) plus new flag bits catering to the additional features of the processor. The flag register has all the flag bits that the 8086 has – the additional bits are for use in the protected mode. Figure 14.2 shows the flag register bits.

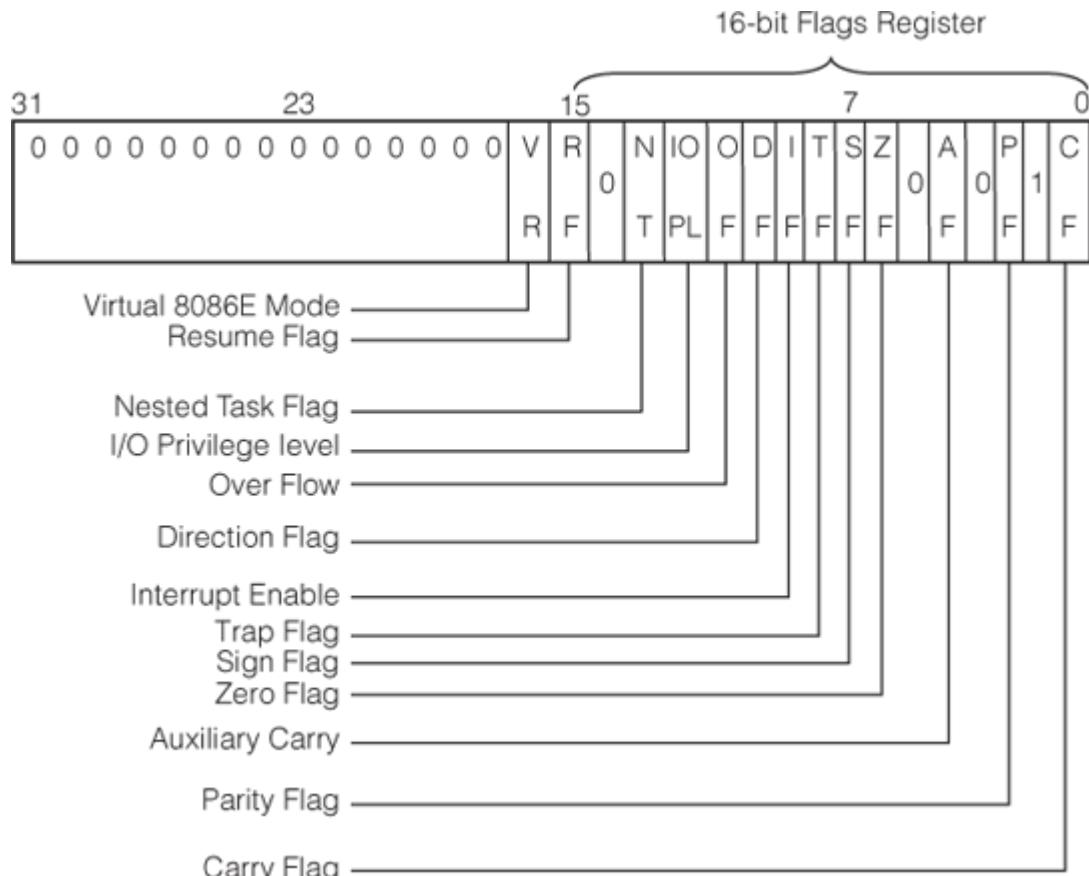


Figure 14.2 | Flag register of the 80386

14.4 | Programming Enhancements

The processor has a number of new instructions and some enhancements with respect to the instructions available for its predecessor. It has all the instructions that the 8086, '186 and '286 has, plus a few more. Here we will see the new instructions that can be used in the **real mode**.

1. The '386 has an interesting enhancement with regard to what we have until now defined as 'pointer registers' which carry addresses. **Here all the general-purpose registers too can hold addresses that can be used in the register indirect, register relative and based indexed mode of addressing.** For example, the following instructions are allowed.

```
MOV EAX, [EBX],  
MOV CX, WORD PTR [EAX]  
MOV AL, BYTE PTR [ECX]
```

2. A new and powerful mode of addressing called scaled index addressing mode has been introduced for the 80386. In this mode, each of the 32-bit registers (except the ESP) can be used as a pointer that is multiplied by a scale factor.

Now, let us do a few programming exercises using the new instructions of 80386.

Note 1 This processor has 32-bit registers, and hence, all the programming for this will be done in MASM32, which is the 32-bit Windows based assembler. The installation and use of this assembler is described in Appendix H available online at www.pearsoned.co.in/lylabdas.

Example 14.1

Find the effective address of the operand for the following instructions, given ECX = 233411E2H, EDI = 10008967H and EAX = 78960900H

1. MOV [ECX*2], AX
2. MOV EBX, [2341H + EDI*4]
3. MOV BL, [EAX*8]

Solution

Remember that the ‘effective address’ calculation does not need the base address of the segment. It gives the offset of the operand from the base address.

1. The effective address is obtained by multiplying the value in ECX by 2, which gives 466823C4H. The word from AX is moved to this address (offset).
2. The effective address is obtained by multiplying the value in EDI by 4 and adding 2341H, i.e., 4002259CH + 2341H = 400248DDH. The double word in this address is moved to EBX.
3. The effective address is obtained by multiplying the value in EAX by 8, which is 3C4B04800H. The byte in this address is moved to BL.

Example 14.2

Add a set of 32-bit words stored in memory assuming that the result will not exceed 64 bits.

Note 2 This program has been run in MASM 32 and results have been observed using the Olly debugger, the details of which are available in Appendix H.

```
.386
.MODEL FLAT, STDCALL
.CODE
DATS DD 00123456H, 00EF3456H, 07860EACFH,
      0A45674FH, 08976ABCH

STRT: MOV EAX, 0          ;EAX = 0
       MOV EBX, 0          ;EBX = 0
       MOV ECX, 00000005    ;ECX = 5, the count
AGN:  ADD EAX, [DATS + EBX*4] ;add to EAX the contents of memory
       INC EBX             ;increment the pointer
       LOOP AGN            ;repeat until ECX = 0
END STRT
```

Here, after adding the five 32-bit data stored in locations starting from DATS, the sum is found to be 8C3F 2586H, which is in EAX. Note that for the instruction ADD EAX, [DATS + EBX*4], the scale factor used is 4. This is because the data is 4 bytes long, and therefore, the address of each data is displaced from the next data by 4.

If the problem was one of adding words (2 bytes), we would have used a scale factor of 2. Thus, the scaled addressing mode is useful in adding tables stored in memory, when the content of the tables are either 2 or 4 bytes long.

14.4.1 | New Instructions of 80386

Bit Test and Modify Instructions

This group of instructions operates on a single bit that can be in memory or in a general register. The location of the bit is specified as an offset from the low-order end of the operand. The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register. These instructions first assign the value of the selected bit to CF, the carry flag. Then, a new value is assigned to the selected bit as determined by the operation. OF, SF, ZF, AF, PF are left in an undefined state. Table 14.1 defines these instructions.

Example 14.3

```

.386
.MODEL FLAT, STDCALL
.CODE
DATS DW 0560H

STRT: LEA ESI, DATS      ;let ESI point to the data
      MOV AX, [ESI]       ;AX = 0560H
      BTC AX, 1           ;AX = 0562H, CF = 0
      BTR AX, 1           ;AX = 0560H, CF = 1
      BTS AX, 0EH         ;AX = 4560H, CF = 0
      END STRT

```

Table 14.1 | Bit Test and Modify Instructions

Mnemonic	Instruction	Carry flag after execution	Effect on the selected bit after execution
BIT	Bit test	CF = bit	None
BTS	Bit test and set	CF = bit	BIT = 1
BTR	Bit test and reset	CF = bit	BIT = 0
BTC	Bit test and complement	CF = bit	BIT = not BIT

The effect of each of the operations is shown in the comment field.

14.4.2 | Bit Scan Instructions

These instructions scan a word or doubleword for a ‘1’-bit and store the index of the first set bit into a register. The bit string being scanned may be either in a register or in memory. The ZF flag is set if the entire word is zero (no set bits are found); ZF is cleared if a ‘1’-bit is found. If no set bit is found, the value of the destination register is undefined.

BSF (Bit Scan Forward)

This instruction scans from low-order to high-order (starting from bit index zero)

BSR (Bit Scan Reverse)

This instruction scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

Example 14.4

```

.386
.MODEL FLAT, STDCALL
.CODE

DATS      DW 0560H

STRT:    LEA ESI, DATS      ;let ESI point to the data
        MOV AX, [ESI]      ;AX = 0560H
        BSR BX, AX        ;BX = 000AH meaning that D10 = 1
        BSF CX, AX        ;CX = 0005 meaning that D5 = 1
END STRT

```

14.4.3 | MOVSX and MOVZX Instructions

Two other new and useful instructions are the MOVSX and MOVZX instructions. In MOVSX, the sign bit of a register (or memory) can be extended to any register. MOVZX zero extends the contents of a register or memory location. The MOVSX instruction is used for signed arithmetic and MOVZX for unsigned arithmetic.

Example 14.5

```

.386
.MODEL FLAT, STDCALL
.CODE
        DAT1 DW -269
        DAT2 DB 56
        DAT3 DB - 123

STRT:    MOV AX, DAT1      ;AX = FEF3H
        MOVSX ECX, AX      ;ECX = FFFFFFFE3H
        MOVSX EBX, DAT2      ;EBX = 000000038H
        MOVZX ECX, DAT1      ;ECX = 0000FEF3H
        MOVSX AX, DAT3      ;AX = FF85H
END STRT

```

In Example 14.5, the numbers are expressed in the decimal format. In the hex format, they are

DAT1 = FEF3H
 DAT2 = 38H
 DAT3 = 85H

Note The first number is stored as a word, and the next two are stored as bytes. The result of each of the operations is shown in the comment field of the program.

14.5 | Hardware Features of 80386

The 80386 has 132 pins and is packaged as a pin grade array (PGA). Figure 14.3 shows the processor with the functional grouping of pins. The pin numbers are not mentioned – only the pin functionality is discussed. Only new pins which are not present in 8086 will be discussed. The pins that are not discussed may be assumed to have the same functionality at they have in the 8086.

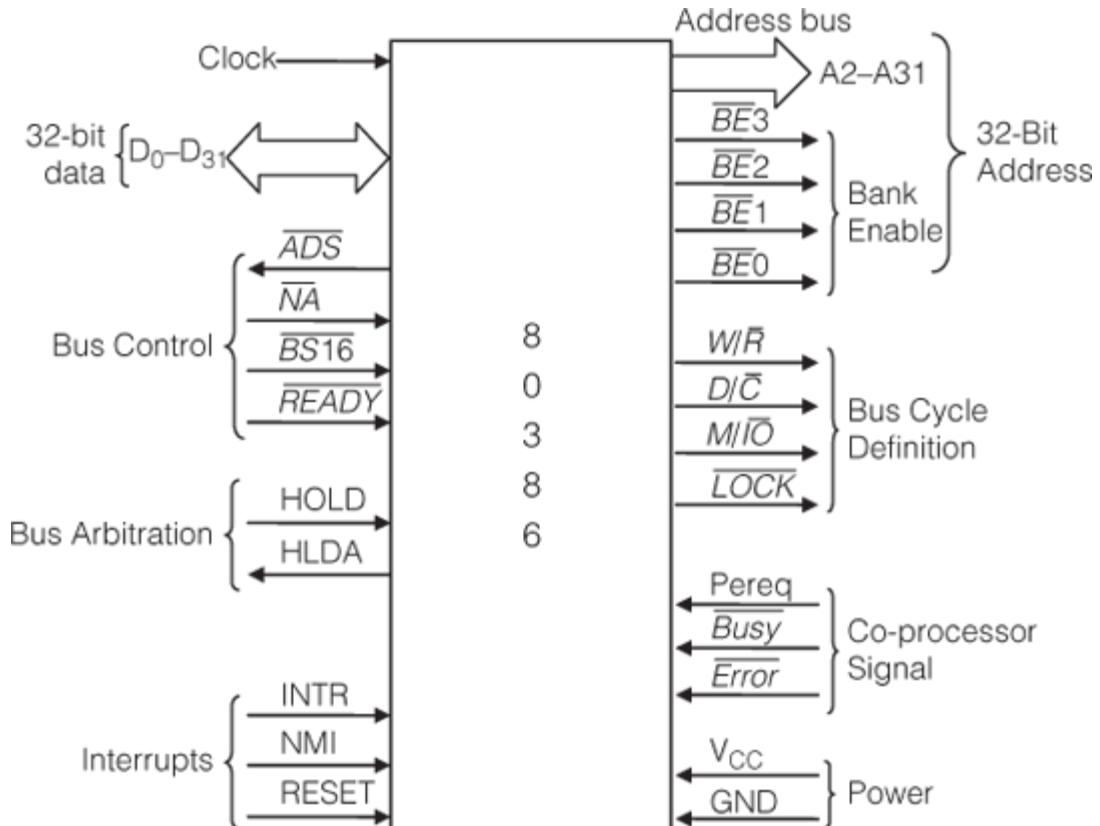


Figure 14.3 | Functional pin diagram of the 80386

Pin Grade Array

A pin grid array (PGA) refers to the arrangement of pins on the integrated circuit packaging. In a PGA, the pins are arranged in a square array that may or may not cover the bottom of the package. PGAs are primarily used in applications that require more pins than what older packages such as the dual-in-line package (DIP) provide. Figure 14.4 shows a PGA.

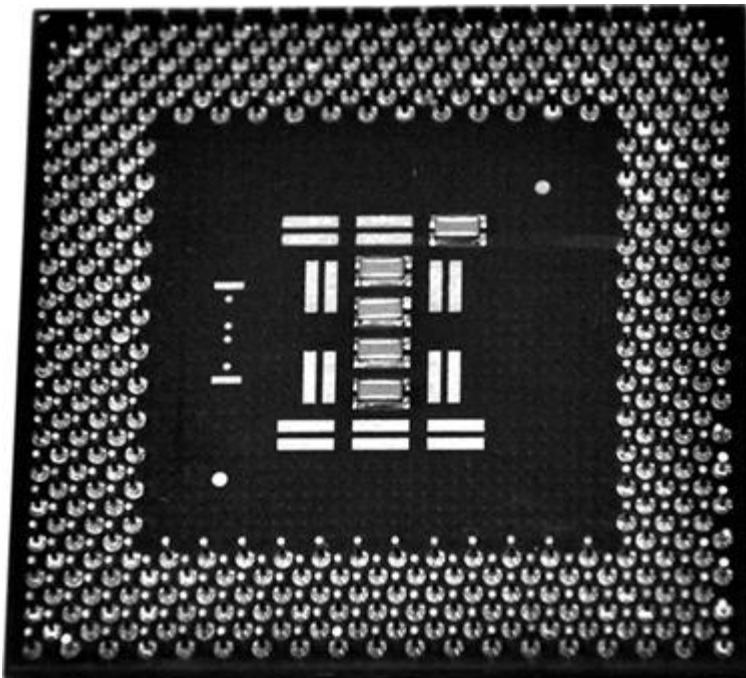


Figure 14.4 | A pin grade array (PGA)

Data Bus

The data bus is 32-bit wide and is designated as D_0 to D_{31} . It is not multiplexed, as in the 8086, and carries only data. Since data can be used as 8, 16 and 32 bits, the data bus is grouped into four 8-bit combinations, and access of each of these combinations is selected by a separate enable signals for the memory/ IO banks of the system.

Address Bus

The address of the processor is 32 bits wide, but the dedicated address lines are only A_2 to A_{31} , i.e., 30 lines. The address lines A_0 to A_1 are absent but they are encoded as four bus enable signals BE_0 to BE_3 (active low), which separately enable four memory banks. Each memory bank has a data bus width of 8 bits and data in the banks can be accessed by selective activation of the banks. See Fig. 14.5.

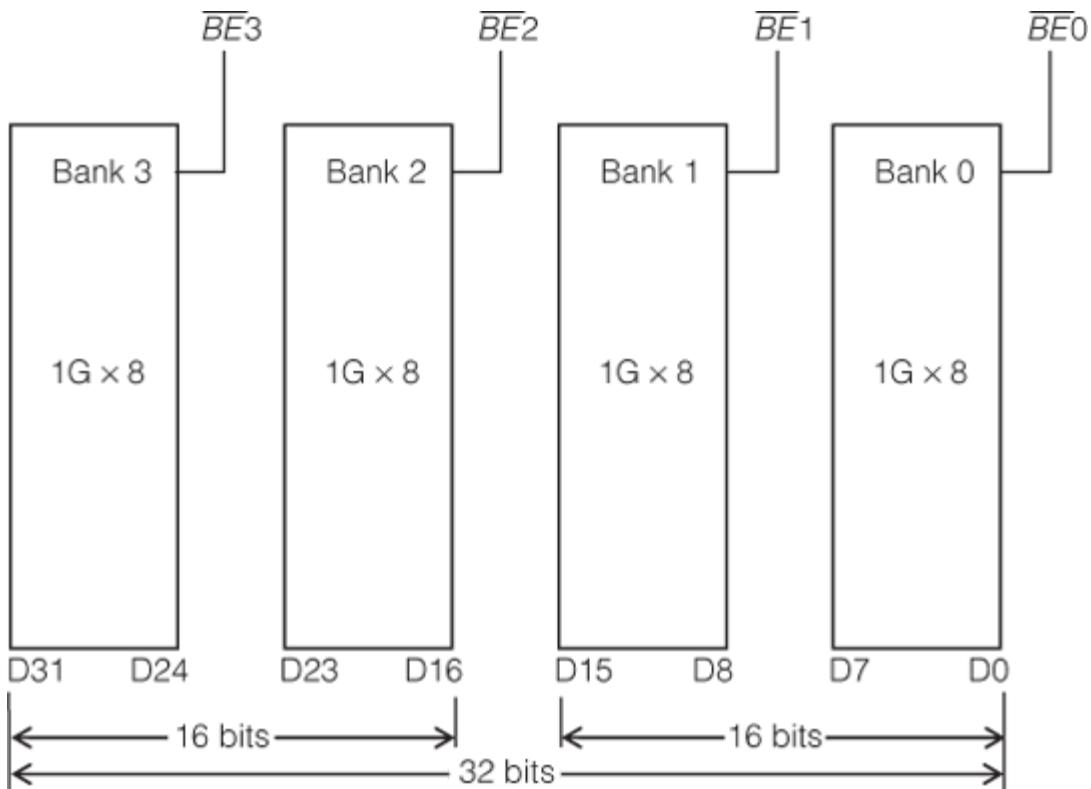


Figure 14.5 | Memory banks for the 80386

There are four bank enable signals BE_0 to BE_3 . They are selectively enabled depending on the banks that contain the data byte, word or double word. If a double word has to be read, all four bank enable signals are to be activated.

Example 14.6

For accessing a word, given the physical address as FFFF FFFCCH, which banks have to be enabled and what should be the status of the bus enable signals?

Solution

To read a word, two bytes have to be obtained, i.e., two banks have to be enabled. The specified address and the next address will be activated.

FFFF FFFCCH and FFFF FFFDH. These addresses are in Bank 0 and Bank1. Hence, the corresponding bank enable signals have to be made low. Thus, BE_0 and BE_1 have to be low and BE_2 and BE_3 are to be high (addresses with the lowest two bits 00 will be of bank 0, with 01 in bank 1, 10 in bank 2 and 11 in bank 3).

In this set, only the D/ C signal is new.

D/C (Data/Control)

This signal being high indicates that the data bus contains data for memory or I/O. The low status of this pin indicates that the processor is halted or is executing an interrupt acknowledge cycle.

Bus Control

These signals are used for efficient bus control circuitry.

ADS: Address Data Strobe: This signal becomes active whenever the processor issues a valid memory or I/O address.

NA: Next address: This signal causes the processor to output the address of the next instruction or data in the current bus cycle itself. This is used for address pipelining.

B16: Bus size 16 selects either a 32-bit data bus ($B16 = 1$) or a 16-bit data bus ($B16 = 0$).

Coprocessor Pins

These are pins receiving signals from the coprocessor, which is external to the 80386. The 80387 is a compatible coprocessor operating similar to the 8087 (arithmetic coprocessor) discussed in chapter 13.

PEREQ: Co-processor request is a request from a coprocessor to the 80386 to relinquish control of the bus so that the 80387 coprocessor gets a direct connection.

BUSY: This is a direct connection between the 80386 and the coprocessor, which indicates that the coprocessor is busy. This signal is used by the WAIT instruction to wait until the co-processor finishes its task.

ERROR: This is a signal from the coprocessor that an error has been detected.

Clock

This provides the timing for the processor. The system frequency is half the frequency of the clock connected to this pin. For the system frequency to be 33 MHz, the frequency to be given to the pin should be

66 MHz. There is no internal clock generator – so an external clock chip is to be used.

Reset

This is a high level sensitive input signal. When reset, control goes to the location FFFF FFFOH. The processor wakes up in the real mode and the upper 12 bits of the address remain high until a far jump is executed. This pin is included in the list of interrupt pins because resetting has a pre-defined vector to which control is branched.

The other pins that are shown in Fig. 14.4 have the same functions as they have in the 8086, and have been discussed in chapter 6.

14.5.1 | Real Mode Operation of the 80386

The 80386 can operate in the real mode as well as in the protected mode. (There is also a third mode called the virtual 8086 mode.) On start up, the processor is in the real mode. In the real mode, it acts as a faster 8086 processor. The speed increase is due to the increased clock speed and the hardware enhancements. It takes the '386 only two cycles to perform a read/write (with zero wait states), while the 8086 requires four cycles. Besides being faster, the '386 in the real mode can also use the enhanced instructions the processor provides. It can use the 32-bit registers and data bus, and so the bandwidth is double that of the 8086.

The main feature of the real mode is that it has access only to 1 MB of physical memory, and address calculation is done as in the 8086, with the base address being directly added to the offset. The register set available in real-address mode includes all the registers defined for the 8086 plus the new registers introduced by the 80386, i.e., FS, GS, debug registers, control registers, and test registers.

When the processor is reset or after power on, it is in the real mode, with the reset vector being the address FFFF FFFOH. The upper 12 address bits remain high until a FAR call or jump is executed. The first far JMP or CALL instruction causes A31-A20 to drop low, and thus, the 80386 can execute instructions in the lower one megabyte of physical memory (00000 to FFFFFH). This automatic lowering of address lines A31-A20 allows systems designers to use a ROM at the high end of the address space (of 1 MB) to initialize the system. The first instruction has an address in the upper part of the 1 MB physical memory and that is usually ROM – thus, the processor continues to be in the real mode, working within the physical memory of 1 MB.

We will now discuss the protected mode features of the processor, starting with memory management. Before we do that, it is necessary to understand a very important concept regarding memory.

14.6 | Virtual Memory

Virtual memory is a concept by which the processor (in effect, the user) is made to believe that it has much more memory than is physically available. In effect, there is this idea of an ‘enlarged address space’, which is called a ‘virtual address space’. This space maps into a physical address space, ultimately. When a user writes a program, he uses ‘virtual addresses’ or ‘logical addresses’. The idea is that an application does not get to see where the data is physically located. It is up to the address translation mechanism to map these virtual addresses to a physical address. This is an operating system concept that keeps the user free of the worry of exhausting the available memory, when he writes an application.

Recollect that whenever we executed DOS based programs, we used physical memory, which is RAM/ROM. We noted that for the 8086, with 20 address lines, the memory space available is 2^{20} bytes, i.e., 1 MB. Similarly, for the 80286, with 24 address lines, the physical memory it can address is $2^{24} = 16$ MB. For the 80386, naturally, the physical address space is 2^{32} , i.e., 4 GB.

Consider using the 80386 for an application – does it mean that we might have to limit our data and code to fit into 4 GB? We know that in the computers that we use now, we never have experienced a ‘memory crunch’ or the feeling that our application cannot be run because the memory capacity is insufficient, though there might have been times when we have felt that our application runs too slowly. All this leads us to the concept of what is called ‘virtual memory’.

In a computer system there is a memory hierarchy, and a two level hierarchy is shown in Fig. 14.6. Let us discuss the concept of virtual memory using just a two level hierarchy, Semiconductor memory, which is reasonably fast (RAM/ROM), is what we usually designate as ‘primary memory’, ‘main memory’ or ‘physical memory’. Keep in mind that we will frequently use these terms interchangeably and all the three refer to the memory that is addressable by the address lines of the processor. We then designate the hard disk (and other memory types) as ‘secondary memory’. The processor directly communicates with the main memory only, but data movement between the secondary and main memory is possible. Whenever an application is to be run, the processor expects to

find the associated data and code in the main memory. In case it is not found there, it should be brought thereto from the hard disk. This takes a small amount of time. This is the reason why we find our computer ‘slow’ if the RAM size is low. However, even if the RAM is of a large capacity, it is not necessary that our application files are currently present there. Managing the system such that important files are found in the main memory most of the time, is part of the ‘cleverness’ of the operating system.



Figure 14.6 | Memory hierarchy in a computer system

Does this mean that the ’386 processor does all this? Definitely not, but the processor provides the extra hardware, extra instructions, and an intelligent ‘memory management unit’ that allows the OS to handle many of such tasks efficiently.

14.6.1 | Protected Virtual Addressing Mode (PVAM)

Now, we will go through the features of the 80386 processor, which has made all this possible. Please be warned that understanding these features is not very easy and cannot be done in one reading. Repetitive reading and contemplating on the intricacies of the mechanisms involved, is the only way to get a clear understanding of this processor. Since the same features are also present in the 80486 and Pentium as well, it is important to get a good insight into all this, at this point itself. It is also important to keep in mind that the ’386 is an upgraded version of the basic 8086 – as such, it will have all the features of 8086 plus additional features. The ‘additional’ features in the protected mode are what we will discuss now.

The following are the topics we will cover:

1. Memory Management
2. Protection
3. Multitasking
4. Exceptions and Interrupts

Even though we discuss each of them separately, they are not really disjoint – for example, protection is a part of memory management;

memory management and protection have to be taken into consideration when multitasking is done, and so on.

14.7 | Memory Management Unit

The 80386 transforms logical addresses (i.e., addresses as viewed by programmers) into physical addresses (i.e., actual addresses in physical memory) in two steps:

1. Segment translation, in which a logical address (consisting of a segment selector and segment offset) is converted to a linear address. Here, memory is considered to be divided into segments, which are large units of storage.
2. Page translation, in which a linear address is converted to a physical address. This step is optional, and usable on the discretion of systems-software designers. In this step, memory is divided into smaller storage areas, each of which is designated as a page.



Figure 14.7 | Address translation using paging and segmentation

These translations are performed in a way that is not visible to applications programmers. The 80386 has both segmentation and paging. If both these schemes are active, the logical address given by the program is converted to a linear address by the address translation hardware of the segmentation unit. This linear address is interpreted as a page address and given to the paging unit, which converts it to a physical address. **But paging can be switched off by certain register settings. In that case, only segmentation is effective and then the logical address is directly converted to a physical address.** We will deal with ‘segmentation’ and ‘paging’ separately.

14.7.1 | Segmentation

A typical instruction could be

MOV EAX, COST

where EAX is a 32-bit register and COST is a 32-bit offset in a segment.

This implies that the segment has a 32-bit base address, because all addresses are 32-bit long. However, to access the location called COST,

we have to get the base address of the segment it is contained in. COST is the offset of the data from the base address. The physical address calculation involves just adding the offset to the base address. In the real mode (as in 8086), this address calculation is very direct (Fig. 14.8), but it is not so in the protected mode.

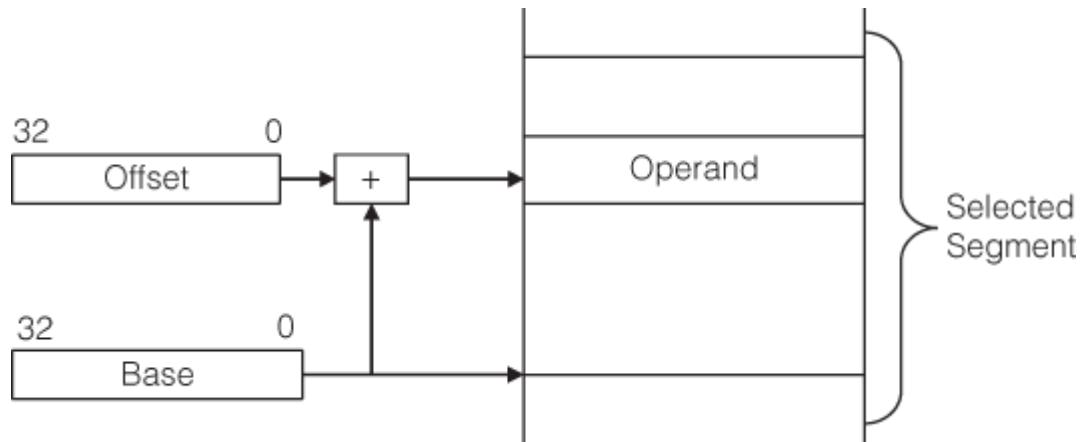


Figure 14.8 | Address calculation in the real address mode

The reason for the additional levels of calculation in the protected mode is the necessity for incorporating virtual memory concepts into memory management and ensuring protection features. Our problem is now of getting a physical address when the program supplies a virtual address, which is the logical address in a virtual memory system. In the instruction cited as an example, the label COST is the logical address of the operand to be accessed. COST will be a 32-bit address – a number (say, 0987 7654H).

Any location that is to be accessed will be in a particular segment. A segment can be of different types and it also has many attributes. All this is put together in what is called the ‘descriptor’ of the segment. Thus, any segment has a descriptor, which gives all information about the segment, i.e., it describes the segment. See in Fig. 14.9, the structure of a typical descriptor of a data/code segment used in an application program. (For an operating system segment, some bits of the descriptor will be different from this.)

Base (B31–B24)	G	D	0	A V L	Limit (L19–L16)	6
Access Byte				Base (B23–B16)		4
				Base (B15–B0)		2
				Limit (L15–L0)		0

15

0

Figure 14.9 | Descriptor of a data/code segment of an application program

A descriptor is an 8-byte (numbered as 0 to 7) entity, which describes the segment pointed by it. Let us see the contents of the descriptor in Fig. 14.9

1. It contains the 32-bit base address of the particular segment. This address is obtained as the concatenation of bytes 2, 3, 4, and 7 of the descriptor.
2. Bytes 0, 1 and 4 bits of byte 6 constitute the limit of the segment. This is a 20-bit number and specifies the maximum extent of the segment. There are two cases for the limit and this is decided by the G (granularity) bit of byte 6. If G = 0, it imposes a maximum limit of 64 K for the segment size. Then, the limit bits L₁₆ to L₁₉ are to be taken as zeroes, and the bits L₀ to L₁₅ give the size of this particular segment. Remember that with 16 bits, the maximum segment size is 64 K.
But if the G bit = 1, the number L₀ to L₁₉ is to be a multiple of 4 K (2¹²). Thus, the maximum possible size of a segment is $2^{20} \times 2^{12} = 2^{32} = 4$ GB. This is how the maximum size of a segment in the protected mode becomes 4 GB. In the descriptor, the number corresponding to the limit, gives the size of this particular segment.
3. Byte 5 of the descriptor is called the access byte. It contains the bits which specify the type and access rights to the segment pointed by this descriptor.

The various bits in the access byte mean the following. Refer Fig. 14.10.

P	DPL	S	E	X	R/W	A
7					0	

Figure 14.10 | Contents of the access byte in a descriptor

Bit 0 A: Accessed

This bit is set when the corresponding segment is accessed. The operating system tests this bit periodically to find out the frequency of usage of this segment – this is necessary for algorithms that need to have a measure of the ‘importance’ of this segment, meaning that, if this is a segment, which is accessed very frequently, it is best to retain it in the main memory.

Bit 4 S: Segment

indicates that this is a data or code segment descriptor for an application program ($S = 1$), or is a system segment descriptor ($S = 0$).

Bit 3: Executable E

This bit indicates if the segment is executable ($E = 1$) i.e., code segment, or non-executable ($E = 0$), i.e., data segment.

Bits 1, 2: Type

The type value depends on the E bit (bit 3 of the access byte) and the corresponding definitions are given in Table 14.2.

Table 14.2 | The Designation of the Bits of ‘Type’ Field of the Access Byte

Bit no.	If bit 3, E = 1 meaning executable (code segment)	If bit 3, E = 0 meaning non-executable (data segment)
2	C, Conforming, C = 1 Non conforming, C=0	ED Expand down. ED = 1, stack seg- ment Expand up ED = 0, data segment
1	R, Readable R = 1, may be read R = 0, may not be read	W, writable W = 1, writable W = 0, may not be written to

Bits 5, 6 DPL

This is a two-bit field, which stands for ‘descriptor privilege level’, and is part of the ‘protection mechanism’, which we will discuss in greater detail in Section 14.10.

Bit 7 P: Present Bit

The setting of this bit shows that the segment corresponding to this descriptor is now present in the main memory. If $P = 0$, it means that the corresponding segment is not in main memory. Hence, a type 11

interrupt is generated, which initiates an ISR to bring the segment into the main memory.

Byte 6

Four bits are used as part of the limit field. Byte 6 has four bits other than the limit bits

G: Granularity

This bit (discussed earlier) decides if the limit field has to be multiplied by 1 or 4 K. If G = 0, the multiplier is 1. If G = 1, the multiplier is 4 K.

D

Selects the default instruction mode for code segment descriptors. If D = 0, the registers and memory are 16 bits wide as in the 80286; if D = 1, they are 32 bits wide as in the 80386.

AVL

This bit can be used by the OS in any way it needs. It is often used simply to specify that the described segment is available. Bit 5 of byte 6 is to be 0.

14.7.2 | System Descriptors

Segments that are used by the operating system are separate and somewhat different from application program segments. They have a descriptor, which is slightly different. Figure 14.11 shows the general format of a descriptor used for operating system uses. If Fig. 14.11 is compared with Fig. 14.9, what will be noted is that only the bits in the access byte and the four bits of byte 6 are different. The difference in the access byte is that, 4 bytes are used to specify the type of the segment being described. There are 16 possible types, of which a few have reference to the 80286, some refer to future uses, and only a few indicate the types of system segments of the 80386. It is not worth going to the details of these types, since it will only lead to cluttering of this discussion with unnecessary detail.

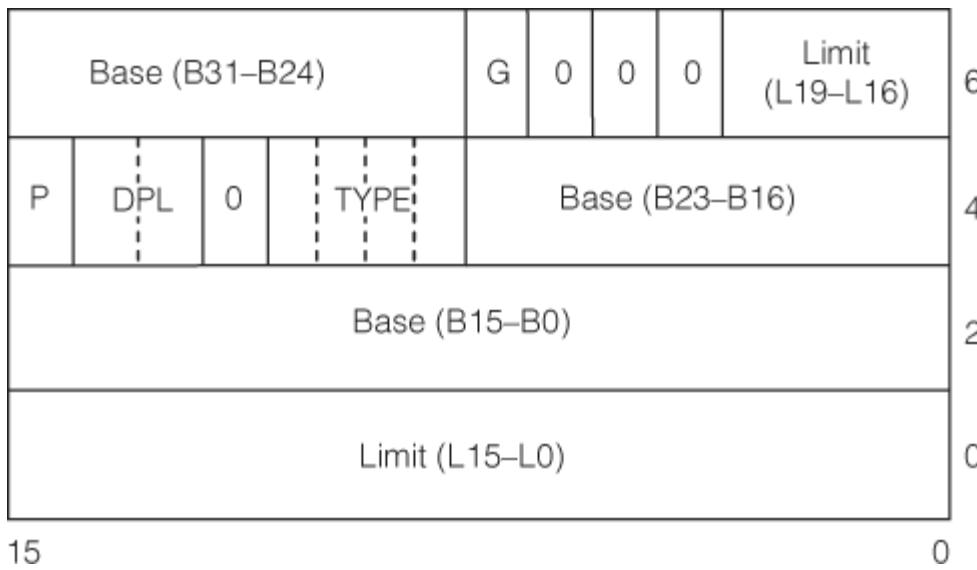


Figure 14.11 | Descriptor for a system segment

14.7.3 | Descriptor Tables

So, we see that each and every segment has a descriptor. However, who creates descriptors? Definitely, not the application programmer. When the programmer creates a segment, the system software creates the corresponding descriptor with the help of compilers and loaders and linkers – in effect; it is an operating system job.

In any system, there will be a number of segments of various types created for various applications – thus, there should be as many descriptors too. These descriptors are stored in tables called ‘descriptor tables’. There are two types of descriptor tables – the local descriptor table and the global descriptor table.

14.7.3.1 | Local Descriptor Table (LDT)

Suppose the system has 10 application tasks whose segments and corresponding descriptors have been created. For each of these tasks, there will be a descriptor table designated as the LDT of the particular task. Thus, we may name these tables as the LDT of task 1, task 2 and so on. There will be as many LDTs as there are application tasks.

14.7.3.2 | Global Descriptor Table (GDT)

There are segments and corresponding descriptors, which do not belong to any specific application but cater to the system as a whole – they are global in nature. These descriptors are stored in the GDT. **There will be only one GDT in a system.**

For executing a particular application task, say task A, segments from the LDT of task A are required and some operating system services are also required. Figure 14.12 shows how an application task needs the GDT as well as its own LDT.

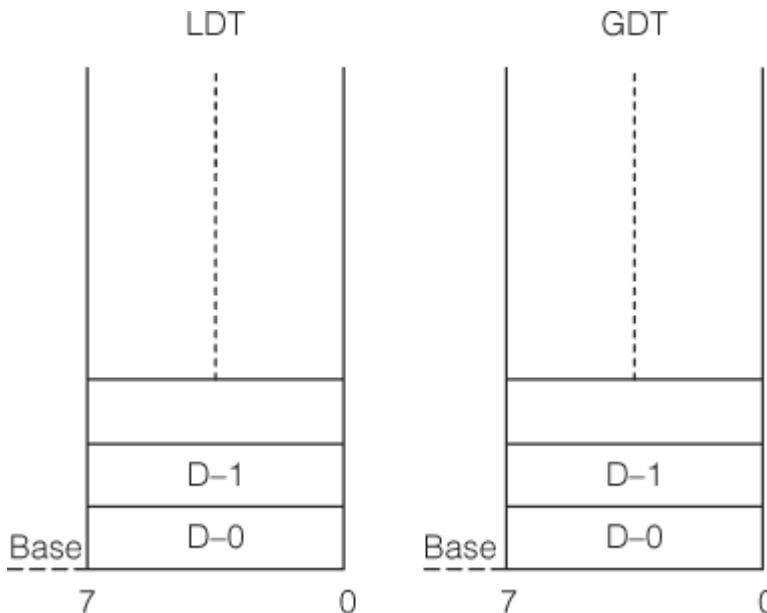


Figure 14.12 | LDT of a task and the GDT of the system

The next question is – where are these tables stored? The answer is – in the main memory, naturally. Remember also that a descriptor table is a memory array of 8-byte entries and that the size of a table is variable. Thus, these tables are also ‘segments’ and each of them also should have a descriptor. This may sound a bit confusing at first, but on going ahead, this confusion will soon be overcome.

This takes us to the discussion of the processor registers which are used in the protected mode. At a particular time, the descriptor of the GDT will be available in a processor register called the ‘Global Descriptor Table Register (GDTR)’. This is to be ensured when the system is switched to the protected mode. Since descriptors are 8-byte entities, these registers have to be 8 bytes in length, obviously. In fact, the loading of the GDTR with the descriptor of the GDT occurs when the system is put into the protected mode.

If an application task is running, the descriptor of the LDT of that task is available in the Local Descriptor Table Register (LDTR). We will soon get to the details of all the processor registers involved in the address translation mechanism.

14.7.4 | Selector

Remember that any logical address for the 8086 could be specified in the form

Segment base address: Offset

For the 80386 protected mode, the format of any logical address is modified to be

Segment Selector: Offset

The offset is 32 bits long as mentioned earlier, but the selector is only 16 bits long and has the structure as shown in Fig. 14.13. It contains a 13-bit number called the index. This number is an indicator of where in the descriptor table, the particular segment's descriptor is located. Thus, for each descriptor, the selector is a pointer to its position in the descriptor table. Since the index is 13 bits long, the number of descriptors it can point to is 2^{13} , i.e., 8192, this means that any descriptor table can have 8192 descriptors stored in it.



Figure 14.13 | Format of a selector

In the selector, there is a bit called TI that stands for ‘Table Indicator’. This indicates whether the selector corresponds to a descriptor in the LDT (TI = 1) or the GDT (TI = 0).

RPL

This is a two-bit field ‘Requested Privilege Level’, which is part of the protection mechanism, which will be discussed in detail in Section 14.10.4. The selector value is fixed up by linker or loader software.

Registers for Address Translation

To understand the concept of address translation in the protected mode, we should have a clear idea of the registers that are involved in this. Corresponding to the type of segments in use, there are the segment registers as shown in Fig. 14.14. Each segment register has a two-byte (16-bit) visible portion and an 8-byte hidden part in which the corresponding descriptor is loaded, when the corresponding segment is being accessed.

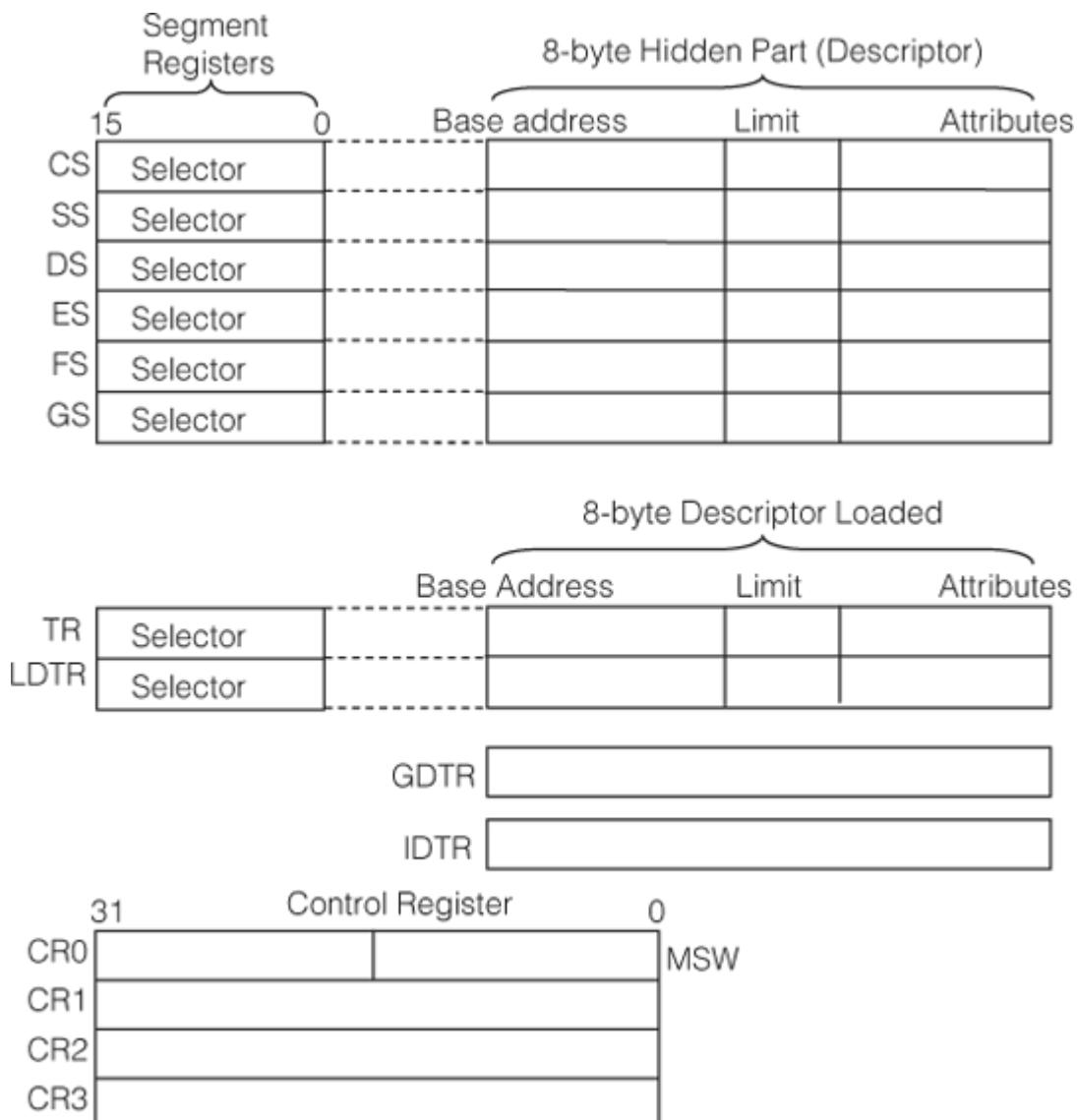


Figure 14.14 | Address translation registers

Now, let us consider a simple case of address translation. Note the following points. The GDT and LDT are, as mentioned earlier, memory arrays or segments. So they too have descriptors.

The GDT contains the descriptors of the segments used in system programs – there is only one GDT for the whole system. As such, there is no selector for the GDT. On startup, the descriptor of the GDT will be loaded into the GDTR. Once this is set up, the system can start running application tasks. Figure 14.15 shows the descriptor of a GDT loaded into the GDTR. Note that the GDT can contain a maximum of 8192 descriptors and the limit field indicates the size of the GDT (here, a maximum sized GDT is shown).

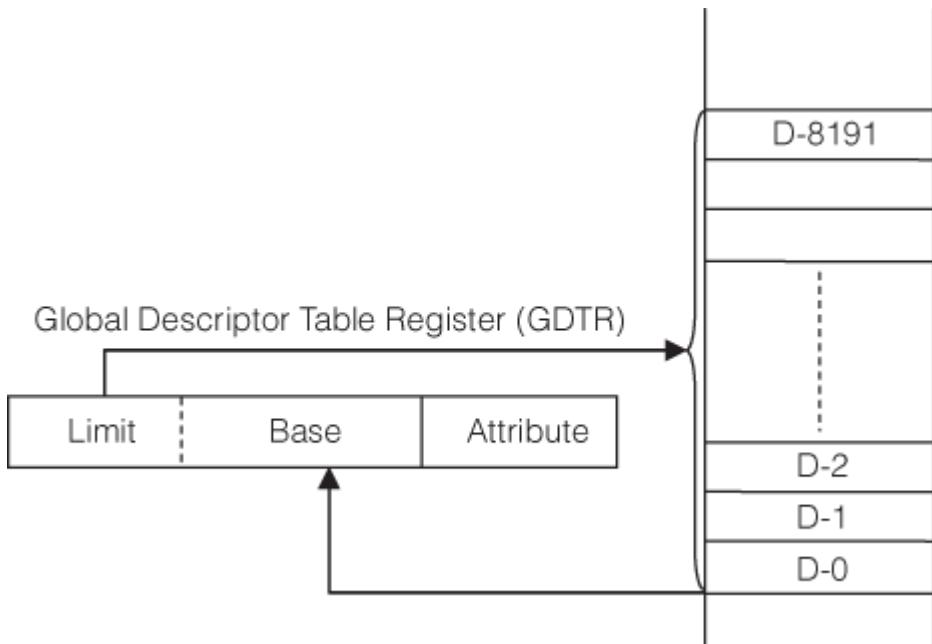


Figure 14.15 | GDT and GDTR

There are as many LDTs as there are tasks in the system. The descriptors of all the LDTs are in the GDT. When an application task is taken up, the selector of its LDT will be loaded into the visible part of the LDTR and then the corresponding descriptor is taken from the GDT and loaded in the hidden part of the LDTR. It is then that this task becomes active. (When this task is abandoned and a new one is taken up, the content of the LDTR will be changed to favor the new task).

14.8 | Converting a Logical Address to a Physical Address

Now consider an instruction for an application program, which is

MOV EAX, COST

This instruction needs an operand from a data segment to be moved to the 32-bit register EAX. The segment has a descriptor as well as a selector created by the system software. The needed data has an offset labeled COST which corresponds to, say, 0012 5674H. Looking at Fig.14.16, try to follow the steps outlined below. The steps in accessing data are:

1. Since it is a data segment, which is to be accessed, the selector of the data segment will be loaded into the visible part of the DS register, i.e., the segment register part. The selector contains a 13-bit number – the index. This is multiplied by 8 and added to the base address of the descriptor table (GDT or LDT, depending on the value of the TI bit in the selector).

2. Note that the base address of the descriptor table is available in the GDT register (GDTR), or LDT register (LDTR) of the processor. Thus, the descriptor of the data segment is obtained from the descriptor table, and this is loaded into the hidden part of the DS register. Since an application program is considered here, the descriptor of the data segment will be taken from the LDT of the application task.
3. The descriptor of the data segment has the base address of the data segment. This is used to locate the data segment. This base address plus the offset mentioned in the instruction, i.e., 0012 5674H (with label COST) are added to get the physical address of the operand, which is then moved to the register EAX.

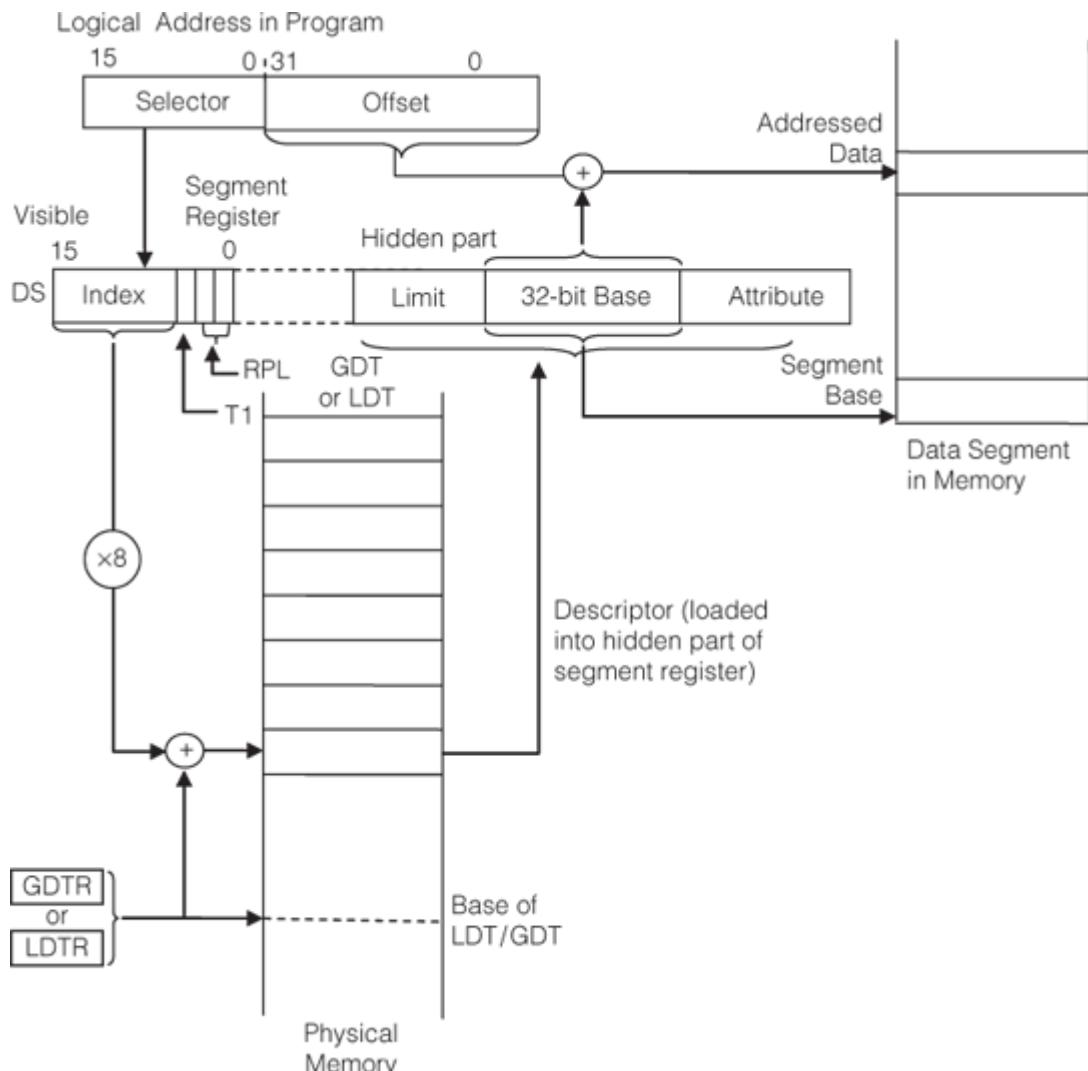


Figure 14.16 | Address translation of a logical address to a physical address

The above explanation should be sufficient to understand the translation of a logical address specified in the instruction to a physical address in memory, in the case of segmentation.

14.9 | Calculating the Size of the Logical Address Space

In a segmented model of memory, the address space as viewed by an applications program (called the logical address space) is a much larger space of up to 2^{46} (64 terabytes). How is this figure obtained? Let us make a calculation.

A selector has 13 bits as index for selecting one of the descriptors from a descriptor table. Thus, it can access up to 2^{13} descriptors with just one descriptor table. Since two descriptor tables (the GDT and one LDT) are active at a time, 2×2^{13} segments can be accessed. Each descriptor corresponds to a segment with a maximum size of 4 GB or 2^{32} bytes. The total available memory space is now calculated to be $2 \times 2^{13} \times 2^{32}$ bytes = 2^{46} or 64 terabytes ($2^6 \times 2^{40} = 64$ TB). Thus, the logical or virtual address space is 64 TB, which is much larger than the actual physical memory of 4 GB.

This calculation makes clear the idea of how the virtual memory concept creates an enlarged memory size. The address translation mechanism discussed above has the duty of mapping this large virtual memory space to a much smaller physical memory. Applications programmers do not need to know the details of this mapping.

As a matter of fact, the user usually thinks that the memory available to him is infinite. This illusion is created by having a secondary memory and shuttling data between the main and secondary memories according to the needs of the user. This underlying mechanism is not the concern of the user – the OS has the duty of performing this task efficiently.

14.9.1 | Paging

The mechanism of paging is in many ways, similar to segmentation – however, a smaller chunk of memory is handled in this case. Remember that a segment can have a variable size with a maximum physical size of 4 GB, but the size of a page is fixed to be 4 KB. This small size is what makes paging so attractive.

Just as in segmentation, whenever we want to run an application, we expect the required page to be in the main memory – if not, it has to be brought to the main memory. The charm of paging is that it allows just one page (i.e., just 4 K) to be brought in (or swapped out, if necessary). In contrast, segmentation requires that the whole segment be handled.

Paging is not a new concept, but 80286 did not allow paging – the '386 allows both paging and segmentation – which implies that any segment can be considered to be composed of a number of pages, and handled as one segment or as a number of separate pages. This is quite logical because we know that sometimes we might need only a part of the available data/code segment for an application.

14.9.2 | Page Translation

The 80386 has segmentation as a basic concept in its memory management unit. The **logical address** from a program is converted into a '**linear address**' by the segmentation unit, and given to the paging unit. This **linear address** is to be translated to a **physical address**, and this entails two levels of translation. A linear address can refer directly to a physical address so the segment and offset can be located directly, as is done in Section 14.8. This is the case when only segmentation is used. If paging is also activated, a linear address indirectly specifies a physical address (refer Fig. 14.7) by specifying a page table, a page within that table, and an offset within that page. A 32-bit linear address, as shown in Fig 14.17, is composed of the following fields:

1. A 10-bit field called DIR which points to a page descriptor in a page directory.
2. A 10-bit PAGE field which is used as an index into the page table determined by the page directory.
3. A 12-bit offset which points to the required byte within the located page.

A page or page frame, as it may be referred to, is a 4 K-byte unit of contiguous addresses of physical memory. See Fig. 14.18 for understanding the two levels of address translation in paging. Note that the base address of the current page directory is to be in the CR3 register. This register is in the processor and is also called the Page Directory Base Register (PDBR). The page directory contains page descriptors, which are similar to segment descriptors, except that these are 4-byte entries, i.e., 32 bits long. Thus, the page directory contains the descriptors for page tables. A page table is also a page (4 K in size) – thus, it has a 4-byte descriptor, which is in the page directory.



Figure 14.17 | Format of the 32-bit linear address

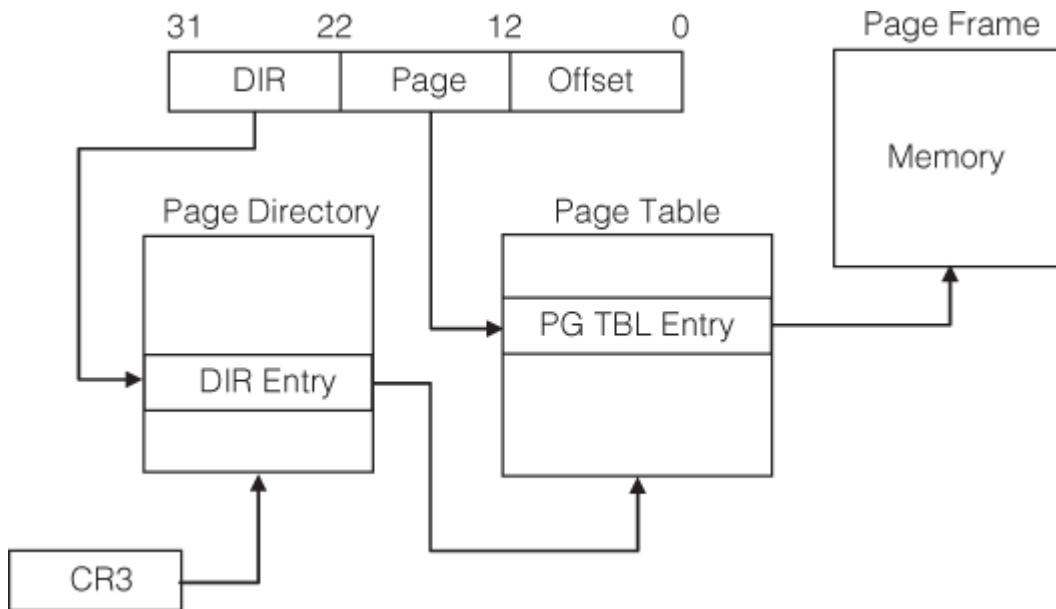


Figure 14. 18 | Two level address translation for paging

In the first level of address translation, the bits of the DIR field of the linear address is used as an index to locate a page descriptor in the current page directory. This will contain, among other things, the base address of a page table. In the next level of translation, the PAGE field of the linear address is used as an index to locate an entry in the page table. This entry contains the address of the required page in the physical memory. Adding the OFFSET field (in the linear address) to this address, gives the physical address of the required data. Note that this is a two level address translation mechanism, and the two levels are similar. In the first level, a page directory is accessed, and in the second, a page table is accessed. Both the page directory as well as the page table are 4 K-byte long pages.

Now to get a feel about the amount of memory addressable with paging, let us do a bit of calculation. Go back to the format of the linear address. Since the DIR field is 10 bits long, 2^{10} or 1 K page tables are possible. Each entry in the page table is indexed by the 10-bit PAGE field of the virtual address – hence, it has 2^{10} entries of page descriptors – which means 2^{10} pages can be addressed. Each page can contain 2^{12} (4 K) bytes. Thus, the total memory is $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ bytes or 4 GB, which is the complete physical memory of the 80386 processor (and also the maximum size of a segment in the protected mode).

14.9.3 | Format of a Page Table Entry

Figure 14.19 shows the format of a page table entry. Let us discuss its contents.

31	12 11	0
Page Frame Address 31...12	Avail 0 0 D A 0 0 U / S R / W P	

P – Present

R/W – Read/Write

U/S – User/Supervisor

D – Dirty

Avail – Available for systems programmer use

Figure 14.19 | Page table entry

Page Frame Address

The page frame address specifies the physical starting address of a page. Because pages are located on 4 K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the base address of a page table. In a second-level page table, the page frame address is the address of the page that contains the desired memory operand.

P bit

This bit has the same function as the P bit in segmentation, i.e., P = 1, indicates a valid page access. P = 0 indicates that this page is not in physical memory, and hence, it generates an exception, which initiates an ISR to bring the page to physical memory.

R/W

Read / Write-used for protection.

U/S

User/Supervisor-used in the protection scheme.

D

Dirty is undefined for page directory but used in the page table. It means the page has been modified.

A

Accessed is set to 1, whenever the page directory is accessed. Certain bits are left to be 0, which indicates Intel reserved.

14.9.4 | Translation Lookaside Buffer

It is cumbersome and time consuming to calculate the physical address from the linear address for every memory access. A translation lookaside buffer (TLB) simplifies the process. A TLB is a table in the processor which stores the physical addresses of the 32 recently accessed page table entries. The TLB is a ‘hardware cache’ for the physical addresses of the most recently used virtual address. It is a four way set associative cache inside the ’386 chip.

The paging unit receives a 32-bit linear address from the segmentation unit. The upper 20 bits of the linear address is compared with all 32 entries in the TLB to check if it matches with any of the entries. If it matches, the 32-bit physical address is calculated from the matching TLB entry and placed on the address bus. See Fig. 14.20a. If it does not match, the processor must fetch into the TLB, the page table entry from memory.

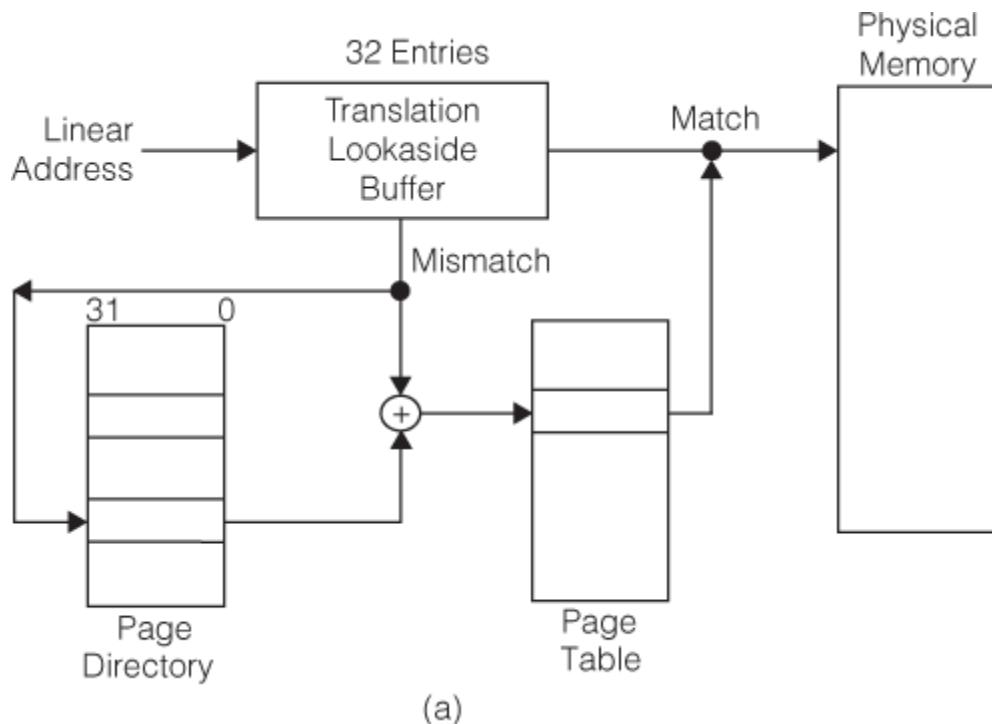


Figure 14.20a | Translation lookaside buffer inside the 80386 chip

Linear address (X)	Physical address(Y)
X1	Y1
X2	Y2
X3	Y3
.....
.....

(b)

Figure 14.20b | A conceptual view of the TLB

Corresponding to any linear address, there is a physical address obtained by page translation. Once page translation has been done once, the corresponding physical address is available. This is saved in the TLB (for the recent entries only). Conceptually, the TLB is shown in Fig. 14.20b, which is a table showing the physical address corresponding to a linear address in the case where address calculation has been done recently. For a linear address X, the physical address is Y.

14.9.5 | Combining Paging and Segmentation

Since 80386 has both segmentation and paging, it is easy to visualize a system in which a segment can be composed of a number of pages. To access a data in a page, what is needed is to have a technique by which the logical address corresponding to segmentation locates a segment, and then this logical address is converted to a linear address. This linear address is converted to a physical address as we have just seen. This is shown in Fig. 14.21.

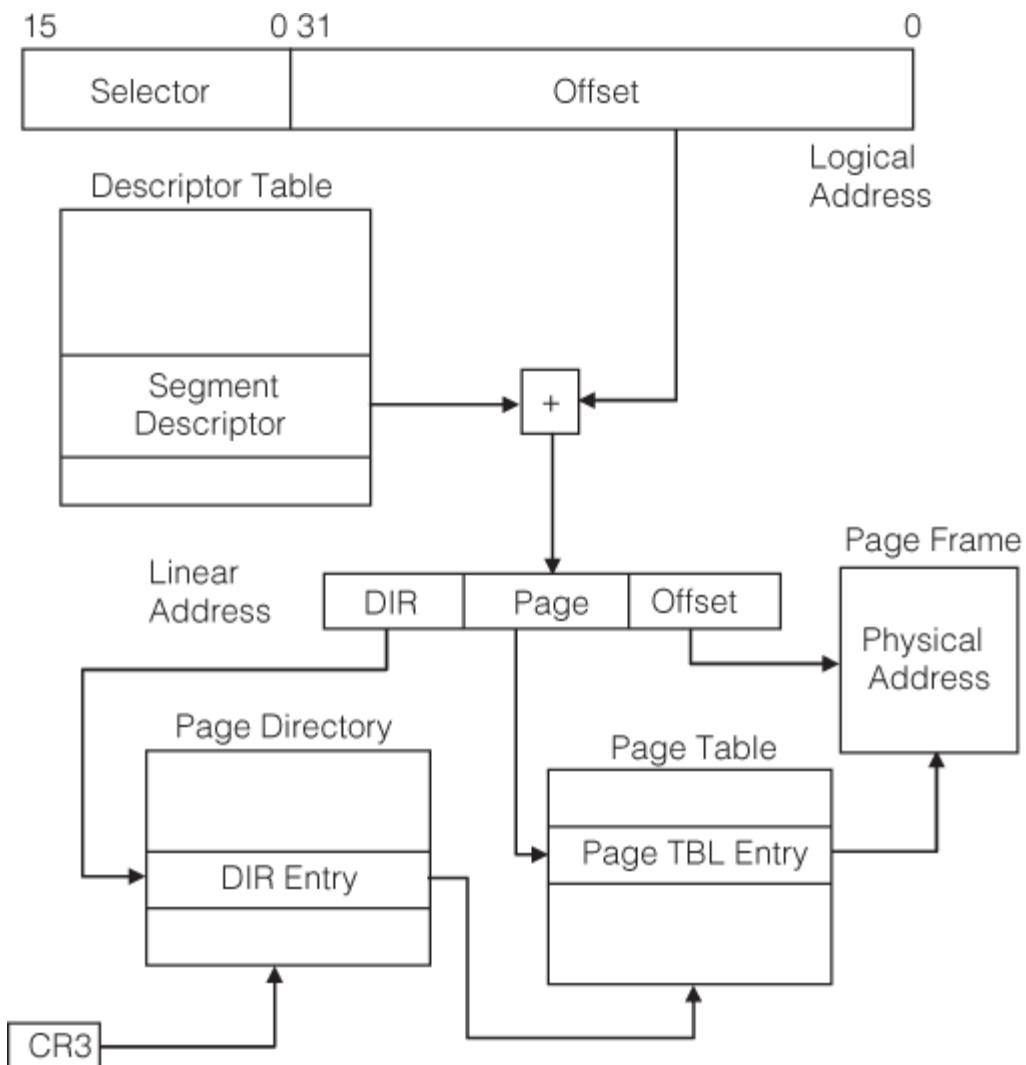


Figure 14.21 | Segmentation and paging combined

14.9.6 | Flat Model

In a '386 based system, it is not mandatory to use paging and segmentation together. There is the option to turn off the paging mechanism that leaves a segmented architecture. However, a mechanism to turn off segmentation is not inherently present. However, this can be achieved by programming techniques.

When the 80386 is used to execute software designed for architectures that do not have segments, it may be expedient to effectively 'turn off' the segmentation features of the 80386. The 80386 does not have a mode that disables segmentation, but the same effect can be achieved by loading the segment registers with selectors for descriptors that have a base addresses of 0, and by setting the limits to allow access the entire 32-bit address space. Once this is done, there is no longer a need to change the segment registers. The 32-bit offsets used by the 80386 instructions, in protected mode, are sufficient to access the entire linear

address space. This is called the ‘flat’ model. In a ‘flat’ model of memory organization, the applications programmer sees a single array of up to 2^{32} (4 GB) bytes.

14.10 | Protection

In the previous section, you have seen the word ‘protection’ being used without being told what exactly it is and how it is achieved. So the first thing to be done in this section, is to get a clear idea of the word ‘protection’, in the way it is used in the protected virtual addressing mode of 80386. There are many aspects to protection in this context.

Protection Between Tasks

A system can have a number of tasks, some of which are active, and some of which are not. One requirement is to protect user tasks from each other – this means that the code and data of one user is to be isolated from all other users – since code and data are in segments, this implies that segments of one task should not have access to the segments of other user tasks. This protection is implemented by having separate LDTs for each task. The descriptors of segments of each task are kept in separate local descriptor tables. At a time, only one user task is active and only one LDT can be used.

Protection between user and system tasks

The system programs, i.e., tasks associated with the operating system, manage the computer system as a whole. Thus, these tasks are very important and should not be allowed to be tromped on by users, i.e., application tasks. This is achieved by the concept of privilege levels. System tasks are allotted a high privilege level, while user tasks are at a lower privilege level – and direct access of segments of a higher privilege level task is not permitted. Thus, user tasks are prevented from accessing system level segments.

Capturing address violation errors

This aspect ensures that addresses beyond the limit of a segment are not accessed. Any such attempt is immediately aborted. It should be remembered that all these features are provided in the hardware of the processor, and when an operating system is designed using these hardware features, the design of the operating system becomes relatively easy. Now, let us get into the details.

14.10.1 | Protection Checks

The important aspects of protection in the 80386 are as follows:

- Type checking.
- Limit checking.
- Restriction in the accessible domain.
- Restriction in the entry points of procedures.
- Restriction in the usable instructions.

Address translation and memory access have already been discussed. protection hardware is an integral part of that unit, which applies to both segmentation and paging. It is, thus, necessary to keep in mind that protection is discussed separately, only for ease of understanding. Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria. All these checks are made before a memory cycle is started. Any violation prevents that cycle from starting, and results in an exception. Since the checks are performed concurrently with address formation, there is no performance penalty. Invalid attempts to access memory, results in an exception (an exception is an interrupt caused by an error).

Role of descriptors

The protection parameters are placed in the descriptor by system software at the time a descriptor is created. When a segment selector is loaded into a segment register, the descriptor of the segment is automatically loaded into the hidden portion of the segment register – thus, all the protection parameters are available in the segment register inside the processor and so subsequent protection checks on the same segment do not consume additional clock cycles. The first check that needs to be done is on the P bit in the descriptor. If P = 0, it implies that the referred segment is not present in main memory, and an exception is generated, which initiates an ISR to bring the corresponding segment to the main memory.

14.10.2 | Type Checking

There is a type field for a descriptor within the access byte (bits 1, 2 and 3). Refer Fig. 14.10. This field has two functions.

1. It distinguishes between different descriptor formats.
2. It specifies the intended use of a segment.

Programming errors can occur if a wrong segment selector is loaded into a segment register. Keep in mind that a code segment alone is an executable segment. It can be allowed to be read but cannot be written into. Data segments must be readable, and stack segments are to be - writable. Thus, the following conditions are to be verified.

1. The CS register can be loaded only with a selector of an executable segment.
2. Selectors of executable segments that are not readable cannot be loaded into data-segment registers.
3. Only selectors of writable data segments can be loaded into SS.

When an instruction with memory reference is to be executed, type checking is performed to confirm that

1. no instruction may write into an executable segment,
2. no instruction may write into a data segment if the writable bit is not set,
3. no instruction may read an executable segment unless the readable bit is set.

14.10.3 | Limit Checking

There is a limit field in the descriptor and the limit of a segment depends on the value of the G-bit. Before allowing a memory access, this field is used to confirm that memory access outside the segment does not occur. The value of the limit field has to be interpreted differently for expand-up segments (data), and expand-down segments (stack). Catching and preventing address violation errors goes a long way in avoiding system crash.

14.10.4 | Privilege Levels

The concept of privilege levels is all about trust and protection.

Trust

When a task is to be performed, it is code that gets executed – to execute code, data from various segments are required – also, code from other segments may be required when a branching to a different segment is to be done. The task that does this has to be a ‘trusted’ task – the code segment performing the task has to be a trusted segment. Then only it can be allowed to access other segments. The privilege level accorded to this segment depends on how much of trust is placed on it. If it is accorded the highest privilege level, it means that is allowed access to any other segment that it needs. Such a high privilege level is accorded

only to segments of operating system tasks, i.e., tasks with supervisor status, which are the most trusted tasks. Application level tasks usually are at low privilege levels – they are not trusted enough to allow them to use all available segments. All this means that all segments are given a particular privilege level (usually a number), and a task at a low privilege level can only access segments at the same level or at a lower privilege level.

Protection

Now think of a data segment or even a code segment which contains a procedure to be accessed from other code segments. Let us call it a target segment. What is the extent of protection it has? Can it be used by just any task? In that case, it is a segment at the lowest privilege level and hence the ‘least protected’. If it is highly protected, it will be accorded the highest privilege level and then, only tasks with the supervisor status can use it.

14.10.4.1 | Privilege Levels of 80386

The ’386 processor defines 4 levels of privilege numbered as 0 to 3 – the smaller the number, the higher the privilege level. Figure 14.22 shows the 4 level privilege hierarchy. Note that the kernel of the OS has been accorded the highest level of 0, while application tasks are at level 3. OS level services are at privilege level 1, and utilities and custom services of lesser importance are at level 2. In the figure, three application tasks A, B and C have been shown. It is not mandatory that an 80386 based system use all the 4 privilege levels. The system software can reduce it to 3 or 2 as needed.

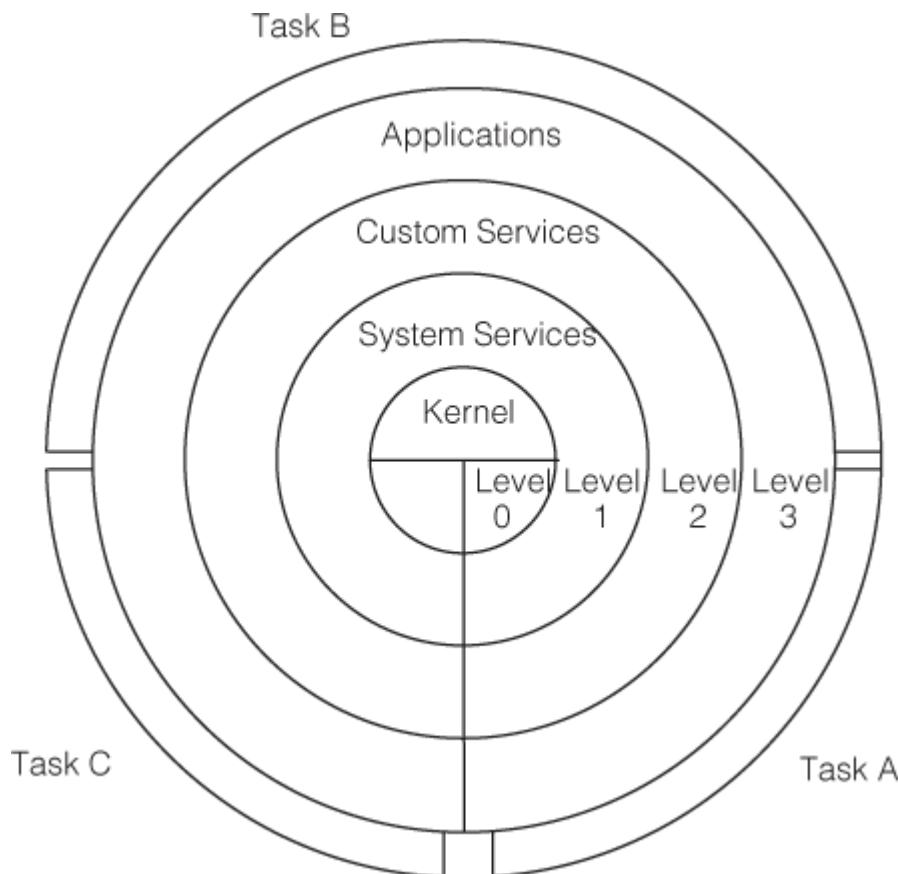


Figure 14.22 | Privilege level hierarchy of 80386

Three different types of privilege levels enter into the privilege level checks.

CPL (Current Privilege Level)

CPL is stored in the selector of the currently executing code segment register (bits 0 and 1); **CPL represents the privilege level of the currently executing task**, i.e., the program or procedure. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. Thus, this is also the privilege level in the access byte of the descriptor of this code segment (except for what is called a ‘conforming’ code segment). The CPL is also designated as TPL or the Task Privilege Level.

DPL (Descriptor Privilege Level)

DPL is the privilege level of an object, which is being attempted to be accessed by the current task. It is the privilege level of the target segment, and is contained in the access byte of the descriptor of the corresponding segment. The above discussion might seem a bit complex, but what is actually done is only a simple comparison of numbers. In simple terms, what is done is that the CPL of

the task in execution is compared with the DPL of the target segment. After comparison is done, access is allowed only if the CPL is higher or the same as the DPL of the target segment. Keep in mind that a higher privilege level corresponds to a lower number.

RPL (Requestor Privilege Level)

It is seen that the lowest two bits of any selector are reserved for the RPL field – these bits have an effect on privilege validation. This RPL field of a target segment's selector can be used to ‘weaken’ the CPL, if desired. If RPL of the selector is 2 while the CPL of the task is 1, the effective privilege level is the numerical higher of them, i.e., 2. Thus, the task becomes less privileged. The task is effectively excluded from accessing a data segment that the task might otherwise have been privileged to use. (This may seem unnecessary, but it is generally used to prevent inappropriate use of pointers that could corrupt the operation of more privileged code or data from a less privileged level.)

But the opposite effect of making RPL = 0 does not have any effect. If the selector's RPL is numerically less than the CPL, it has no effect.

14.10.5 | Accessing Data Segments

Assume that a task needs to get data from a data segment. The privilege level checks are performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register. (Keep in mind that the higher privileges are represented by smaller numbers and vice versa.) As Fig. 14.23 shows, three different privilege levels enter into this type of privilege checking mechanism.

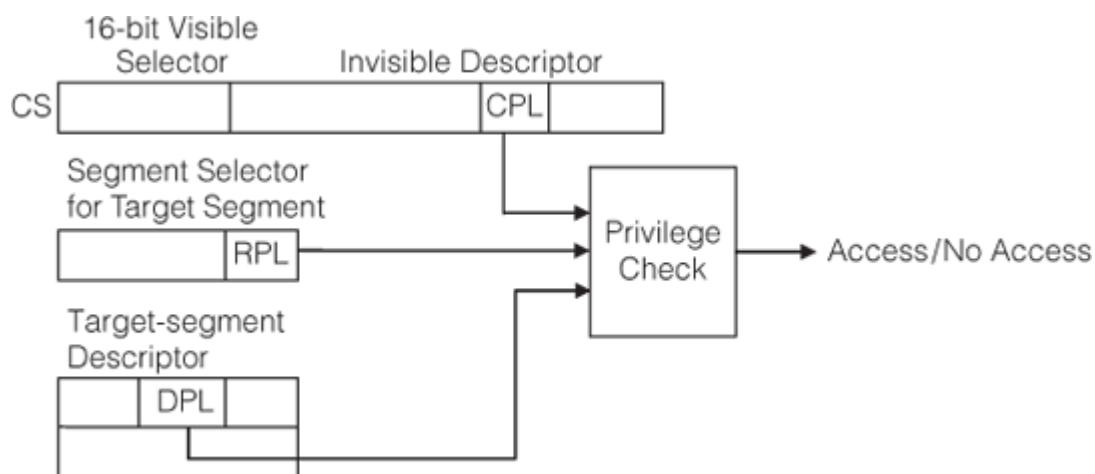


Figure 14.23 | Privilege level checks for a task when accessing a data segment

1. The CPL.
2. The RPL of the selector used to specify the target segment.
3. The DPL of the descriptor of the target segment.

Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is **numerically greater** than or equal to the maximum of the CPL and the selector's RPL. In **other words, a code segment can only access data that is at the same or less privileged level**.

The data segment has a DPL in the access byte of its descriptor, and this number designates the privilege level of the data segment. Then, the data segment must be at the same or lower privilege level in comparison with the CPL of the task. The accessible domain of a task varies according to its CPL. If the CPL of a task is zero, data segments at all privilege levels are accessible; when CPL is one, only data segments at privilege levels one through three are accessible; when CPL is three, only data segments at privilege level three are accessible. This property of the protection mechanism is used, for example, to prevent applications procedures from reading or changing tables of the operating system.

In short, the following numeric equation should be satisfied: $CPL \leq DPL$. However, CPL can be weakened by the RPL. So we define the Effective Privilege Level (EPL) as the numeric maximum of RPL and CPL. Thus, the final condition for access is that $EPL \leq DPL$.

Figure 14.23 shows the concept of privilege level checking when a task attempts to access a data segment. If the required condition is satisfied, access is allowed, otherwise it is not.

14.10.6 | Privilege Checks for Control Transfers

We have been talking of code segments having to access segments that contain data. However, many times code segments have to access other code segments. This happens when a far call or jump or ret instruction is encountered. (It also occurs in the case of interrupts but that will be discussed separately). There are a few issues in this type of control transfer.

- We can go to a code segment of lower privilege level, but on returning, it causes privilege level violations, so in effect this type of access is not possible.
- Going to code segments of higher privilege level is not permitted, but there is an indirect way of doing this and that is using ‘gates’.

Before proceeding further, a clarification on ‘conforming’ and ‘non-conforming’ code segments needs to be given.

Conforming Code Segment

A conforming code segment is one that does not have a privilege level of its own. It can be called by programs at any privilege level and then, it acquires the CPL of the calling program. Bit 2 of the access byte of a code segment descriptor specifies if it is a conforming ($C = 1$) or non-conforming ($C = 0$) segment. This mechanism is well suited to handle programs that share code but run at different privilege levels, e.g., shared libraries.

It permits sharing of procedures, so they can be called from various privilege levels. They are usually used for math library and exception handlers.

For a conforming code segment, the selector’s lowest two bits need not be the same as the CPL bits in the access byte of the code segment’s descriptor.

Non-Conforming Code Segment

But most code segments are not conforming. The basic rules of privilege mean that, for nonconforming segments, control can be transferred only to executable segments at the same level of privilege. There is a need, however, to transfer control to higher privilege levels – for example, when a low privilege application program wants to use a system service which is at a high privilege level. An example will be when an application program wants to write data to disk, which involves the use of a device driver. In this case, an indirect access is allowed through a ‘gate’, which is stored in the LDT or GDT.

There are three types of gates for this:

- Task gate
- Interrupt gate
- Call gate

In this section, only the call gate will be described.

To allow a program to jump to a more privileged level, it must go through Call gates, which basically defines the entry points for the privileged code. Corresponding security checks will be conducted at

those entry points to decide whether the invoking code has sufficient rights. These security checks are enforced by operating systems.

14.10.7 | Call Gates

A call gate is an 8-byte entity which contains the following information:

- Segment selector of the code segment to be accessed.
- Entry point for a procedure in the specified code segment (offset in segment).
- Privilege level required for a caller trying to access the procedure (DPL).
- Parameter count – if a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.

Access Control Policy for Call Gates

- In general, $CPL \leq DPL$ of the call gate (numerically).
- For a CALL instruction: DPL of the target code segment $\leq CPL$ (only calls to the more privileged code segment are allowed through a gate) – in numerical terms.
- For a JMP instruction: DPL of the code segment = CPL . Gates cannot be used for jumping to a different privilege level.

This indirect access mechanism permits access only if a certain check is done. The privilege level of the calling program (i.e., CPL) is compared with that of the call gate. If the former has a privilege level higher than or equal to the DPL of the gate, access is allowed – otherwise it is denied. This can be elaborated this way – a call gate with a DPL of 2 can be used by a program with CPL 2 to access a system service at level 0 – but the same gate cannot be used by a program with a CPL of 3. Figure 14.24 shows a call gate.

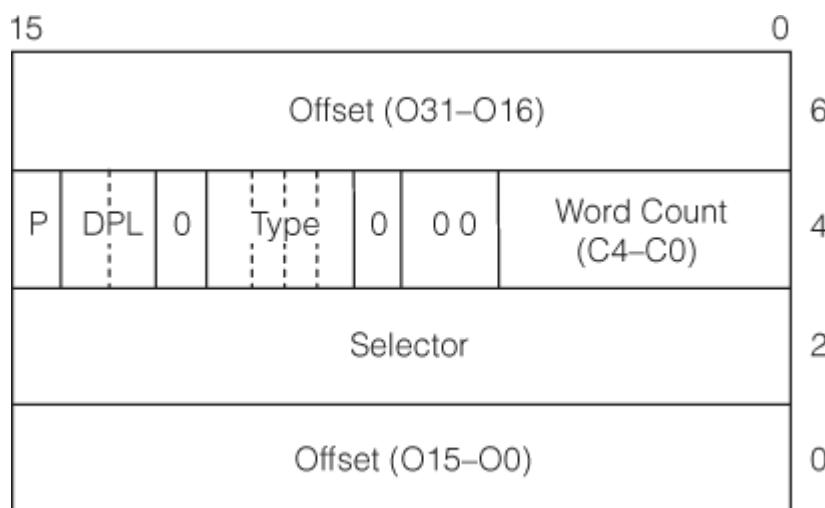


Figure 14.24 | Structure of a call gate

How to use call gates?

The instruction Call xxxx may be used where xxxx may be specified as a gate name but usually generates a selector leading us to an entry in the GDT or LDT, where the gate is stored. In the call gate, the detail of the target code segment is available and using this, the calling program enters the code segment at a higher privilege but only at the defined (in the gate) entry point. Remember that a gate cannot be used to enter code segments at lower privilege levels.

14.11 | Multi Tasking

What is meant by the word multitasking in the context of computers and processors? If a computer can handle multiple tasks at the same time, it has the capability of multitasking. However, the phrase ‘at the same time’ is a bit tricky. Can a single processor perform the task of executing multiple programs at the same time, i.e., in parallel?

Unlikely. Most of us have observed Windows OS allowing multiple tasks to be attended to, simultaneously. Then, there are systems with just one processor but multiple terminals and multiple users. It seems like all the users’ programs are being done concurrently. However, actually, that is not the case. A single processor can perform the execution of only one code sequence. When multiple tasks are to be handled, task switching is being done.

A program in execution is called a process, or task. Suppose there are 10 tasks that the system has to handle – the OS has to arrange a sequence of how to organize the execution of these tasks. This multitasking OS can use one of the many concepts available. There is the ‘time slice’ method in which one task is given a certain time to execute – after this, the task has to give way to the execution of the next task and so on. In this way, all the ten tasks are executed in one round, and the user/users feel that all tasks are running concurrently. In the next round, once again all the tasks get executed and this continues.

In priority-based task scheduling, tasks which are more important are allowed to execute, while low priority tasks are made to wait. There is also the co-operative scheduling system in which a task is allowed to execute until its completion. It is the OS which decides the task scheduling mechanism. Windows is a multiuser OS while DOS is a single user OS.

Task Switching

When one task is temporarily abandoned and another is taken up, context switching or task switching is said to have occurred.

14.11.1 | Issues in Multi-Tasking Systems

One main issue in task switching is that, before abandoning a task temporarily, its ‘context’ is to be saved so as to resume execution at the point at which it had left off. The context consists of the content of the registers being used, the state of the stack and all the information necessary to resume the old task. This is similar to returning from an interrupt, but more variables and more parameters have to be saved before switching to a new task.

If a processor is said to have ‘multitasking’ capability, it means that it has hardware features to handle all the issues associated with task switching. It should have the necessary mechanisms, instructions, registers and necessary additional hardware. These features are available in the 80386 and are useful in designing a multitasking OS for this processor.

TSS and LDTs

It was mentioned earlier that if there are ‘n’ tasks in a system, there will be n LDTs – i.e., for each task there is an LDT and each LDT holds the descriptors of the segments required to run the particular task. When a particular task is taken up for execution, the selector for its LDT is loaded into the LDTR in the processor. Thus, we need to remember that an LDT is associated with a particular task and that there are as many LDTs in the system, as there are tasks. The descriptors of all the LDTs of a system are stored in the GDT of the system.

Similar to this, there is another structure associated with a particular task and that is its ‘Task State Segment’ or TSS. Each task has a TSS. All the information pertaining to a particular task is stored in its TSS, which is also a memory array or segment stored in main memory.

The TSS has the following fields.

1. A dynamic set which changes every time a task switch occurs. They are:

1. The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI).
2. The segment registers (ES, CS, SS, DS, FS, and GS).
3. The flags register (EFLAGS).

4. The instruction pointer (EIP).
 5. The selector of the TSS of the previously executing task (updated only when a return is expected).
2. A static set that the processor reads but does not change. This set includes:
- o. The selector of the task's LDT.
 1. The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
 2. Pointers to the stacks for privilege levels 0-2.
 3. The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs.
 4. The I/O map base.

Figure 14.25 shows the structure of a typical TSS.

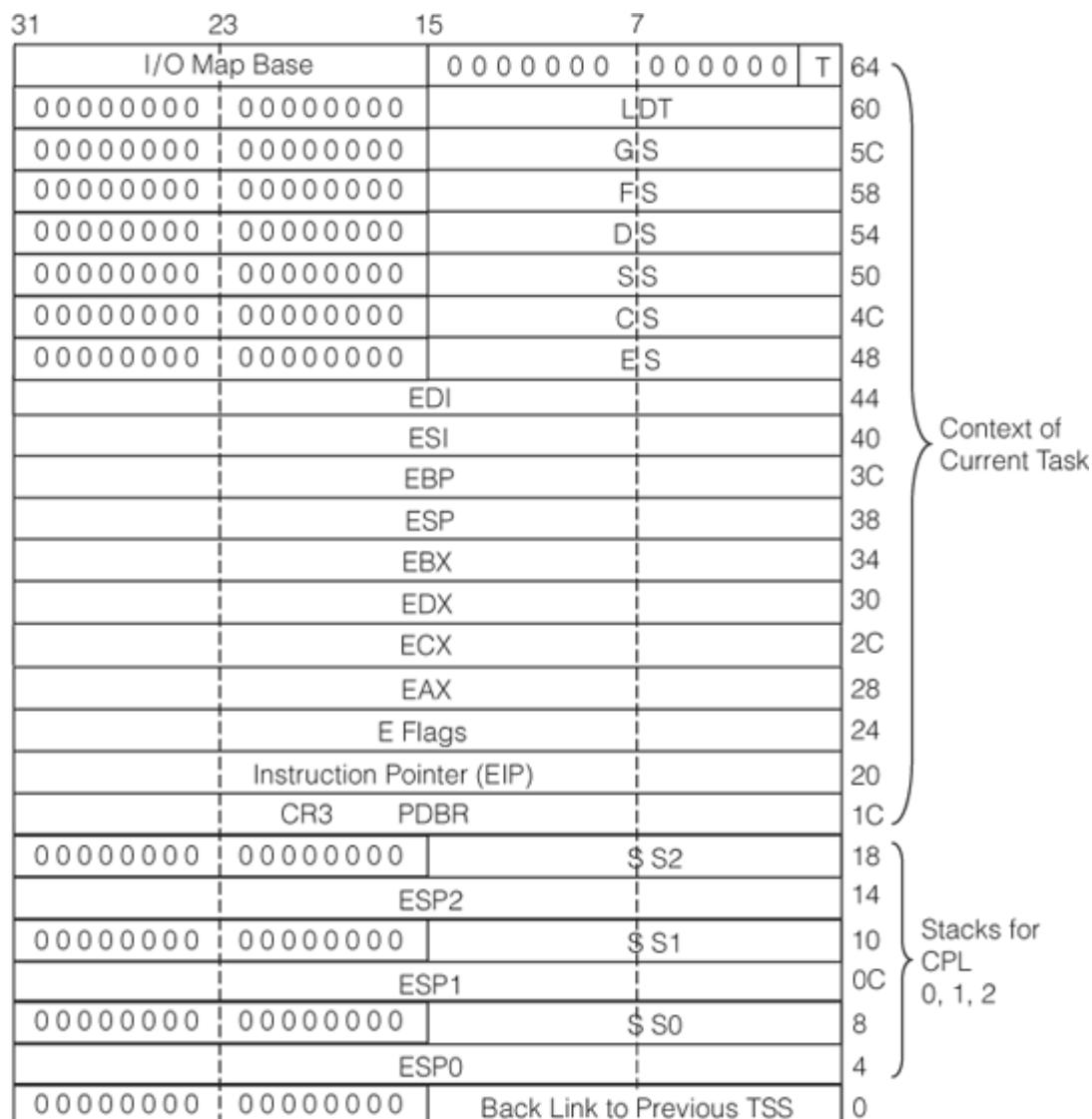


Figure 14.25 | Structure of a typical TSS

The TSS is a segment, and hence, it has a descriptor that is stored in the GDT of the system (just as the LDT's descriptor is). The descriptor of a TSS is similar to a segment descriptor except that it has a 'B' or busy bit. The B-bit in the type field indicates whether the task is busy. The B-bit allows the processor to detect an attempt to switch to a task that is already busy.

14.11.2 | Task Register

The processor has a register called a task register (TR) which points to the current task. When a particular task is taken up, the selector of the TSS is loaded into the visible portion of the task register. The selector contains the index that locates the descriptor of the corresponding TSS in the GDT. Then, the descriptor of the TSS of this task gets loaded into the hidden part of the register.

Two important points regarding the execution of the current task is that it has the TSS selector in the Task register (TR) and the LDT selector in the LDT register (LDTR). This initiates the actions necessary to access the contents of the LDT and TSS that belong to that task. When task switching occurs, the contents of these registers are changed.

14.11.3 | Task Switching

How can task switching be done?

- The current task executes a JMP or CALL that refers to a TSS descriptor.
- The current task executes a JMP or CALL that refers to a task gate.
- An interrupt or exception vectors to a task gate in the Interrupt Descriptor Table.
- The current task executes an IRET when the NT flag is set.

Let us discuss the above four cases in more detail.

1. In the first case, using instructions like JMP taskname or CALL taskname can load the selector of the corresponding TSS into the task register (TR) which gets the TSS descriptor from the GDT.

2. In the second case, the task switching is indirect. There is a 'task gate' involved just like the call gate we had discussed in the previous section. In both cases, a new task is embarked upon.
3. Task switching can also occur by the occurrence of an interrupt. This is also logical, but it involves an 'interrupt descriptor table', which we are yet to discuss. In any case, we know that an interrupt or exception causes the current task to be abandoned and a new one to be taken up. Thus, it can

lead to a task switching when there is the mechanism to access the TSS of a new task.

4. Whether invoked as a task or as a procedure of the interrupted task, an interrupt handler always returns control to the interrupted procedure in the interrupted task. If the Nested Task (NT) flag is set, however, the handler is an interrupt task, and the IRET switches back to the interrupted task.

What are the actions that follow a task switch?

1. If the switching occurs by a JMP or CALL instruction, the usual rules of privilege levels apply for the TSS descriptor or task gate, and task switching occurs only if the conditions are satisfied. Checking of the P bit and limit bit of the TSS descriptor is also done.
2. The context of the current task is saved, i.e., contents of the registers are copied to the TSS. The EIP register, which is now stored in the TSS, will be pointing to the next instruction after the one that caused the task switch. These actions correspond to ‘saving the context’ of the abandoned task.
3. Next, the actions for taking up the new task are begun. The Task Register (TR) is loaded with the new task’s selector. The selector is either the operand of a control transfer instruction or is taken from a task gate, which gets the TSS descriptor and then the TSS from memory.
4. Now the new task can be proceeded with. For that, its state is loaded from the TSS into the corresponding registers. The registers loaded are the LDT register, the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, the segment registers ES, CS, SS, DS, FS, and GS, and PDBR.
5. Every task switch also sets the TS (task switched) bit in the MSW (machine status word), i.e., CR0.

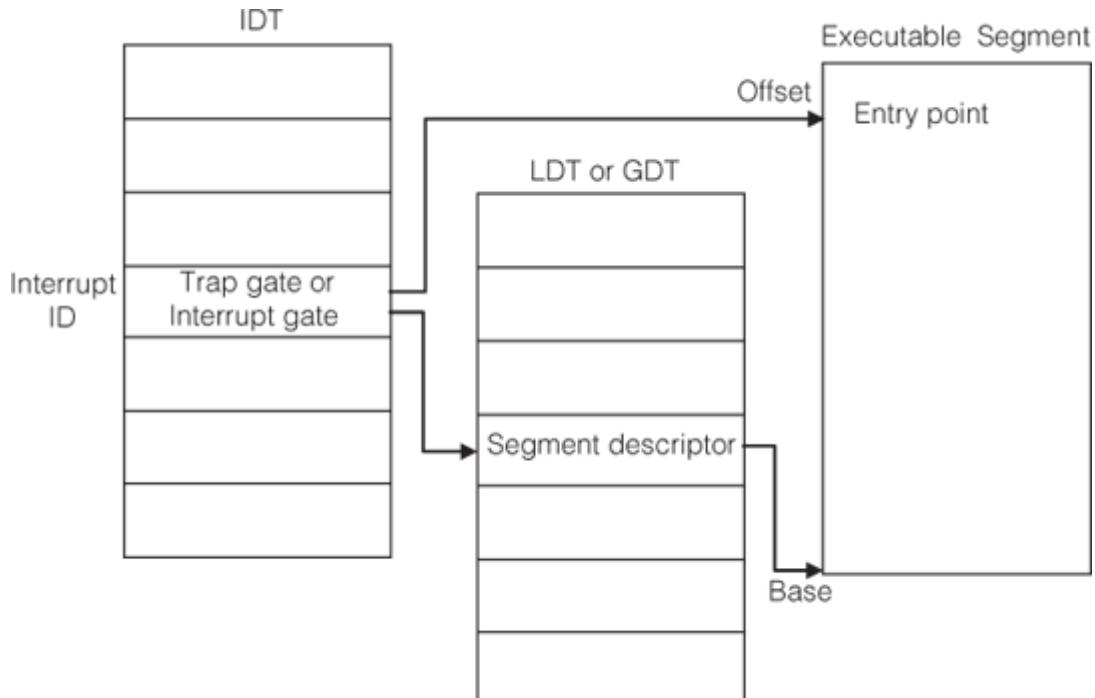
14.11.4 | Task Linking

The back-link field of the TSS and the NT (Nested Task) bit of the flag word together allow the 80386 to automatically return to a task that called another task or was interrupted by another task. When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task’s TSS and, at the same time, sets the NT bit in the new task’s flag register. The NT flag indicates whether the back-link field is valid. The new task releases control by executing an IRET instruction. When interpreting an IRET, the 80386 examines the NT flag. If NT is set, the 80386 switches back to the task selected by the back-link field.

14.12 | Interrupts of 80386

The interrupts of 80386 function exactly in the same way as the 8086 interrupts and the categorization is also similar, i.e., the hardware interrupts are NMI and INTR and the software interrupts can be used in the format INT n . There are interrupts generated due to errors and they are called exceptions. The interrupt response is also similar, i.e., the interrupt vector is in a table. On reset, this table is at location 0 and has a maximum size of 1 K.

Now, for the differences – instead of the ‘interrupt vector table’, here it is the ‘Interrupt Descriptor Table (IDT)’ that has the interrupt vectors. There is one and only one IDT for a system. It can be located anywhere in the physical memory, and that is specified by the descriptor of this table which will be in the GDT. When the system is in protected mode, the descriptor of the IDT is loaded into the IDT register (IDTR). Figure 14.26 shows the interrupt structure of 80386.



Task gates have been discussed already. An interrupt can cause task switching if the target segment corresponds to a different task. That is why we can have task gates also in the IDT.

We know that a gate contains an entry point into a code segment. It contains a selector of the destination code segment and the offset of the allowed entry point. Call gates have already been discussed. An interrupt is similar to a call – the response is similar, but the initiation mechanism is different. Also an interrupt causes the flags to be saved on the stack, and the return instruction is IRET rather than RET. So, we do not have call gates in the IDT but, instead, have trap and interrupt gates.

The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF. The figure shows the channeling of an interrupt from the IDT to the interrupt vector. Because descriptors are involved in this sequence, privilege level checking is done for interrupts as well. The privilege rule that governs interrupt procedures is similar to that for procedure calls.

14.12.1 | Exceptions

Exceptions are caused by error conditions. For the case of the 8086, we know that a divide by zero error causes a type 0 interrupt. We can call it an exception. The 80386 has more error conditions under which interrupts can occur. Table 14.3 shows the list of the 80386 interrupts. Compare it to Table 8.2, which gives the interrupt listing for a PC.

Table 14.3 | List of 80386 interrupts

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupts
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(Reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(Reserved)
16	Coprocessor error
17-31	(Reserved)
32-255	Available for external interrupts via INTR pin

14.13 | Privileged Instructions

The instructions that affect system data structures can only be executed in the protected mode and it is also to be ensured that they are used only at the highest privilege level, i.e., when CPL is zero. These instructions include:

CLTS — Clear Task Switched Flag

HLT — Halt Processor

LGDT — Load GDT Register

LIDT — Load IDT Register

LLDT — Load LDT Register

LMSW — Load Machine Status Word

LTR — Load Task Register

MOV to/from CRn – Move to Control Register n

MOV to/from DRn – Move to Debug Register n

MOV to/from TRn – Move to Test Register n

In fact, all the instructions involving load/store of protected mode registers are privileged instructions.

14.13.1 | Switching to Protected Mode

It has been mentioned that when the processor is powered on, it is in the real mode. How is it switched to the protected mode?

To switch it to protected mode, a number of initialization operations need to be done. These can be done in the real mode itself. (These things can be done in the protected mode as well, but then it must be ensured that the initialization procedures must not use protected-mode features that are not yet initialized.) For example, the GDTR must point to a valid GDT, the IDTR must point to a valid IDT and so on – after all the initializations are done, the switching may be done.

Next, to switch it to protected mode, the PE (Protection Enable) bit of control register CR0 must be set. Those who are familiar with the 80286 will remember that there is a LMSW (Load Machine Status Word) instruction, which was executed with PE = 1. It is this same register MSW that has been re-named as CR0 for the '386.

The difference between the '286 and '386 is that in the case of the former, it is not possible to return to the real mode without doing a hardware reset. However, in the case of the '386, returning to the real mode is possible by clearing the PE bit and loading it to CR0 again. Of course, various conditions will have to be arranged before switching back to the real mode. All these conditions can be put inside a procedure, with the final instruction being the clearing of the PE bit of CR0. Thus, the processor returns to real mode.

14.13.2 | Virtual 8086 Mode

We know that even though the protected mode is important, running programs in MS-DOS has its charms and to facilitate the use of both these modes, the '386 has a virtual mode which allows switching between the real and protected modes. Thus, a multitasking system can have users running programs in the protected mode as well as users

running real mode 8086 programs. Time slices are allotted for all these programs, and thus, there is continuous switching between real and protected mode. Virtual 8086 mode divides the computer into multiple address spaces and maintains virtual registers for each virtual machine. This mode is entered by setting the VM (Virtual Machine) bit in the EFlags register.

The desire to allow execution of MS-DOS applications under the control of a protected-mode environment, (such as Windows) has led to the inclusion of virtual-mode to all of Intel's 32-bit processors. When the processor is running in the virtual machine mode, it behaves as if it were an 8086 equipped with protection, multitasking and paging support.

Note that the virtual 8086 mode has nothing to do with 'virtual memory' but is associated with the term 'virtual machine', which can be explained as follows. 'A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization. The main advantage of system virtual machines is that multiple OS environments can co-exist on the same computer, in strong isolation from each other'.

14.13.3 | Conclusion

With this, we come to the end of our discussion on 80386. Most of the important features have been explained but to be able to write instructions to use all these features, more study and more information (from the programming manual) is required. The attempt here has to give an idea of the capabilities of one of the more sophisticated processors of the Intel family. The 80486 and Pentium have all these features, so these sections have to be understood for appreciating the characteristics of Pentium.

14.14 | Enhanced Features of 80486

1. The 80486 processor was introduced by Intel in 1989. The first important difference that the '486 has in comparison with its predecessor is that it has an integrated floating point unit on the chip itself. Previous processors had the arithmetic unit as a separate unit – the 8086 had 8087, the 80186 had the 80187, the 80286 had the 80287, the '386 had the '387 and so on. Logically, the 80486 should have had an arithmetic co-processor 80487 external to the chip, but instead, Intel placed the arithmetic co-processor inside the chip and called the whole chip as 80486DX. To improve its market segment, Intel also sold a '486 processor without an FPU (floating point unit, as the arithmetic co-processor is

called). It was named as 80486SX, but in fact, it was just the 80486 processor with its FPU turned off.

2. The second important enhancement is that of adding an 8 KB of cache **on the chip** for data as well as code. All predecessor processors had cache outside the processor. With this enhancement, on-chip cache came to be designated as L1 (level 1) cache while off-chip cache is called L2 (level 2) cache. Control register CR0 is used to control the internal cache with two new control bits which were not present in the 80386 microprocessor.
3. The internal cache improved the memory access speed substantially but later versions had something called ‘clock doubling’. New editions were released with higher clock frequencies as they hit on the idea of *doubling* the *internal* clock frequency in relation to the external clock. These double-clocked processors were given the name 80486DX2. A very popular model in this series had an external clock frequency of 33 MHz while working at 66 MHz internally. This principle (*over-clocking*) has been allowed in one way or another in all later generations of CPU’s, though there are problems associated with it. It is just an option allowed for a processor and is specified in the motherboard spec.
4. Compared to the 80386, the ’486 is a heavily pipelined processor. It has a 5 stage pipeline as shown in Fig 14.27. Each stage takes one clock cycle, but once the pipeline is full, each instruction will execute in a single clock. Referring to Fig. 14.27, the stages in the pipeline are as PF-Pre-fetch, D1-decode1, D2-decode2, EX-Execute and WB-Write back. I₁ to I₅ correspond to five instructions in the pipeline. As per this figure, there are two decoding stages. This is because of the varied addressing modes of 80486 and the necessity for protection checks before any access is allowed. This processor has a 5 stage pipeline – if the pipelining stages are split into smaller and smaller steps, it is called ‘super pipelining’, but such an idea is not used in the 80486.
5. ’486 processors and above support burst mode transfers. This feature is helpful in saving time when accessing data from consecutive addresses. Let us try to understand this important feature. Figure 14.28 shows the read cycle timing for an ordinary memory read cycle. Two clock cycles are required to transfer data (which is 32 bits long). The first clock cycle T₁ provides the memory address and control signals, while in T₂, data is transferred between the memory and the microprocessor. The figure shows the case of two 32-bit words being transferred. ADS is address strobe. Thus, two clock cycles are required to transfer 32 bits of data. If 128 bytes of data is to be transferred, obviously 8 clock cycles will be required. Now, let us contrast this with a burst cycle. A burst cycle will require only 5 clock cycles to transfer 128 bytes of data. How do we get such a number? See Fig 14.29.

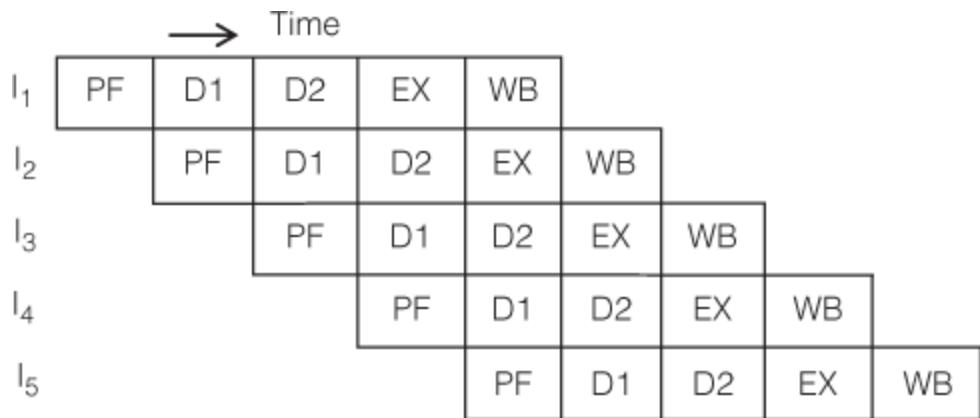


Figure 14.27 | Five stage pipeline of the 80486

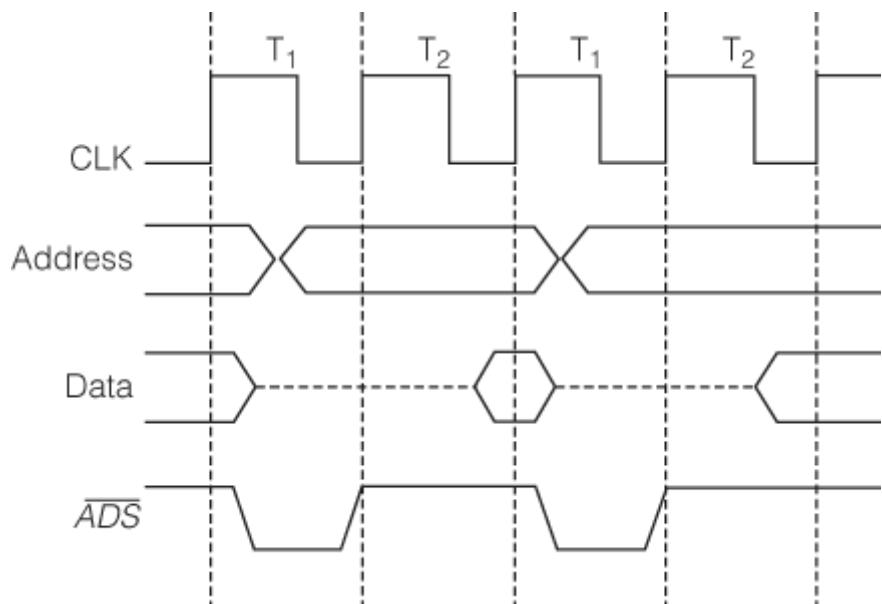


Figure 14.28 | Non-burst bus cycle

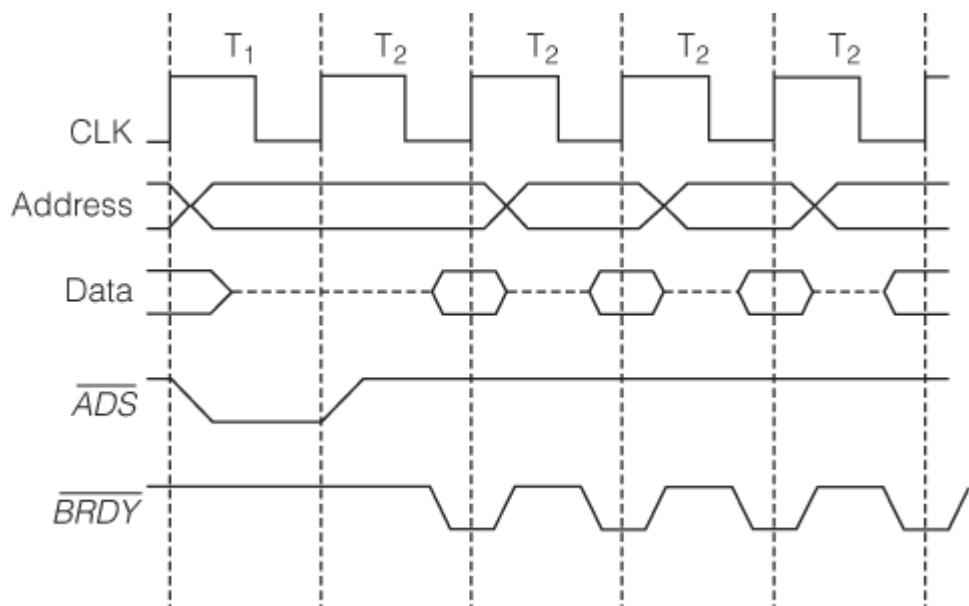


Figure 14.29 | Burst mode data transfer

A burst cycle transfers multiple bytes of data across the bus during one long memory cycle. For example, a transfer of a 128-bit data item across a 32-bit bus would normally occur in four groups, each group containing one 32-bit data. But in the burst mode, the initial address (of the first byte) is used by the processor to compute the remaining addresses for the subsequent data bytes. There are various ways of accomplishing this, but to understand it, think of it this way – a sequential burst order from address zero would first transfer the 32-bit data residing at address zero. The next transfers would be to the data at addresses 4, 8 and 12, in order.

Figure 14.29 shows a burst cycle which reads four 32-bit words (a total of 128 bytes) in five clocking periods. To start the process, the processor sends out the first address and asserts the BLAST signal (not shown in the figure) high. The *BRDY* (Burst ready) signal is asserted in the first T_2 . The processor reads the data word in two cycles and outputs the next address. Since data are at successive addresses, only the lower address bits need to be changed. When the processor has read the required number of data words, it asserts the BLAST signal low to terminate the burst mode. Such a burst read cycle is referred to as a 2-1-1-1 read where four data reads are accomplished in 5 clock cycles.

6. The processor has a major addition in terms of four pins (one for each memory bank) for parity, which is generated by it, during each write cycle. Parity is generated as even parity and a parity bit is provided for each byte of memory. The parity check bits appear on pins DP0–DP3, which are parity inputs as well as parity outputs. These are typically stored in memory during each write cycle and read from memory during each read cycle. On a read, the microprocessor checks parity and generates a parity check error if it occurs on the PCHK pin. A parity error causes no change in processing unless the user applies the PCHK signal to an interrupt input. Figure 14.30 shows the memory banks of 80486 with the bank enable signals (*BE*'s) and the parity check bit for each channel.

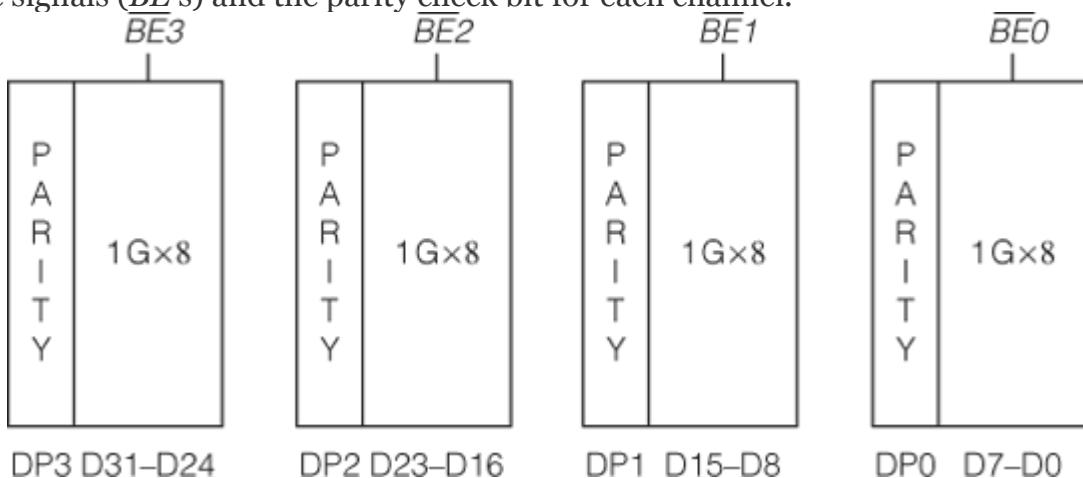


Figure 14.30 | Memory banks of 80486

7. Another enhancement, which needs mention, is the addition of a few new instructions which are listed in Table 14.4.

Table 14.4 | New instructions in the 80486

Mnemonic	Function performed
BSWAP	Byte Swap
XADD	Exchange and add
CMPXCHG	Compare and Exchange
INVD	Invalidate cache
INVLPG	Invalidate TLB entry
WBINVD	Write back and invalidate cache

Of these, the last three instructions relate to cache operations. The first instruction BSWAP is very useful for systems which use big endian format instead of the little endian format used by Intel. Changing from one format to the other just involves the switching of the bytes of a word.

Now, before going ahead to the next topic, let us try to understand the meaning and implications of an issue that needs careful consideration.

14.15 | Data Alignment

In Section 7.3 the idea of memory banks was presented, and the importance of using an even address when accessing 16-bit data was explained. It was made clear that when the memory address is even, memory access takes only one bus cycle, while an odd addressed word incurs a penalty of one more bus cycle. That was for the case of the 8086 where the maximum data width was 16 bits. However, for '386 onwards, the bus width is 32 bits and data is divided into 4 memory banks. For Pentium, the external data bus width is 64 bits. In all these cases, a bus penalty will occur if the memory address is not 'aligned'. A performance penalty occurs due to misaligned data. What is meant by data alignment?

2-byte Data

A 16-bit data item is aligned if it is stored at an address that is a multiple of two. This implies that the least significant bit of the address must be 0. This is the case that we have discussed for the 8086, where the LSB of the address being 0, which means that the address is an even number.

4-byte Data

Let us think of it this way – if all the 4 bytes are in one row, they can be accessed in one cycle, otherwise one more cycle is required. This means that a 32-bit data item is aligned if it is stored at an address that is a multiple of 4. This implies that the least significant two bits of the address must be zero.

8-byte Data

A 64-bit data item is aligned if it is stored at an address that is a multiple of 8. That means that the least significant three bits of the address must be 0. This alignment is important for processors such as Pentium that have a 64-bit data bus.

14.16 | The Pentium Processor

Now, we will come to Intel's most popular name in processors, i.e., Pentium. In 1993, Intel developed its fifth generation x86 processor and decided to call it 80586 (according to the convention followed till then), but to keep away competitors such as AMD and a few others, Intel wanted to trade mark this processor title. However, the lawsuit that followed, declared that numbers cannot be trademarked, and hence, Intel had to find a new name for its new processor, and that is how the word Pentium came into being. It is also called P5. The name Pentium was derived from the Greek word *pente*, meaning 'five', and the Latin ending -ium. Vinod Dham, an engineer from Pune who worked in Intel, is frequently hailed as 'the father of Pentium'.

The Pentium family of processors, which has its roots in the Intel 486 processor, uses the Intel 486 instruction set (with a few additional instructions). The term 'Pentium processor' refers to a family of microprocessors that share a common architecture and instruction set, but each new version of Pentium has newer and newer features. The first Pentium processor was released in the year 1993 and it had a lot of new features compared to its predecessor, i.e., 80486. The foremost enhancement is that it has a 64-bit data bus, while continuing to be a 32-bit processor internally. See the list below.

Features of the Pentium Processor family:

- 32-bit Microprocessor (internal registers of 32 bits)
 - 32-bit address bus
 - 64-bit data bus

- Superscalar architecture
 - Two pipelined integer units
 - Capable of ‘less than’ one clock per instruction
 - Pipelined floating point unit
- Separate code and data caches
 - 8 K code cache, 8 K write back data
- Advanced design features
 - Branch prediction

Now, let us go into the details of each of these enhancements.

14.16.1 | Data Bus of 64 Bits

The data bus width has been doubled. What is the advantage of that? It is that data can be received as 64 bits in one go. What can be done with 64-bit data when the processor registers are only 32 bits in width? The answer is that since the bus bandwidth has doubled, double the amount of data can be sent/received from/to the memory.

See Fig. 14.31. There are 8 memory banks activated by eight *BE* signals. From this memory bank array, data can be accessed for internal processing as 8, 16 or 32 bits. However, sending or receiving data outside the chip can be at the rate of 64 bits per memory cycle. This is a major enhancement.

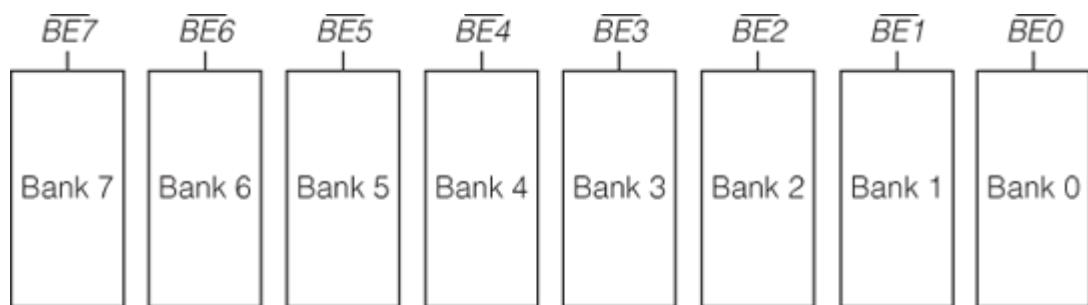


Figure 14.31 | Memory banks of Pentium

14.16.2 | Superscalar Architecture

Pentium is a ‘superscalar processor’, which is pipelined. Pipelining is still only of 5 stages. Being superscalar means it has more than one executing unit – here, there are two integer execution units (each of them is pipelined). They are called the U and V pipes – both these units are slightly different in their processing power, which means that one unit can handle more complex operations than the other. It is the U pipe which can handle more complex operations. See Fig. 14.32, which shows the execution units of the Pentium and the ’486. Note the differences.

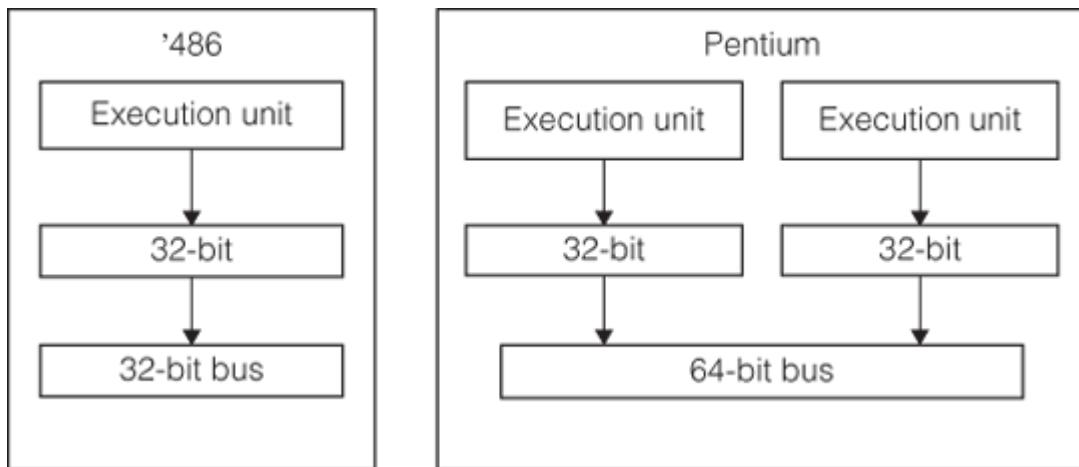


Figure 14.32 | Execution units of pentium and the '486 compared

14.16.3 | Faster Floating Point Unit

The floating-point unit has been completely redesigned over the Intel '486 CPU. Faster operations provide up to 10x speed for common operations including add, multiply and load. The FPU has an eight stage pipeline to speed up operations. Figure 14.33 shows that the units responsible for fetch and decode are common – once decoding is over, it goes to the execution phase. In this phase, the execution occurs in either the U pipe, V pipe or the floating point unit, depending on the data type and complexity of the required operation. Once this is done, the result is written into the data cache.

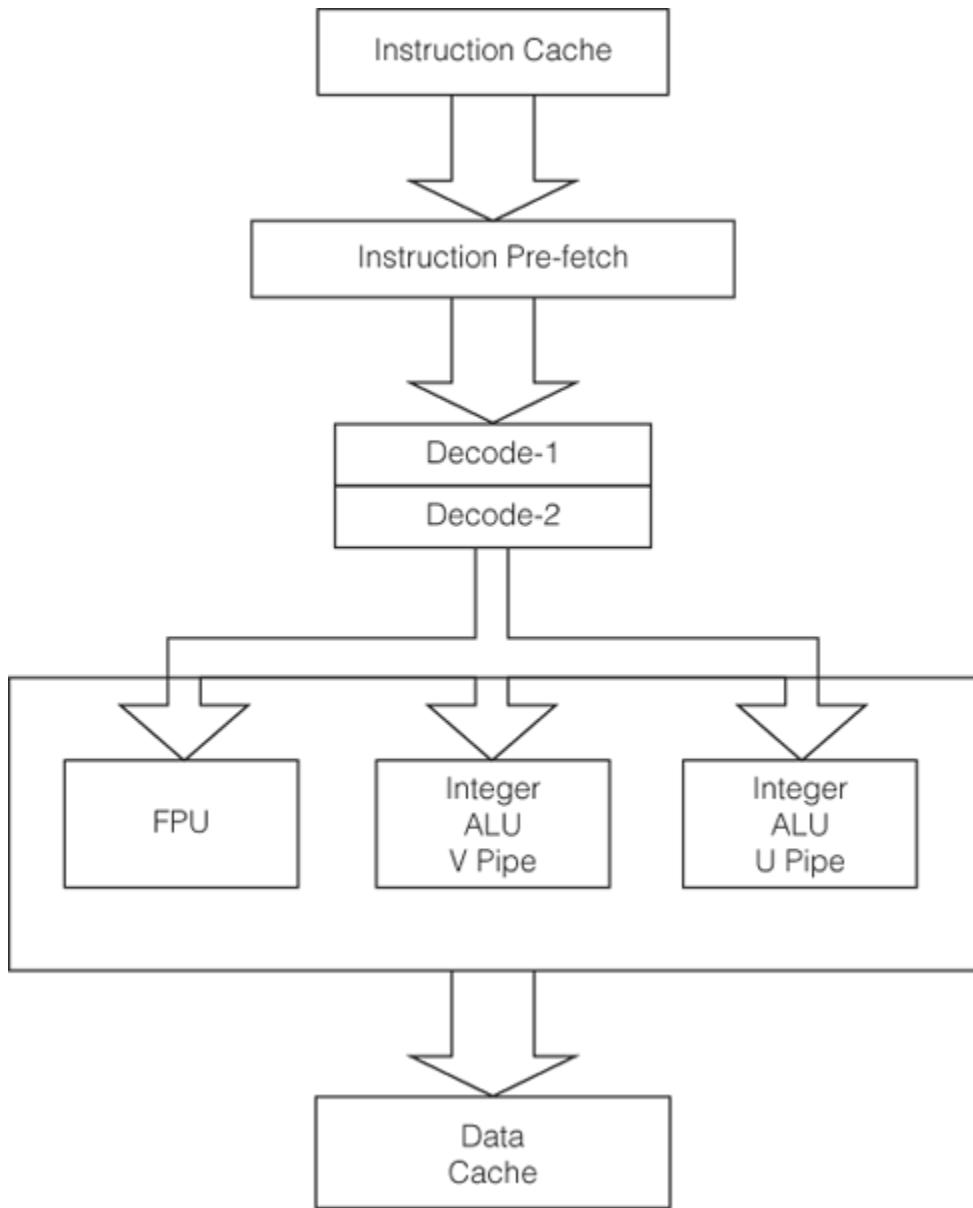


Figure 14.33 | Execution unit of Pentium

14.16.4 | Separate Data and Instruction Cache

Figure 14.33 also shows that there are separate instruction and data caches. Pentium has two separate on-chip caches, one for data and one for code. Each cache is 8 KB in size with a 32-byte line size and is 2-way set associative. Each cache has a dedicated Translation Lookaside Buffer (TLB) to translate linear addresses to physical addresses. The code (instruction) cache is an inherently write-protected cache. The caches can be enabled or disabled by software or hardware.

14.16.5 | Data Dependency Problem

Increasing the number of execution units brings with it a new set of problems. Programs are written by a programmer for serial execution. However, when attempting to run this program on two parallel execution units, the problem of ‘data dependency’ can arise for certain program sequences. For example, if the first program line asks for two numbers to be added and the second line needs the result of the first instruction for its completion, obviously these two instructions cannot be executed in parallel. In such cases, reordering or rescheduling of instructions has to be done, and at present, this problem has been handed over to the compiler to resolve.

14.16.6 | Branch Prediction Unit

Another new feature that Pentium has is the branch prediction unit. This is useful in preventing ‘pipeline stalling’ in the case of a branch instruction occurring in the sequence.

Let us take a closer look at what is called the ‘branch unit’. This unit includes the ‘branch execution unit’ and the ‘branch prediction unit’. Think of what should happen if a conditional branch instruction is encountered. First, the condition is to be evaluated, then the branch address calculated and then the code should be fetched from the new branch address and instruction execution should proceed. All these steps involve a delay. To avoid the delay, a technique called ‘speculative execution’ is implemented. In this, a calculated risk is taken and one particular branch is taken for continuing the instruction execution in sequence. In the meanwhile, the branch condition is evaluated. If the ‘speculative execution’ had taken the correct branch, all is well and good. Otherwise, the pipeline will have to be flushed and everything re-done. This is indeed a minor catastrophe considering that the delay so incurred causes a performance penalty. Thus, it is imperative that the branch prediction unit does a good job.

There are two methods of branch prediction—static and dynamic. In the first case, the assumption is that most branch conditions pertain to whether to repeat a loop or not. In most cases, a loop’s exit condition is false; hence, repeating the loop is something that is normally assumed to be necessary. This type of static prediction is simple and fast, but its success rate is good only in cases of loop intensive programs.

The other technique, i.e., dynamic prediction, relies on using the past history of branching, for the program. There is a ‘Branch History Table’ as well as a ‘Branch Target Buffer’. The latter stores the target addresses of previously executed branches, so that speculative execution can use

any of these target addresses when branching has to occur. This assumes that previously used target addresses are likely to be needed again.

Pentium uses both static and dynamic branch prediction and it is found that a success rate of around 75 to 85% is possible.

14.16.7 | Page Size

Remember that the '386 paging unit supported pages with a maximum size of 4 K only. This is so for 80486 as well. The Pentium processors' Memory Management Unit contains optional extensions to the architecture which allow 2 MB and 4 MB page sizes.

14.16.8 | System Management Mode

The first Pentium was released in 1993, and a newer version was released in 1994, which had a new mode of operation besides the real, virtual and protected modes. This mode is the System Management mode, which is an additional feature for efficient power management and to run specific proprietary code.

In this mode, the power dissipation is reduced by switching off all peripherals and even the system when it is not in use. This is not the only feature of this mode, which operates as a high level manager, and has also extra features for enhanced security.

KEY POINTS OF THIS CHAPTER

- The 80286 had many improved features compared to the 8086 and fuelled the development of the PC-AT and the first Windows based operating system – Windows 3.1.
- The 80386 was the next processor of Intel in which the major change is the increase in the number of address and data pins to 32.
- This processor can address 4 GB of physical memory.
- It has a number of new instructions, and, also enhancements to some existing instructions.
- The general-purpose registers of 80386 are 32 bits long.
- It has a number of additional pins compared to its predecessors.
- The idea of virtual memory is of having an enlarged memory space, which is mapped to a smaller physical memory.
- The memory management unit of the 80386 has the capability of paging and segmentation.
- Descriptors are 8-byte entities in which the information corresponding to a segment are stored.

- Selectors are 16 bits long and point to the descriptor it is associated with.
- The paging unit of the MMU can be switched off but not the segmentation unit.
- Protection has many aspects, including protecting tasks from each other, and also the necessity of protecting system level tasks from user tasks.
- The concept of privilege levels is used for prescribing the level of trust, and the level of protection for each segment.
- Call gates are used for accessing code segments at a higher privilege level.
- The 80386 is a multitasking CPU meaning that it can take care of multiple tasks.
- The tasks are not executed concurrently; the CPU time is allotted to different tasks.
- The interrupt vector table is replaced by an Interrupt descriptor table while in the protected mode.
- The 80486 processor has a number of enhanced features compared to its predecessor 80386.
- It has the burst mode of data transfer which increases data transfer speed significantly.
- It has level one cache placed on the chip which was a big step in microprocessor history.
- Intel's first Pentium processor was released in 1993.
- One important enhancement is that it has a 64-bit data bus, while the processor is internally 32-bit.
- It has a superscalar architecture which means that it has two integer execution units.
- It has a faster floating point unit.
- Its instruction and data caches are separate.
- It can have a page size of 2 MB/4 MB.

QUESTIONS

1. What were the major changes incorporated in the 80286 compared to the 8086?
2. What was the major complaint against the 80286?
3. Name the major change that made the 80386 a big leap in processor technology.
4. What is the 'scaled indexed 'mode of addressing?
5. Where can the MOVSX and MOVZX instructions be used?
6. Why is it that the '386 has address lines only from A2 to A31?
7. Does the 80386 have an inbuilt floating point processor?
8. What is the importance of the virtual memory concept for a user?
9. Why is address translation in the PVAM so complex?
10. What does a descriptor mean to a segment?
11. What is the role of a selector in address translation?

12. How many LDTs and GDTs does a system have?
13. Why do we say that a segment with a CPL = 0 is a highly trusted segment?
14. What is the role of RPL in allowing access to a segment?
15. Distinguish between a conforming and a non-conforming segment.
16. Under what condition is a call gate used?
17. What is the main issue to be taken care of in task switching?
18. What does the task register contain?
19. What is meant by the term 'back link to the previous TSS'?
20. What is an IDT? How many IDTs can a system have?
21. What is the difference between 80386 DX and SX?
22. What is L1 cache and how is it different from L2 cache?
23. What is overclocking? What are the possible pitfalls in this idea?
24. How does a burst access cycle of memory contribute to memory speed?
25. What is the penalty incurred in data misalignment?
26. What advantage is obtained by having a 64-bit data bus for a 32-bit processor?
27. What do the following words mean in the context of a Pentium processor?
 1. superscalar architecture
 2. branch prediction
 3. code cache and instruction cache
 4. pipeline stalling

EXERCISES

1. Write a program that uses MOVSX and MOVZX instructions to
 1. add an 8-bit negative number to a sixteen bit negative number,
 2. add a 32-bit negative number to a 16-bit positive number.
2. Write a program to test bits 4, 8, 9 of a 16-bit number in memory.
3. Add the contents of a table of size 27 containing 16-bit numbers using the scaled indexed mode of addressing.
4. Find the difference between the processors 80386DX and 80386SX.
5. In virtual 8086 mode, can the BIOS and DOS instructions be used by 8086 based programs? How?

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)

14

Advanced Processors—80286 to Pentium

IN THIS CHAPTER, YOU WILL LEARN

- The impact that 80286 made to the realm of PCs.
- The major changes that the 80386 architecture offered.
- The programming enhancements of 80386.
- The features of PVAM and the concept of virtual memory.
- The mechanism of address translation in the protected mode.
- The importance and implementation of protection mechanisms in the 80386.
- The issues and methods of task switching.
- The use of the interrupt descriptor table and exceptions of 80386.
- The features of 80486 which are an enhanced set of the '386 features.
- The idea of burst mode data transfers for both 80486 and Pentium.
- Why data alignment is important for multibyte processors.
- The features of Pentium.
- The cache structure of Pentium.

Introduction

In this chapter, we will discuss the 80286, 80386 and 80486 processors, as well as pentium. The first 32-bit processor of Intel, i.e., the 80386 is discussed with the aim of cracking the intricacies of a complex processor. Once the '386 is understood well, the '486 and Pentium have nothing more to it than a few 'enhancements'. This means that the internal architecture of the '486 and Pentium is more or less the same as that of the '386. They have the same memory management unit, protection mechanisms and multitasking units. Thus, a discussion on these features is redundant and we will concentrate on the additional features of the 80486 and Pentium, in addition to that of 80386.

14.1 | The 80286 Processor

You know that the first Intel processor used in a personal computer by IBM was the 8088. The PC based on 8088 and manufactured by IBM came to be known as IBM PC. It had only a floppy disk drive internally but could have an external hard disk. All PCs with similar hardware and

standards came to be called IBM PC clones. Intel later released the PC-XT (extended technology), which included a standard hard disk internally.

The next important processor of Intel was 80186. However, we have seen in the previous section that the 80186 has never been used as a processor in a PC but instead found applications in the field of embedded systems.

Soon after that, in 1982, Intel released the 80286, which came to be used in PCs that came to be called PC-AT where AT stands for ‘advanced technology’. This turned out to be a big leap for Intel and for PCs as well. The PC-AT used 80286 as the processor and its technology could run all the software written for its predecessor. It could be operated at frequencies of 6 MHz, 8 MHz, 10 MHz and 12.5 MHz. The 80286 turned out to be a big improvement over the 8086/8088. It could boast of a performance double that of the 8086. It had a number of features that were entirely new to the realm of processors.

The major enhancements can be listed as:

1. The data bus remained the same at 16 bits (internally and externally), but there was no need for the multiplexed address/data bus as separate pins were made available for them. The pin count increased so that the packaging could no longer be a Dual in line package like the 8086. Instead, the packaging was a pin grade array (PGA).
2. The address bus was increased to 24 bits, which could address 2^{24} bytes or 16 Mbytes of physical memory.
3. The memory cycle time was reduced from 4 cycles to 2 cycles, which made a direct improvement to the speed factor. Calculation of the more complex addressing modes (such as base + index) had less clock penalty because it was performed by a special circuit. The 8086, its predecessor, had to perform effective address calculation in the general ALU, taking many cycles. Also, complex mathematical operations (such as MUL / DIV) took fewer clock cycles compared to the 8086.
4. The biggest change and enhancement, however, was in the form of the introduction of what was called the ‘Protected Virtual Addressing Mode (PVAM)’. Let us try to understand what exactly this is. For the 8086, there is no ‘protection’ provided between users or between system software and application software. MS-DOS allowed access to anyone to any programs or any hardware through the mechanism of interrupts. Also, there was no facility to use the processor in a multi-tasking environment. 80286 reversed all this – it provided new mechanisms and instructions which provided protection and multi-tasking. Also, it could use the concept of ‘virtual

memory' whereby the memory as seen by the user becomes much larger than the actual physical memory present. This is the essence of PVAM.

When the processor is powered on, it is in the 'real mode' where it behaves as an accelerated 8086 – it then has only 20 active address lines and 1 MB of physical memory. From this mode, a special instruction switches it into the protected mode where the active address lines become 24 in number, the physical memory becomes 16 MB and all the protection and virtual memory mechanisms become activated. With all these enhancements and changes, this processor was popular for a short while, but one complaint against it was that it could not be switched back to real mode without resetting it.

But it was this processor that made the real change in the way a computer could be designed and used. The concepts in the protected mode allowed the first GUI (graphic user interface) based operating system of Microsoft, i.e., windows 3.1 to be designed comfortably for a PC. The design of the 80286 was followed up in Intel leading to a better and bigger (in terms of data and address size) processor named as the 80386 (1985). All the features of the 80286 were used here, and more features were added as well – we can say that the features of 80286 are a subset of the features of that of the 80386. Thus, it will be to our advantage to go into the details of the architecture of the 80386, which will give us all the wisdom necessary to understand the '486 and Pentium which followed very soon.

14.2 | The 80386

In 1985, Intel released the 80386 processor which carried in it all the new features it had introduced in its previous processor and also made one great leap – that was the doubling of the data bus width to 32 bits and the increase in the address bus width to 32 bits. Thus, the data band width doubled, and the addressable physical memory became 2^{32} , i.e., 4 GB. Thus, 80386 was Intel's first 32-bit processor. Intel also manufactured a model called 80386 SX, which was a 32-bit processor internally but had an external data bus width of 16 bits. However, our discussion will be on the 80386 DX, which is a 32-bit processor internally and externally.

When powered on, the '386 is in the real mode with 20 address lines and capability to access only 1 MB of physical memory. From this state, it can be taken to the protected mode using a special instruction, the details of which we will soon come to.

14.2.1 | 80386 Enhancements

The 80386 has all the features of 80286 – additional instructions, protected virtual addressing mode – plus a lot more. It has more instructions, higher memory addressing capability, double the data bus width and address bus and also includes the use of memory as ‘pages’ in addition to segmentation. Let us list out the new features.

- The external data bus is now 32 bits in width.
- The registers and, hence, the internal data bus width is 32 bits in width.
- The address bus is 32 bits in size, and hence, the memory addressing capability is 2^{32} or 4 GB.
- All addresses are 32 bits long, and hence, the address registers are also 32 bits long.
- A new addressing mode called the scaled addressing mode is available.
- A set of bit manipulation instructions is available.
- The ’386 design overcame the biggest complaint against its predecessor – this processor can switch from protected mode to real mode without a hardware reset.
- The processor has all the features of PVAM that its predecessor had – as well as an additional ‘paging’ mode.

Let us go into the details of this processor – first the internal architecture and then the hardware features.

14.3 | Internal Architecture

14.3.1 | Registers

Let us first view the working registers of the processor – the ones used for computation, address calculation and the like (see Fig. 14.1). The processor has 16 registers, which consist of scratchpad (general-purpose) registers, segment registers, flag register and instruction pointer.

Comparing it with 8086 shows that the 16-bit registers AX to DX have been re-designated as EAX to EDX, making them 32-bit registers. These registers can be used as 8 bits or 16 bits as well, but only the lower portions of the registers can be used, i.e., the register EAX can be used as EAX (32-bit), AX (16-bit), AH (8-bit) and AL (8-bit). Similar is the case for the registers EBX, ECX and EDX.

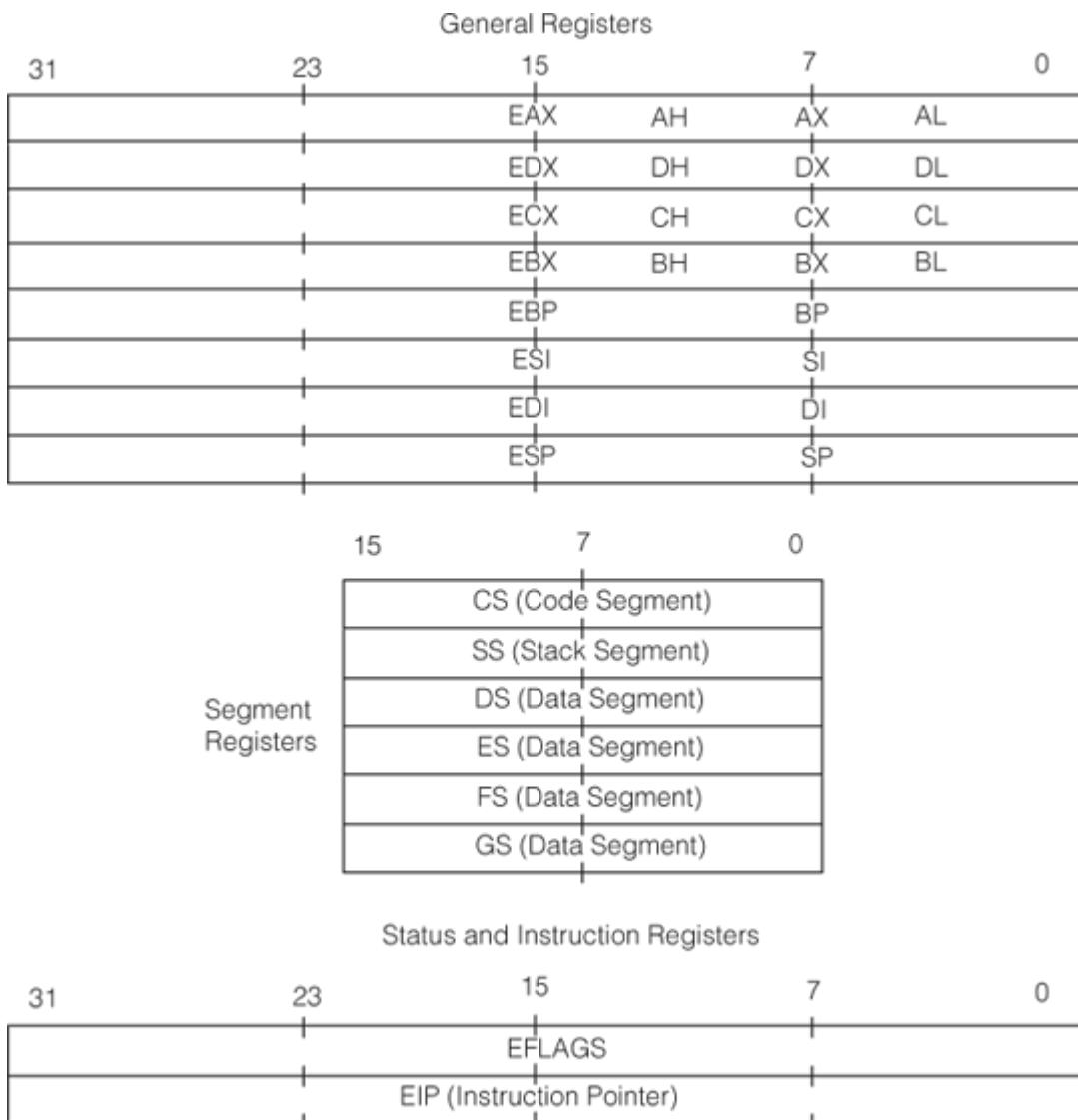


Figure 14.1 | General registers and address registers used in the real mode

The registers EBP, ESI, EDI and ESP are 32-bit address registers as all addresses are 32 bits in length. The instruction pointer EIP is 32 bits long as each memory address is 32 bits in size. The same is the case of the stack pointer, which is now designated as ESP.

14.3.1.1 | Segment Registers

The segment registers are 16-bit, but we will see later that they have what is called a ‘hidden part’, which will then make them totally 48 bits long (this becomes effective only in the protected mode). The visible part of the segment registers are still 16 bits and you can see that there are two new segment registers FS and GS. They are used to store the base addresses of the GS and FS segments, which are data segments itself.

14.3.1.2 | Control and Status Registers

The flag register, i.e., now EFLAG, contains the values of the status flags (i.e., C and Z), the control flags (i.e., DF, IF and TF) plus new flag bits catering to the additional features of the processor. The flag register has all the flag bits that the 8086 has – the additional bits are for use in the protected mode. Figure 14.2 shows the flag register bits.

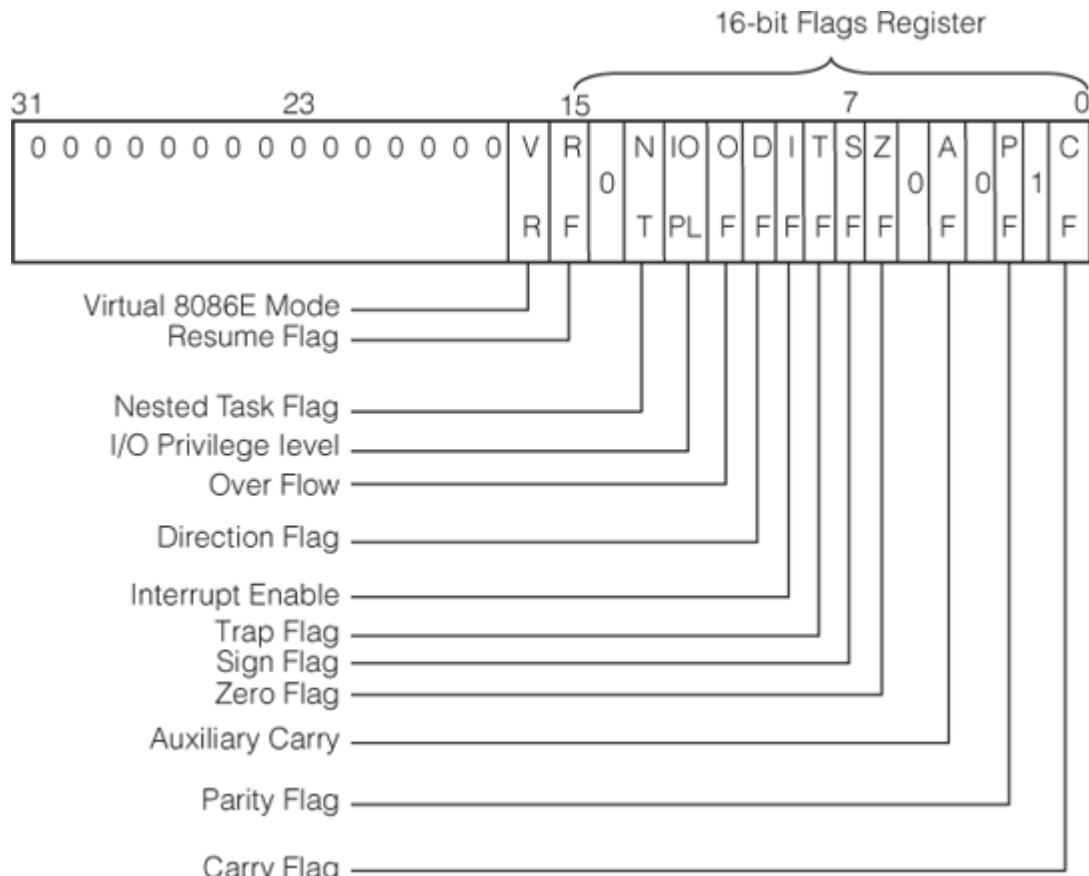


Figure 14.2 | Flag register of the 80386

14.4 | Programming Enhancements

The processor has a number of new instructions and some enhancements with respect to the instructions available for its predecessor. It has all the instructions that the 8086, '186 and '286 has, plus a few more. Here we will see the new instructions that can be used in the **real mode**.

1. The '386 has an interesting enhancement with regard to what we have until now defined as 'pointer registers' which carry addresses. **Here all the general-purpose registers too can hold addresses that can be used in the register indirect, register relative and based indexed mode of addressing.** For example, the following instructions are allowed.

```
MOV EAX, [EBX],  
MOV CX, WORD PTR [EAX]  
MOV AL, BYTE PTR [ECX]
```

2. A new and powerful mode of addressing called scaled index addressing mode has been introduced for the 80386. In this mode, each of the 32-bit registers (except the ESP) can be used as a pointer that is multiplied by a scale factor.

Now, let us do a few programming exercises using the new instructions of 80386.

Note 1 This processor has 32-bit registers, and hence, all the programming for this will be done in MASM32, which is the 32-bit Windows based assembler. The installation and use of this assembler is described in Appendix H available online at www.pearsoned.co.in/lylabdas.

Example 14.1

Find the effective address of the operand for the following instructions, given ECX = 233411E2H, EDI = 10008967H and EAX = 78960900H

1. MOV [ECX*2], AX
2. MOV EBX, [2341H + EDI*4]
3. MOV BL, [EAX*8]

Solution

Remember that the ‘effective address’ calculation does not need the base address of the segment. It gives the offset of the operand from the base address.

1. The effective address is obtained by multiplying the value in ECX by 2, which gives 466823C4H. The word from AX is moved to this address (offset).
2. The effective address is obtained by multiplying the value in EDI by 4 and adding 2341H, i.e., 4002259CH + 2341H = 400248DDH. The double word in this address is moved to EBX.
3. The effective address is obtained by multiplying the value in EAX by 8, which is 3C4B04800H. The byte in this address is moved to BL.

Example 14.2

Add a set of 32-bit words stored in memory assuming that the result will not exceed 64 bits.

Note 2 This program has been run in MASM 32 and results have been observed using the Olly debugger, the details of which are available in Appendix H.

```
.386
.MODEL FLAT, STDCALL
.CODE
DATS DD 00123456H, 00EF3456H, 07860EACFH,
      0A45674FH, 08976ABCH

STRT: MOV EAX, 0          ;EAX = 0
       MOV EBX, 0          ;EBX = 0
       MOV ECX, 00000005    ;ECX = 5, the count
AGN:  ADD EAX, [DATS + EBX*4] ;add to EAX the contents of memory
       INC EBX             ;increment the pointer
       LOOP AGN            ;repeat until ECX = 0
END STRT
```

Here, after adding the five 32-bit data stored in locations starting from DATS, the sum is found to be 8C3F 2586H, which is in EAX. Note that for the instruction ADD EAX, [DATS + EBX*4], the scale factor used is 4. This is because the data is 4 bytes long, and therefore, the address of each data is displaced from the next data by 4.

If the problem was one of adding words (2 bytes), we would have used a scale factor of 2. Thus, the scaled addressing mode is useful in adding tables stored in memory, when the content of the tables are either 2 or 4 bytes long.

14.4.1 | New Instructions of 80386

Bit Test and Modify Instructions

This group of instructions operates on a single bit that can be in memory or in a general register. The location of the bit is specified as an offset from the low-order end of the operand. The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register. These instructions first assign the value of the selected bit to CF, the carry flag. Then, a new value is assigned to the selected bit as determined by the operation. OF, SF, ZF, AF, PF are left in an undefined state. Table 14.1 defines these instructions.

Example 14.3

```

.386
.MODEL FLAT, STDCALL
.CODE
DATS DW 0560H

STRT: LEA ESI, DATS      ;let ESI point to the data
      MOV AX, [ESI]       ;AX = 0560H
      BTC AX, 1           ;AX = 0562H, CF = 0
      BTR AX, 1           ;AX = 0560H, CF = 1
      BTS AX, 0EH         ;AX = 4560H, CF = 0
      END STRT

```

Table 14.1 | Bit Test and Modify Instructions

Mnemonic	Instruction	Carry flag after execution	Effect on the selected bit after execution
BIT	Bit test	CF = bit	None
BTS	Bit test and set	CF = bit	BIT = 1
BTR	Bit test and reset	CF = bit	BIT = 0
BTC	Bit test and complement	CF = bit	BIT = not BIT

The effect of each of the operations is shown in the comment field.

14.4.2 | Bit Scan Instructions

These instructions scan a word or doubleword for a ‘1’-bit and store the index of the first set bit into a register. The bit string being scanned may be either in a register or in memory. The ZF flag is set if the entire word is zero (no set bits are found); ZF is cleared if a ‘1’-bit is found. If no set bit is found, the value of the destination register is undefined.

BSF (Bit Scan Forward)

This instruction scans from low-order to high-order (starting from bit index zero)

BSR (Bit Scan Reverse)

This instruction scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

Example 14.4

```

.386
.MODEL FLAT, STDCALL
.CODE

DATS      DW 0560H

STRT:    LEA ESI, DATS      ;let ESI point to the data
        MOV AX, [ESI]      ;AX = 0560H
        BSR BX, AX        ;BX = 000AH meaning that D10 = 1
        BSF CX, AX        ;CX = 0005 meaning that D5 = 1
END STRT

```

14.4.3 | MOVSX and MOVZX Instructions

Two other new and useful instructions are the MOVSX and MOVZX instructions. In MOVSX, the sign bit of a register (or memory) can be extended to any register. MOVZX zero extends the contents of a register or memory location. The MOVSX instruction is used for signed arithmetic and MOVZX for unsigned arithmetic.

Example 14.5

```

.386
.MODEL FLAT, STDCALL
.CODE
        DAT1 DW -269
        DAT2 DB 56
        DAT3 DB - 123

STRT:    MOV AX, DAT1      ;AX = FEF3H
        MOVSX ECX, AX      ;ECX = FFFFFFFE3H
        MOVSX EBX, DAT2      ;EBX = 000000038H
        MOVZX ECX, DAT1      ;ECX = 0000FEF3H
        MOVSX AX, DAT3      ;AX = FF85H
END STRT

```

In Example 14.5, the numbers are expressed in the decimal format. In the hex format, they are

DAT1 = FEF3H
 DAT2 = 38H
 DAT3 = 85H

Note The first number is stored as a word, and the next two are stored as bytes. The result of each of the operations is shown in the comment field of the program.

14.5 | Hardware Features of 80386

The 80386 has 132 pins and is packaged as a pin grade array (PGA). Figure 14.3 shows the processor with the functional grouping of pins. The pin numbers are not mentioned – only the pin functionality is discussed. Only new pins which are not present in 8086 will be discussed. The pins that are not discussed may be assumed to have the same functionality at they have in the 8086.

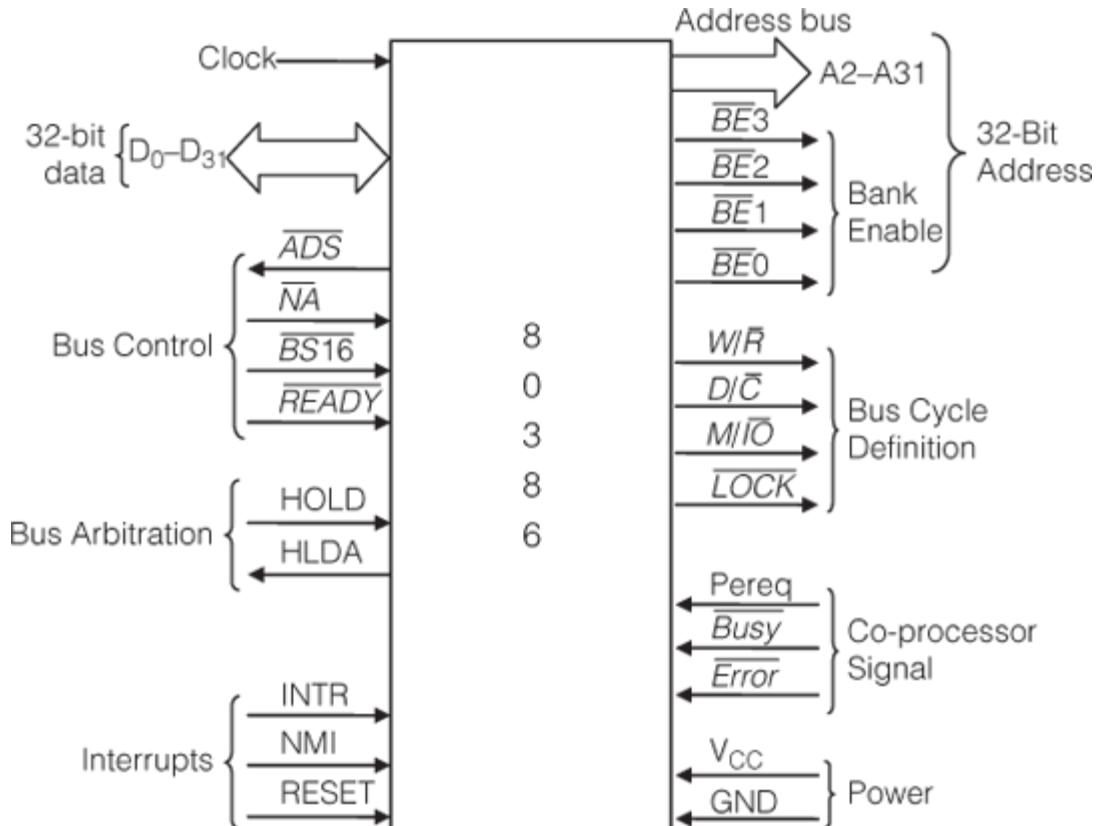


Figure 14.3 | Functional pin diagram of the 80386

Pin Grade Array

A pin grid array (PGA) refers to the arrangement of pins on the integrated circuit packaging. In a PGA, the pins are arranged in a square array that may or may not cover the bottom of the package. PGAs are primarily used in applications that require more pins than what older packages such as the dual-in-line package (DIP) provide. Figure 14.4 shows a PGA.

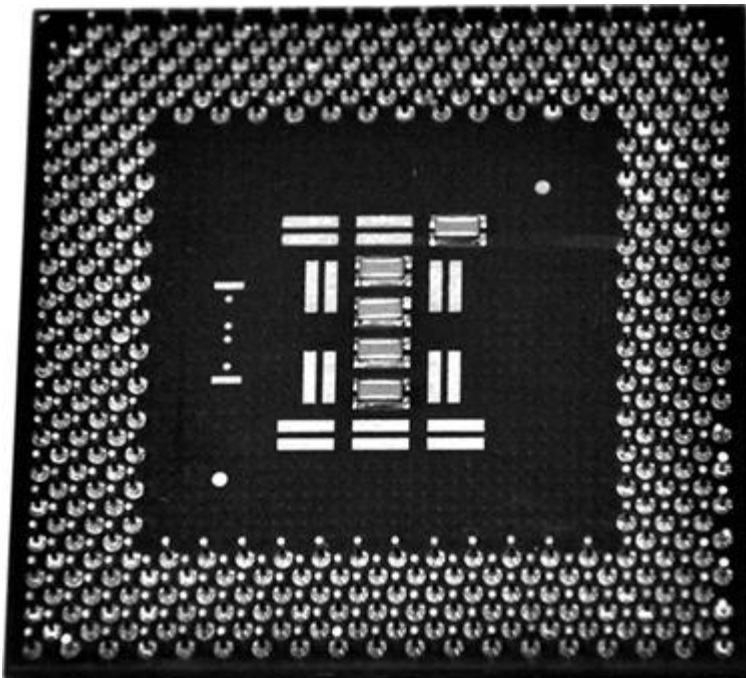


Figure 14.4 | A pin grade array (PGA)

Data Bus

The data bus is 32-bit wide and is designated as D_0 to D_{31} . It is not multiplexed, as in the 8086, and carries only data. Since data can be used as 8, 16 and 32 bits, the data bus is grouped into four 8-bit combinations, and access of each of these combinations is selected by a separate enable signals for the memory/ IO banks of the system.

Address Bus

The address of the processor is 32 bits wide, but the dedicated address lines are only A_2 to A_{31} , i.e., 30 lines. The address lines A_0 to A_1 are absent but they are encoded as four bus enable signals BE_0 to BE_3 (active low), which separately enable four memory banks. Each memory bank has a data bus width of 8 bits and data in the banks can be accessed by selective activation of the banks. See Fig. 14.5.

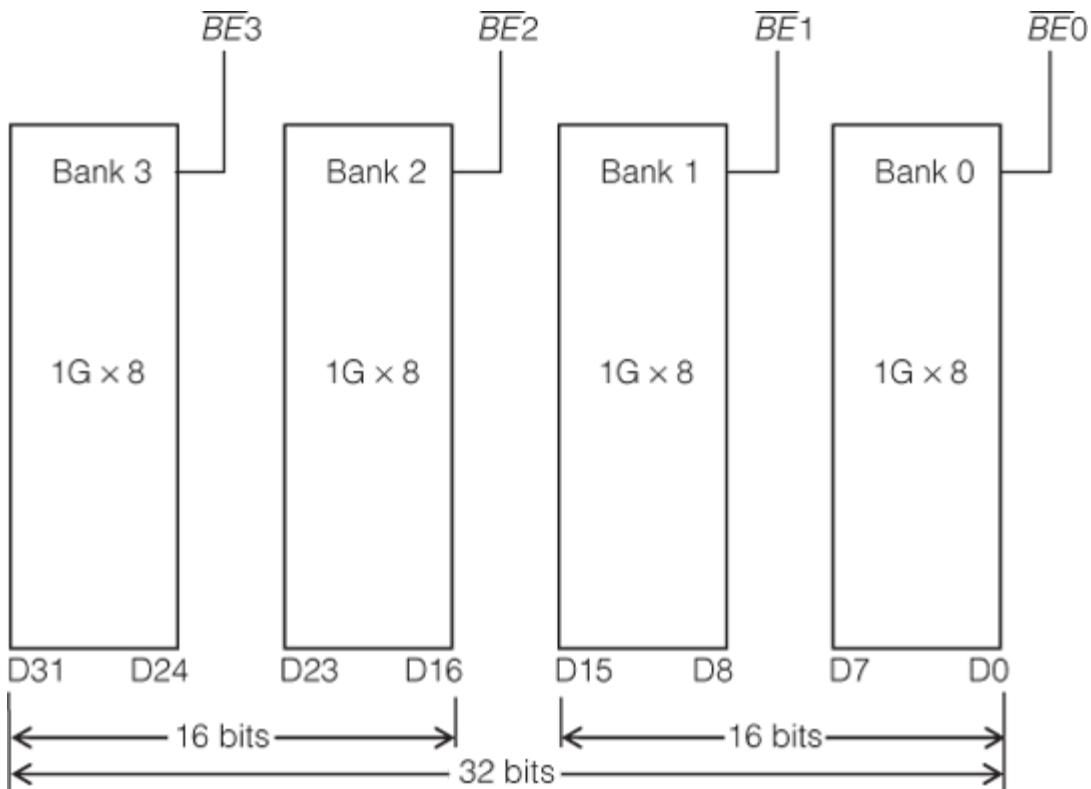


Figure 14.5 | Memory banks for the 80386

There are four bank enable signals BE_0 to BE_3 . They are selectively enabled depending on the banks that contain the data byte, word or double word. If a double word has to be read, all four bank enable signals are to be activated.

Example 14.6

For accessing a word, given the physical address as FFFF FFFCCH, which banks have to be enabled and what should be the status of the bus enable signals?

Solution

To read a word, two bytes have to be obtained, i.e., two banks have to be enabled. The specified address and the next address will be activated.

FFFF FFFCCH and FFFF FFFDH. These addresses are in Bank 0 and Bank1. Hence, the corresponding bank enable signals have to be made low. Thus, BE_0 and BE_1 have to be low and BE_2 and BE_3 are to be high (addresses with the lowest two bits 00 will be of bank 0, with 01 in bank 1, 10 in bank 2 and 11 in bank 3).

In this set, only the D/ C signal is new.

D/C (Data/Control)

This signal being high indicates that the data bus contains data for memory or I/O. The low status of this pin indicates that the processor is halted or is executing an interrupt acknowledge cycle.

Bus Control

These signals are used for efficient bus control circuitry.

ADS: Address Data Strobe: This signal becomes active whenever the processor issues a valid memory or I/O address.

NA: Next address: This signal causes the processor to output the address of the next instruction or data in the current bus cycle itself. This is used for address pipelining.

B16: Bus size 16 selects either a 32-bit data bus ($B16 = 1$) or a 16-bit data bus ($B16 = 0$).

Coprocessor Pins

These are pins receiving signals from the coprocessor, which is external to the 80386. The 80387 is a compatible coprocessor operating similar to the 8087 (arithmetic coprocessor) discussed in chapter 13.

PEREQ: Co-processor request is a request from a coprocessor to the 80386 to relinquish control of the bus so that the 80387 coprocessor gets a direct connection.

BUSY: This is a direct connection between the 80386 and the coprocessor, which indicates that the coprocessor is busy. This signal is used by the WAIT instruction to wait until the co-processor finishes its task.

ERROR: This is a signal from the coprocessor that an error has been detected.

Clock

This provides the timing for the processor. The system frequency is half the frequency of the clock connected to this pin. For the system frequency to be 33 MHz, the frequency to be given to the pin should be

66 MHz. There is no internal clock generator – so an external clock chip is to be used.

Reset

This is a high level sensitive input signal. When reset, control goes to the location FFFF FFFOH. The processor wakes up in the real mode and the upper 12 bits of the address remain high until a far jump is executed. This pin is included in the list of interrupt pins because resetting has a pre-defined vector to which control is branched.

The other pins that are shown in Fig. 14.4 have the same functions as they have in the 8086, and have been discussed in chapter 6.

14.5.1 | Real Mode Operation of the 80386

The 80386 can operate in the real mode as well as in the protected mode. (There is also a third mode called the virtual 8086 mode.) On start up, the processor is in the real mode. In the real mode, it acts as a faster 8086 processor. The speed increase is due to the increased clock speed and the hardware enhancements. It takes the '386 only two cycles to perform a read/write (with zero wait states), while the 8086 requires four cycles. Besides being faster, the '386 in the real mode can also use the enhanced instructions the processor provides. It can use the 32-bit registers and data bus, and so the bandwidth is double that of the 8086.

The main feature of the real mode is that it has access only to 1 MB of physical memory, and address calculation is done as in the 8086, with the base address being directly added to the offset. The register set available in real-address mode includes all the registers defined for the 8086 plus the new registers introduced by the 80386, i.e., FS, GS, debug registers, control registers, and test registers.

When the processor is reset or after power on, it is in the real mode, with the reset vector being the address FFFF FFFOH. The upper 12 address bits remain high until a FAR call or jump is executed. The first far JMP or CALL instruction causes A31-A20 to drop low, and thus, the 80386 can execute instructions in the lower one megabyte of physical memory (00000 to FFFFFH). This automatic lowering of address lines A31-A20 allows systems designers to use a ROM at the high end of the address space (of 1 MB) to initialize the system. The first instruction has an address in the upper part of the 1 MB physical memory and that is usually ROM – thus, the processor continues to be in the real mode, working within the physical memory of 1 MB.

We will now discuss the protected mode features of the processor, starting with memory management. Before we do that, it is necessary to understand a very important concept regarding memory.

14.6 | Virtual Memory

Virtual memory is a concept by which the processor (in effect, the user) is made to believe that it has much more memory than is physically available. In effect, there is this idea of an ‘enlarged address space’, which is called a ‘virtual address space’. This space maps into a physical address space, ultimately. When a user writes a program, he uses ‘virtual addresses’ or ‘logical addresses’. The idea is that an application does not get to see where the data is physically located. It is up to the address translation mechanism to map these virtual addresses to a physical address. This is an operating system concept that keeps the user free of the worry of exhausting the available memory, when he writes an application.

Recollect that whenever we executed DOS based programs, we used physical memory, which is RAM/ROM. We noted that for the 8086, with 20 address lines, the memory space available is 2^{20} bytes, i.e., 1 MB. Similarly, for the 80286, with 24 address lines, the physical memory it can address is $2^{24} = 16$ MB. For the 80386, naturally, the physical address space is 2^{32} , i.e., 4 GB.

Consider using the 80386 for an application – does it mean that we might have to limit our data and code to fit into 4 GB? We know that in the computers that we use now, we never have experienced a ‘memory crunch’ or the feeling that our application cannot be run because the memory capacity is insufficient, though there might have been times when we have felt that our application runs too slowly. All this leads us to the concept of what is called ‘virtual memory’.

In a computer system there is a memory hierarchy, and a two level hierarchy is shown in Fig. 14.6. Let us discuss the concept of virtual memory using just a two level hierarchy, Semiconductor memory, which is reasonably fast (RAM/ROM), is what we usually designate as ‘primary memory’, ‘main memory’ or ‘physical memory’. Keep in mind that we will frequently use these terms interchangeably and all the three refer to the memory that is addressable by the address lines of the processor. We then designate the hard disk (and other memory types) as ‘secondary memory’. The processor directly communicates with the main memory only, but data movement between the secondary and main memory is possible. Whenever an application is to be run, the processor expects to

find the associated data and code in the main memory. In case it is not found there, it should be brought thereto from the hard disk. This takes a small amount of time. This is the reason why we find our computer ‘slow’ if the RAM size is low. However, even if the RAM is of a large capacity, it is not necessary that our application files are currently present there. Managing the system such that important files are found in the main memory most of the time, is part of the ‘cleverness’ of the operating system.



Figure 14.6 | Memory hierarchy in a computer system

Does this mean that the ’386 processor does all this? Definitely not, but the processor provides the extra hardware, extra instructions, and an intelligent ‘memory management unit’ that allows the OS to handle many of such tasks efficiently.

14.6.1 | Protected Virtual Addressing Mode (PVAM)

Now, we will go through the features of the 80386 processor, which has made all this possible. Please be warned that understanding these features is not very easy and cannot be done in one reading. Repetitive reading and contemplating on the intricacies of the mechanisms involved, is the only way to get a clear understanding of this processor. Since the same features are also present in the 80486 and Pentium as well, it is important to get a good insight into all this, at this point itself. It is also important to keep in mind that the ’386 is an upgraded version of the basic 8086 – as such, it will have all the features of 8086 plus additional features. The ‘additional’ features in the protected mode are what we will discuss now.

The following are the topics we will cover:

1. Memory Management
2. Protection
3. Multitasking
4. Exceptions and Interrupts

Even though we discuss each of them separately, they are not really disjoint – for example, protection is a part of memory management;

memory management and protection have to be taken into consideration when multitasking is done, and so on.

14.7 | Memory Management Unit

The 80386 transforms logical addresses (i.e., addresses as viewed by programmers) into physical addresses (i.e., actual addresses in physical memory) in two steps:

1. Segment translation, in which a logical address (consisting of a segment selector and segment offset) is converted to a linear address. Here, memory is considered to be divided into segments, which are large units of storage.
2. Page translation, in which a linear address is converted to a physical address. This step is optional, and usable on the discretion of systems-software designers. In this step, memory is divided into smaller storage areas, each of which is designated as a page.



Figure 14.7 | Address translation using paging and segmentation

These translations are performed in a way that is not visible to applications programmers. The 80386 has both segmentation and paging. If both these schemes are active, the logical address given by the program is converted to a linear address by the address translation hardware of the segmentation unit. This linear address is interpreted as a page address and given to the paging unit, which converts it to a physical address. **But paging can be switched off by certain register settings. In that case, only segmentation is effective and then the logical address is directly converted to a physical address.** We will deal with ‘segmentation’ and ‘paging’ separately.

14.7.1 | Segmentation

A typical instruction could be

MOV EAX, COST

where EAX is a 32-bit register and COST is a 32-bit offset in a segment.

This implies that the segment has a 32-bit base address, because all addresses are 32-bit long. However, to access the location called COST,

we have to get the base address of the segment it is contained in. COST is the offset of the data from the base address. The physical address calculation involves just adding the offset to the base address. In the real mode (as in 8086), this address calculation is very direct (Fig. 14.8), but it is not so in the protected mode.

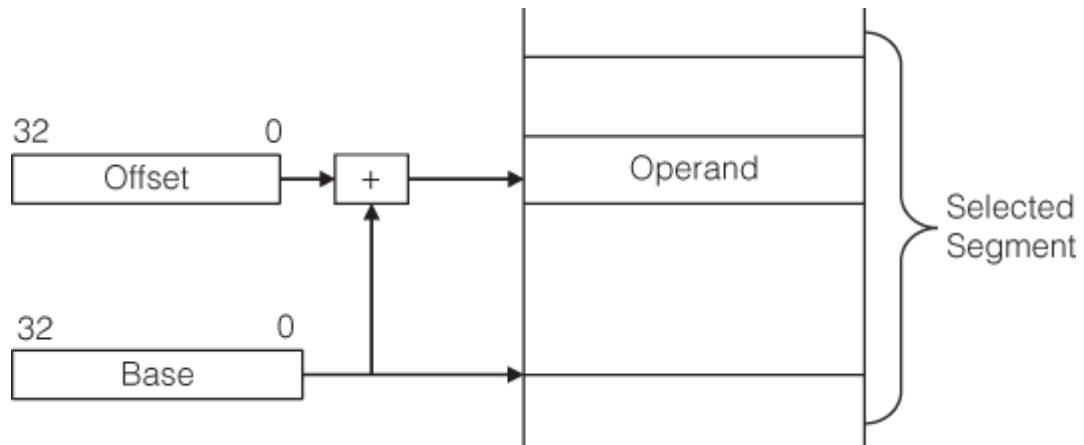


Figure 14.8 | Address calculation in the real address mode

The reason for the additional levels of calculation in the protected mode is the necessity for incorporating virtual memory concepts into memory management and ensuring protection features. Our problem is now of getting a physical address when the program supplies a virtual address, which is the logical address in a virtual memory system. In the instruction cited as an example, the label COST is the logical address of the operand to be accessed. COST will be a 32-bit address – a number (say, 0987 7654H).

Any location that is to be accessed will be in a particular segment. A segment can be of different types and it also has many attributes. All this is put together in what is called the ‘descriptor’ of the segment. Thus, any segment has a descriptor, which gives all information about the segment, i.e., it describes the segment. See in Fig. 14.9, the structure of a typical descriptor of a data/code segment used in an application program. (For an operating system segment, some bits of the descriptor will be different from this.)

Base (B31–B24)	G	D	0	A V L	Limit (L19–L16)	6
Access Byte				Base (B23–B16)		4
				Base (B15–B0)		2
				Limit (L15–L0)		0

15

0

Figure 14.9 | Descriptor of a data/code segment of an application program

A descriptor is an 8-byte (numbered as 0 to 7) entity, which describes the segment pointed by it. Let us see the contents of the descriptor in Fig. 14.9

1. It contains the 32-bit base address of the particular segment. This address is obtained as the concatenation of bytes 2, 3, 4, and 7 of the descriptor.
2. Bytes 0, 1 and 4 bits of byte 6 constitute the limit of the segment. This is a 20-bit number and specifies the maximum extent of the segment. There are two cases for the limit and this is decided by the G (granularity) bit of byte 6. If G = 0, it imposes a maximum limit of 64 K for the segment size. Then, the limit bits L₁₆ to L₁₉ are to be taken as zeroes, and the bits L₀ to L₁₅ give the size of this particular segment. Remember that with 16 bits, the maximum segment size is 64 K.
But if the G bit = 1, the number L₀ to L₁₉ is to be a multiple of 4 K (2¹²). Thus, the maximum possible size of a segment is $2^{20} \times 2^{12} = 2^{32} = 4$ GB. This is how the maximum size of a segment in the protected mode becomes 4 GB. In the descriptor, the number corresponding to the limit, gives the size of this particular segment.
3. Byte 5 of the descriptor is called the access byte. It contains the bits which specify the type and access rights to the segment pointed by this descriptor.

The various bits in the access byte mean the following. Refer Fig. 14.10.

P	DPL	S	E	X	R/W	A
7					0	

Figure 14.10 | Contents of the access byte in a descriptor

Bit 0 A: Accessed

This bit is set when the corresponding segment is accessed. The operating system tests this bit periodically to find out the frequency of usage of this segment – this is necessary for algorithms that need to have a measure of the ‘importance’ of this segment, meaning that, if this is a segment, which is accessed very frequently, it is best to retain it in the main memory.

Bit 4 S: Segment

indicates that this is a data or code segment descriptor for an application program ($S = 1$), or is a system segment descriptor ($S = 0$).

Bit 3: Executable E

This bit indicates if the segment is executable ($E = 1$) i.e., code segment, or non-executable ($E = 0$), i.e., data segment.

Bits 1, 2: Type

The type value depends on the E bit (bit 3 of the access byte) and the corresponding definitions are given in Table 14.2.

Table 14.2 | The Designation of the Bits of ‘Type’ Field of the Access Byte

Bit no.	If bit 3, E = 1 meaning executable (code segment)	If bit 3, E = 0 meaning non-executable (data segment)
2	C, Conforming, C = 1 Non conforming, C=0	ED Expand down. ED = 1, stack seg- ment Expand up ED = 0, data segment
1	R, Readable R = 1, may be read R = 0, may not be read	W, writable W = 1, writable W = 0, may not be written to

Bits 5, 6 DPL

This is a two-bit field, which stands for ‘descriptor privilege level’, and is part of the ‘protection mechanism’, which we will discuss in greater detail in Section 14.10.

Bit 7 P: Present Bit

The setting of this bit shows that the segment corresponding to this descriptor is now present in the main memory. If $P = 0$, it means that the corresponding segment is not in main memory. Hence, a type 11

interrupt is generated, which initiates an ISR to bring the segment into the main memory.

Byte 6

Four bits are used as part of the limit field. Byte 6 has four bits other than the limit bits

G: Granularity

This bit (discussed earlier) decides if the limit field has to be multiplied by 1 or 4 K. If G = 0, the multiplier is 1. If G = 1, the multiplier is 4 K.

D

Selects the default instruction mode for code segment descriptors. If D = 0, the registers and memory are 16 bits wide as in the 80286; if D = 1, they are 32 bits wide as in the 80386.

AVL

This bit can be used by the OS in any way it needs. It is often used simply to specify that the described segment is available. Bit 5 of byte 6 is to be 0.

14.7.2 | System Descriptors

Segments that are used by the operating system are separate and somewhat different from application program segments. They have a descriptor, which is slightly different. Figure 14.11 shows the general format of a descriptor used for operating system uses. If Fig. 14.11 is compared with Fig. 14.9, what will be noted is that only the bits in the access byte and the four bits of byte 6 are different. The difference in the access byte is that, 4 bytes are used to specify the type of the segment being described. There are 16 possible types, of which a few have reference to the 80286, some refer to future uses, and only a few indicate the types of system segments of the 80386. It is not worth going to the details of these types, since it will only lead to cluttering of this discussion with unnecessary detail.

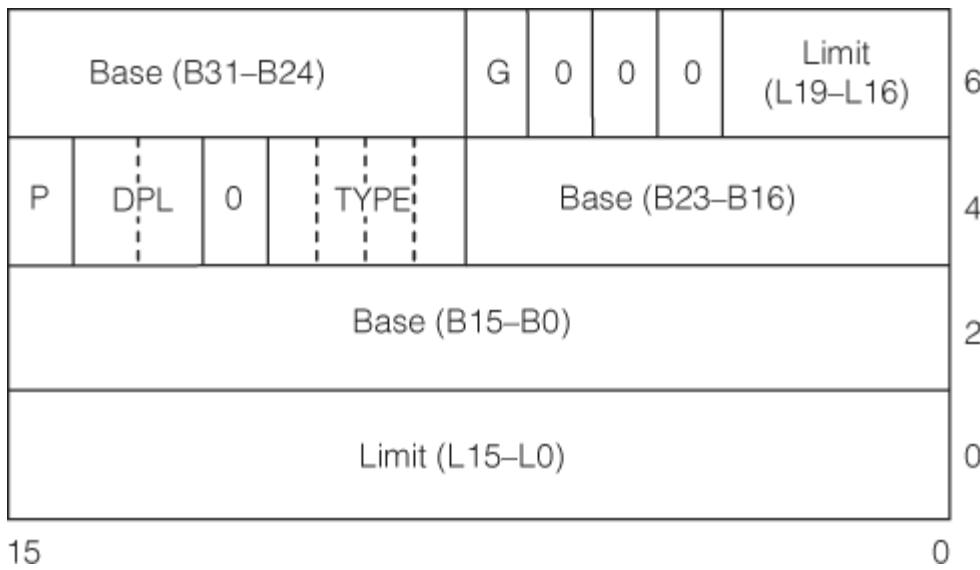


Figure 14.11 | Descriptor for a system segment

14.7.3 | Descriptor Tables

So, we see that each and every segment has a descriptor. However, who creates descriptors? Definitely, not the application programmer. When the programmer creates a segment, the system software creates the corresponding descriptor with the help of compilers and loaders and linkers – in effect; it is an operating system job.

In any system, there will be a number of segments of various types created for various applications – thus, there should be as many descriptors too. These descriptors are stored in tables called ‘descriptor tables’. There are two types of descriptor tables – the local descriptor table and the global descriptor table.

14.7.3.1 | Local Descriptor Table (LDT)

Suppose the system has 10 application tasks whose segments and corresponding descriptors have been created. For each of these tasks, there will be a descriptor table designated as the LDT of the particular task. Thus, we may name these tables as the LDT of task 1, task 2 and so on. There will be as many LDTs as there are application tasks.

14.7.3.2 | Global Descriptor Table (GDT)

There are segments and corresponding descriptors, which do not belong to any specific application but cater to the system as a whole – they are global in nature. These descriptors are stored in the GDT. **There will be only one GDT in a system.**

For executing a particular application task, say task A, segments from the LDT of task A are required and some operating system services are also required. Figure 14.12 shows how an application task needs the GDT as well as its own LDT.

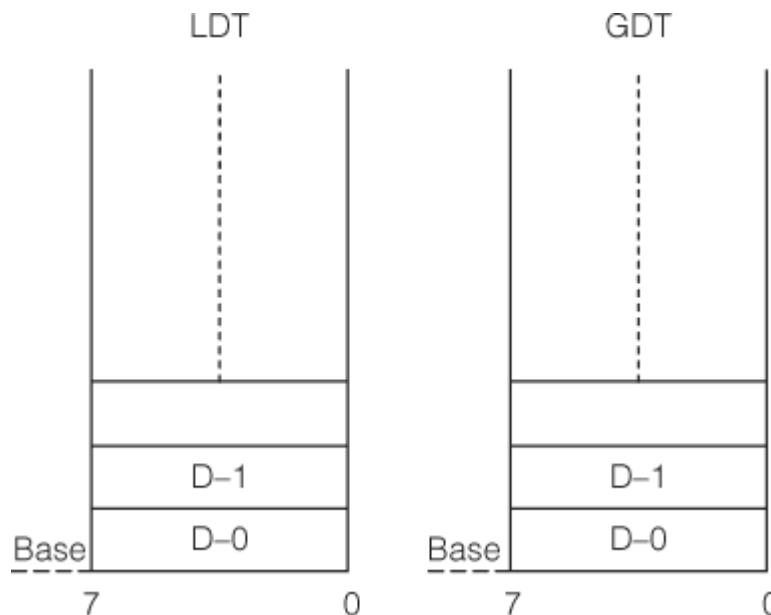


Figure 14.12 | LDT of a task and the GDT of the system

The next question is – where are these tables stored? The answer is – in the main memory, naturally. Remember also that a descriptor table is a memory array of 8-byte entries and that the size of a table is variable. Thus, these tables are also ‘segments’ and each of them also should have a descriptor. This may sound a bit confusing at first, but on going ahead, this confusion will soon be overcome.

This takes us to the discussion of the processor registers which are used in the protected mode. At a particular time, the descriptor of the GDT will be available in a processor register called the ‘Global Descriptor Table Register (GDTR)’. This is to be ensured when the system is switched to the protected mode. Since descriptors are 8-byte entities, these registers have to be 8 bytes in length, obviously. In fact, the loading of the GDTR with the descriptor of the GDT occurs when the system is put into the protected mode.

If an application task is running, the descriptor of the LDT of that task is available in the Local Descriptor Table Register (LDTR). We will soon get to the details of all the processor registers involved in the address translation mechanism.

14.7.4 | Selector

Remember that any logical address for the 8086 could be specified in the form

Segment base address: Offset

For the 80386 protected mode, the format of any logical address is modified to be

Segment Selector: Offset

The offset is 32 bits long as mentioned earlier, but the selector is only 16 bits long and has the structure as shown in Fig. 14.13. It contains a 13-bit number called the index. This number is an indicator of where in the descriptor table, the particular segment's descriptor is located. Thus, for each descriptor, the selector is a pointer to its position in the descriptor table. Since the index is 13 bits long, the number of descriptors it can point to is 2^{13} , i.e., 8192, this means that any descriptor table can have 8192 descriptors stored in it.



Figure 14.13 | Format of a selector

In the selector, there is a bit called TI that stands for ‘Table Indicator’. This indicates whether the selector corresponds to a descriptor in the LDT (TI = 1) or the GDT (TI = 0).

RPL

This is a two-bit field ‘Requested Privilege Level’, which is part of the protection mechanism, which will be discussed in detail in Section 14.10.4. The selector value is fixed up by linker or loader software.

Registers for Address Translation

To understand the concept of address translation in the protected mode, we should have a clear idea of the registers that are involved in this. Corresponding to the type of segments in use, there are the segment registers as shown in Fig. 14.14. Each segment register has a two-byte (16-bit) visible portion and an 8-byte hidden part in which the corresponding descriptor is loaded, when the corresponding segment is being accessed.

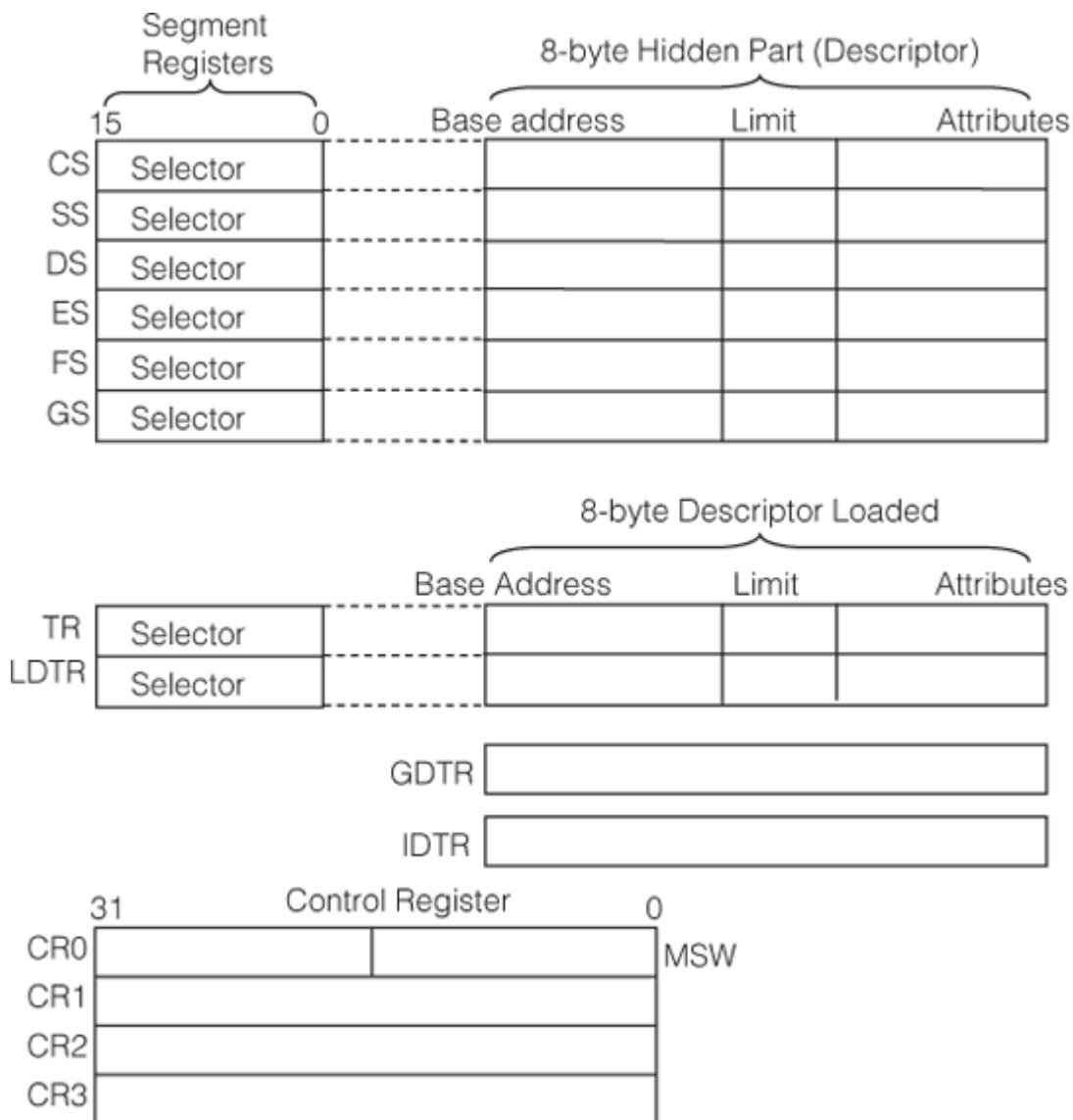


Figure 14.14 | Address translation registers

Now, let us consider a simple case of address translation. Note the following points. The GDT and LDT are, as mentioned earlier, memory arrays or segments. So they too have descriptors.

The GDT contains the descriptors of the segments used in system programs – there is only one GDT for the whole system. As such, there is no selector for the GDT. On startup, the descriptor of the GDT will be loaded into the GDTR. Once this is set up, the system can start running application tasks. Figure 14.15 shows the descriptor of a GDT loaded into the GDTR. Note that the GDT can contain a maximum of 8192 descriptors and the limit field indicates the size of the GDT (here, a maximum sized GDT is shown).

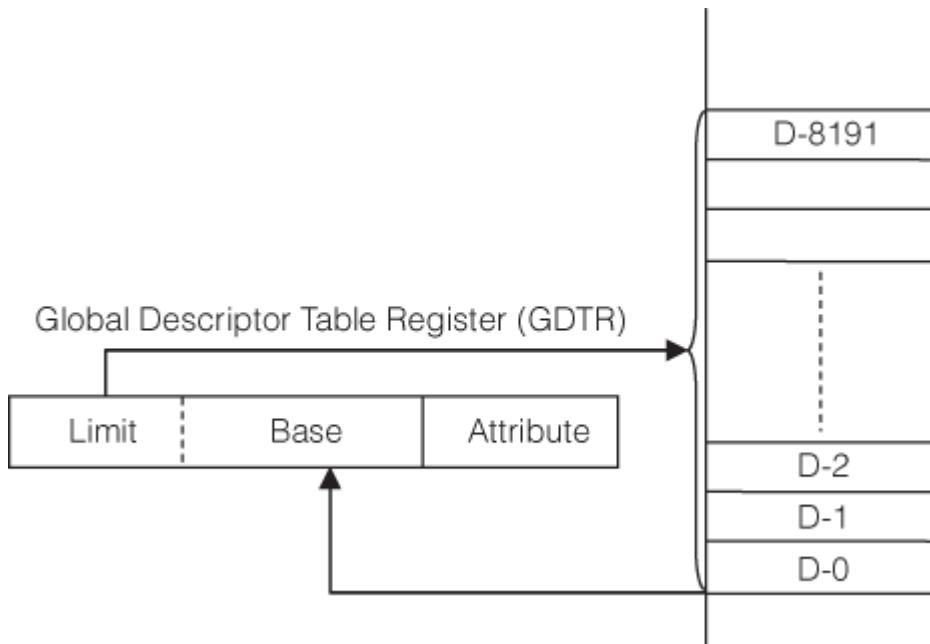


Figure 14.15 | GDT and GDTR

There are as many LDTs as there are tasks in the system. The descriptors of all the LDTs are in the GDT. When an application task is taken up, the selector of its LDT will be loaded into the visible part of the LDTR and then the corresponding descriptor is taken from the GDT and loaded in the hidden part of the LDTR. It is then that this task becomes active. (When this task is abandoned and a new one is taken up, the content of the LDTR will be changed to favor the new task).

14.8 | Converting a Logical Address to a Physical Address

Now consider an instruction for an application program, which is

MOV EAX, COST

This instruction needs an operand from a data segment to be moved to the 32-bit register EAX. The segment has a descriptor as well as a selector created by the system software. The needed data has an offset labeled COST which corresponds to, say, 0012 5674H. Looking at Fig.14.16, try to follow the steps outlined below. The steps in accessing data are:

1. Since it is a data segment, which is to be accessed, the selector of the data segment will be loaded into the visible part of the DS register, i.e., the segment register part. The selector contains a 13-bit number – the index. This is multiplied by 8 and added to the base address of the descriptor table (GDT or LDT, depending on the value of the TI bit in the selector).

2. Note that the base address of the descriptor table is available in the GDT register (GDTR), or LDT register (LDTR) of the processor. Thus, the descriptor of the data segment is obtained from the descriptor table, and this is loaded into the hidden part of the DS register. Since an application program is considered here, the descriptor of the data segment will be taken from the LDT of the application task.
3. The descriptor of the data segment has the base address of the data segment. This is used to locate the data segment. This base address plus the offset mentioned in the instruction, i.e., 0012 5674H (with label COST) are added to get the physical address of the operand, which is then moved to the register EAX.

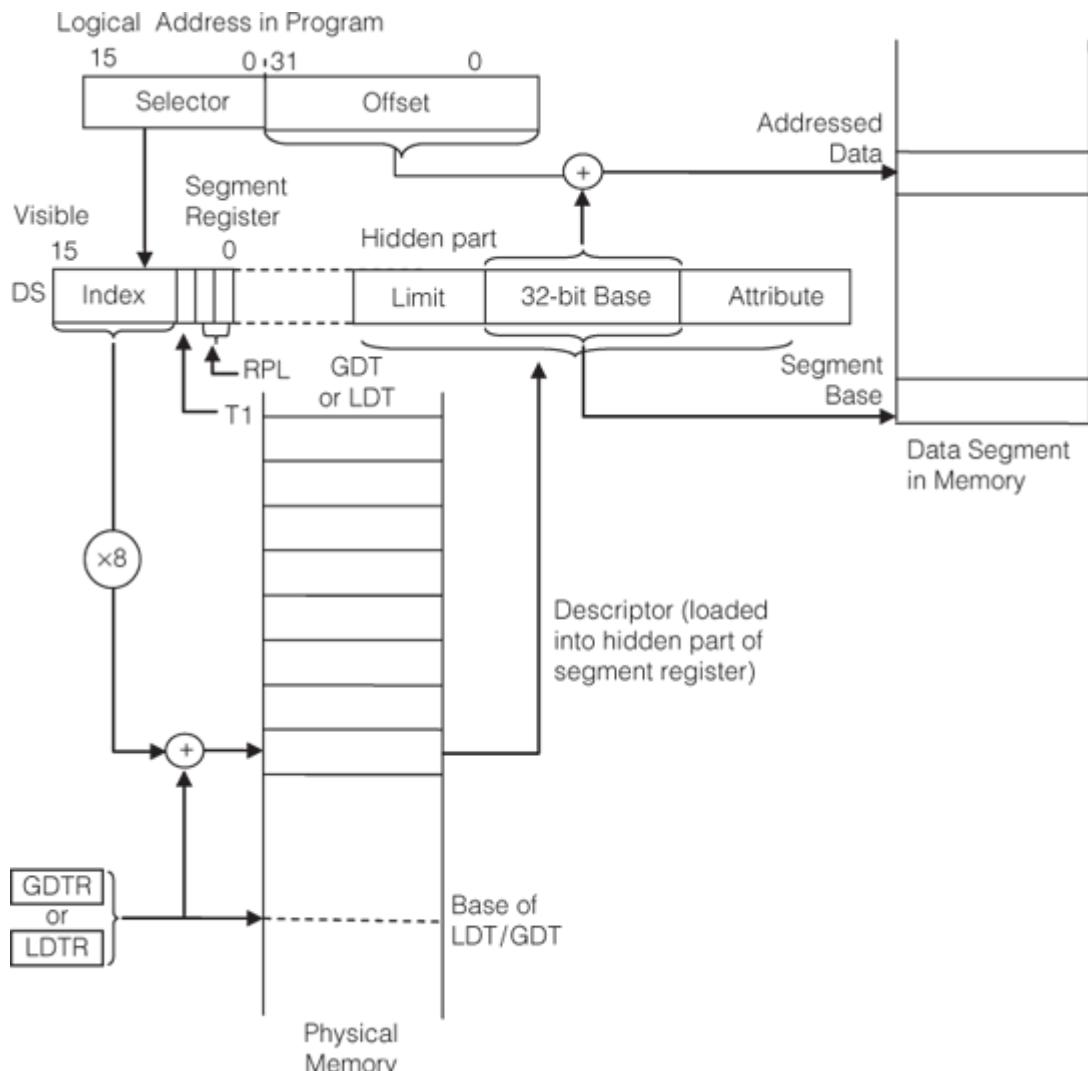


Figure 14.16 | Address translation of a logical address to a physical address

The above explanation should be sufficient to understand the translation of a logical address specified in the instruction to a physical address in memory, in the case of segmentation.

14.9 | Calculating the Size of the Logical Address Space

In a segmented model of memory, the address space as viewed by an applications program (called the logical address space) is a much larger space of up to 2^{46} (64 terabytes). How is this figure obtained? Let us make a calculation.

A selector has 13 bits as index for selecting one of the descriptors from a descriptor table. Thus, it can access up to 2^{13} descriptors with just one descriptor table. Since two descriptor tables (the GDT and one LDT) are active at a time, 2×2^{13} segments can be accessed. Each descriptor corresponds to a segment with a maximum size of 4 GB or 2^{32} bytes. The total available memory space is now calculated to be $2 \times 2^{13} \times 2^{32}$ bytes = 2^{46} or 64 terabytes ($2^6 \times 2^{40} = 64$ TB). Thus, the logical or virtual address space is 64 TB, which is much larger than the actual physical memory of 4 GB.

This calculation makes clear the idea of how the virtual memory concept creates an enlarged memory size. The address translation mechanism discussed above has the duty of mapping this large virtual memory space to a much smaller physical memory. Applications programmers do not need to know the details of this mapping.

As a matter of fact, the user usually thinks that the memory available to him is infinite. This illusion is created by having a secondary memory and shuttling data between the main and secondary memories according to the needs of the user. This underlying mechanism is not the concern of the user – the OS has the duty of performing this task efficiently.

14.9.1 | Paging

The mechanism of paging is in many ways, similar to segmentation – however, a smaller chunk of memory is handled in this case. Remember that a segment can have a variable size with a maximum physical size of 4 GB, but the size of a page is fixed to be 4 KB. This small size is what makes paging so attractive.

Just as in segmentation, whenever we want to run an application, we expect the required page to be in the main memory – if not, it has to be brought to the main memory. The charm of paging is that it allows just one page (i.e., just 4 K) to be brought in (or swapped out, if necessary). In contrast, segmentation requires that the whole segment be handled.

Paging is not a new concept, but 80286 did not allow paging – the '386 allows both paging and segmentation – which implies that any segment can be considered to be composed of a number of pages, and handled as one segment or as a number of separate pages. This is quite logical because we know that sometimes we might need only a part of the available data/code segment for an application.

14.9.2 | Page Translation

The 80386 has segmentation as a basic concept in its memory management unit. The **logical address** from a program is converted into a '**linear address**' by the segmentation unit, and given to the paging unit. This **linear address** is to be translated to a **physical address**, and this entails two levels of translation. A linear address can refer directly to a physical address so the segment and offset can be located directly, as is done in Section 14.8. This is the case when only segmentation is used. If paging is also activated, a linear address indirectly specifies a physical address (refer Fig. 14.7) by specifying a page table, a page within that table, and an offset within that page. A 32-bit linear address, as shown in Fig 14.17, is composed of the following fields:

1. A 10-bit field called DIR which points to a page descriptor in a page directory.
2. A 10-bit PAGE field which is used as an index into the page table determined by the page directory.
3. A 12-bit offset which points to the required byte within the located page.

A page or page frame, as it may be referred to, is a 4 K-byte unit of contiguous addresses of physical memory. See Fig. 14.18 for understanding the two levels of address translation in paging. Note that the base address of the current page directory is to be in the CR3 register. This register is in the processor and is also called the Page Directory Base Register (PDBR). The page directory contains page descriptors, which are similar to segment descriptors, except that these are 4-byte entries, i.e., 32 bits long. Thus, the page directory contains the descriptors for page tables. A page table is also a page (4 K in size) – thus, it has a 4-byte descriptor, which is in the page directory.



Figure 14.17 | Format of the 32-bit linear address

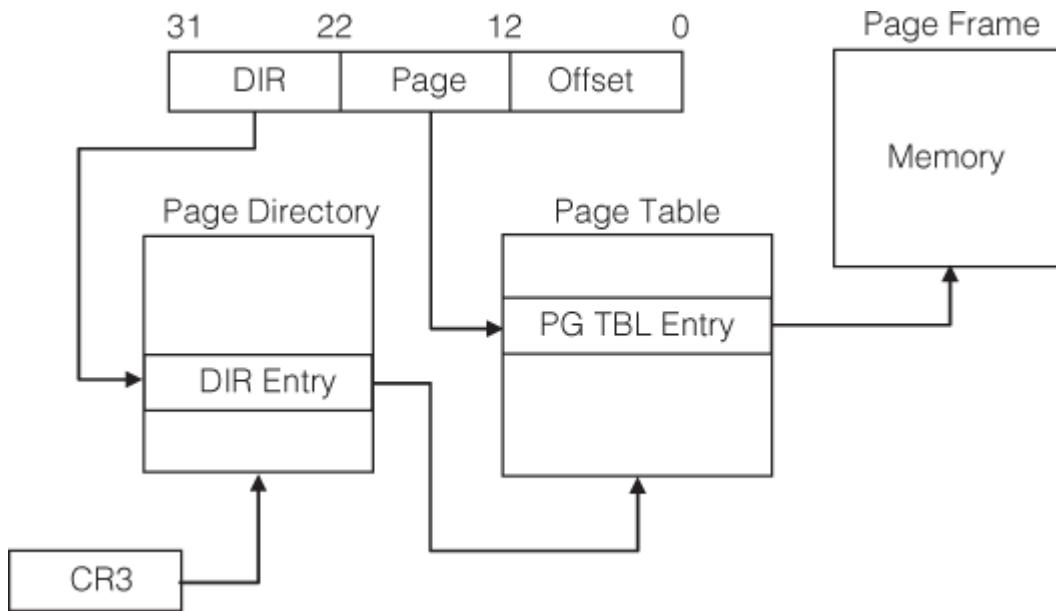


Figure 14. 18 | Two level address translation for paging

In the first level of address translation, the bits of the DIR field of the linear address is used as an index to locate a page descriptor in the current page directory. This will contain, among other things, the base address of a page table. In the next level of translation, the PAGE field of the linear address is used as an index to locate an entry in the page table. This entry contains the address of the required page in the physical memory. Adding the OFFSET field (in the linear address) to this address, gives the physical address of the required data. Note that this is a two level address translation mechanism, and the two levels are similar. In the first level, a page directory is accessed, and in the second, a page table is accessed. Both the page directory as well as the page table are 4 K-byte long pages.

Now to get a feel about the amount of memory addressable with paging, let us do a bit of calculation. Go back to the format of the linear address. Since the DIR field is 10 bits long, 2^{10} or 1 K page tables are possible. Each entry in the page table is indexed by the 10-bit PAGE field of the virtual address – hence, it has 2^{10} entries of page descriptors – which means 2^{10} pages can be addressed. Each page can contain 2^{12} (4 K) bytes. Thus, the total memory is $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ bytes or 4 GB, which is the complete physical memory of the 80386 processor (and also the maximum size of a segment in the protected mode).

14.9.3 | Format of a Page Table Entry

Figure 14.19 shows the format of a page table entry. Let us discuss its contents.

31	12 11	0
Page Frame Address 31...12	Avail 0 0 D A 0 0 U / S R / W P	

P – Present

R/W – Read/Write

U/S – User/Supervisor

D – Dirty

Avail – Available for systems programmer use

Figure 14.19 | Page table entry

Page Frame Address

The page frame address specifies the physical starting address of a page. Because pages are located on 4 K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the base address of a page table. In a second-level page table, the page frame address is the address of the page that contains the desired memory operand.

P bit

This bit has the same function as the P bit in segmentation, i.e., P = 1, indicates a valid page access. P = 0 indicates that this page is not in physical memory, and hence, it generates an exception, which initiates an ISR to bring the page to physical memory.

R/W

Read / Write-used for protection.

U/S

User/Supervisor-used in the protection scheme.

D

Dirty is undefined for page directory but used in the page table. It means the page has been modified.

A

Accessed is set to 1, whenever the page directory is accessed. Certain bits are left to be 0, which indicates Intel reserved.

14.9.4 | Translation Lookaside Buffer

It is cumbersome and time consuming to calculate the physical address from the linear address for every memory access. A translation lookaside buffer (TLB) simplifies the process. A TLB is a table in the processor which stores the physical addresses of the 32 recently accessed page table entries. The TLB is a ‘hardware cache’ for the physical addresses of the most recently used virtual address. It is a four way set associative cache inside the ’386 chip.

The paging unit receives a 32-bit linear address from the segmentation unit. The upper 20 bits of the linear address is compared with all 32 entries in the TLB to check if it matches with any of the entries. If it matches, the 32-bit physical address is calculated from the matching TLB entry and placed on the address bus. See Fig. 14.20a. If it does not match, the processor must fetch into the TLB, the page table entry from memory.

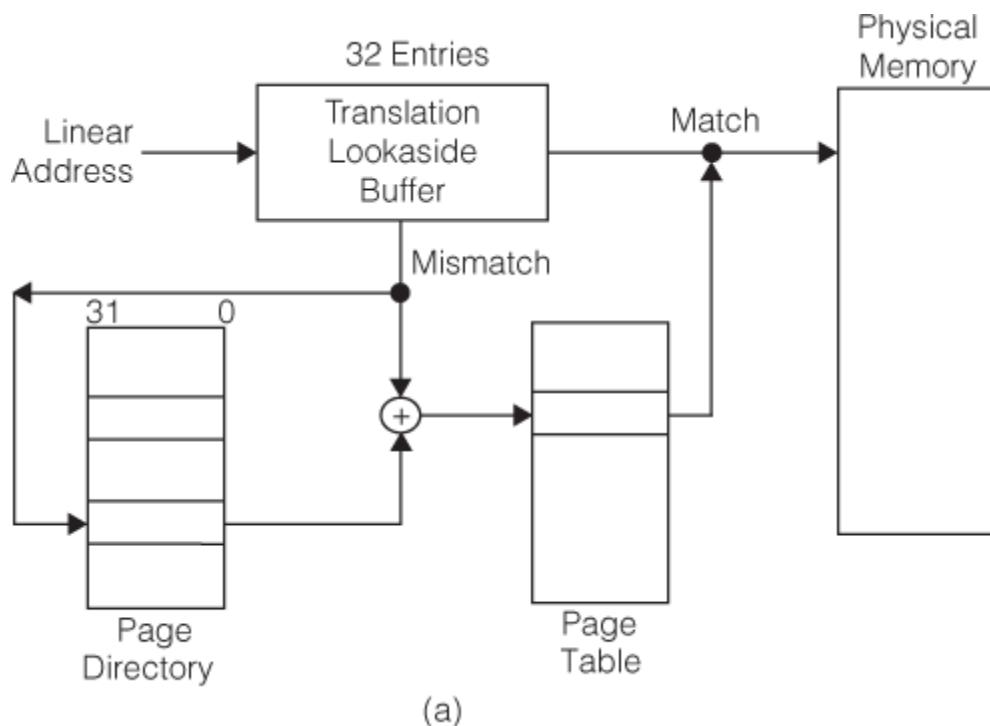


Figure 14.20a | Translation lookaside buffer inside the 80386 chip

Linear address (X)	Physical address(Y)
X1	Y1
X2	Y2
X3	Y3
.....
.....

(b)

Figure 14.20b | A conceptual view of the TLB

Corresponding to any linear address, there is a physical address obtained by page translation. Once page translation has been done once, the corresponding physical address is available. This is saved in the TLB (for the recent entries only). Conceptually, the TLB is shown in Fig. 14.20b, which is a table showing the physical address corresponding to a linear address in the case where address calculation has been done recently. For a linear address X, the physical address is Y.

14.9.5 | Combining Paging and Segmentation

Since 80386 has both segmentation and paging, it is easy to visualize a system in which a segment can be composed of a number of pages. To access a data in a page, what is needed is to have a technique by which the logical address corresponding to segmentation locates a segment, and then this logical address is converted to a linear address. This linear address is converted to a physical address as we have just seen. This is shown in Fig. 14.21.

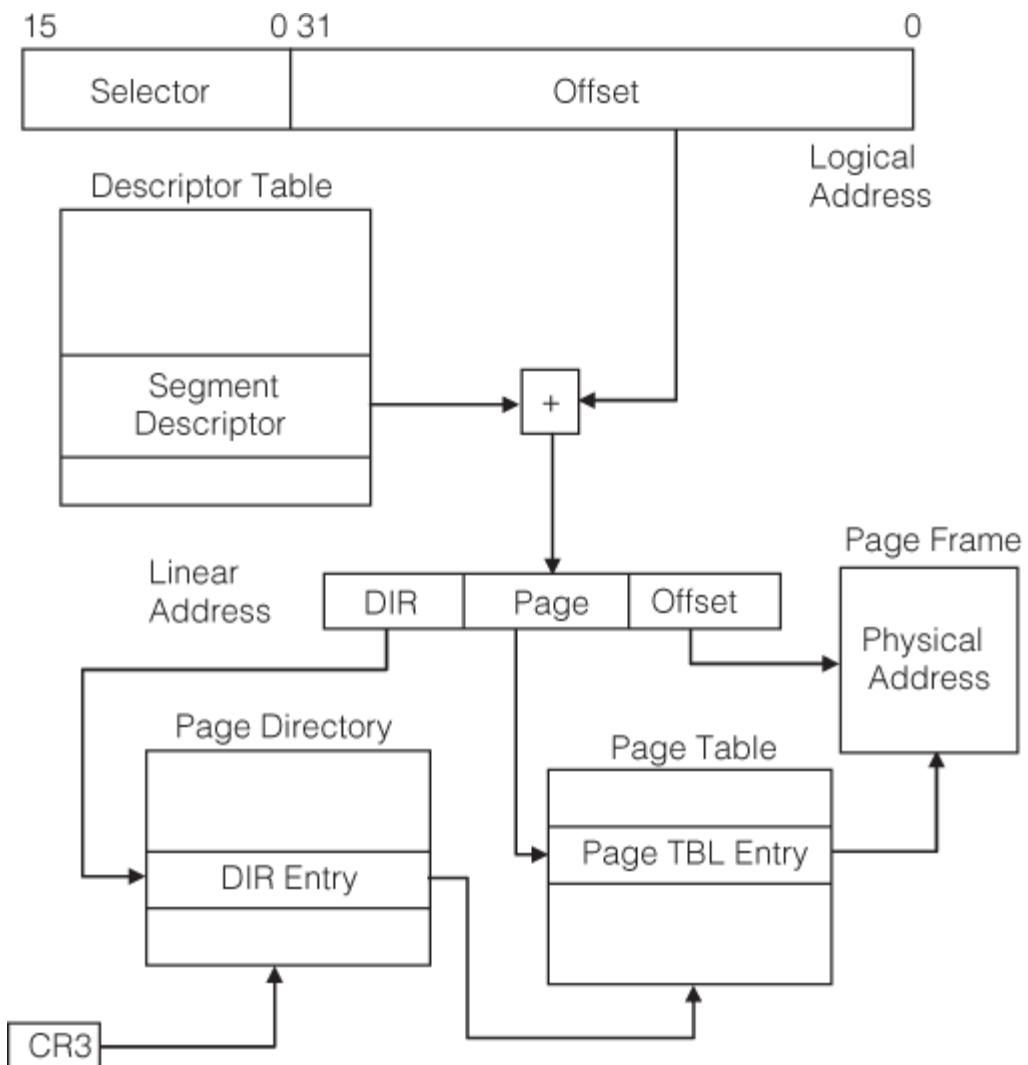


Figure 14.21 | Segmentation and paging combined

14.9.6 | Flat Model

In a '386 based system, it is not mandatory to use paging and segmentation together. There is the option to turn off the paging mechanism that leaves a segmented architecture. However, a mechanism to turn off segmentation is not inherently present. However, this can be achieved by programming techniques.

When the 80386 is used to execute software designed for architectures that do not have segments, it may be expedient to effectively 'turn off' the segmentation features of the 80386. The 80386 does not have a mode that disables segmentation, but the same effect can be achieved by loading the segment registers with selectors for descriptors that have a base addresses of 0, and by setting the limits to allow access the entire 32-bit address space. Once this is done, there is no longer a need to change the segment registers. The 32-bit offsets used by the 80386 instructions, in protected mode, are sufficient to access the entire linear

address space. This is called the ‘flat’ model. In a ‘flat’ model of memory organization, the applications programmer sees a single array of up to 2^{32} (4 GB) bytes.

14.10 | Protection

In the previous section, you have seen the word ‘protection’ being used without being told what exactly it is and how it is achieved. So the first thing to be done in this section, is to get a clear idea of the word ‘protection’, in the way it is used in the protected virtual addressing mode of 80386. There are many aspects to protection in this context.

Protection Between Tasks

A system can have a number of tasks, some of which are active, and some of which are not. One requirement is to protect user tasks from each other – this means that the code and data of one user is to be isolated from all other users – since code and data are in segments, this implies that segments of one task should not have access to the segments of other user tasks. This protection is implemented by having separate LDTs for each task. The descriptors of segments of each task are kept in separate local descriptor tables. At a time, only one user task is active and only one LDT can be used.

Protection between user and system tasks

The system programs, i.e., tasks associated with the operating system, manage the computer system as a whole. Thus, these tasks are very important and should not be allowed to be tromped on by users, i.e., application tasks. This is achieved by the concept of privilege levels. System tasks are allotted a high privilege level, while user tasks are at a lower privilege level – and direct access of segments of a higher privilege level task is not permitted. Thus, user tasks are prevented from accessing system level segments.

Capturing address violation errors

This aspect ensures that addresses beyond the limit of a segment are not accessed. Any such attempt is immediately aborted. It should be remembered that all these features are provided in the hardware of the processor, and when an operating system is designed using these hardware features, the design of the operating system becomes relatively easy. Now, let us get into the details.

14.10.1 | Protection Checks

The important aspects of protection in the 80386 are as follows:

- Type checking.
- Limit checking.
- Restriction in the accessible domain.
- Restriction in the entry points of procedures.
- Restriction in the usable instructions.

Address translation and memory access have already been discussed. protection hardware is an integral part of that unit, which applies to both segmentation and paging. It is, thus, necessary to keep in mind that protection is discussed separately, only for ease of understanding. Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria. All these checks are made before a memory cycle is started. Any violation prevents that cycle from starting, and results in an exception. Since the checks are performed concurrently with address formation, there is no performance penalty. Invalid attempts to access memory, results in an exception (an exception is an interrupt caused by an error).

Role of descriptors

The protection parameters are placed in the descriptor by system software at the time a descriptor is created. When a segment selector is loaded into a segment register, the descriptor of the segment is automatically loaded into the hidden portion of the segment register – thus, all the protection parameters are available in the segment register inside the processor and so subsequent protection checks on the same segment do not consume additional clock cycles. The first check that needs to be done is on the P bit in the descriptor. If P = 0, it implies that the referred segment is not present in main memory, and an exception is generated, which initiates an ISR to bring the corresponding segment to the main memory.

14.10.2 | Type Checking

There is a type field for a descriptor within the access byte (bits 1, 2 and 3). Refer Fig. 14.10. This field has two functions.

1. It distinguishes between different descriptor formats.
2. It specifies the intended use of a segment.

Programming errors can occur if a wrong segment selector is loaded into a segment register. Keep in mind that a code segment alone is an executable segment. It can be allowed to be read but cannot be written into. Data segments must be readable, and stack segments are to be - writable. Thus, the following conditions are to be verified.

1. The CS register can be loaded only with a selector of an executable segment.
2. Selectors of executable segments that are not readable cannot be loaded into data-segment registers.
3. Only selectors of writable data segments can be loaded into SS.

When an instruction with memory reference is to be executed, type checking is performed to confirm that

1. no instruction may write into an executable segment,
2. no instruction may write into a data segment if the writable bit is not set,
3. no instruction may read an executable segment unless the readable bit is set.

14.10.3 | Limit Checking

There is a limit field in the descriptor and the limit of a segment depends on the value of the G-bit. Before allowing a memory access, this field is used to confirm that memory access outside the segment does not occur. The value of the limit field has to be interpreted differently for expand-up segments (data), and expand-down segments (stack). Catching and preventing address violation errors goes a long way in avoiding system crash.

14.10.4 | Privilege Levels

The concept of privilege levels is all about trust and protection.

Trust

When a task is to be performed, it is code that gets executed – to execute code, data from various segments are required – also, code from other segments may be required when a branching to a different segment is to be done. The task that does this has to be a ‘trusted’ task – the code segment performing the task has to be a trusted segment. Then only it can be allowed to access other segments. The privilege level accorded to this segment depends on how much of trust is placed on it. If it is accorded the highest privilege level, it means that is allowed access to any other segment that it needs. Such a high privilege level is accorded

only to segments of operating system tasks, i.e., tasks with supervisor status, which are the most trusted tasks. Application level tasks usually are at low privilege levels – they are not trusted enough to allow them to use all available segments. All this means that all segments are given a particular privilege level (usually a number), and a task at a low privilege level can only access segments at the same level or at a lower privilege level.

Protection

Now think of a data segment or even a code segment which contains a procedure to be accessed from other code segments. Let us call it a target segment. What is the extent of protection it has? Can it be used by just any task? In that case, it is a segment at the lowest privilege level and hence the ‘least protected’. If it is highly protected, it will be accorded the highest privilege level and then, only tasks with the supervisor status can use it.

14.10.4.1 | Privilege Levels of 80386

The ’386 processor defines 4 levels of privilege numbered as 0 to 3 – the smaller the number, the higher the privilege level. Figure 14.22 shows the 4 level privilege hierarchy. Note that the kernel of the OS has been accorded the highest level of 0, while application tasks are at level 3. OS level services are at privilege level 1, and utilities and custom services of lesser importance are at level 2. In the figure, three application tasks A, B and C have been shown. It is not mandatory that an 80386 based system use all the 4 privilege levels. The system software can reduce it to 3 or 2 as needed.

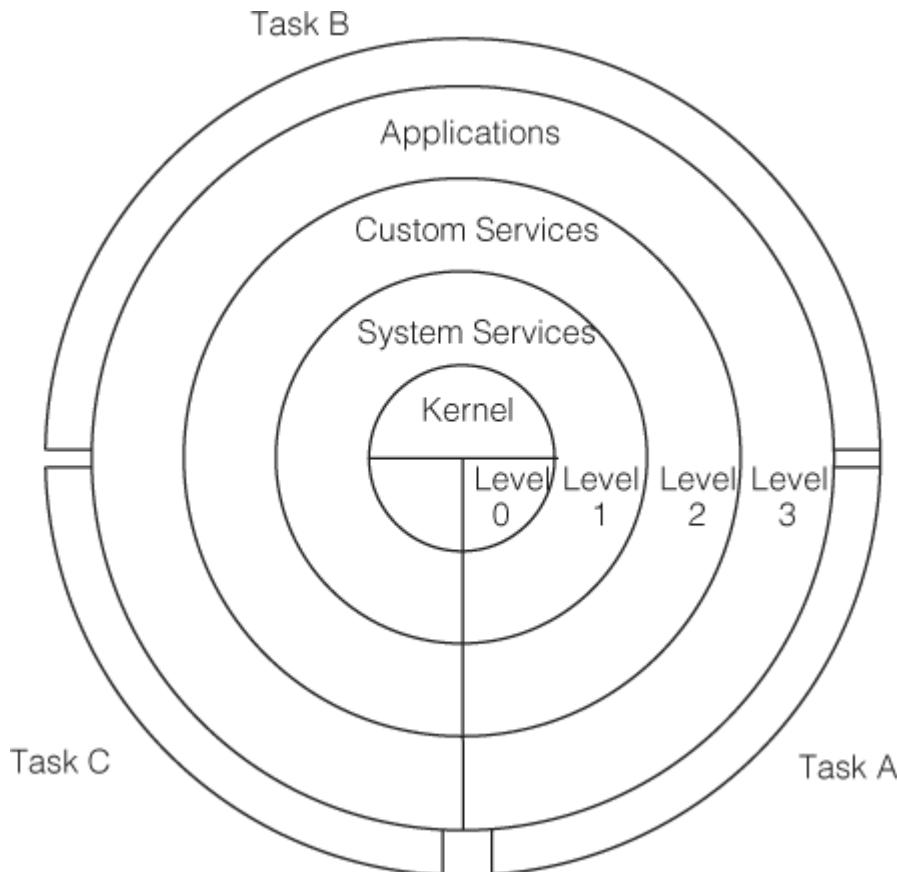


Figure 14.22 | Privilege level hierarchy of 80386

Three different types of privilege levels enter into the privilege level checks.

CPL (Current Privilege Level)

CPL is stored in the selector of the currently executing code segment register (bits 0 and 1); **CPL represents the privilege level of the currently executing task**, i.e., the program or procedure. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. Thus, this is also the privilege level in the access byte of the descriptor of this code segment (except for what is called a ‘conforming’ code segment). The CPL is also designated as TPL or the Task Privilege Level.

DPL (Descriptor Privilege Level)

DPL is the privilege level of an object, which is being attempted to be accessed by the current task. It is the privilege level of the target segment, and is contained in the access byte of the descriptor of the corresponding segment. The above discussion might seem a bit complex, but what is actually done is only a simple comparison of numbers. In simple terms, what is done is that the CPL of

the task in execution is compared with the DPL of the target segment. After comparison is done, access is allowed only if the CPL is higher or the same as the DPL of the target segment. Keep in mind that a higher privilege level corresponds to a lower number.

RPL (Requestor Privilege Level)

It is seen that the lowest two bits of any selector are reserved for the RPL field – these bits have an effect on privilege validation. This RPL field of a target segment's selector can be used to ‘weaken’ the CPL, if desired. If RPL of the selector is 2 while the CPL of the task is 1, the effective privilege level is the numerical higher of them, i.e., 2. Thus, the task becomes less privileged. The task is effectively excluded from accessing a data segment that the task might otherwise have been privileged to use. (This may seem unnecessary, but it is generally used to prevent inappropriate use of pointers that could corrupt the operation of more privileged code or data from a less privileged level.)

But the opposite effect of making RPL = 0 does not have any effect. If the selector's RPL is numerically less than the CPL, it has no effect.

14.10.5 | Accessing Data Segments

Assume that a task needs to get data from a data segment. The privilege level checks are performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register. (Keep in mind that the higher privileges are represented by smaller numbers and vice versa.) As Fig. 14.23 shows, three different privilege levels enter into this type of privilege checking mechanism.

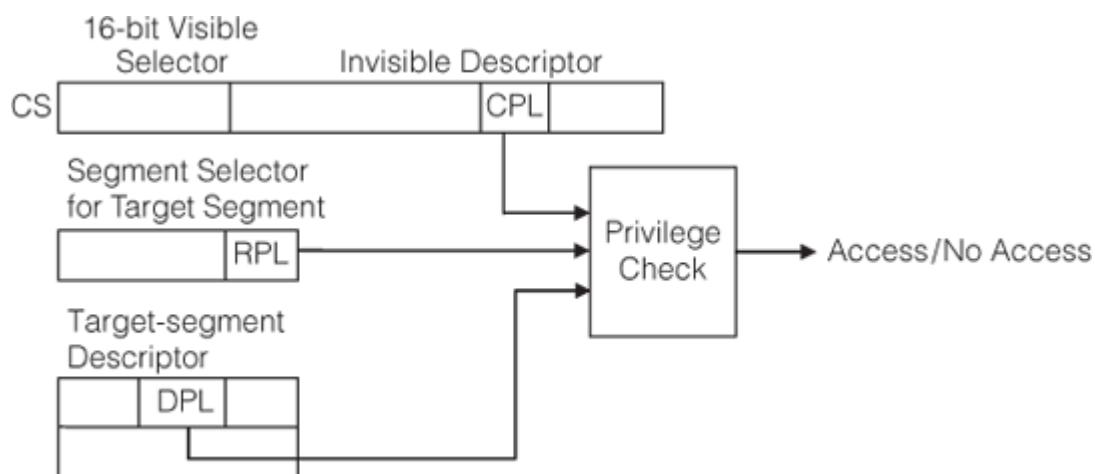


Figure 14.23 | Privilege level checks for a task when accessing a data segment

1. The CPL.
2. The RPL of the selector used to specify the target segment.
3. The DPL of the descriptor of the target segment.

Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is **numerically greater** than or equal to the maximum of the CPL and the selector's RPL. In **other words, a code segment can only access data that is at the same or less privileged level**.

The data segment has a DPL in the access byte of its descriptor, and this number designates the privilege level of the data segment. Then, the data segment must be at the same or lower privilege level in comparison with the CPL of the task. The accessible domain of a task varies according to its CPL. If the CPL of a task is zero, data segments at all privilege levels are accessible; when CPL is one, only data segments at privilege levels one through three are accessible; when CPL is three, only data segments at privilege level three are accessible. This property of the protection mechanism is used, for example, to prevent applications procedures from reading or changing tables of the operating system.

In short, the following numeric equation should be satisfied: $CPL \leq DPL$. However, CPL can be weakened by the RPL. So we define the Effective Privilege Level (EPL) as the numeric maximum of RPL and CPL. Thus, the final condition for access is that $EPL \leq DPL$.

Figure 14.23 shows the concept of privilege level checking when a task attempts to access a data segment. If the required condition is satisfied, access is allowed, otherwise it is not.

14.10.6 | Privilege Checks for Control Transfers

We have been talking of code segments having to access segments that contain data. However, many times code segments have to access other code segments. This happens when a far call or jump or ret instruction is encountered. (It also occurs in the case of interrupts but that will be discussed separately). There are a few issues in this type of control transfer.

- We can go to a code segment of lower privilege level, but on returning, it causes privilege level violations, so in effect this type of access is not possible.
- Going to code segments of higher privilege level is not permitted, but there is an indirect way of doing this and that is using ‘gates’.

Before proceeding further, a clarification on ‘conforming’ and ‘non-conforming’ code segments needs to be given.

Conforming Code Segment

A conforming code segment is one that does not have a privilege level of its own. It can be called by programs at any privilege level and then, it acquires the CPL of the calling program. Bit 2 of the access byte of a code segment descriptor specifies if it is a conforming ($C = 1$) or non-conforming ($C = 0$) segment. This mechanism is well suited to handle programs that share code but run at different privilege levels, e.g., shared libraries.

It permits sharing of procedures, so they can be called from various privilege levels. They are usually used for math library and exception handlers.

For a conforming code segment, the selector’s lowest two bits need not be the same as the CPL bits in the access byte of the code segment’s descriptor.

Non-Conforming Code Segment

But most code segments are not conforming. The basic rules of privilege mean that, for nonconforming segments, control can be transferred only to executable segments at the same level of privilege. There is a need, however, to transfer control to higher privilege levels – for example, when a low privilege application program wants to use a system service which is at a high privilege level. An example will be when an application program wants to write data to disk, which involves the use of a device driver. In this case, an indirect access is allowed through a ‘gate’, which is stored in the LDT or GDT.

There are three types of gates for this:

- Task gate
- Interrupt gate
- Call gate

In this section, only the call gate will be described.

To allow a program to jump to a more privileged level, it must go through Call gates, which basically defines the entry points for the privileged code. Corresponding security checks will be conducted at

those entry points to decide whether the invoking code has sufficient rights. These security checks are enforced by operating systems.

14.10.7 | Call Gates

A call gate is an 8-byte entity which contains the following information:

- Segment selector of the code segment to be accessed.
- Entry point for a procedure in the specified code segment (offset in segment).
- Privilege level required for a caller trying to access the procedure (DPL).
- Parameter count – if a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.

Access Control Policy for Call Gates

- In general, $CPL \leq DPL$ of the call gate (numerically).
- For a CALL instruction: DPL of the target code segment $\leq CPL$ (only calls to the more privileged code segment are allowed through a gate) – in numerical terms.
- For a JMP instruction: DPL of the code segment = CPL . Gates cannot be used for jumping to a different privilege level.

This indirect access mechanism permits access only if a certain check is done. The privilege level of the calling program (i.e., CPL) is compared with that of the call gate. If the former has a privilege level higher than or equal to the DPL of the gate, access is allowed – otherwise it is denied. This can be elaborated this way – a call gate with a DPL of 2 can be used by a program with CPL 2 to access a system service at level 0 – but the same gate cannot be used by a program with a CPL of 3. Figure 14.24 shows a call gate.

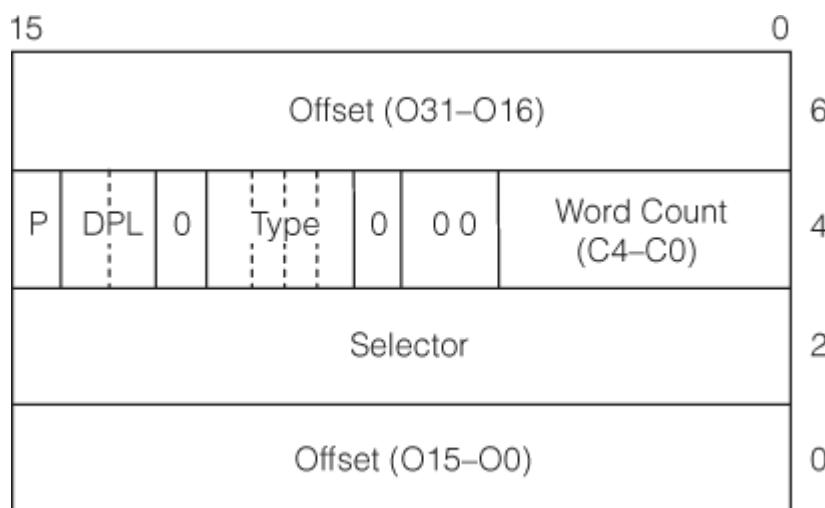


Figure 14.24 | Structure of a call gate

How to use call gates?

The instruction Call xxxx may be used where xxxx may be specified as a gate name but usually generates a selector leading us to an entry in the GDT or LDT, where the gate is stored. In the call gate, the detail of the target code segment is available and using this, the calling program enters the code segment at a higher privilege but only at the defined (in the gate) entry point. Remember that a gate cannot be used to enter code segments at lower privilege levels.

14.11 | Multi Tasking

What is meant by the word multitasking in the context of computers and processors? If a computer can handle multiple tasks at the same time, it has the capability of multitasking. However, the phrase ‘at the same time’ is a bit tricky. Can a single processor perform the task of executing multiple programs at the same time, i.e., in parallel?

Unlikely. Most of us have observed Windows OS allowing multiple tasks to be attended to, simultaneously. Then, there are systems with just one processor but multiple terminals and multiple users. It seems like all the users’ programs are being done concurrently. However, actually, that is not the case. A single processor can perform the execution of only one code sequence. When multiple tasks are to be handled, task switching is being done.

A program in execution is called a process, or task. Suppose there are 10 tasks that the system has to handle – the OS has to arrange a sequence of how to organize the execution of these tasks. This multitasking OS can use one of the many concepts available. There is the ‘time slice’ method in which one task is given a certain time to execute – after this, the task has to give way to the execution of the next task and so on. In this way, all the ten tasks are executed in one round, and the user/users feel that all tasks are running concurrently. In the next round, once again all the tasks get executed and this continues.

In priority-based task scheduling, tasks which are more important are allowed to execute, while low priority tasks are made to wait. There is also the co-operative scheduling system in which a task is allowed to execute until its completion. It is the OS which decides the task scheduling mechanism. Windows is a multiuser OS while DOS is a single user OS.

Task Switching

When one task is temporarily abandoned and another is taken up, context switching or task switching is said to have occurred.

14.11.1 | Issues in Multi-Tasking Systems

One main issue in task switching is that, before abandoning a task temporarily, its ‘context’ is to be saved so as to resume execution at the point at which it had left off. The context consists of the content of the registers being used, the state of the stack and all the information necessary to resume the old task. This is similar to returning from an interrupt, but more variables and more parameters have to be saved before switching to a new task.

If a processor is said to have ‘multitasking’ capability, it means that it has hardware features to handle all the issues associated with task switching. It should have the necessary mechanisms, instructions, registers and necessary additional hardware. These features are available in the 80386 and are useful in designing a multitasking OS for this processor.

TSS and LDTs

It was mentioned earlier that if there are ‘n’ tasks in a system, there will be n LDTs – i.e., for each task there is an LDT and each LDT holds the descriptors of the segments required to run the particular task. When a particular task is taken up for execution, the selector for its LDT is loaded into the LDTR in the processor. Thus, we need to remember that an LDT is associated with a particular task and that there are as many LDTs in the system, as there are tasks. The descriptors of all the LDTs of a system are stored in the GDT of the system.

Similar to this, there is another structure associated with a particular task and that is its ‘Task State Segment’ or TSS. Each task has a TSS. All the information pertaining to a particular task is stored in its TSS, which is also a memory array or segment stored in main memory.

The TSS has the following fields.

1. A dynamic set which changes every time a task switch occurs. They are:

1. The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI).
2. The segment registers (ES, CS, SS, DS, FS, and GS).
3. The flags register (EFLAGS).

4. The instruction pointer (EIP).
 5. The selector of the TSS of the previously executing task (updated only when a return is expected).
2. A static set that the processor reads but does not change. This set includes:
- o. The selector of the task's LDT.
 1. The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
 2. Pointers to the stacks for privilege levels 0-2.
 3. The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs.
 4. The I/O map base.

Figure 14.25 shows the structure of a typical TSS.

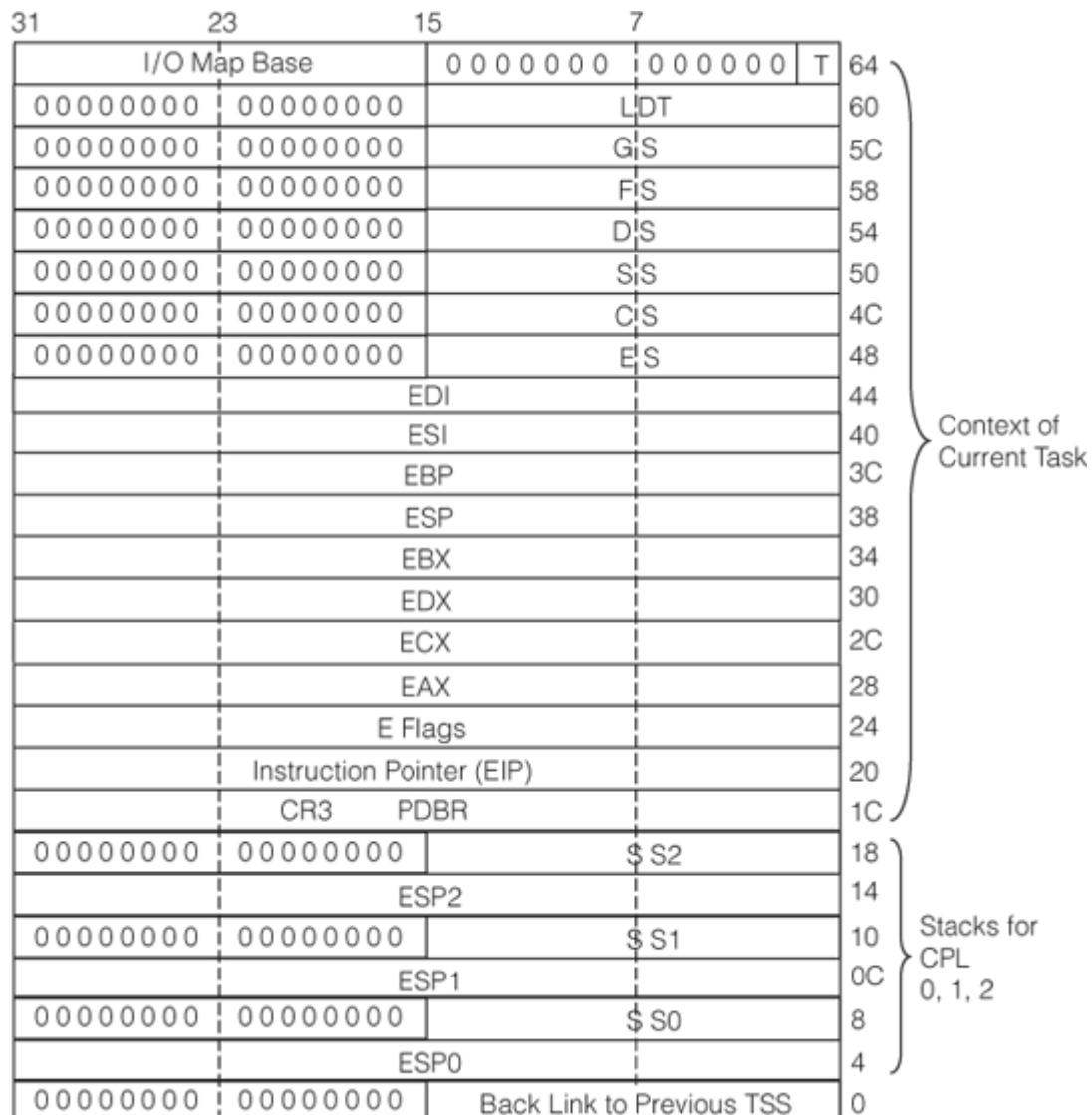


Figure 14.25 | Structure of a typical TSS

The TSS is a segment, and hence, it has a descriptor that is stored in the GDT of the system (just as the LDT's descriptor is). The descriptor of a TSS is similar to a segment descriptor except that it has a 'B' or busy bit. The B-bit in the type field indicates whether the task is busy. The B-bit allows the processor to detect an attempt to switch to a task that is already busy.

14.11.2 | Task Register

The processor has a register called a task register (TR) which points to the current task. When a particular task is taken up, the selector of the TSS is loaded into the visible portion of the task register. The selector contains the index that locates the descriptor of the corresponding TSS in the GDT. Then, the descriptor of the TSS of this task gets loaded into the hidden part of the register.

Two important points regarding the execution of the current task is that it has the TSS selector in the Task register (TR) and the LDT selector in the LDT register (LDTR). This initiates the actions necessary to access the contents of the LDT and TSS that belong to that task. When task switching occurs, the contents of these registers are changed.

14.11.3 | Task Switching

How can task switching be done?

- The current task executes a JMP or CALL that refers to a TSS descriptor.
- The current task executes a JMP or CALL that refers to a task gate.
- An interrupt or exception vectors to a task gate in the Interrupt Descriptor Table.
- The current task executes an IRET when the NT flag is set.

Let us discuss the above four cases in more detail.

1. In the first case, using instructions like JMP taskname or CALL taskname can load the selector of the corresponding TSS into the task register (TR) which gets the TSS descriptor from the GDT.

2. In the second case, the task switching is indirect. There is a 'task gate' involved just like the call gate we had discussed in the previous section. In both cases, a new task is embarked upon.
3. Task switching can also occur by the occurrence of an interrupt. This is also logical, but it involves an 'interrupt descriptor table', which we are yet to discuss. In any case, we know that an interrupt or exception causes the current task to be abandoned and a new one to be taken up. Thus, it can

lead to a task switching when there is the mechanism to access the TSS of a new task.

4. Whether invoked as a task or as a procedure of the interrupted task, an interrupt handler always returns control to the interrupted procedure in the interrupted task. If the Nested Task (NT) flag is set, however, the handler is an interrupt task, and the IRET switches back to the interrupted task.

What are the actions that follow a task switch?

1. If the switching occurs by a JMP or CALL instruction, the usual rules of privilege levels apply for the TSS descriptor or task gate, and task switching occurs only if the conditions are satisfied. Checking of the P bit and limit bit of the TSS descriptor is also done.
2. The context of the current task is saved, i.e., contents of the registers are copied to the TSS. The EIP register, which is now stored in the TSS, will be pointing to the next instruction after the one that caused the task switch. These actions correspond to ‘saving the context’ of the abandoned task.
3. Next, the actions for taking up the new task are begun. The Task Register (TR) is loaded with the new task’s selector. The selector is either the operand of a control transfer instruction or is taken from a task gate, which gets the TSS descriptor and then the TSS from memory.
4. Now the new task can be proceeded with. For that, its state is loaded from the TSS into the corresponding registers. The registers loaded are the LDT register, the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, the segment registers ES, CS, SS, DS, FS, and GS, and PDBR.
5. Every task switch also sets the TS (task switched) bit in the MSW (machine status word), i.e., CR0.

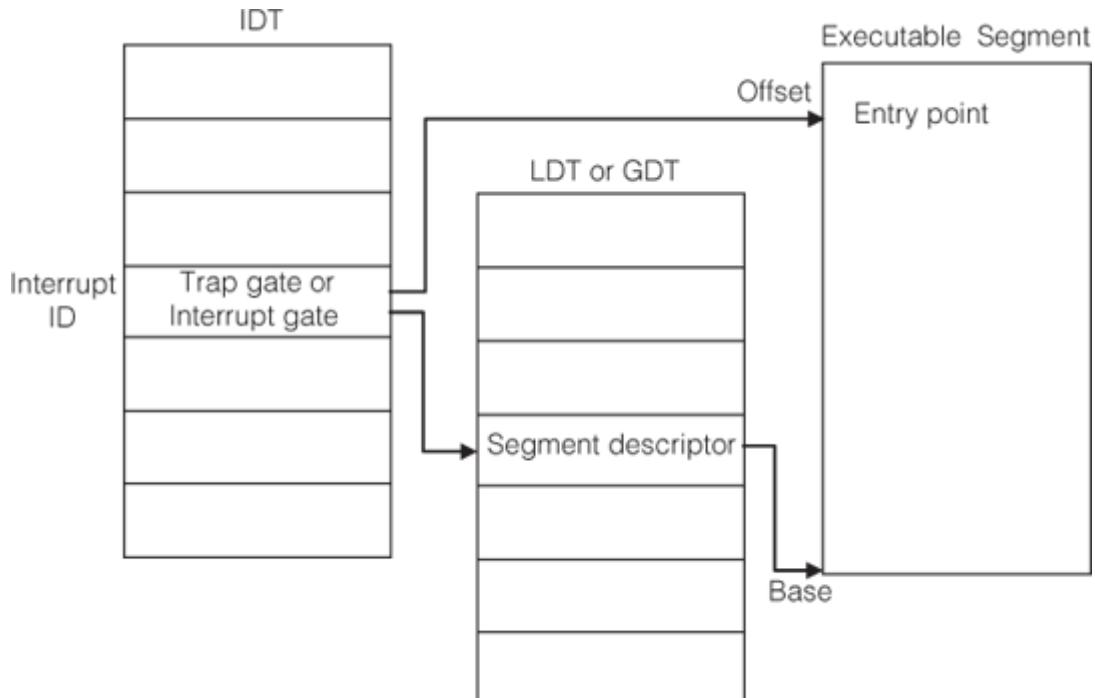
14.11.4 | Task Linking

The back-link field of the TSS and the NT (Nested Task) bit of the flag word together allow the 80386 to automatically return to a task that called another task or was interrupted by another task. When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task’s TSS and, at the same time, sets the NT bit in the new task’s flag register. The NT flag indicates whether the back-link field is valid. The new task releases control by executing an IRET instruction. When interpreting an IRET, the 80386 examines the NT flag. If NT is set, the 80386 switches back to the task selected by the back-link field.

14.12 | Interrupts of 80386

The interrupts of 80386 function exactly in the same way as the 8086 interrupts and the categorization is also similar, i.e., the hardware interrupts are NMI and INTR and the software interrupts can be used in the format INT n . There are interrupts generated due to errors and they are called exceptions. The interrupt response is also similar, i.e., the interrupt vector is in a table. On reset, this table is at location 0 and has a maximum size of 1 K.

Now, for the differences – instead of the ‘interrupt vector table’, here it is the ‘Interrupt Descriptor Table (IDT)’ that has the interrupt vectors. There is one and only one IDT for a system. It can be located anywhere in the physical memory, and that is specified by the descriptor of this table which will be in the GDT. When the system is in protected mode, the descriptor of the IDT is loaded into the IDT register (IDTR). Figure 14.26 shows the interrupt structure of 80386.



Task gates have been discussed already. An interrupt can cause task switching if the target segment corresponds to a different task. That is why we can have task gates also in the IDT.

We know that a gate contains an entry point into a code segment. It contains a selector of the destination code segment and the offset of the allowed entry point. Call gates have already been discussed. An interrupt is similar to a call – the response is similar, but the initiation mechanism is different. Also an interrupt causes the flags to be saved on the stack, and the return instruction is IRET rather than RET. So, we do not have call gates in the IDT but, instead, have trap and interrupt gates.

The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF. The figure shows the channeling of an interrupt from the IDT to the interrupt vector. Because descriptors are involved in this sequence, privilege level checking is done for interrupts as well. The privilege rule that governs interrupt procedures is similar to that for procedure calls.

14.12.1 | Exceptions

Exceptions are caused by error conditions. For the case of the 8086, we know that a divide by zero error causes a type 0 interrupt. We can call it an exception. The 80386 has more error conditions under which interrupts can occur. Table 14.3 shows the list of the 80386 interrupts. Compare it to Table 8.2, which gives the interrupt listing for a PC.

Table 14.3 | List of 80386 interrupts

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupts
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(Reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(Reserved)
16	Coprocessor error
17-31	(Reserved)
32-255	Available for external interrupts via INTR pin

14.13 | Privileged Instructions

The instructions that affect system data structures can only be executed in the protected mode and it is also to be ensured that they are used only at the highest privilege level, i.e., when CPL is zero. These instructions include:

CLTS — Clear Task Switched Flag

HLT — Halt Processor

LGDT — Load GDT Register

LIDT — Load IDT Register

LLDT — Load LDT Register

LMSW — Load Machine Status Word

LTR — Load Task Register

MOV to/from CRn – Move to Control Register n

MOV to/from DRn – Move to Debug Register n

MOV to/from TRn – Move to Test Register n

In fact, all the instructions involving load/store of protected mode registers are privileged instructions.

14.13.1 | Switching to Protected Mode

It has been mentioned that when the processor is powered on, it is in the real mode. How is it switched to the protected mode?

To switch it to protected mode, a number of initialization operations need to be done. These can be done in the real mode itself. (These things can be done in the protected mode as well, but then it must be ensured that the initialization procedures must not use protected-mode features that are not yet initialized.) For example, the GDTR must point to a valid GDT, the IDTR must point to a valid IDT and so on – after all the initializations are done, the switching may be done.

Next, to switch it to protected mode, the PE (Protection Enable) bit of control register CR0 must be set. Those who are familiar with the 80286 will remember that there is a LMSW (Load Machine Status Word) instruction, which was executed with PE = 1. It is this same register MSW that has been re-named as CR0 for the '386.

The difference between the '286 and '386 is that in the case of the former, it is not possible to return to the real mode without doing a hardware reset. However, in the case of the '386, returning to the real mode is possible by clearing the PE bit and loading it to CR0 again. Of course, various conditions will have to be arranged before switching back to the real mode. All these conditions can be put inside a procedure, with the final instruction being the clearing of the PE bit of CR0. Thus, the processor returns to real mode.

14.13.2 | Virtual 8086 Mode

We know that even though the protected mode is important, running programs in MS-DOS has its charms and to facilitate the use of both these modes, the '386 has a virtual mode which allows switching between the real and protected modes. Thus, a multitasking system can have users running programs in the protected mode as well as users

running real mode 8086 programs. Time slices are allotted for all these programs, and thus, there is continuous switching between real and protected mode. Virtual 8086 mode divides the computer into multiple address spaces and maintains virtual registers for each virtual machine. This mode is entered by setting the VM (Virtual Machine) bit in the EFlags register.

The desire to allow execution of MS-DOS applications under the control of a protected-mode environment, (such as Windows) has led to the inclusion of virtual-mode to all of Intel's 32-bit processors. When the processor is running in the virtual machine mode, it behaves as if it were an 8086 equipped with protection, multitasking and paging support.

Note that the virtual 8086 mode has nothing to do with 'virtual memory' but is associated with the term 'virtual machine', which can be explained as follows. 'A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization. The main advantage of system virtual machines is that multiple OS environments can co-exist on the same computer, in strong isolation from each other'.

14.13.3 | Conclusion

With this, we come to the end of our discussion on 80386. Most of the important features have been explained but to be able to write instructions to use all these features, more study and more information (from the programming manual) is required. The attempt here has to give an idea of the capabilities of one of the more sophisticated processors of the Intel family. The 80486 and Pentium have all these features, so these sections have to be understood for appreciating the characteristics of Pentium.

14.14 | Enhanced Features of 80486

1. The 80486 processor was introduced by Intel in 1989. The first important difference that the '486 has in comparison with its predecessor is that it has an integrated floating point unit on the chip itself. Previous processors had the arithmetic unit as a separate unit – the 8086 had 8087, the 80186 had the 80187, the 80286 had the 80287, the '386 had the '387 and so on. Logically, the 80486 should have had an arithmetic co-processor 80487 external to the chip, but instead, Intel placed the arithmetic co-processor inside the chip and called the whole chip as 80486DX. To improve its market segment, Intel also sold a '486 processor without an FPU (floating point unit, as the arithmetic co-processor is

called). It was named as 80486SX, but in fact, it was just the 80486 processor with its FPU turned off.

2. The second important enhancement is that of adding an 8 KB of cache **on the chip** for data as well as code. All predecessor processors had cache outside the processor. With this enhancement, on-chip cache came to be designated as L1 (level 1) cache while off-chip cache is called L2 (level 2) cache. Control register CR0 is used to control the internal cache with two new control bits which were not present in the 80386 microprocessor.
3. The internal cache improved the memory access speed substantially but later versions had something called ‘clock doubling’. New editions were released with higher clock frequencies as they hit on the idea of *doubling* the *internal* clock frequency in relation to the external clock. These double-clocked processors were given the name 80486DX2. A very popular model in this series had an external clock frequency of 33 MHz while working at 66 MHz internally. This principle (*over-clocking*) has been allowed in one way or another in all later generations of CPU’s, though there are problems associated with it. It is just an option allowed for a processor and is specified in the motherboard spec.
4. Compared to the 80386, the ’486 is a heavily pipelined processor. It has a 5 stage pipeline as shown in Fig 14.27. Each stage takes one clock cycle, but once the pipeline is full, each instruction will execute in a single clock. Referring to Fig. 14.27, the stages in the pipeline are as PF-Pre-fetch, D1-decode1, D2-decode2, EX-Execute and WB-Write back. I₁ to I₅ correspond to five instructions in the pipeline. As per this figure, there are two decoding stages. This is because of the varied addressing modes of 80486 and the necessity for protection checks before any access is allowed. This processor has a 5 stage pipeline – if the pipelining stages are split into smaller and smaller steps, it is called ‘super pipelining’, but such an idea is not used in the 80486.
5. ’486 processors and above support burst mode transfers. This feature is helpful in saving time when accessing data from consecutive addresses. Let us try to understand this important feature. Figure 14.28 shows the read cycle timing for an ordinary memory read cycle. Two clock cycles are required to transfer data (which is 32 bits long). The first clock cycle T₁ provides the memory address and control signals, while in T₂, data is transferred between the memory and the microprocessor. The figure shows the case of two 32-bit words being transferred. ADS is address strobe. Thus, two clock cycles are required to transfer 32 bits of data. If 128 bytes of data is to be transferred, obviously 8 clock cycles will be required. Now, let us contrast this with a burst cycle. A burst cycle will require only 5 clock cycles to transfer 128 bytes of data. How do we get such a number? See Fig 14.29.

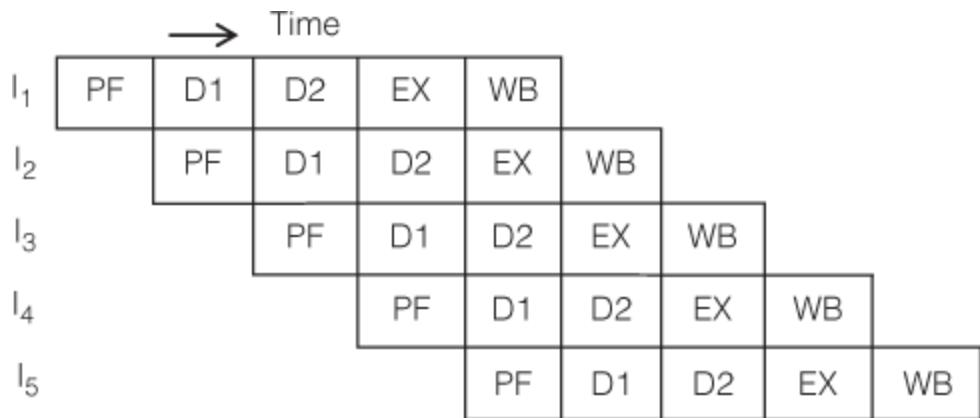


Figure 14.27 | Five stage pipeline of the 80486

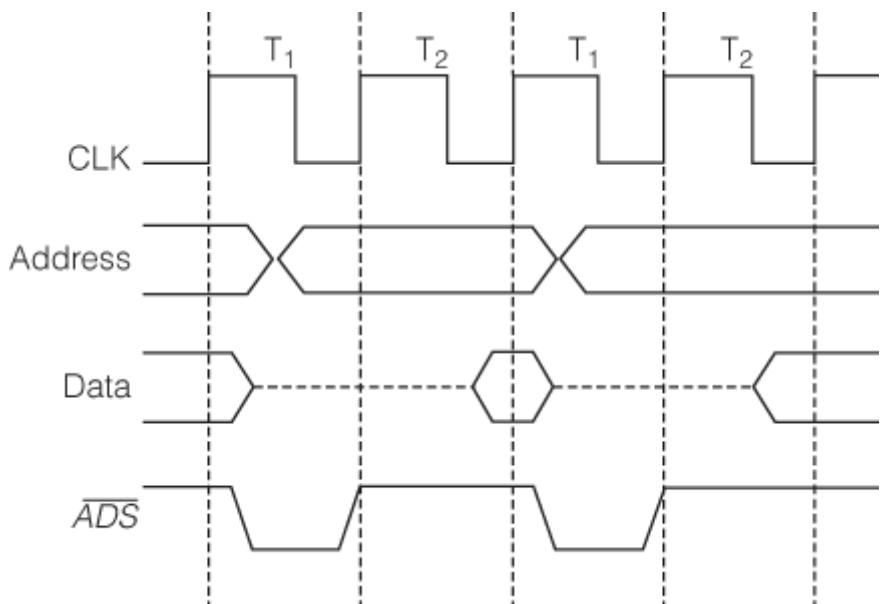


Figure 14.28 | Non-burst bus cycle

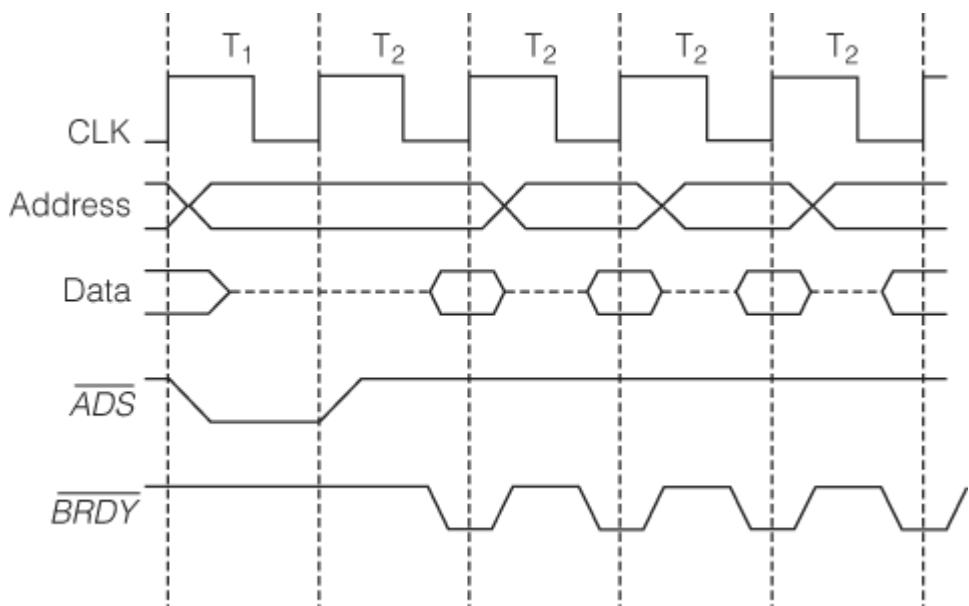


Figure 14.29 | Burst mode data transfer

A burst cycle transfers multiple bytes of data across the bus during one long memory cycle. For example, a transfer of a 128-bit data item across a 32-bit bus would normally occur in four groups, each group containing one 32-bit data. But in the burst mode, the initial address (of the first byte) is used by the processor to compute the remaining addresses for the subsequent data bytes. There are various ways of accomplishing this, but to understand it, think of it this way – a sequential burst order from address zero would first transfer the 32-bit data residing at address zero. The next transfers would be to the data at addresses 4, 8 and 12, in order.

Figure 14.29 shows a burst cycle which reads four 32-bit words (a total of 128 bytes) in five clocking periods. To start the process, the processor sends out the first address and asserts the BLAST signal (not shown in the figure) high. The *BRDY* (Burst ready) signal is asserted in the first T_2 . The processor reads the data word in two cycles and outputs the next address. Since data are at successive addresses, only the lower address bits need to be changed. When the processor has read the required number of data words, it asserts the BLAST signal low to terminate the burst mode. Such a burst read cycle is referred to as a 2-1-1-1 read where four data reads are accomplished in 5 clock cycles.

6. The processor has a major addition in terms of four pins (one for each memory bank) for parity, which is generated by it, during each write cycle. Parity is generated as even parity and a parity bit is provided for each byte of memory. The parity check bits appear on pins DP0–DP3, which are parity inputs as well as parity outputs. These are typically stored in memory during each write cycle and read from memory during each read cycle. On a read, the microprocessor checks parity and generates a parity check error if it occurs on the PCHK pin. A parity error causes no change in processing unless the user applies the PCHK signal to an interrupt input. Figure 14.30 shows the memory banks of 80486 with the bank enable signals (*BE*'s) and the parity check bit for each channel.

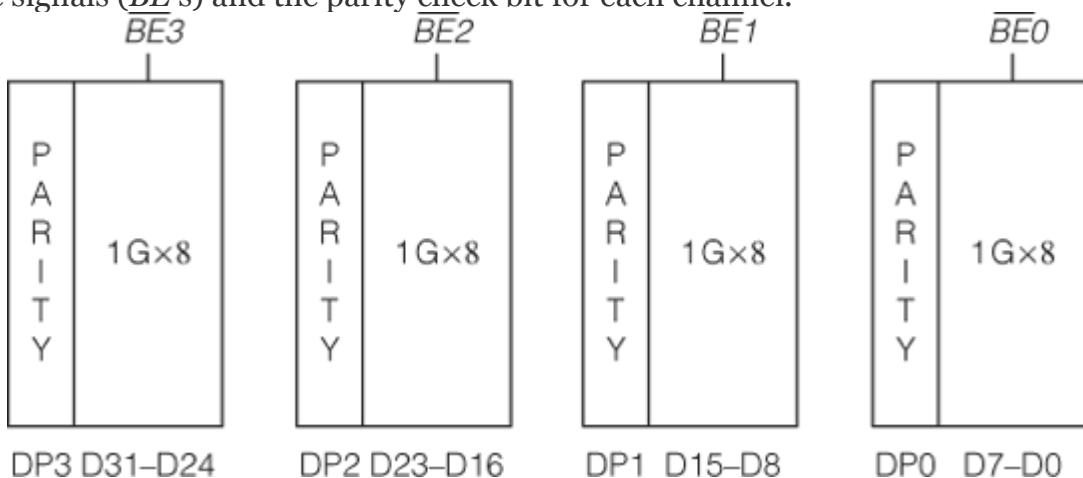


Figure 14.30 | Memory banks of 80486

7. Another enhancement, which needs mention, is the addition of a few new instructions which are listed in Table 14.4.

Table 14.4 | New instructions in the 80486

Mnemonic	Function performed
BSWAP	Byte Swap
XADD	Exchange and add
CMPXCHG	Compare and Exchange
INVD	Invalidate cache
INVLPG	Invalidate TLB entry
WBINVD	Write back and invalidate cache

Of these, the last three instructions relate to cache operations. The first instruction BSWAP is very useful for systems which use big endian format instead of the little endian format used by Intel. Changing from one format to the other just involves the switching of the bytes of a word.

Now, before going ahead to the next topic, let us try to understand the meaning and implications of an issue that needs careful consideration.

14.15 | Data Alignment

In Section 7.3 the idea of memory banks was presented, and the importance of using an even address when accessing 16-bit data was explained. It was made clear that when the memory address is even, memory access takes only one bus cycle, while an odd addressed word incurs a penalty of one more bus cycle. That was for the case of the 8086 where the maximum data width was 16 bits. However, for '386 onwards, the bus width is 32 bits and data is divided into 4 memory banks. For Pentium, the external data bus width is 64 bits. In all these cases, a bus penalty will occur if the memory address is not 'aligned'. A performance penalty occurs due to misaligned data. What is meant by data alignment?

2-byte Data

A 16-bit data item is aligned if it is stored at an address that is a multiple of two. This implies that the least significant bit of the address must be 0. This is the case that we have discussed for the 8086, where the LSB of the address being 0, which means that the address is an even number.

4-byte Data

Let us think of it this way – if all the 4 bytes are in one row, they can be accessed in one cycle, otherwise one more cycle is required. This means that a 32-bit data item is aligned if it is stored at an address that is a multiple of 4. This implies that the least significant two bits of the address must be zero.

8-byte Data

A 64-bit data item is aligned if it is stored at an address that is a multiple of 8. That means that the least significant three bits of the address must be 0. This alignment is important for processors such as Pentium that have a 64-bit data bus.

14.16 | The Pentium Processor

Now, we will come to Intel's most popular name in processors, i.e., Pentium. In 1993, Intel developed its fifth generation x86 processor and decided to call it 80586 (according to the convention followed till then), but to keep away competitors such as AMD and a few others, Intel wanted to trade mark this processor title. However, the lawsuit that followed, declared that numbers cannot be trademarked, and hence, Intel had to find a new name for its new processor, and that is how the word Pentium came into being. It is also called P5. The name Pentium was derived from the Greek word *pente*, meaning 'five', and the Latin ending -ium. Vinod Dham, an engineer from Pune who worked in Intel, is frequently hailed as 'the father of Pentium'.

The Pentium family of processors, which has its roots in the Intel 486 processor, uses the Intel 486 instruction set (with a few additional instructions). The term 'Pentium processor' refers to a family of microprocessors that share a common architecture and instruction set, but each new version of Pentium has newer and newer features. The first Pentium processor was released in the year 1993 and it had a lot of new features compared to its predecessor, i.e., 80486. The foremost enhancement is that it has a 64-bit data bus, while continuing to be a 32-bit processor internally. See the list below.

Features of the Pentium Processor family:

- 32-bit Microprocessor (internal registers of 32 bits)
 - 32-bit address bus
 - 64-bit data bus

- Superscalar architecture
 - Two pipelined integer units
 - Capable of ‘less than’ one clock per instruction
 - Pipelined floating point unit
- Separate code and data caches
 - 8 K code cache, 8 K write back data
- Advanced design features
 - Branch prediction

Now, let us go into the details of each of these enhancements.

14.16.1 | Data Bus of 64 Bits

The data bus width has been doubled. What is the advantage of that? It is that data can be received as 64 bits in one go. What can be done with 64-bit data when the processor registers are only 32 bits in width? The answer is that since the bus bandwidth has doubled, double the amount of data can be sent/received from/to the memory.

See Fig. 14.31. There are 8 memory banks activated by eight *BE* signals. From this memory bank array, data can be accessed for internal processing as 8, 16 or 32 bits. However, sending or receiving data outside the chip can be at the rate of 64 bits per memory cycle. This is a major enhancement.

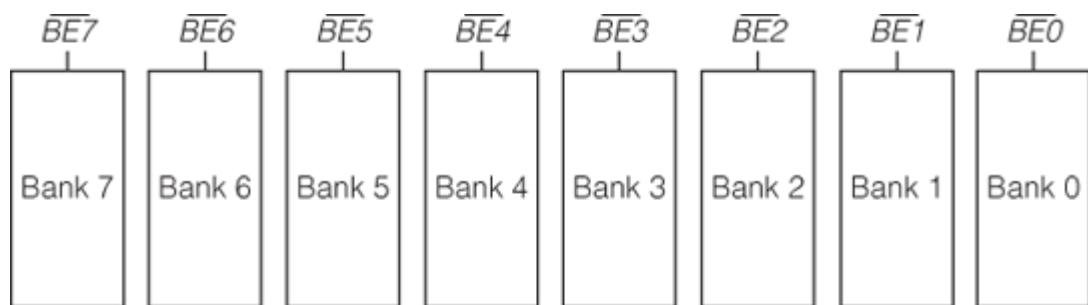


Figure 14.31 | Memory banks of Pentium

14.16.2 | Superscalar Architecture

Pentium is a ‘superscalar processor’, which is pipelined. Pipelining is still only of 5 stages. Being superscalar means it has more than one executing unit – here, there are two integer execution units (each of them is pipelined). They are called the U and V pipes – both these units are slightly different in their processing power, which means that one unit can handle more complex operations than the other. It is the U pipe which can handle more complex operations. See Fig. 14.32, which shows the execution units of the Pentium and the ’486. Note the differences.

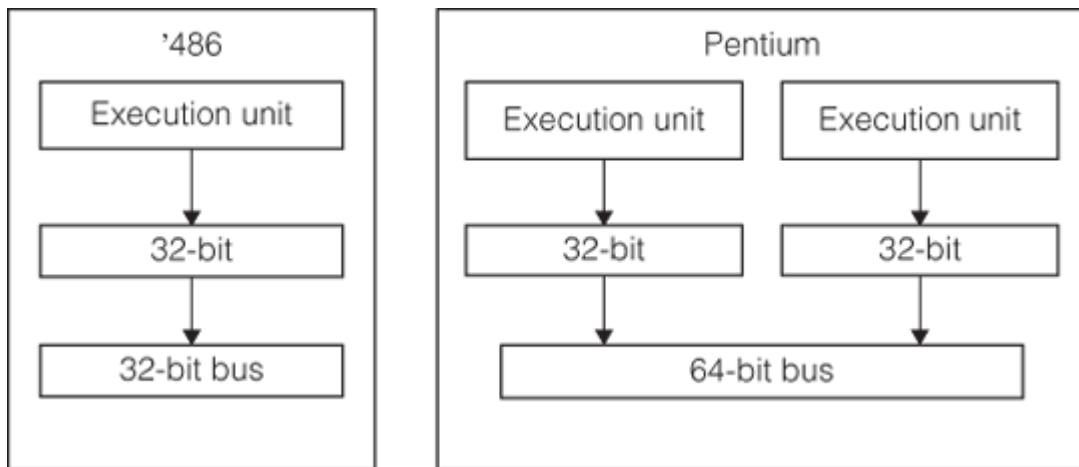


Figure 14.32 | Execution units of pentium and the '486 compared

14.16.3 | Faster Floating Point Unit

The floating-point unit has been completely redesigned over the Intel '486 CPU. Faster operations provide up to 10x speed for common operations including add, multiply and load. The FPU has an eight stage pipeline to speed up operations. Figure 14.33 shows that the units responsible for fetch and decode are common – once decoding is over, it goes to the execution phase. In this phase, the execution occurs in either the U pipe, V pipe or the floating point unit, depending on the data type and complexity of the required operation. Once this is done, the result is written into the data cache.

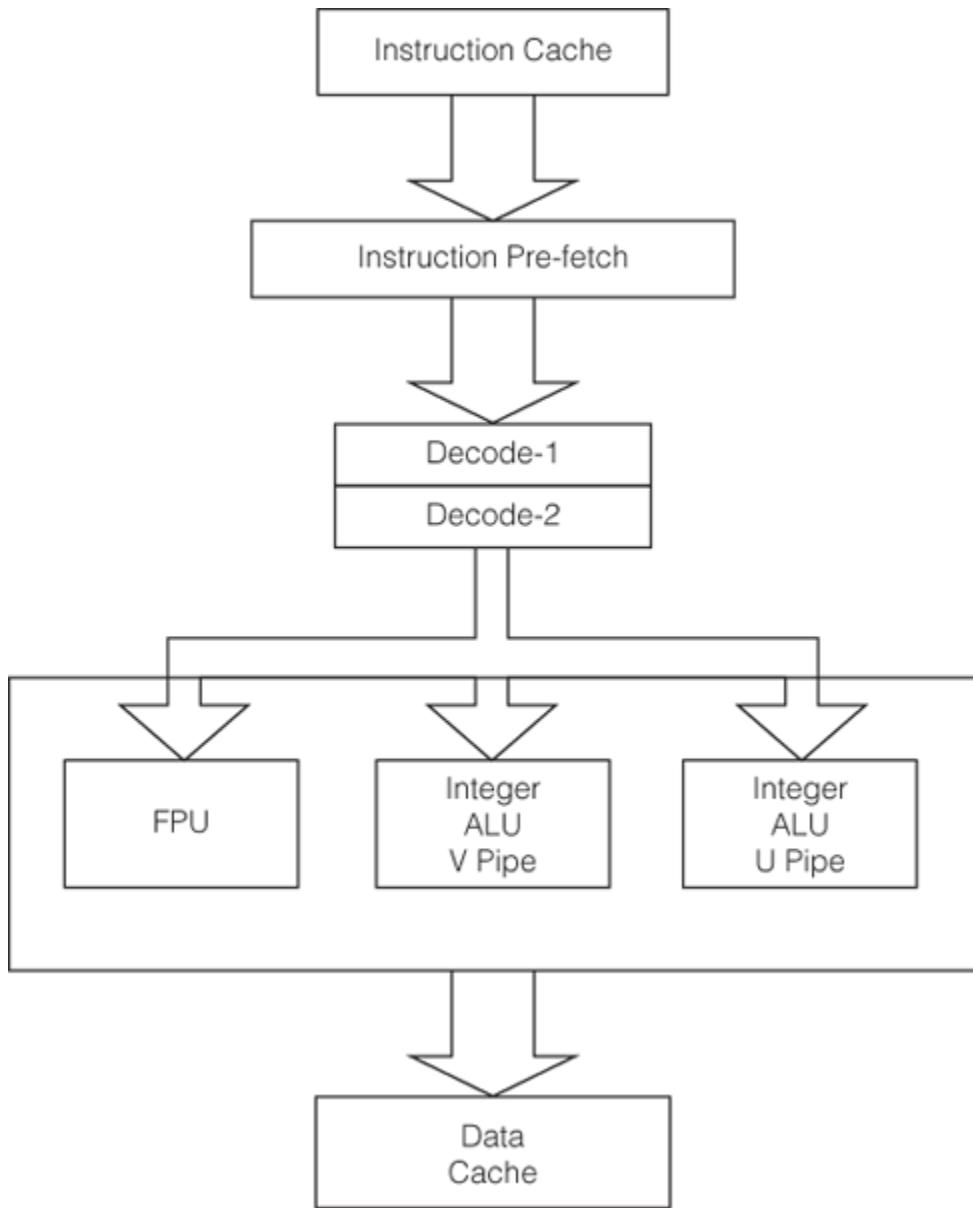


Figure 14.33 | Execution unit of Pentium

14.16.4 | Separate Data and Instruction Cache

Figure 14.33 also shows that there are separate instruction and data caches. Pentium has two separate on-chip caches, one for data and one for code. Each cache is 8 KB in size with a 32-byte line size and is 2-way set associative. Each cache has a dedicated Translation Lookaside Buffer (TLB) to translate linear addresses to physical addresses. The code (instruction) cache is an inherently write-protected cache. The caches can be enabled or disabled by software or hardware.

14.16.5 | Data Dependency Problem

Increasing the number of execution units brings with it a new set of problems. Programs are written by a programmer for serial execution. However, when attempting to run this program on two parallel execution units, the problem of ‘data dependency’ can arise for certain program sequences. For example, if the first program line asks for two numbers to be added and the second line needs the result of the first instruction for its completion, obviously these two instructions cannot be executed in parallel. In such cases, reordering or rescheduling of instructions has to be done, and at present, this problem has been handed over to the compiler to resolve.

14.16.6 | Branch Prediction Unit

Another new feature that Pentium has is the branch prediction unit. This is useful in preventing ‘pipeline stalling’ in the case of a branch instruction occurring in the sequence.

Let us take a closer look at what is called the ‘branch unit’. This unit includes the ‘branch execution unit’ and the ‘branch prediction unit’. Think of what should happen if a conditional branch instruction is encountered. First, the condition is to be evaluated, then the branch address calculated and then the code should be fetched from the new branch address and instruction execution should proceed. All these steps involve a delay. To avoid the delay, a technique called ‘speculative execution’ is implemented. In this, a calculated risk is taken and one particular branch is taken for continuing the instruction execution in sequence. In the meanwhile, the branch condition is evaluated. If the ‘speculative execution’ had taken the correct branch, all is well and good. Otherwise, the pipeline will have to be flushed and everything re-done. This is indeed a minor catastrophe considering that the delay so incurred causes a performance penalty. Thus, it is imperative that the branch prediction unit does a good job.

There are two methods of branch prediction—static and dynamic. In the first case, the assumption is that most branch conditions pertain to whether to repeat a loop or not. In most cases, a loop’s exit condition is false; hence, repeating the loop is something that is normally assumed to be necessary. This type of static prediction is simple and fast, but its success rate is good only in cases of loop intensive programs.

The other technique, i.e., dynamic prediction, relies on using the past history of branching, for the program. There is a ‘Branch History Table’ as well as a ‘Branch Target Buffer’. The latter stores the target addresses of previously executed branches, so that speculative execution can use

any of these target addresses when branching has to occur. This assumes that previously used target addresses are likely to be needed again.

Pentium uses both static and dynamic branch prediction and it is found that a success rate of around 75 to 85% is possible.

14.16.7 | Page Size

Remember that the '386 paging unit supported pages with a maximum size of 4 K only. This is so for 80486 as well. The Pentium processors' Memory Management Unit contains optional extensions to the architecture which allow 2 MB and 4 MB page sizes.

14.16.8 | System Management Mode

The first Pentium was released in 1993, and a newer version was released in 1994, which had a new mode of operation besides the real, virtual and protected modes. This mode is the System Management mode, which is an additional feature for efficient power management and to run specific proprietary code.

In this mode, the power dissipation is reduced by switching off all peripherals and even the system when it is not in use. This is not the only feature of this mode, which operates as a high level manager, and has also extra features for enhanced security.

KEY POINTS OF THIS CHAPTER

- The 80286 had many improved features compared to the 8086 and fuelled the development of the PC-AT and the first Windows based operating system – Windows 3.1.
- The 80386 was the next processor of Intel in which the major change is the increase in the number of address and data pins to 32.
- This processor can address 4 GB of physical memory.
- It has a number of new instructions, and, also enhancements to some existing instructions.
- The general-purpose registers of 80386 are 32 bits long.
- It has a number of additional pins compared to its predecessors.
- The idea of virtual memory is of having an enlarged memory space, which is mapped to a smaller physical memory.
- The memory management unit of the 80386 has the capability of paging and segmentation.
- Descriptors are 8-byte entities in which the information corresponding to a segment are stored.

- Selectors are 16 bits long and point to the descriptor it is associated with.
- The paging unit of the MMU can be switched off but not the segmentation unit.
- Protection has many aspects, including protecting tasks from each other, and also the necessity of protecting system level tasks from user tasks.
- The concept of privilege levels is used for prescribing the level of trust, and the level of protection for each segment.
- Call gates are used for accessing code segments at a higher privilege level.
- The 80386 is a multitasking CPU meaning that it can take care of multiple tasks.
- The tasks are not executed concurrently; the CPU time is allotted to different tasks.
- The interrupt vector table is replaced by an Interrupt descriptor table while in the protected mode.
- The 80486 processor has a number of enhanced features compared to its predecessor 80386.
- It has the burst mode of data transfer which increases data transfer speed significantly.
- It has level one cache placed on the chip which was a big step in microprocessor history.
- Intel's first Pentium processor was released in 1993.
- One important enhancement is that it has a 64-bit data bus, while the processor is internally 32-bit.
- It has a superscalar architecture which means that it has two integer execution units.
- It has a faster floating point unit.
- Its instruction and data caches are separate.
- It can have a page size of 2 MB/4 MB.

QUESTIONS

1. What were the major changes incorporated in the 80286 compared to the 8086?
2. What was the major complaint against the 80286?
3. Name the major change that made the 80386 a big leap in processor technology.
4. What is the 'scaled indexed 'mode of addressing?
5. Where can the MOVSX and MOVZX instructions be used?
6. Why is it that the '386 has address lines only from A2 to A31?
7. Does the 80386 have an inbuilt floating point processor?
8. What is the importance of the virtual memory concept for a user?
9. Why is address translation in the PVAM so complex?
10. What does a descriptor mean to a segment?
11. What is the role of a selector in address translation?

12. How many LDTs and GDTs does a system have?
13. Why do we say that a segment with a CPL = 0 is a highly trusted segment?
14. What is the role of RPL in allowing access to a segment?
15. Distinguish between a conforming and a non-conforming segment.
16. Under what condition is a call gate used?
17. What is the main issue to be taken care of in task switching?
18. What does the task register contain?
19. What is meant by the term 'back link to the previous TSS'?
20. What is an IDT? How many IDTs can a system have?
21. What is the difference between 80386 DX and SX?
22. What is L1 cache and how is it different from L2 cache?
23. What is overclocking? What are the possible pitfalls in this idea?
24. How does a burst access cycle of memory contribute to memory speed?
25. What is the penalty incurred in data misalignment?
26. What advantage is obtained by having a 64-bit data bus for a 32-bit processor?
27. What do the following words mean in the context of a Pentium processor?
 1. superscalar architecture
 2. branch prediction
 3. code cache and instruction cache
 4. pipeline stalling

EXERCISES

1. Write a program that uses MOVSX and MOVZX instructions to
 1. add an 8-bit negative number to a sixteen bit negative number,
 2. add a 32-bit negative number to a 16-bit positive number.
2. Write a program to test bits 4, 8, 9 of a 16-bit number in memory.
3. Add the contents of a table of size 27 containing 16-bit numbers using the scaled indexed mode of addressing.
4. Find the difference between the processors 80386DX and 80386SX.
5. In virtual 8086 mode, can the BIOS and DOS instructions be used by 8086 based programs? How?

- [Copy](#)
- [Add Highlight](#)
- [Add Note](#)